EXPERT INSIGHT

Mastering Blockchain

Inner workings of blockchain, from cryptography and decentralized identities, to DeFi, NFTs and Web3

Fourth Edition



Imran Bashir

‹packt›

Mastering Blockchain

Fourth Edition

Inner workings of blockchain, from cryptography and decentralized identities, to DeFi, NFTs and Web3

Imran Bashir

‹packt›

BIRMINGHAM—MUMBAI

Mastering Blockchain

Fourth Edition

Copyright © 2023 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Devika Battike
Acquisition Editor – Peer Reviews: Gaurav Gavas
Project Editor: Meenakshi Vijay
Content Development Editor: Rebecca Robinson
Copy Editor: Safis Editing
Technical Editor: Karan Sonawane
Proofreader: Safis Editing
Indexer: Manju Arasan
Presentation Designer: Rajesh Shirsath
Developer Relations Marketing Executive: Vidhi Vashisth

First published: March 2017
Second edition: March 2018
Third edition: August 2020
Fourth edition: March 2023

Production reference: 2040723

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.
ISBN: 978-1-80324-106-7

www.packt.com

Disclaimer: The information and viewpoints expressed in this book are those of the author and not necessarily those of any of the author's employers or their affiliates.

This book is dedicated with immeasurable love and gratitude to my beloved father.
The most affectionate, selfless, and hardworking man, who sacrificed everything for me.



> *Don't worry, my son! Adversities come and go.*
>
> *—Scientist Bashir Ahmed Khan*

76 → 363 → 1245 → 1003 → 275 → 77 → 60 → 588 → 1 → 12 → 12 → 2215 → ∞

Contributors

About the author

Imran Bashir has an MSc in Information Security from Royal Holloway, University of London. He has a background in software development, solution architecture, infrastructure management, information security, and IT service management. His current focus is on the latest technologies, such as blockchain and quantum computing. He is a member of the Institute of Electrical and Electronics Engineers (IEEE). He has worked in various senior technology roles for different organizations around the world.

I would like to thank Edward Doxey for his dedication and Packt for their support and guidance throughout this project. I also want to thank John Cornyn, who read the previous edition of this book and suggested some corrections, as well as the reviewers of this edition.

I want to thank my wife and children for their support while I was writing, especially during the time I was supposed to spend with them.

Finally, I want to thank my father, who has always been there for me and always will be. And my mother, whose encouragement makes difficult obstacles easy to surmount for me. My parents' prayers and unconditional love for me have enabled me to achieve anything I desire.

Disclaimer: The information and viewpoints expressed in this book are those of the author and not necessarily those of any of the author's employers or their affiliates.

About the reviewers

Brian Wu is a senior blockchain architect and consultant. Brian has over 20 years of hands-on experience across various technologies, including blockchain, big data, cloud, AI, systems, and infrastructure. He has worked on more than 50 projects in his career.

He has written nine books, published by O'Reilly, Packt, and Apress, on popular fields within the blockchain domain, including *Learn Ethereum, First Edition*; *Learn Ethereum, Second Edition*; *Blockchain for Teens*; *Hands-On Smart Contract Development with Hyperledger Fabric V2*; *Hyperledger Cookbook*; *Blockchain Quick Start Guide*; *Security Tokens and Stablecoins Quick Start Guide*; *Blockchain By Example*; and *Seven NoSQL Databases in a Week*.

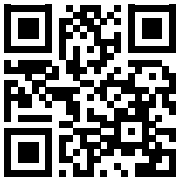
Shailesh B. Nair is a computer engineer and has worked in software development for the last 21 years. He has been involved in blockchain development for about 8 years using different blockchain frameworks, like Ethereum (L1, L2, L3), Substrate, Polkadot, Solana, Cosmos (IBC), CasperLabs, Tezos, etc using various programming languages like C++, Rust, Golang, Ocaml, and Haskell.

He has worked with many different crypto startups since 2014 and as a blockchain consultant, he has worked with Mina Protocol, FTX, and others.

I would like to thank my friends who have worked with me on projects related to blockchain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

Table of Contents

Preface xxvii

Chapter 1: Blockchain 101 1

The growth of blockchain technology 1
- Progress towards maturity • 2
- Rising interest • 3

Distributed systems 4
- CAP theorem • 6
- PACELC theorem • 8

The history of blockchain 8
- Bitcoin • 9
- Electronic cash • 9

Introducing blockchain 11
- Blockchain architecture • 12
  - *Blockchain by layers • 12*
  - *Blockchain in business • 14*
- Generic elements of a blockchain • 14
- Blockchain functionality • 18
- Benefits and features of blockchain • 20
- Limitations of blockchain technology • 21

Types of blockchain 23
- Distributed ledgers • 23
- Shared ledger • 24
- Public blockchains • 24
- Private blockchains • 24
- Semi-private blockchains • 24
- Permissioned ledger • 25

Fully private and proprietary blockchains • 25
Tokenized blockchains • 25
Tokenless blockchains • 25
Layer 1 blockchains • 26
Monolithic and polylithic blockchains • 26
Layer 2 blockchains • 26
Sidechains • 26
Summary **27**

Chapter 2: Decentralization **29**

Introducing decentralization **29**
Methods of decentralization • 33
Disintermediation • 33
Contest-driven decentralization • 34
Quantifying decentralization • 34
Benefits of decentralization • 36
Evaluating requirements • 37
Full-ecosystem decentralization **38**
Storage • 38
Communication • 39
Computing power • 40
Decentralization in practice **42**
Smart contracts • 42
Autonomous agents • 42
Decentralized organizations • 43
Decentralized autonomous organizations • 43
Decentralized autonomous corporations • 43
Decentralized autonomous societies • 44
Decentralized applications • 44
Criteria for a DApp • 46
Operations of a DApp • 46
Design of a DApp • 46
Innovative trends **48**
Decentralized web • 48
Web 1 • 49
Web 2 • 49
Web 3 • 49
Summary **49**

Chapter 3: Symmetric Cryptography 51

Introducing cryptography **52**
Services provided by cryptography • 52
Cryptographic primitives **54**
Keyless primitives • 55
Random numbers • 55
Hash functions • 56
Symmetric key primitives • 67
Message authentication codes • 68
Secret key ciphers • 69
Advanced Encryption Standard **74**
Data Encryption Standard • 74
How AES works • 74
Encrypting and decrypting using AES • 76
Summary **77**

Chapter 4: Asymmetric Cryptography 79

Foundational mathematics **79**
Asymmetric cryptography **80**
Public and private keys • 80
Asymmetric cryptography algorithms • 82
Integer factorization • 82
Discrete logarithm • 82
Elliptic curves • 83
Integrated encryption scheme • 83
Introducing RSA **83**
Encrypting and decrypting with RSA • 85
Introducing ECC **87**
Mathematics behind ECC • 87
Point addition • 88
Point doubling • 91
Point multiplication • 92
The discrete logarithm problem • 93
Generating keys with ECC • 95
Digital signatures **98**
RSA digital signature algorithms • 98

Generating RSA digital signatures • 100
The elliptic curve digital signature algorithm • 100
Generating ECDSA digital signatures • 102
Different types of digital signatures • 104
Blind signatures • 104
Multisignatures • 105
Threshold signatures • 106
Aggregate signatures • 108
Ring signatures • 108
Cryptographic constructs and blockchain technology 109
Homomorphic encryption • 109
Secret sharing • 109
Commitment schemes • 110
Zero-knowledge proofs • 111
zk-SNARKs • 113
zk-STARKs • 115
Zero-knowledge range proofs • 116
Encoding schemes • 116
Base64 • 117
base58 • 117
Verifiable random functions • 117
Summary 118

Chapter 5: Consensus Algorithms 119

Introducing consensus 119
Fault tolerance • 120
FLP impossibility • 121
Analysis and design 122
Model • 122
Processes • 122
Timing assumptions • 123
Classification 123
Algorithms 124
CFT algorithms • 124
Paxos • 124
Raft • 127
BFT algorithms • 129

Practical Byzantine Fault Tolerance • 129
Istanbul Byzantine Fault Tolerance • 134
Tendermint • 137
Nakamoto consensus • 144
Variants of PoW • 146
HotStuff • 151
Choosing an algorithm 155
Finality • 155
Speed, performance, and scalability • 156
Summary 156

Chapter 6: Bitcoin Architecture 157

Introducing Bitcoin 157
Cryptographic keys 159
Private keys in Bitcoin • 159
Public keys in Bitcoin • 160
Addresses 161
Typical Bitcoin addresses • 161
Advanced Bitcoin addresses • 163
Transactions 163
Coinbase transactions • 164
The transaction lifecycle • 165
Transaction validation • 166
Transaction fees • 166
The transaction data structure • 167
Metadata • 169
Inputs • 169
Outputs • 170
Verification • 170
The Script language • 170
Opcodes • 171
Standard transaction scripts • 171
Contracts • 174
Transaction bugs • 175
Blockchain 176
Structure • 176
The genesis block • 178

Stale and orphan blocks • 179
Forks • 179
Properties • 180
Miners **181**
Proof of Work (PoW) • 182
Mining systems • 184
CPU • 184
GPU • 184
FPGAs • 185
ASICs • 185
Mining pools • 186
Network **186**
Types of messages • 187
Client software • 192
Bloom filters • 192
Wallets **194**
Summary **196**

Chapter 7: Bitcoin in Practice **197**

Bitcoin in the real world **197**
Bitcoin payments **198**
Innovation in Bitcoin **200**
Bitcoin improvement proposals • 201
Advanced protocols • 202
Segregated Witness • 202
Bitcoin Cash • 204
Bitcoin Unlimited • 205
Bitcoin Gold • 205
Taproot • 205
Extended protocols on top of Bitcoin • 206
Colored coins • 206
Counterparty • 207
Altcoins from Bitcoin • 208
Bitcoin client installation **209**
Types of clients and tools • 209
Setting up a Bitcoin node • 210
Setting up the source code • 210

Setting up bitcoin.conf • 211
Starting up a node in the testnet • 211
Starting up a node in regtest • 212
Experimenting further with bitcoin-cli 214
Using the Bitcoin command-line tool • 216
Using the JSON-RPC interface • 217
Using the HTTP REST interface • 218
Bitcoin programming 219
Summary 219

Chapter 8: Smart Contracts 221

Introducing smart contracts 221
Definitions • 222
Properties • 222
Real-world application • 224
Ricardian contracts 225
Smart contract templates 229
Oracles 231
Software-and network-assisted proofs • 233
TLSNotary • 233
TLS-N-based mechanism • 233
Hardware device-assisted proofs • 234
Android proof • 234
Ledger proof • 234
Trusted hardware-assisted proofs • 235
Types of blockchain oracles • 236
Inbound oracles • 236
Outbound oracles • 238
Cryptoeconomic oracles • 239
Blockchain oracle services • 239
Deploying smart contracts 240
The DAO 241
Advances in smart contract technology 242
Solana Sealevel • 242
Digital Asset Modeling Language • 243
Summary 245

Chapter 9: Ethereum Architecture 247

Introducing Ethereum **247**
Cryptocurrency **250**
Keys and addresses **250**
Accounts **254**
Transactions and messages **255**
MPTs • 256
Transaction components • 257
Recursive Length Prefix • 261
Gas • 262
Transaction types • 264
Simple transactions • 264
Contract creation transactions • 264
Message call transactions • 265
Messages • 265
Transaction validation and execution • 266
State and storage in the Ethereum blockchain • 267
The world state • 268
The account state • 268
Transaction receipts • 269
Ethereum virtual machine **270**
Execution environment • 273
The machine state • 273
Blocks and blockchain **274**
The genesis block • 276
Block validation, finalization, and processing • 276
Block difficulty mechanism • 278
Nodes and miners **279**
The consensus mechanism • 279
Forks in the blockchain • 281
The Ethereum network **281**
Main net • 282
Test nets • 282
Private nets • 282
Precompiled smart contracts **286**
Programming languages • 287

Solidity • 287
Runtime bytecode • 288
Opcodes • 288
Wallets and client software 289
Wallets • 289
Geth • 289
Light clients • 289
Supporting protocols 290
Whisper • 290
Swarm • 290
Summary 292

Chapter 10: Ethereum in Practice 293

Ethereum payments 294
Innovations in Ethereum 295
Difficulty time bomb • 295
EIP-1559 • 296
The merge and upcoming upgrades • 298
Programming with Geth 298
Installing and configuring the Geth client • 299
Creating a Geth new account • 299
Querying the blockchain using Geth • 301
Geth console • 301
Geth attach • 301
Geth JSON RPC API • 302
Setting up a development environment 304
Connecting to test networks • 305
Creating a private network • 305
Starting up the private network • 307
Experimenting with the Geth JavaScript console • 310
Mining and sending transactions • 312
Introducing Remix IDE 318
Interacting with the Ethereum Blockchain with MetaMask 321
Installing MetaMask • 321
Creating and funding an account with MetaMask • 322
Using MetaMask and Remix IDE to deploy a smart contract • 324
Adding a custom network to MetaMask and connecting it with Remix IDE • 325

Importing accounts into MetaMask using keystore files • 328
Deploying a contract with MetaMask • 331
Interacting with a contract through MetaMask using Remix IDE • 336
Summary 342

Chapter 11: Tools, Languages, and Frameworks for Ethereum Developers 343

Languages 344
The Solidity compiler 344
Installing solc • 344
Experimenting with solc • 345
Tools, libraries, and frameworks 347
Node.js • 347
Ganache • 348
ganache-cli • 348
Ganache UI • 349
Truffle • 351
Drizzle • 352
Other tools • 352
Contract development and deployment 353
Writing smart contracts • 353
Testing smart contracts • 353
Deploying smart contracts • 354
The Solidity language 354
Functions • 355
Variables • 359
Local variables • 359
Global variables • 359
State variables • 360
Data types • 361
Value types • 361
Reference types • 363
Control structures • 365
Events • 366
Inheritance • 367
Libraries • 367
Error handling • 368
Summary 369

Chapter 12: Web3 Development Using Ethereum 371

Interacting with contracts using Web3 and Geth 371
Deploying contracts • 372
Using solc to generate ABI and code • 376
Querying contracts with Geth • 377
Interacting with Geth using POST requests • 380
Interacting with contracts via frontends 381
Installing the web3.js JavaScript library • 382
Creating a web3 object • 383
Creating an app.js JavaScript file • 384
Creating a frontend webpage • 387
Calling contract functions • 388
Creating a frontend webpage • 389
Deploying and interacting with contracts using Truffle 391
Installing and initializing Truffle • 392
Compiling, testing, and migrating using Truffle • 393
Interacting with the contract • 398
Using Truffle to test and deploy smart contracts • 399
Deployment on decentralized storage using IPFS • 404
Summary 406

Chapter 13: The Merge and Beyond 407

Introduction 407
Ethereum after The Merge 408
The Beacon Chain • 409
Beacon nodes • 410
Consensus client • 411
Execution client • 411
Validator client • 411
Proof-of-stake • 415
P2P interface (networking) • 421
The Merge 422
Sharding 432
The future roadmap of Ethereum 440
Summary 441

Chapter 14: Hyperledger 443

Projects under Hyperledger 444
Distributed ledgers • 444
Fabric • 444
Sawtooth • 445
Iroha • 445
Indy • 446
Besu • 446
Libraries • 446
Aries • 446
Transact • 447
Ursa • 447
AnonCreds • 447
Tools • 447
Cello • 448
Caliper • 448
Domain-specific • 448
Grid • 448
Hyperledger reference architecture 449
Hyperledger design principles • 451
Hyperledger Fabric 452
Key concepts • 452
Membership service • 453
Blockchain services • 454
Smart contract services • 456
APIs and CLIs • 456
Components • 456
Peers/nodes • 457
Clients • 457
Channels • 457
World state database • 457
Private data collections • 458
Transactions • 458
Membership Service Provider • 458
Smart contracts • 459
Crypto service provider • 459

Applications • 459
Chaincode implementation • 460
The application model • 462
Consensus mechanism • 462
Transaction lifecycle • 463
Fabric 2.0 465
New chaincode lifecycle management • 465
New chaincode application patterns • 466
Summary 468

Chapter 15: Tokenization 469

Tokenization on a blockchain 470
Advantages of tokenization • 470
Disadvantages of tokenization • 472
Types of tokens 473
Fungible tokens • 473
Non-fungible tokens • 473
Stable tokens • 474
Security tokens • 475
Process of tokenization 475
Token offerings 476
Initial coin offerings • 476
Security token offerings • 477
Initial exchange offerings • 477
Equity token offerings • 477
Decentralized autonomous initial coin offering • 478
Other token offerings • 478
Token standards 479
ERC-20 • 479
ERC-223 • 480
ERC-777 • 480
ERC-721 • 480
ERC-884 • 480
ERC-1400 • 481
ERC-1404 • 481
ERC-1155 • 482
ERC-4626 • 482

Building an ERC-20 token **483**
Building the Solidity contract • 483
Deploying the contract on the Remix JavaScript virtual machine • 488
Adding tokens in MetaMask • 493
Emerging concepts **495**
Tokenomics/token economics • 495
Token engineering • 495
Token taxonomy • 496
Summary **496**

Chapter 16: Enterprise Blockchain 497

Enterprise solutions and blockchain **498**
Success factors • 499
Limiting factors • 500
Requirements **501**
Privacy • 502
Performance • 502
Access governance • 503
Further requirements • 503
Compliance • 503
Interoperability • 504
Integration • 505
Ease of use • 505
Monitoring • 505
Secure off-chain computation • 505
Better tools • 506
Enterprise blockchain versus public blockchain **506**
Enterprise blockchain architecture **507**
Designing enterprise blockchain solutions **509**
TOGAF • 510
Architecture development method (ADM) • 511
Blockchain in the cloud • 513
Currently available enterprise blockchains **515**
Enterprise blockchain challenges **517**
Interoperability • 517
Lack of standardization • 517
Compliance • 518

Business challenges • 518
VMware Blockchain 518
Components • 519
Consensus protocol • 519
Architecture • 520
VMware Blockchain for Ethereum • 522
Quorum 522
Architecture • 522
Nodes • 523
Privacy manager • 524
Cryptography • 525
Privacy • 525
Enclave encryption • 527
Transaction propagation to transaction managers • 527
Enclave decryption • 528
Access control with permissioning • 529
Performance • 531
Pluggable consensus • 531
Setting up a Quorum network with IBFT 532
Installing and running Quorum Wizard • 532
Running a private transaction • 535
Attaching Geth to nodes • 535
Viewing the transaction in Cakeshop • 538
Further investigation with Geth • 539
Other Quorum projects 542
Remix plugin • 542
Pluggable architecture • 542
Summary 543

Chapter 17: Scalability 545

What is scalability? 545
Blockchain trilemma • 546
Methods for improving scalability • 548
Layer 0 – multichain solutions • 549
Polkadot • 549
Layer 1 – on-chain scaling solutions • 551
Layer 2 – off-chain solutions • 555

Layer 2 • 555
Rollups • 559
Data validity • 560
Data availability • 560
How rollups work • 561
Types of rollups • 563
Optimistic rollups • 563
ZK-rollups • 564
Technologies used for building ZK-rollups • 566
ZK-ZK-rollups • 573
Optimistic rollups vs ZK-rollups • 573
Fraud and validity proof-based classification of rollups • 575
Example • 577
Layer 3 and beyond • 579
Summary 580

Chapter 18: Blockchain Privacy 583

Privacy 583
Anonymity • 584
Confidentiality • 584
Techniques to achieve privacy 585
Layer 0 • 586
Tor • 586
I2P • 586
Indistinguishability obfuscation • 586
Homomorphic encryption • 587
Secure multiparty computation • 587
Trusted hardware-assisted confidentiality • 587
Mixing protocols • 588
CoinSwap • 589
TumbleBit • 590
Dandelion • 590
Confidential transactions • 592
MimbleWimble • 592
Zkledger • 593
Attribute-based encryption • 593
Anonymous signatures • 593

Zether • 594
Privacy using Layer 2 protocols • 594
Privacy managers • 594
Privacy using zero-knowledge • 594
Cryptographic Commitments • 595
Zero-knowledge proofs • 597
Building ZK-SNARKs • 601
Example • 610
Summary 616

Chapter 19: Blockchain Security 619

Security 619
Blockchain layers and attacks 621
Hardware layer • 622
Network layer • 623
Blockchain layer • 624
Attacks on transactions • 624
Transaction replay attacks • 625
Attacks on consensus protocols • 626
Double-spending • 627
Selfish mining • 627
Forking and chain reorganization • 627
Blockchain application layer • 628
Smart contract vulnerabilities • 628
DeFi attacks • 631
Interface layer • 631
Oracle attacks/oracle manipulation attacks • 632
Attacks on wallets • 632
Attacks on layer 2 blockchains 634
Cryptography layer • 635
Attacking public key cryptography • 635
Attacking hash functions • 636
Key management-related vulnerabilities and attacks • 636
ZKP-related attacks • 637
Security analysis tools and mechanism 638
Formal verification • 639
Formal verification of smart contracts • 640

Model checking • 641
Smart contract security • 644
Oyente • 645
Solgraph • 646
Threat modeling 646
Regulation and compliance 649
Summary 649

Chapter 20: Decentralized Identity 651

Identity 651
Digital identity 652
Centralized identity model • 652
Federated identity model • 653
Decentralized identity model • 657
Self-sovereign identity • 658
Components of SSI • 659
Identity in Ethereum 672
Identity in the world of Web3, DeFi, and Metaverse 672
SSI-specific blockchain projects 675
Hyperledger Indy, Aries, Ursa, and AnonCreds • 675
Other projects • 676
Some other initiatives • 676
Challenges 676
Summary 677

Chapter 21: Decentralized Finance 679

Introduction 679
Financial markets 681
Trading • 681
Exchanges • 682
Orders and order properties • 682
Order management and routing systems • 683
Components of a trade • 683
Trade lifecycle • 684
Applications of blockchain in finance 685
Insurance • 685
Post-trade settlement • 685

Financial crime prevention • 686
Payments • 688
Decentralized finance 690
Properties of DeFi • 691
DeFi layers • 692
DeFi primitives • 693
DeFi services • 694
Asset tokenization • 694
Decentralized exchanges • 695
Flash loans • 701
Derivatives • 702
Money streaming • 703
Yield farming • 703
Insurance • 704
Decentralized lending – lending and borrowing • 704
Benefits of DeFi • 707
Uniswap • 708
Swap the token • 708
Uniswap liquidity pool • 710
Summary 714

Chapter 22: Blockchain Applications and What's Next 715

Use cases 715
IoT • 716
IoT architecture 716
The physical object layer • 718
The device layer • 718
The network layer • 718
The management layer • 718
The application layer • 718
Benefits of IoT and blockchain convergence • 719
Implementing blockchain-based IoT in practice 722
Setting up Raspberry Pi • 723
Setting up the first node • 725
Setting up the Raspberry Pi node • 726
Installing Node.js • 727
Building the electronic circuit • 728

Developing and running a Solidity contract • 729
Government **735**
Border control • 735
Elections • 737
Citizen identification • 737
Health **738**
Media **739**
Blockchain and AI **740**
Some emerging trends **741**
Some challenges **743**
Summary **745**

Index **747**

Preface

The goal of this book is to teach the theory and practice of distributed ledger technology to anyone interested in learning this fascinating new subject. Anyone can benefit from this book, whether a seasoned technologist, student, business executive, or enthusiast. To this end, I aim to provide a comprehensive and in-depth reference of distributed ledger technology that serves the expert and is also accessible to beginners. I primarily focus on describing the core characteristics of blockchain so that readers can build a strong foundation on which to build further knowledge and expertise. The main topics include core blockchain principles, cryptography, consensus algorithms, distributed systems theory, and smart contracts. In addition, practical topics such as programming smart contracts in solidity, building blockchain networks, using blockchain development frameworks such as Truffle, and writing decentralized applications and descriptions constitute a significant part of this book. Moreover, many types of blockchains, related use cases, and cross-industry applications of blockchain technology are discussed in detail.

This book is a unique blend of theoretical principles and hands-on application. Readers will not only be able to understand the technical underpinnings of this technology, but they will also be able to write code for smart contracts and build blockchain networks. Practitioners can use this book as a reference, and it can also serve as a textbook for students wishing to learn this technology. Indeed, some institutions have adopted previous editions of this book as a primary textbook for their courses on blockchain technology.

This book has six new chapters on the latest topics in blockchain, including scalability, security, privacy, the Ethereum Merge, decentralized identity, and decentralized finance.

I hope that this work will serve well technologists, teachers, students, scientists, developers, business executives, and anyone who wants to learn this fascinating technology for many years to come.

Who this book is for

This book is for anyone who wants to understand blockchain technology in depth. It can also be used as a reference resource by developers who are developing applications for blockchain. It can also be used as a textbook for courses related to blockchain technology and cryptocurrencies, as well as being a learning resource for various examinations and certifications related to cryptocurrency and blockchain technology.

What this book covers

Chapter 1, *Blockchain 101*, introduces the basic concepts of distributed computing, which blockchain technology is based on. It also covers the history, definitions, features, types, and benefits of blockchains, along with various consensus mechanisms that are at the core of blockchain technology.

Chapter 2, *Decentralization*, covers the concept of decentralization and its relationship with blockchain technology. Various methods and platforms that can be used to decentralize a process or a system will also be introduced.

Chapter 3, *Symmetric Cryptography*, introduces the theoretical foundations of symmetric cryptography, which is necessary to understand how various security services such as confidentiality and integrity are provided.

Chapter 4, *Asymmetric Cryptography*, introduces concepts such as public and private keys, digital signatures, and hash functions with practical examples.

Chapter 5, *Consensus Algorithms*, covers the fundamentals of consensus algorithms and describes the design and inner workings of several consensus algorithms. It covers both traditional consensus protocols and blockchain consensus protocols.

Chapter 6, *Bitcoin Architecture*, covers Bitcoin, the first and largest blockchain. It introduces technical concepts related to Bitcoin cryptocurrency in detail.

Chapter 7, *Bitcoin in Practice*, covers the Bitcoin network, relevant protocols, and various Bitcoin wallets. Moreover, advanced protocols, Bitcoin trading, and payments are also introduced. Moreover, various Bitcoin clients and programming APIs that can be used to build Bitcoin applications are covered.

Chapter 8, *Smart Contracts*, provides an in-depth discussion on smart contracts. Topics such as the history, the definition of smart contracts, Ricardian contracts, Oracles, and the theoretical aspects of smart contracts are presented in this chapter.

Chapter 9, *Ethereum Architecture*, introduces the design and architecture of the Ethereum blockchain in detail. It covers various technical concepts related to the Ethereum blockchain and explains the underlying principles, features, and components of this platform in depth. Other topics covered are related to the Ethereum Virtual Machine, mining, and supporting protocols for Ethereum.

Chapter 10, *Ethereum in Practice*, covers the topics related to setting up private networks for Ethereum smart contract development and programming.

Chapter 11, *Tools, Languages, and Frameworks for Ethereum Developers*, provides a detailed practical introduction to the Solidity programming language and different relevant tools and frameworks that are used for Ethereum development.

Chapter 12, *Web3 Development Using Ethereum*, covers the development of decentralized applications and smart contracts using the Ethereum blockchain. A detailed introduction to the Web3 API is provided along with multiple practical examples and a final project.

Chapter 13, *The Merge and Beyond*, introduces the latest development in Ethereum, such as the Beacon Chain, sharding, and future upgrades.

Chapter 14, *Hyperledger*, presents a discussion about the Hyperledger project from the Linux Foundation, which includes different blockchain projects introduced by its members.

Chapter 15, *Tokenization*, introduces the topic of tokenization, stable coins, and other relevant ideas such as initial coin offerings and token development standards.

Chapter 16, *Enterprise Blockchain*, covers the use and application of blockchain technology in enterprise settings and covers DLT platforms such as Quorum.

Chapter 17, *Scalability*, is dedicated to a discussion of one of the challenges, that is, scalablity, faced by blockchain technology and how to address it. We focus on layer 2 solutions to address this problem, however, other solutions are also discussed.

Chapter 18, *Blockchain Privacy*, introduces the problem of lack of privacy in blockchains and explains various techniques to address this limitation. We cover solutions to achieve confidentiality and anonymity in blockchains using techniques such as ZK-SNARKs, mixers, and various other methods.

Chapter 19, Blockchain Security, introduces the various security challenges in blockchains and how to solve them. These include smart contract security, formal verification, security concerns, and best practices at each layer of the blockchain system.

Chapter 20, *Decentralized Identity*, covers one of the hottest topics in the blockchain world. Decentralized identity is a cornerstone of the Web3 ecosytem. In this chapter, we explore the methods, techniques, and ecosystems that underpin the Web3 and decentralized identity landscape.

Chapter 21, *Decentralized Finance*, covers the use and application of decentralized finance, its various aspects, the use cases of blockchain in fianance, and different DeFi protocols.

Chapter 22, *Blockchain Applications and What's Next*, provides a practical and detailed introduction to the applications of blockchain technology in fields other than cryptocurrency, including the Internet of Things, government, media, and finance. It is aimed at providing information about the current landscape, projects, and research efforts related to blockchain technology.

Chapter 23, *Alternative Blockchains*, introduces alternative blockchain solutions and platforms as bonus content that is available online. It covers technical details and features of alternative blockchains and relevant platforms. This is a online chapter and you can read about it at the following link: `https://packt.link/OceZK`.

To get the most out of this book

In order to get the most out of this book, some familiarity with computer science and basic knowledge of a programming language is desirable.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Blockchain-Fourth-Edition`.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`.

Check them out!

Download the color images

We provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://packt.link/5y4vk`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: "Tapscript also enables easier future soft fork upgrades by using the new `OP_SUCCESS` opcode."

A block of code is set as follows:

```
function ()
{
    throw;
}
```

Any command-line input or output is written as follows:

```
"Please send 0.00033324 BTC to address 1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3".
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "**ACCOUNTS & KEYS** provides options to configure balance and the number of accounts to generate."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit `http://www.packtpub.com/submit-errata`, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `http://authors.packtpub.com`.

Share Your Thoughts

Once you've read *Mastering Blockchain - Fourth Edition*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781803241067`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Blockchain 101

It is very likely that if you are reading this book, you already have heard about blockchain and have some fundamental appreciation of its enormous potential. If not, then let me tell you that this is a technology that has promised to positively alter the existing paradigms of nearly all industries, including, but not limited to, the IT, finance, government, media, medical, and law sectors, by making them more efficient, trustworthy, and transparent.

This chapter introduces blockchain technology, its technical foundations, the theory behind it, and various technologies that have contributed to building what is known today as blockchain. The theoretical foundations of distributed systems are described first. Next, the precursors of Bitcoin are presented. Finally, blockchain technology is introduced. This approach is a logical way of understanding blockchain technology, as the roots of blockchain are in distributed systems and cryptography. We will be covering a lot of ground quickly here, but don't worry—we will go over a great deal of this material in much greater detail as you move throughout the book.

In this chapter, we'll focus on:

- The growth of blockchain technology
- Distributed systems
- The history of blockchain
- Introducing blockchain
- Types of blockchain

The growth of blockchain technology

With the invention of Bitcoin in 2008, the world was introduced to a new concept that revolutionized the whole of society. It was something that promised to have an impact upon every industry. This new concept was blockchain, the underlying technology that underpins Bitcoin.

Some describe blockchain as a revolution, whereas another school of thought believes that it is going to be more evolutionary, and it will take many years before any practical benefits of blockchain reach fruition. This thinking is correct to some extent, but, in my opinion, the revolution has already begun.

It is a technology that has an impact on current technologies too and possesses the ability to change them at a fundamental level.

Progress towards maturity

Around 2013, some ideas emerged regarding using blockchain technology for other applications instead of only cryptocurrencies. Then, in 2014, some research and experimentation began, which led to proofs of concept, further research, and full-scale trial projects between 2015 and 2017. In 2015, Ethereum was launched as the first smart contract programmable blockchain, which unlocked many possibilities. Interest in enterprise-grade blockchains originated around the same time. Around that time, we also saw several projects, such as Everledger, Quorum, and Corda.

In addition, the development of novel monolithic and multichain architectures rapidly evolved after 2018. Some key examples of novel protocols include Solana, Avalanche, and Polkadot, which we will discuss later in this book.

Currently, as of the second quarter of 2023, **decentralized finance** (**DeFi**), n**on-fungible tokens** (**NFTs**), and tokenization in general are very popular applications of blockchain. They are already in mainstream usage by millions of users around the world who are engaging in DeFi services. It is expected that within three years or so, DeFi will stabilize into a mature mainstream technology. With all the activity and adoption in DeFi and NFT trading, we can say that to some extent blockchain is already part of our daily lives. Of course, further maturity is required, especially from a regulation and security perspective, but millions of users are already using blockchain regularly, either to make payments, trade NFTs, get loans, or play games.



See the following article for details on which platforms these millions are making use of: `https://beincrypto.com/ethereum-defi-users-reach-new-highs-over-4m-growing-roughly-8x-in-a-year/`

This trend is only expected to grow and, during 2022 and onwards, more and more research and development will be carried out. There will be more focus on the regulation and standardization of blockchain technology. Numerous projects are already production ready, and more adoption is expected in the coming years.



Progress in blockchain technology almost feels like the internet dot-com boom of the late 1990s.

Research in the scalability of blockchains, where blockchains can handle many transactions like traditional financial networks, has led to the development of **layer 2 architectures** and **multi-chain architectures**. Advancement in making zero-knowledge proofs practical helped tremendously in making layer 2 solutions a reality. Layer 2 solutions are under heavy research and development now, and many mechanisms have been introduced, such as Plasma, rollups, sidechains, Lightning Network, and many others.

As of 2022, blockchain is already something that many people use every day. With DeFi, NFTs, cryptocurrencies, and Metaverses, blockchain has attracted millions of users. Most of the attraction is because of financial incentives gained by the trading and investment of digital assets such as NFTs and other DeFi services; however, researchers and academics are also interested in studying and developing the theory of these applications and relevant protocols.

Undoubtedly, further maturation and the adoption of blockchain technology is expected in the coming years.

Rising interest

Interest in blockchain technology has risen quite significantly over the last few years. Once dismissed by some simply as "geek money" from a cryptocurrency point of view or as something that was just not considered worth pursuing, blockchain is now being researched by the largest companies and organizations around the world. Millions of dollars are being spent to adopt and experiment with this technology.

Also, the interest in blockchain within academia is astounding, and many educational establishments—including prestigious universities around the world—are conducting research and development on blockchain technology. There are not only educational courses being offered by many institutions, but academics are also conducting high-quality research and producing many insightful research papers on the topic. There are also several research groups and conferences worldwide that specifically focus on blockchain research. This research community is beneficial for the growth of the entire blockchain ecosystem. A simple online search of "blockchain research groups" would reveal hundreds, if not thousands, of these research groups.



There are also various consortiums, such as **Enterprise Ethereum Alliance** (**EEA**) at `https://entethalliance.org` and Hyperledger at `https://www.hyperledger.org`, that have been established for the research, development, and standardization of blockchain technology. Moreover, the **Institute of Electrical and Electronics Engineers** (**IEEE**) and the **International Standards Organization** (**ISO**) have also started their attempts to standardize various aspects of blockchain technology.

Many start-ups are providing blockchain-based solutions already. A simple trend search on Google reveals the immense scale of interest in blockchain technology over the last few years. Hot topics include **decentralized autonomous organizations** (**DAOs**), fully autonomous and transparent member-governed entities developed using smart contracts with no central authority. Meanwhile, NFTs, through which digital art is routinely bought and sold for millions of dollars, and Metaverses have been making the news.



A Metaverse is a computer-simulated three-dimensional environment. A convergence of our digital and real-world life, they provide a virtual world for the users to perform activities such as social interactions, engaging in business, and shopping. This virtual world is usually accessible via specialized hardware called VR headsets, enhancing the virtual world experience.

This is not a new concept; in the Web 2.0 days (the usual internet that we know and use daily), Second Life, World of Warcraft, and quite a few other similar platforms rose to prominence, as entities centrally owned by shareholders. In the Web 3.0 era (post blockchain), Decentraland and the Sandbox, among many others, are becoming popular, based on decentralized foundations and community governance.

In this book, we are going to learn what exactly blockchain technology is and how it can reshape businesses, multiple industries, and indeed everyday life by bringing about a plenitude of benefits such as decentralized trust, efficiency, cost savings, transparency, and security. We will also explore what decentralization is, smart contracts, and how solutions can be developed and implemented using blockchain platforms such as Ethereum.

While there are many benefits of blockchain technology, there are some challenges too that can cause hurdles in adoption and are being actively researched, such as scalability, privacy, and security. We'll also take a critical look at blockchain and discuss its limitations and challenges.

We shall begin our exploration of blockchain by looking at distributed systems in the following section. This is a foundational paradigm on which blockchain is based, and we must have a good grasp of what distributed systems are before we can meaningfully discuss blockchain in detail.

Distributed systems

Understanding distributed systems is essential to our understanding of blockchain, as blockchain is a distributed system at its core. It is a distributed ledger that can be centralized or decentralized. A blockchain is originally intended to be and is usually used as a decentralized platform. It can be thought of as a system that has properties of both decentralized and distributed paradigms. It is a decentralized-distributed system.

A distributed system is a computing paradigm whereby two or more nodes work with each other in a coordinated fashion to achieve a common outcome. It is modeled in such a way that end users see it as a single logical platform. For example, Google's search engine is based on a large distributed system; however, to a user, it looks like a single, coherent platform. It is composed of processes (nodes) and channels (communication channels) where nodes communicate by passing messages. A blockchain is a message-passing distributed system.

A **node** is an individual player (a computer) in a distributed system. All nodes can send and receive messages to and from each other. Nodes can be honest, faulty, or malicious, and they have memory and a processor. A node that exhibits arbitrary behavior is known as a *Byzantine node* after the **Byzantine Generals** problem.



The Byzantine Generals problem

In 1982, a thought experiment was proposed by Lamport et al. in their research paper, *The Byzantine Generals Problem*, which is available here: `https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/`

In this problem, a group of army generals who lead different parts of the Byzantine army is planning to attack or retreat from a city. The only way of communicating with them is via a messenger. They need to agree to strike at the same time to win. The issue is that one or more generals might be traitors who could send a misleading message. Moreover, the messenger could be captured by the city, resulting in no message delivery. Therefore, there is a need for a viable mechanism that allows agreement among the generals, even in the presence of the treacherous ones, and message loss, so that the attack can still take place at the same time. As an analogy for distributed systems, the generals can be considered as honest nodes, the traitors as Byzantine nodes (that is, nodes with arbitrary behavior), the messenger can be thought of as a channel of communication with the generals, and a captured messenger as a delayed or lost message. Several solutions were presented to this problem in the paper by Lamport et al. in 1982.

This type of inconsistent behavior of Byzantine nodes can be intentionally malicious, which is detrimental to the operation of the network. Any unexpected behavior by a node on the network, whether malicious or not, can be categorized as Byzantine.

A small-scale example of a distributed system is shown in the following diagram. This distributed system has six nodes, of which one (*N4*) is a Byzantine node, leading to possible data inconsistency. *L2* is a link that is broken or slow, and this can lead to a partition in the network:

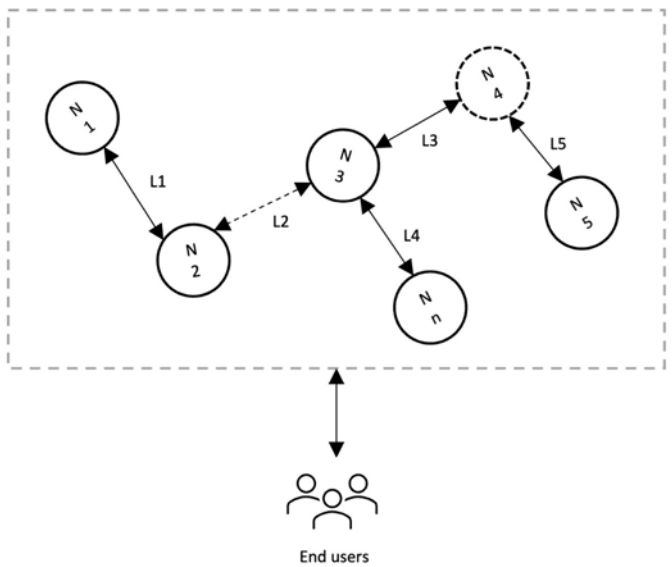


Figure 1.1: Design of a distributed system: N4 is a Byzantine node and L2 is broken or a slow network link

Two key challenges of a distributed system design are the coordination between nodes and fault tolerance. Even if some (a certain threshold dictated by the consensus protocol) of the nodes become faulty or network links break, the distributed system should be able to tolerate this and continue to work to achieve the desired result. This problem has been an active area of distributed system design research for many years, and several algorithms and mechanisms have been proposed to overcome these issues.

Distributed systems are challenging to design. It has been proven that a distributed system cannot have all three of the much-desired properties of consistency, availability, and partition tolerance simultaneously. This principle is known as the **CAP theorem**.

CAP theorem

The **CAP theorem**, also known as Brewer's theorem, was introduced by Eric Brewer in 1998 as a conjecture. In 2002, it was proven as a theorem by Seth Gilbert and Nancy Lynch. The theorem states that any distributed system cannot have consistency, availability, and partition tolerance simultaneously:

- **Consistency** is a property that ensures that all nodes in a distributed system have a single, current, and identical copy of the data. Consistency is achieved using consensus algorithms to ensure that all nodes have the same copy of the data. This is also called **state machine replication** (**SMR**). The blockchain is a means of achieving state machine replication.
- **Availability** means that the nodes in the system are up, accessible, and are accepting incoming requests and responding with data without any failures as and when required. In other words, data is available at each node and the nodes are responding to requests.
- **Partition tolerance** ensures that if a group of nodes is unable to communicate with other nodes due to network failures, the distributed system continues to operate correctly. This can occur due to network and node failures.

A Venn diagram is commonly used to visualize the CAP theorem, as shown below:

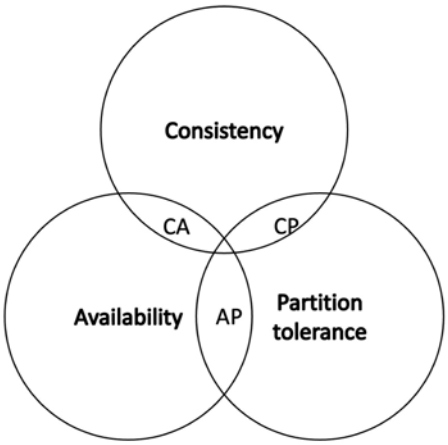


Figure 1.2: CAP theorem

The preceding diagram depicts that only two properties at a time can be attained; either *AP*, *CA*, or *CP*.

In summary:

- If we opt for *CP* (consistency and partition tolerance), we sacrifice availability.
- If we opt for *AP* (availability and partition tolerance), we sacrifice consistency.
- If we opt for *AC* (availability and consistency), we sacrifice partition tolerance.

Note that *AC* does not really exist. The CAP theorem in practice means that in the case of a network partition, a distributed system is either available or consistent. As network partitions cannot be ignored, the choice is between consistency or availability when a network partition occurs.

We can explain this concept with the following example.

Let's imagine that there is a distributed system with two nodes. Now, let's apply the three theorem properties to this smallest of possible distributed systems with only two nodes:

- **Consistency** is achieved if both nodes have the same shared state; that is, they have the same up-to-date copy of the data.
- **Availability** is achieved if both nodes are up and running and responding with the latest copy of data.
- **Partition tolerance** is achieved if, despite communication failure or delay between nodes, the network (distributed system) continues to operate.

Now think of a scenario where a partition occurs and nodes can no longer communicate with each other. If newly updated data comes in now, it can only be updated on one node. If the node accepts the update, then only one node in the network is updated, and consistency is lost. If the node rejects the update, that will result in a loss of availability. This means that either availability or consistency is unachievable due to the network partition. This is strange because somehow, blockchain manages to achieve all these properties, violating the theorem (especially in its most successful implementation, Bitcoin)—or does it?

In blockchains, consistency is sacrificed in favor of availability and partition tolerance. In this scenario, consistency (C) in the blockchain is not achieved simultaneously with partition tolerance (P) and availability (A), but it is achieved over time. This is called eventual consistency, where consistency is achieved due to validation from multiple nodes over time. There can be a temporary disagreement between nodes on the final state, but it is eventually agreed upon. For example, in Bitcoin, multiple transaction confirmations are required to achieve a good confidence level that transactions may not be rolled back in the future. Eventually, a consistent view of transaction history is available in all nodes. Multiple confirmations of a transaction over time provide eventual consistency in Bitcoin. For this purpose, the process of mining was introduced in Bitcoin. Mining is a process that facilitates the achievement of consensus by using the **Proof of Work** (**PoW**) algorithm. At a higher level, mining can be defined as a process that's used to add more blocks to the blockchain. We will cover more on this later in *Chapter 6, Bitcoin Architecture*.

PACELC theorem

An extension of the CAP theorem called PACELC was first proposed by Daniel J. Abadi from Yale University. It states that, in addition to the three properties proposed by CAP, there are also tradeoffs between latency and consistency. It states that tradeoffs between consistency and latency always exist in replicated systems, whereas CAP is only applicable when there are network partitions. In other words, it means that even if no network partitions occur, under normal operation the tradeoff between consistency and latency exists. For example, some databases may choose to give up consistency for lower latency, and some databases could pay the availability and latency costs to achieve consistency. This is true for replicated systems and presents a more inclusive picture of consistency tradeoffs in distributed systems.



PACELC was formally proved in a paper available here: `https://dl.acm.org/doi/10.1145/3197406.3197420`

With a better understanding of distributed systems, let's now talk about blockchain itself. First, we'll begin with a brief rundown of the history of blockchain and Bitcoin.

The history of blockchain

Blockchain was introduced with the invention of Bitcoin in 2008. Its practical implementation then occurred in 2009. Bitcoin will be explored in great depth in *Chapter 6*, *Bitcoin Architecture*. However, it is essential to refer to Bitcoin here because without it, the history of blockchain is not complete.

Now we will look at the early history of computing and computer networks and will discuss how these technologies evolved and contributed to the development of Bitcoin in 2008:

- 1976 – Diffie–Hellman work on securely exchanging cryptographic keys.
- 1978 – Invention of public key cryptography.
- 1979 – Invention of Merkle trees (hashes in a tree structure) by Ralph C. Merkle.
- 1980s – Development of TCP/IP.
- 1980 – Protocols for public key cryptosystems, Ralph C. Merkle.
- 1982 – Blind signatures proposed by David Chaum.
- 1982 – The Byzantine Generals problem.
- 1985 – Work on elliptic curve cryptography by Neal Koblitz and Victor Miller.
- 1991 – Haber and Stornetta work on tamper-proofing document timestamps. This can be considered the earliest idea of a chain of blocks or hash chains.
- 1992 – Cynthia Dwork and Moni Naor publish *Pricing via Processing or Combatting Junk Mail*. This is considered the first use of **PoW**.
- 1993 – Haber, Bayer, and Stornetta upgraded the tamper-proofing of document timestamps system with Merkle trees.

- 1995 – David Chaum's Digicash system (an anonymous electronic cash system) started to be used in some banks.
- 1998 – Bit Gold, a mechanism for decentralized digital currency, invented by Nick Szabo. It used hash chaining and Byzantine Quorums.
- 1999 – Emergence of a file-sharing application mainly used for music sharing, Napster, which is a P2P network, but was centralized with the use of indexing servers.
- 1999 – Development of a secure timestamping service for the Belgian project TIMESEC.
- 2000 – Gnutella file-sharing network, which introduced decentralization.
- 2001 – Emergence of BitTorrent and **Distributed Hash Tables** (**DHTs**).
- 2002 – Hashcash by Adam Back.
- 2004 – Development of B-Money by Wei Dei using Hashcash.
- 2004 – Hal Finney, the invention of the reusable PoW system.
- 2005 – Prevention of Sybil attacks by using computation puzzles, due to James Aspnes et al.
- 2009 – Bitcoin (first blockchain).

These technologies contributed in some way to the development of Bitcoin, even if not directly; the work is relevant to the problem that Bitcoin solved.

Bitcoin

All previous attempts to create anonymous and decentralized digital currency were successful to some extent, but they could not solve the problem of preventing double spending in a completely trustless or permissionless environment. This problem was finally addressed by the Bitcoin blockchain, which introduced the Bitcoin cryptocurrency.

Bitcoin also solves the **SMR problem**, introduced in 1978 by Leslie Lamport and formalized in 1980 by Fred Schneider. SMR is a scheme that's used to implement a fault-tolerant service by replicating data (state) between nodes in a distributed system. Bitcoin solves the problem by allowing the replication of blocks at all correct nodes and ensuring consistency via its PoW mechanism. Here, the agreement is reached between nodes (or replicas) repeatedly to append new blocks to the blockchain.

Electronic cash

The concept of **electronic cash** (**e-cash**), or digital currency, is not new. Since the 1980s, e-cash protocols have existed that are based on a model proposed by David Chaum.

Just as understanding the concepts of distributed systems is necessary to comprehend blockchain technology, the idea of e-cash is also essential to appreciate the first, and astonishingly successful, application of blockchain, Bitcoin, and more broadly, cryptocurrencies in general. To create an effective e-cash system, two fundamental requirements need to be met: **accountability** and **anonymity**.

Accountability is required to ensure that cash is spendable only once (addressing the double-spending problem) and that it can only be spent by its rightful owner. The double-spending problem arises when the same money is spent twice. As it is quite easy to make copies of digital data, this becomes a big issue in digital currencies as you can make many copies of the same digital cash. Spending the same cash twice is known as the double-spending problem.

Anonymity is required to protect users' privacy. With physical cash, it is almost impossible to trace back spending to the individual who actually paid the money, which provides adequate privacy should the consumer choose to hide their identity. In the digital world, however, providing such a level of privacy is difficult due to inherent personalization, tracing, and logging mechanisms in digital payment systems such as credit card payments. This is a required feature for ensuring the security and safety of the financial network, but it is also often seen as a breach of privacy.

This is because end users do not have any control over who their data might be shared with, even without their consent. Nevertheless, this is a solvable problem, and cryptography is used to address such issues. Especially in blockchain networks, the privacy and anonymity of the participants on the blockchain are sought-after features. David Chaum solved both problems during his work in the 1980s by using two cryptographic operations, namely, **blind signatures** and **secret sharing**. These terms and related concepts will be discussed in detail in *Chapter 4*, *Asymmetric Cryptography*. For the moment, it is sufficient to say that *blind signatures* allow the signing of a document without actually seeing it, and a *secret sharing* scheme enables the detection of double-spending.

In 2009, the first practical implementation of an e-cash system named Bitcoin appeared. The term cryptocurrency emerged later. For the very first time, it solved the problem of distributed consensus in a trustless network. It used **public key cryptography** with a PoW mechanism to provide a secure and decentralized method of minting digital currency. The key innovation is the idea of an ordered list of blocks composed of transactions that is cryptographically secured by the PoW mechanism to prevent double-spending in a trustless environment. This concept will be explained in greater detail in *Chapter 6*, *Bitcoin Architecture*.

Looking at all the technologies mentioned previously and their relevant history, it is easy to see how concepts from e-cash schemes and distributed systems were combined to create Bitcoin and what now is known as blockchain. This concept can also be visualized with the help of the following diagram:

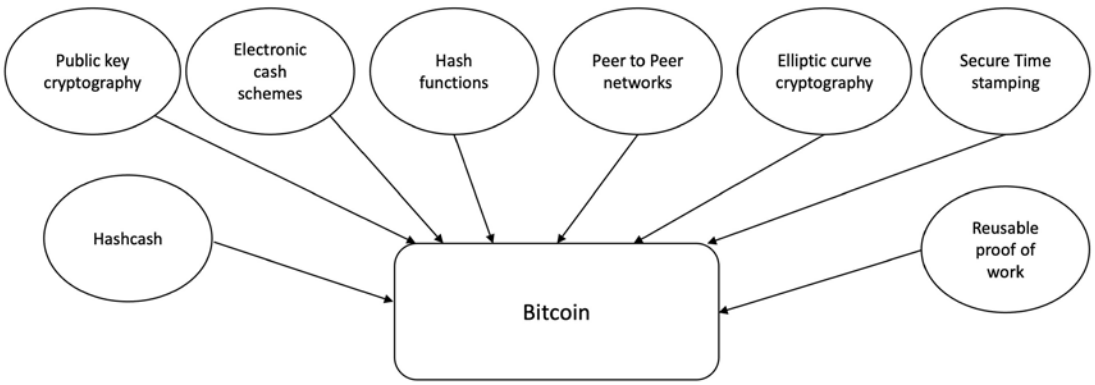


Figure 1.3: The various ideas that supported the invention of Bitcoin and blockchain

With the emergence of e-cash covered, along with the ideas that led to the formation of Bitcoin and blockchain, we can now begin to discuss blockchain itself.

Introducing blockchain

In 2008, a groundbreaking paper, entitled *Bitcoin: A Peer-to-Peer Electronic Cash System*, was written on the topic of peer-to-peer e-cash under the pseudonym of *Satoshi Nakamoto*.



No one knows the actual identity of Satoshi Nakamoto. After introducing Bitcoin in 2009, Nakamoto remained active in the Bitcoin developer community until 2011, before handing over Bitcoin development to its core developers and simply disappearing.

The paper introduced the term **chain of blocks**, later to evolve into "blockchain", where a chronologically ordered sequence of blocks containing transactions is produced by the protocol. The paper is available at `https://bitcoin.org/bitcoin.pdf`.

There are some different ways that blockchain may be defined; the following are two of the most widely accepted definitions:

- **Layman's definition**: Blockchain is an ever-growing, secure, shared recordkeeping system in which each user of the data holds a copy of the records, which can only be updated if a majority of parties involved in a transaction agree to update.
- **Technical definition**: Blockchain is a peer-to-peer, distributed ledger that is cryptographically secure, append-only, immutable (extremely hard to change), and updateable only via consensus among peers.

Now, let's examine things in some more detail. We will look at the keywords from the technical definition one by one:

- **Peer-to-peer**: The first keyword in the technical definition is **peer-to-peer**, or **P2P**. This means that there is no central controller in the network, and all participants (nodes) talk to each other directly. This property allows transactions to be conducted directly among the peers without third-party involvement, such as by a bank.
- **Distributed ledger**: Dissecting the technical definition further reveals that blockchain is a "distributed ledger," which means that a ledger is spread across the network among all peers in the network, and each peer holds a copy of the complete ledger.
- **Cryptographically secure**: Next, we see that this ledger is "cryptographically secure," which means that cryptography has been used to provide security services that make this ledger secure against tampering and misuse. These services include non-repudiation, data integrity, and data origin authentication.
- **Append-only**: Another property that we encounter is that blockchain is "append-only," which means that data can only be added to the blockchain in *time-sequential order*. This property implies that once data is added to the blockchain, it is almost impossible to change that data and it can be considered practically immutable. In other words, blocks added to the blockchain cannot be changed, which allows the blockchain to become an immutable and tamper-proof ledger of transactions.



A blockchain can be changed in rare scenarios where collusion against the blockchain network by bad actors succeeds in gaining more than 51% of the authority. There may also be some legitimate reasons to change data in the blockchain once it has been added, such as the "right to be forgotten" or "right to erasure" (also defined in the GDPR ruling: `https://gdpr-info.eu/art-17-gdpr/`). The right to be forgotten is the right that mandates personal data about a person to be removed from internet records, organizational records, and other associated services. However, those are individual cases that need to be handled separately and that require an elegant technical solution.

- **Updatable via consensus**: The most critical attribute of a blockchain is that it is updateable only via consensus. This is what gives it the power of decentralization. In this scenario, no central authority is in control of updating the ledger. Instead, any update made to the blockchain is validated against strict criteria defined by the blockchain protocol and added to the blockchain only after consensus has been reached among a majority of participating peers/nodes on the network. To achieve consensus, there are various consensus algorithms that ensure all parties agree on the final state of the data on the blockchain network and resolutely agree upon it to be true.

Having detailed the primary features of blockchain, we are now able to begin to look at its actual architecture.

Blockchain architecture

We'll begin by looking at how blockchain acts as a layer within a distributed peer-to-peer network.

Blockchain by layers

Blockchain can be thought of as a layer of a distributed peer-to-peer network running on top of the internet, as can be seen in the following diagram. It is analogous to SMTP, HTTP, or FTP running on top of TCP/IP:

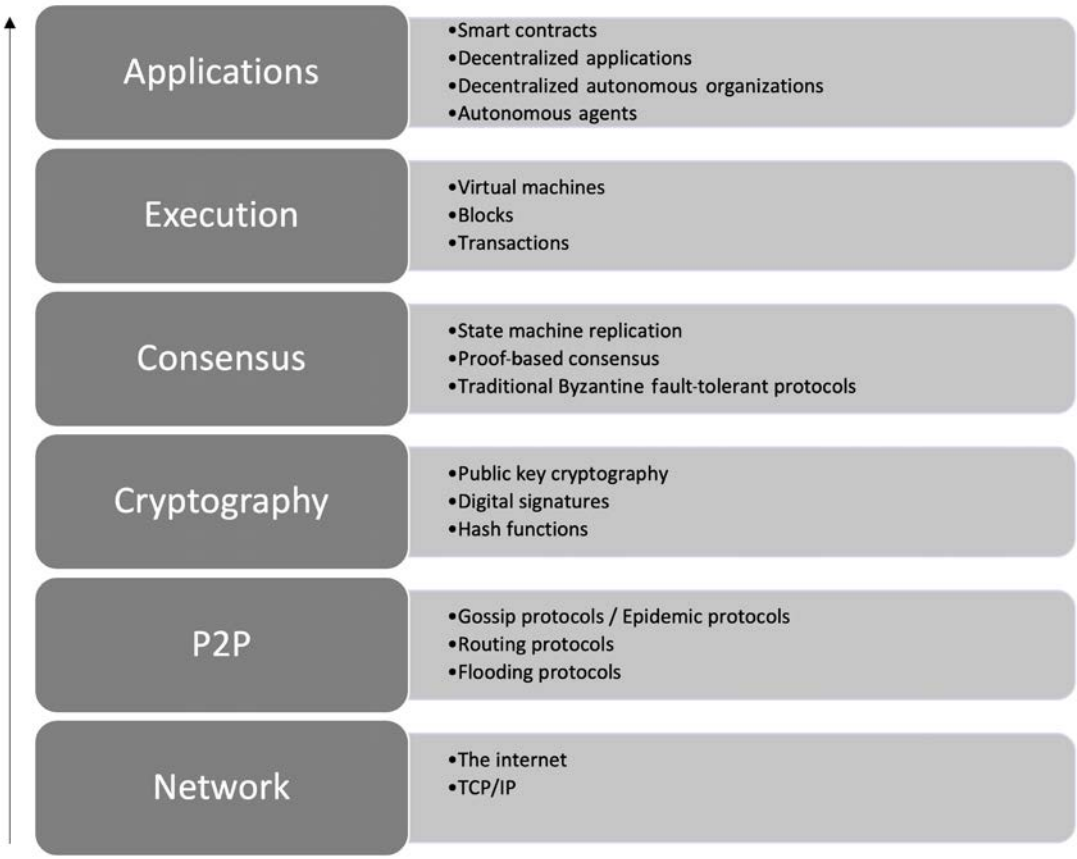


Figure 1.4: The layered architectural view of a generic blockchain

Now we'll discuss all these elements one by one:

- The lowest layer is the **Network** layer, which is usually the internet and provides a base communication layer for any blockchain.
- A **P2P** (peer-to-peer) network runs on top of the **Network** layer, which consists of information propagation protocols such as gossip or flooding protocols.
- After this comes the **Cryptography** layer, which contains crucial cryptographic protocols that ensure the security of the blockchain. These cryptographic protocols play a vital role in the integrity of blockchain processes, secure information dissemination, and blockchain consensus mechanisms. This layer consists of public key cryptography and relevant components such as digital signatures and cryptographic hash functions. Sometimes, this layer is abstracted away, but it has been included in the diagram because it plays a fundamental role in blockchain operations.

- Next comes the **Consensus** layer, which is concerned with the usage of various consensus mechanisms to ensure agreement among different participants of the blockchain. This is another crucial part of the blockchain architecture, which consists of various techniques such as SMR, proof-based consensus mechanisms, or traditional (from traditional distributed systems research) Byzantine fault-tolerant consensus protocols.
- We then have the **Execution** layer, which can consist of virtual machines, blocks, transactions, and smart contracts. This layer, as the name suggests, provides execution services on the blockchain, and performs operations such as value transfer, smart contract execution, and block generation. Virtual machines such as **Ethereum Virtual Machine** (**EVM**), **Ethereum WebAssembly** (**ewasm**), and Zinc VM provide an execution environment for smart contracts to execute.
- Finally, we have the **Applications** layer, which is composed of smart contracts, decentralized applications, DAOs, and autonomous agents. This layer can effectively contain all sorts of various user-level agents and programs that operate on the blockchain. Users interact with the blockchain via decentralized applications.

All these concepts will be discussed in detail later in this book in various chapters. Next, we'll look at blockchain from more of a business-oriented perspective.

Blockchain in business

The current traditional business model is centralized. For example, for cash transfers, banks act as a central trusted third party. In financial trading, a central clearing house acts as a trusted third party between two or more trading parties. From a business standpoint, a blockchain can be defined as a platform where peers can exchange value using transactions without the need for a centrally trusted arbitrator (a trusted third party). This concept is compelling, and, once you absorb it, you will realize the enormous potential of blockchain technology. This disintermediation allows blockchain to be a decentralized mechanism where no single authority controls the network. Immediately, we can see a significant benefit of decentralization here, because if no banks or central clearing houses are required, then it naturally leads to cost savings, faster transaction speeds, transparency, and more trust. Moreover, in the payment business, blockchain can be used to facilitate cross-border and local payments in a decentralized and secure manner.

We've now looked at what blockchain is at a fundamental level. Next, we'll go a little deeper and look at some of the elements that comprise a blockchain.

Generic elements of a blockchain

Now, let's walk through the generic elements of a blockchain. You can use this as a handy reference section if you ever need a reminder about the different parts of a blockchain. More precise elements will be discussed in the context of their respective blockchains in later chapters, for example, the Ethereum blockchain. The structure of a generic blockchain can be visualized with the help of the following diagram:

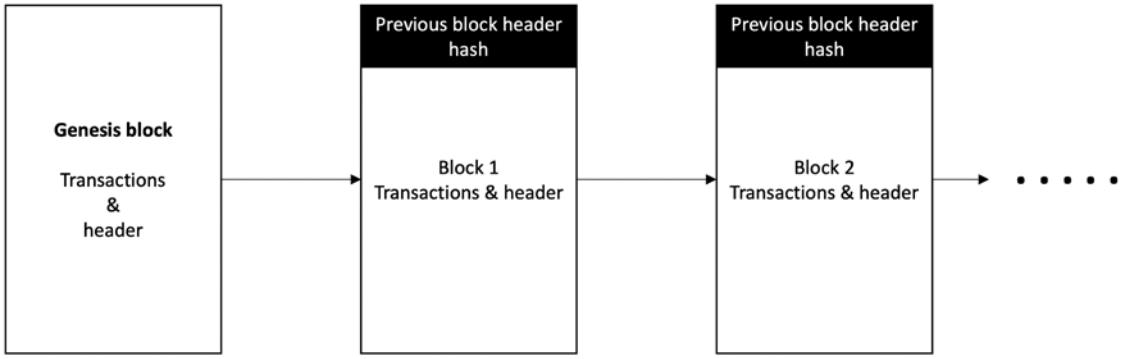


Figure 1.5: Generic structure of a blockchain

Elements of a generic blockchain are described here one by one. These are the elements that you will come across in relation to blockchain:

- **Address**: Addresses are unique identifiers used in a blockchain transaction to denote senders and recipients. An address is usually a public key or derived from a public key.
- **Transaction**: A transaction is the fundamental unit of a blockchain. A transaction represents a transfer of value from one address to another.
- **Block**: A block is composed of multiple transactions and other elements, such as the previous block hash (hash pointer), timestamp, and nonce. A block is composed of a block header and a selection of transactions bundled together and organized logically. A block contains several elements, which we introduce as follows:
    - A reference to a previous block is also included in the block unless it is a genesis block. This reference is the hash of the header of the previous block. A **genesis block** is the first block in the blockchain that is hardcoded at the time the blockchain was first started. The structure of a block is also dependent on the type and design of a blockchain.
    - A **nonce** is a number that is generated and used only once. A nonce is used extensively in many cryptographic operations to provide replay protection, authentication, and encryption. In blockchain, it's used in PoW consensus algorithms and for transaction replay protection. A block also includes the nonce value.
    - A **timestamp** is the creation time of the block.
    - **Merkle root** is a hash of all the nodes of a Merkle tree. In a blockchain block, it is the combined hash of the transactions in the block. Merkle trees are widely used to validate large data structures securely and efficiently. In the blockchain world, Merkle trees are commonly used to allow the efficient verification of transactions. Merkle root in a blockchain is present in the block header section of a block, which is the hash of all transactions in a block. This means that verifying only the Merkle root is required to verify all transactions present in the Merkle tree instead of verifying all transactions one by one.

- In addition to the block header, the block contains transactions that make up the block body. A **transaction** is a record of an event, for example, the event of transferring cash from a sender's account to a beneficiary's account. A block contains transactions, and its size varies depending on the type and design of the blockchain. For example, the Bitcoin block size is limited to one megabyte, which includes the block header of 80 bytes and transactions.

The following structure is a simple block diagram that depicts a generic block:

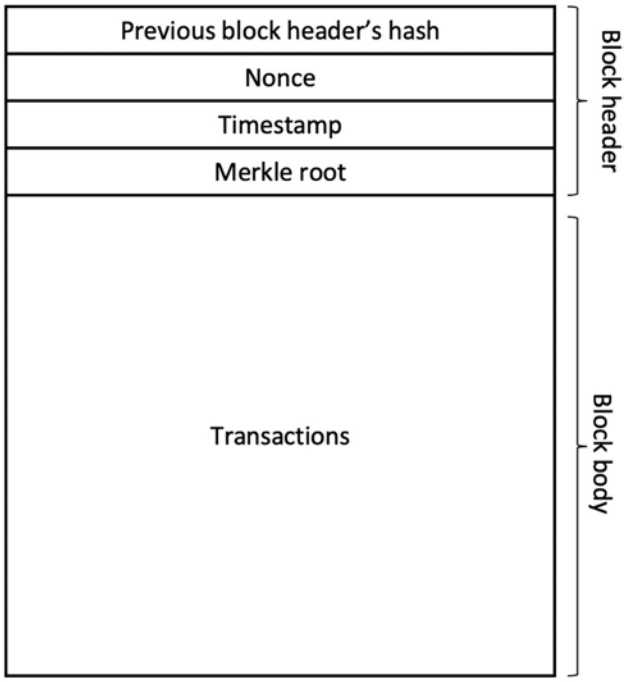


Figure 1.6: The generic structure of a block

Generally, there are just a few attributes that are essential to the functionality of a block: the block header, which is composed of the hash of the previous block's header, the timestamp, nonce, Merkle root, and the block body that contains the transactions. There are also other attributes in a block, but generally, the components introduced in this section are usually available in a block:

- **Peer-to-peer network:** As the name implies, a *peer-to-peer network* is a network topology wherein all peers can communicate with each other directly and send and receive messages.
- **The scripting or programming language:** *Scripts* or *programs* perform various operations on a transaction to facilitate various functions. For example, in Bitcoin, transaction scripts are predefined in a language called **Script**, which consists of sets of commands that allow nodes to transfer bitcoins from one address to another. Script is a limited language in the sense that it only allows essential operations that are necessary for executing transactions, but it does not allow arbitrary program development.



Think of the scripting language as a calculator that only supports standard pre-programmed arithmetic operations. As such, the Bitcoin Script language cannot be called "Turing complete." In simple words, a Turing complete language means that it can perform any computation. It is named after Alan Turing, who developed the idea of a Turing machine, which can run any algorithm, however complex. Turing-complete languages need loops and branching capabilities to perform complex computations. Therefore, Bitcoin's scripting language is not Turing complete, whereas Ethereum's Solidity language is.

To facilitate arbitrary program development on a blockchain, a Turing-complete programming language is needed, and it is now a very desirable feature to have for blockchains. Think of this as a computer that allows the development of any program using programming languages.

- **Virtual machine**: This is an extension of the transaction script introduced previously. A *virtual machine* allows Turing-complete code to be run on a blockchain (as smart contracts), whereas a transaction script is limited in its operation. However, virtual machines are not available on all blockchains. Various blockchains use virtual machines to run programs such as EVM and **Chain Virtual Machine** (**CVM**). EVM is used in the Ethereum blockchain, while CVM is a virtual machine developed for and used in an enterprise-grade blockchain called "Chain Core."
- **State machine**: A blockchain can be viewed as a state transition mechanism whereby a state is modified from its initial form to the next one by nodes on the blockchain network as a result of transaction execution.
- **Smart contracts**: These programs run on top of the blockchain and encapsulate the business logic to be executed when certain conditions are met. These programs are enforceable and automatically executable. The *smart contract* feature is not available on all blockchain platforms, but it is now becoming a very desirable feature due to the flexibility and power that it provides to blockchain applications. Smart contracts have many use cases, including but not limited to identity management, capital markets, trade finance, record management, insurance, and e-governance. Smart contracts will be discussed in more detail in *Chapter 8*, *Smart Contracts*.
- **Node**: A *node* in a blockchain network performs various functions depending on the role that it takes on. A node can propose and validate transactions and perform mining to facilitate consensus and secure the blockchain. This goal is achieved by following a **consensus protocol** (most commonly PoW). Nodes can also perform other functions, such as simple payment verification (lightweight nodes), validation, and many other functions depending on the type of blockchain used and the role assigned to the node. Nodes also perform a transaction signing function. Transactions are first created by nodes and then also digitally signed by nodes using private keys as proof that they are the legitimate owner of the asset that they wish to transfer to someone else on the blockchain network. This asset is usually a token or virtual currency, such as Bitcoin, but it can also be any real-world asset represented on the blockchain by using tokens. There are also now standards related to tokens; for example, on Ethereum, there are ERC20, ERC721, ERC777, and a few others that define the interfaces and semantics of tokenization.

A high-level diagram of blockchain architecture highlighting the key elements mentioned previously is shown as follows:

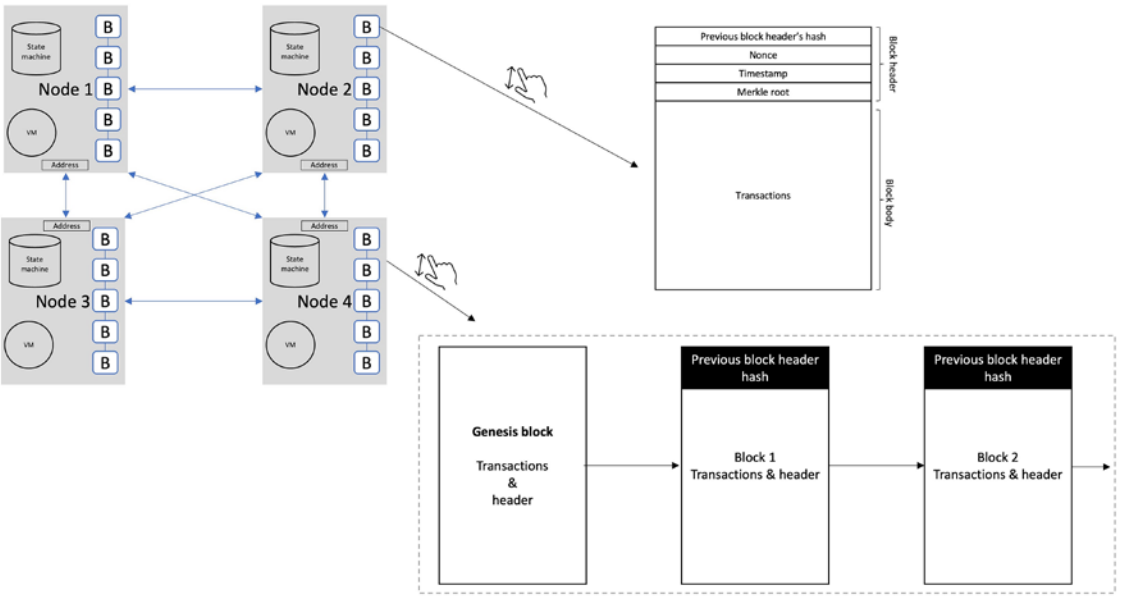


Figure 1.7: Generic structure of a blockchain network

The preceding diagram shows a four-node blockchain network (at the top), each maintaining a chain of blocks, virtual machine, state machine, and address. The blockchain is then further magnified (middle) to show the structure of the chain of blocks, which is again magnified (bottom) to show the structure of a transaction. Note that this is a generic structure of a blockchain; we will see specific blockchains structures in detail in the context of Ethereum and Bitcoin blockchains later in this book.

Blockchain functionality

We have now defined and described blockchain. Now, let's see how a blockchain works. Nodes are either *miners* that create new blocks and mint cryptocurrency (coins) or *block signers* that validate and digitally sign the transactions. A critical decision that every blockchain network must make is to figure out which node will append the next block to the blockchain. This decision is made using a *consensus mechanism*.

Consensus is a process of achieving agreement between distrusting nodes on the final state of data. To achieve consensus, different algorithms are used. It is easy to reach an agreement in a centralized network (in client-server systems, for example), but when multiple nodes are participating in a distributed system and they need to agree on a single value, it becomes quite a challenge to achieve consensus. This process of attaining agreement on a common state or value among multiple nodes is known as **distributed consensus**. If faults are allowed, then we call such a mechanism fault tolerant distributed consensus, where despite the failure of some nodes, agreement is reached between them.

Now, we will look at how a blockchain validates transactions and creates and adds blocks to grow the blockchain, using a general scheme for creating blocks:

1. **Transaction is initiated:** A node starts a transaction by first creating it and then digitally signing it with its private key. A transaction can represent various actions in a blockchain. Most commonly, this is a data structure that represents the transfer of value between users on the blockchain network. The transaction data structure usually consists of some logic of transfer of value, relevant rules, source and destination addresses, and other validation information. Transactions are usually either a cryptocurrency transfer (transfer of value) or a smart contract invocation that can perform any desired operation. A transaction occurs between two or more parties. This will be covered in more detail in specific chapters on Bitcoin and Ethereum later in the book.
2. **Transaction is validated and broadcast**: A transaction is propagated (broadcast) usually by using data-dissemination protocols, such as the Gossip protocol, to other peers that validate the transaction based on preset validity criteria. Before a transaction is propagated, it is also verified to ensure that it is valid.
3. **Find new block:** When the transaction is received and validated by special participants called miners on the blockchain network, it is included in a block, and the process of mining starts. This process is also sometimes referred to as "finding a new block." Here, nodes called miners race to finalize the block they've created by a process known as mining.
4. **New block found:** Once a miner solves a mathematical puzzle (or fulfills the requirements of the consensus mechanism implemented in a blockchain), the block is considered "found" and finalized. At this point, the transaction is considered confirmed. Usually, in cryptocurrency blockchains such as Bitcoin, the miner who solves the mathematical puzzle is also rewarded with a certain number of coins as an incentive for their effort and the resources they spent in the mining process.
5. **Add new block to the blockchain:** The newly created block is validated, transactions or smart contracts within it are executed, and it is propagated to other peers. Peers also validate and execute the block. It now becomes part of the blockchain (ledger), and the next block links itself cryptographically back to this block. This link is called a hash pointer.

This process can be visualized in the diagram as follows:

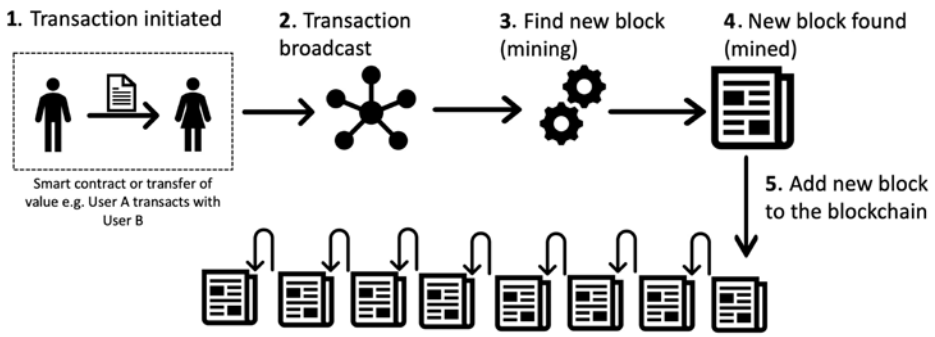


Figure 1.8: How a block is generated

This completes the basic introduction to blockchain. In the next section, you will learn about the benefits and limitations of this technology.

Benefits and features of blockchain

Numerous advantages of blockchain technology have been discussed in many industries and proposed by thought leaders around the world who are participating in the blockchain space. The notable benefits of blockchain technology are as follows:

- **Simplification of current paradigms**: The current blockchain model in many industries, such as finance or health, is somewhat disorganized. In this model, multiple entities maintain their own databases and data sharing can become very difficult due to the disparate nature of the systems. However, as a blockchain can serve as a single shared ledger among many interested parties, this can result in simplifying the model by reducing the complexity of managing the separate systems maintained by each entity.
- **Faster dealings**: In the financial industry, especially in post-trade settlement functions, blockchain can play a vital role by enabling the quick settlement of trades. Blockchain does not require a lengthy process of verification, reconciliation, and clearance because a single version of agreed-upon data is already available on a shared ledger between financial organizations.
- **Cost-saving**: As no trusted third party or clearing house is required in the blockchain model, this can massively reduce overhead costs in the form of the fees, which are paid to such parties.
- **Smart property**: It is possible to link a digital or physical asset to the blockchain in such a secure and precise manner that it cannot be claimed by anyone else. You are in full control of your asset, and it cannot be double-spent or double-owned. Compare this with a digital music file, for example, which can be copied many times without any controls.



While it is true that many **Digital Rights Management** (**DRM**) schemes are being used currently along with copyright laws, none of them are enforceable in the way a blockchain-based DRM can be. Blockchain can provide digital rights management functionality in such a way that it can be enforced fully: if you own an asset, no one else can claim it unless you decide to transfer it. This feature has far-reaching implications, especially in DRM and e-cash systems where double-spend detection is a crucial requirement.

- **Decentralization**: This is a core concept and benefit of blockchain. There is no need for a trusted third party or intermediary to validate transactions; instead, a consensus mechanism is used to agree on the validity of transactions.
- **Transparency and trust**: As blockchains are shared and everyone can see what is on the blockchain, this allows the system to be transparent. As a result, trust is established. This is more relevant in scenarios such as the disbursement of funds or benefits where personal discretion in relation to selecting beneficiaries needs to be restricted.

- **Immutability**: Once the data has been written to the blockchain, it is extremely difficult to change it back. It is not genuinely immutable, but because changing data is so challenging and nearly impossible, this is seen as a benefit to maintaining an immutable ledger of transactions and is especially useful in audit and compliance scenarios.
- **High availability**: As the system is based on thousands of nodes in a peer-to-peer network, and the data is replicated and updated on every node, the system becomes highly available. Even if some nodes leave the network or become inaccessible, the network continues to work, thus making it highly available. This redundancy results in high availability.
- **Highly secure**: All transactions on a blockchain are cryptographically secured and thus provide network integrity. Any transactions posted from the nodes on the blockchain are verified based on a predetermined set of rules. Only valid transactions are selected for inclusion in a block. The blockchain is based on proven cryptographic technology that ensures the integrity and availability of data. Generally, confidentiality is not provided due to the requirements of transparency. This limitation is the leading barrier to its adoption by financial institutions and other industries that require the privacy and confidentiality of transactions. As such, the privacy and confidentiality of transactions on the blockchain are being researched very actively, and advancements are already being made. It could be argued that, in many situations, confidentiality is not needed, and transparency is preferred. For example, with Bitcoin, confidentiality is not an absolute requirement; however, it is desirable in some scenarios. A more recent example is Zcash (`https://z.cash`), which uses zero-knowledge proofs to provide a platform for conducting anonymous transactions. Other security services, such as non-repudiation and authentication, are also provided by blockchain, as all actions are secured using private keys and digital signatures.
- **Platform for smart contracts**: Smart contracts are automated, autonomous programs that reside on the blockchain network and encapsulate the business logic and code needed to execute a required function when certain conditions are met. This is indeed a revolutionary feature of blockchain, as it provides flexibility, speed, security, and automation for real-world scenarios that can lead to a completely trustworthy system with significant cost reductions. Smart contracts can be programmed to perform any application-level actions that blockchain users need and according to their specific business requirements.



Not all blockchains have a mechanism to execute *smart contracts*; however, this is a very desirable feature. However, note that some blockchains may not incorporate smart contract functionality on purpose, citing the reason that hardcoded executions are faster without the complexities of general-purpose smart contracts.

Limitations of blockchain technology

As with any technology, some challenges need to be addressed to make a system more robust, useful, and accessible. Blockchain technology is no exception. In fact, much effort is being made in both academia and industry to overcome the challenges posed by blockchain technology.

The most sensitive blockchain problems are as follows:

- **Scalability**: Currently, blockchain networks are not as scalable as, for example, current financial networks. This is a known area of concern and a very ripe area for research.
- **Regulation**: Due to its decentralized nature, regulation is almost impossible on blockchain. This is sometimes seen as a barrier toward adoption because, traditionally, due to the existence of regulatory authorities, consumers have a certain level of confidence that if something goes wrong they can hold someone accountable. However, in blockchain networks, no such regulatory authority and control exists, which is an inhibiting factor for many consumers.



Note that there are attempts to regulate these networks, including the legalization of cryptocurrency exchanges in the US under the Bank secrecy act, so that these exchanges adhere to requirements such as **anti-money laundering** (**AML**), **Counter Financing of Terrorism** (**CFT**), and enforce strict record keeping and reporting to the authorities. Also, a promising use case enabled by blockchain is **central bank digital currency** (**CBDC**), which is a centralized and regulated form of digital currency issued by a central bank of a country.

- **Privacy**: Privacy is a concern on public blockchains such as Bitcoin where everyone can see every single transaction. This transparency is not desirable in many industries, such as the financial, law, or medical sectors. This is also a known concern and a lot of valuable research with some very impressive solutions has already been developed. A promising technology called zero-knowledge proofs is being utilized on blockchains to provide privacy. Further research continues to make these technologies better and more and more practical and mainstream.
- **Relatively immature technology**: As compared to traditional IT systems that have benefited from decades of research and evolution, blockchain is a comparatively new technology and requires research to achieve maturity. Even though core aspects of blockchains with the latest and most novel chains, such as Solana, Polkadot, and Avalanche, are very much mature; however, some features need to be improved further for example user experience, developer experience, security, and interoperability. From another angle, applications such as DeFi and Metaverses also need further maturity in terms of regulation, governance, and security.
- **Interoperability**: This problem is twofold. First is the interoperability of blockchains with enterprise and legacy systems, and second is the interoperability between different blockchains (cross-chain interoperability). Both these aspects are important to consider because blockchains cannot exist in silos; they must be able to communicate with one another, as well as enterprise networks and legacy systems, to enable enterprises to fully benefit from the technology. This is an active area of research, with many types of solutions becoming available in recent years, such as bridges, hub blockchains, and multi-chain hetereogeneous chains.

- **Adoption:** Often, blockchain is seen as a nascent technology. Even though this perspective has changed rapidly in the last few years, there is still a long way to go before the mass adoption of this technology. Some aspects, such as scalability, security, regulation, and customer confidence, need to be addressed before further adoption. Customer confidence can decrease due to security issues leading to loss of funds. Such problems may inhibit some users from joining the network, who would otherwise be happy to join if this security problem didn't exist. However, note that despite these problems, DeFi is becoming more and more popular, but it might discourage cautious customers.

All these issues and possible solutions will be discussed in detail later in this book.

You now know the basics of blockchain and its benefits and limitations. Now, let's look at the various types of blockchain that exist.

Types of blockchain

Based on the way that blockchain has evolved, it can be divided into multiple categories with distinct, though sometimes partially overlapping, attributes. At a broad level, **Digital Ledger Technology** (**DLT**) is an umbrella term that represents distributed ledger technology, comprising blockchains and distributed ledgers of different types:

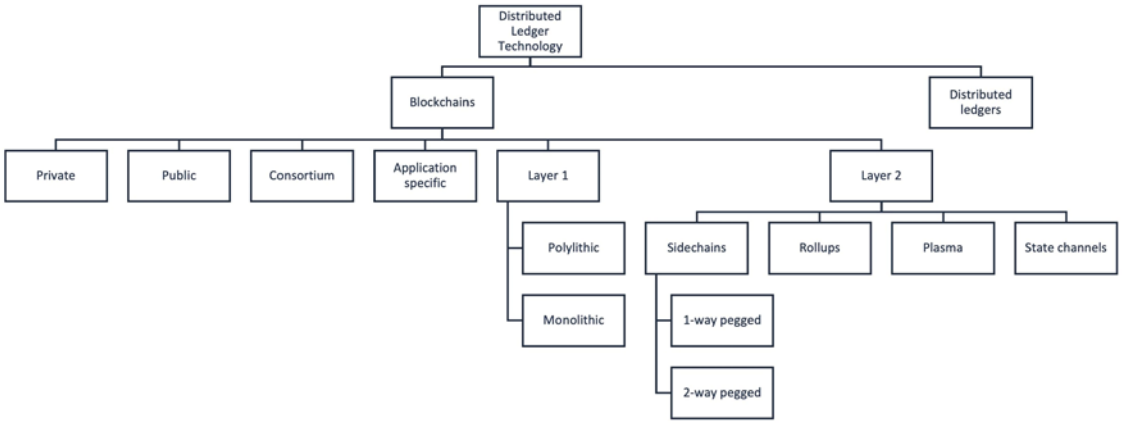


Figure 1.9: DLT types

In this section, we will examine the different types of blockchain from a technical and business perspective.

Distributed ledgers

A *distributed ledger* is a broad term describing shared databases; hence, all blockchains technically fall under the umbrella of shared databases or distributed ledgers. Although all blockchains are fundamentally distributed ledgers, not all distributed ledgers are necessarily blockchains.

A key difference between a distributed ledger and a blockchain is that a distributed ledger does not necessarily consist of blocks of transactions to keep the ledger growing. Rather, a blockchain is a special type of shared database that comprises blocks of transactions. An example of a distributed ledger that does not use blocks of transactions is R3's Corda (`https://www.corda.net`). Corda is a distributed ledger that is developed to record and manage agreements and is especially focused on the financial services industry. On the other hand, more widely known blockchains such as Bitcoin and Ethereum make use of blocks composed of transactions to update the replicated shared database. As the name suggests, a distributed ledger is distributed among its participants and is replicated across multiple nodes, sites, or organizations. This type of ledger can be either private or public.

Shared ledger

This is a generic term that is used to describe any application or database that is shared by the public or a consortium. Generally, all blockchains fall into the category of a shared ledger.

Public blockchains

As the name suggests, public blockchains are not owned by anyone. They are open to the public, and anyone can participate as a node. Users may or may not be rewarded for their participation. All users of these "permissionless" ledgers maintain a copy of the ledger on their local nodes and use a distributed consensus mechanism to decide the eventual state of the ledger. Bitcoin and Ethereum are both examples of public blockchains.

Private blockchains

As the name implies, private blockchains are just that—private. That is, they are open only to a consortium or group of individuals or organizations who have decided to share the ledger among themselves. There are various blockchains now available in this category, such as Hyperledger Fabric and Quorum. Optionally, both blockchains can also be in public mode if required, but their primary purpose is to provide a private blockchain. These blockchains are also called consortium blockchains, or enterprise blockchains.

Semi-private blockchains

With semi-private blockchains, part of the blockchain is private and part of it is public. Note that this is still just a concept today, and no real-world proofs of concept have yet been developed. With a semi-private blockchain, the private part is controlled by a group of individuals, while the public part is open for participation by anyone.

This hybrid model can be used in scenarios where the private part of the blockchain remains internal and shared among known participants, while the public part of the blockchain can still be used by anyone, optionally allowing mining to secure the blockchain. This way, the blockchain can be secured using PoW, thus providing consistency and validity for both the private and public segments. This type of blockchain can also be called a "semi-decentralized" model, where it is controlled by a single entity but still allows multiple users to join the network by following appropriate procedures.

Permissioned ledger

A *permissioned ledger* is a blockchain where participants of the network are already known and trusted. Permissioned ledgers do not need to use a distributed consensus mechanism; instead, an agreement protocol is used to maintain a shared version of the truth about the state of the records on the blockchain. In this case, for verification of transactions on the chain, all verifiers are already preselected by a central authority and, typically, there is no need for a mining mechanism.

There is no requirement for a permissioned blockchain to be private, as it can be a public blockchain but with regulated access control. For example, Bitcoin can become a permissioned ledger if an access control layer is introduced on top of it that verifies the identity of a user and then allows access to the blockchain.

Fully private and proprietary blockchains

There is no mainstream application of these types of blockchains, as they deviate from the core concept of decentralization in blockchain technology. Nonetheless, in specific private settings within an organization, there could be a need to share data and provide some level of guarantee of the authenticity of the data.

An example of this type of blockchain might be to allow collaboration and the sharing of data between various government departments. In that case, no complex consensus mechanism is required, apart from a simple SMR with known central validators. Even in private blockchains, tokens are not really required, but they can be used as a means of transferring value or representing some real-world assets.

Tokenized blockchains

These blockchains are standard blockchains that generate cryptocurrency as a result of a consensus process via mining or initial distribution. Bitcoin and Ethereum are prime examples of this type of blockchain.

Tokenless blockchains

These blockchains are designed in such a way that they do not have the basic unit for the transfer of value. However, they are still valuable in situations where there is no need to transfer value between nodes and only the sharing of data among various trusted parties is required. This is similar to fully private blockchains, the only difference being that the use of tokens is not required. This can also be thought of as a shared distributed ledger used for storing and sharing data between the participants. It does have its benefits when it comes to immutability, tamper-proofing, security, and consensus-driven updates, but is not used for a common blockchain application of value transfer or cryptocurrency. Most of the permissioned blockchains can be seen as an example of tokenless blockchains, for example, Hyperledger Fabric or Quorum. Tokens can be built on these chains as an application implemented using smart contracts, but intrinsically these blockchains do not have a token associated with them. In other words, we can say that there is no native (default) cryptocurrency in tokenless blockchains.

Layer 1 blockchains

Any base layer chain, responsible for consensus is a layer 1 blockchain. For example, Bitcoin and Ethereum. We can also think of two other types of blockchain architectures. One is **monolithic architecture**, which is just one base layer responsible for all operations, and another type is called **polylithic architecture**, which is composed of multiple chains.

Monolithic and polylithic blockchains

The original Bitcoin blockchain is a monolithic chain. Other examples include Ethereum and Solana. These chains are categorized as Layer 1 blockchains as they are base layer single-chain protocols where all functionalities including programmability (smart contracts), consensus protocol, security, and any related functionality are part of the same base blockchain. In other words, no component is off chain.

Polylithic chains include examples such as Polkadot, Avalanche, and Cosmos. In this type of architecture, multiple chains of the same or different types connect to a core chain and form a network of networks. Both types are considered layer 1 chains where a single base layer is the source of canonical truth. In polylithic architectures, there can be multiple chains, but they are horizontal to the core chain, and in some cases the core chain is not strictly needed and subnets can talk to each other directly. We can think of these architectures as multi-chain architectures. If chains connecting to the core chain are all the same type and built using the same rules, we call them homogeneous chains, and if they are of different types and follow different rules, we call them heterogeneous chains. Usually, multichain architectures aim to be heterogeneous architectures.

Layer 2 blockchains

Layer 2 blockchains have also recently emerged as a solution to the scalability and privacy problems on classical layer 1 blockchains, such as Ethereum and Bitcoin. Such solutions are called layer 2 solutions, which is a generic term used to describe solutions that use layer 1 as a base layer for consensus and settlement but execute transactions off chain at the so-called layer 2. These chains run on top of layer 1 chains. Many solutions exist in this space, such as sidechains, zero-knowledge rollups, optimistic rollups, plasma chains, and Lightning Network.

Sidechains

More precisely known as "pegged sidechains," this is a concept whereby coins can be moved from one blockchain to another and then back again. Typical uses include the creation of new *altcoins* (alternative cryptocurrencies) whereby coins are burnt as a proof of an adequate stake. "Burnt" or "burning the coins" in this context means that the coins are sent to an address that is un spendable, and this process makes the "burnt" coins irrecoverable. This mechanism is used to bootstrap a new currency or introduce scarcity, which results in the increased value of the coin.

This mechanism is also called "Proof of Burn" and is used as an alternative method for distributed consensus to PoW and **Proof of Stake** (**PoS**). The example provided previously for burning coins applies to a **one-way pegged sidechain**. The second type is called a **two-way pegged sidechain**, which allows the movement of coins from the main chain to the sidechain and back to the main chain when required.

This process enables the building of smart contracts for the Bitcoin network. Rootstock is one of the leading examples of a sidechain, which enables smart contract development for Bitcoin using this paradigm. It works by allowing a two-way peg for the Bitcoin blockchain, and this results in much faster throughput. Another example is Deku, which is a side chain layer 2 solution for Tezos.

Summary

This chapter introduced blockchain technology at a high level. First, we discussed blockchain's progress toward becoming a mature technology, followed by some basic concepts about distributed systems, and then the history of blockchain was reviewed. Concepts such as e-cash were also discussed.

Various definitions of blockchain from different points of view were presented. Some applications of blockchain technology were also introduced. Next, different types of blockchain were explored. Finally, the benefits and limitations of this new technology were also examined. Some topics, such as blockchain scalability and adoptability issues, were intentionally introduced only lightly, as they will be discussed in depth in later chapters.

In the next chapter, we will introduce the concept of decentralization, which is central to the idea behind blockchains and their vast number of applications.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

2

Decentralization

Decentralization is not a new concept. It has been used in strategy, management, and the government sector for a long time. The basic idea of decentralization is to distribute control and authority to the peripheries of an organization instead of it being concentrated in one central body. This structure produces several benefits for organizations, such as increased efficiency, expedited decision making, better motivation, and a reduced burden on upper management.

In this chapter, we will discuss the concept of decentralization in the context of blockchain, the fundamental basis of which is that no single central authority controls the network. This chapter will present examples of various methods of decentralization and ways to achieve it. Furthermore, we will discuss decentralized applications and platforms for achieving decentralization.

In this chapter, we will cover the following:

- Introducing decentralization
- Full ecosystem decentralization
- Decentralization in practice
- Innovative trends

Introducing decentralization

Decentralization is a core benefit of blockchain technology. By design, blockchain is a perfect vehicle for providing a platform that does not need any intermediaries and that can function with leaders chosen over time via consensus mechanisms. This model allows anyone to compete to become the decision-making authority. This competition is governed by a consensus mechanism, which we will discuss in *Chapter 5*, *Consensus Algorithms*.

Decentralization is applied in varying degrees from a semi-decentralized model to a fully decentralized one depending on the requirements and circumstances. Decentralization can be viewed from a blockchain perspective as a mechanism that provides a way to remodel existing applications and paradigms, or to build new applications, giving full control to users.

IT infrastructure has conventionally been based on a centralized paradigm whereby database or application servers are under the control of a central authority, such as a system administrator. With Bitcoin and the advent of blockchain technology, this model has changed, and now the technology exists to allow anyone to start a decentralized system and operate it with no single point of failure or single trusted authority. It can either be run autonomously or by requiring some human intervention, depending on the type and model of governance used in the decentralized application running on the blockchain.

The following diagram shows the different types of systems that currently exist: central, distributed, and decentralized. This concept was first published by Paul Baran in *On Distributed Communications: Introduction to Distributed Communications Networks* (Rand Corporation, 1964):

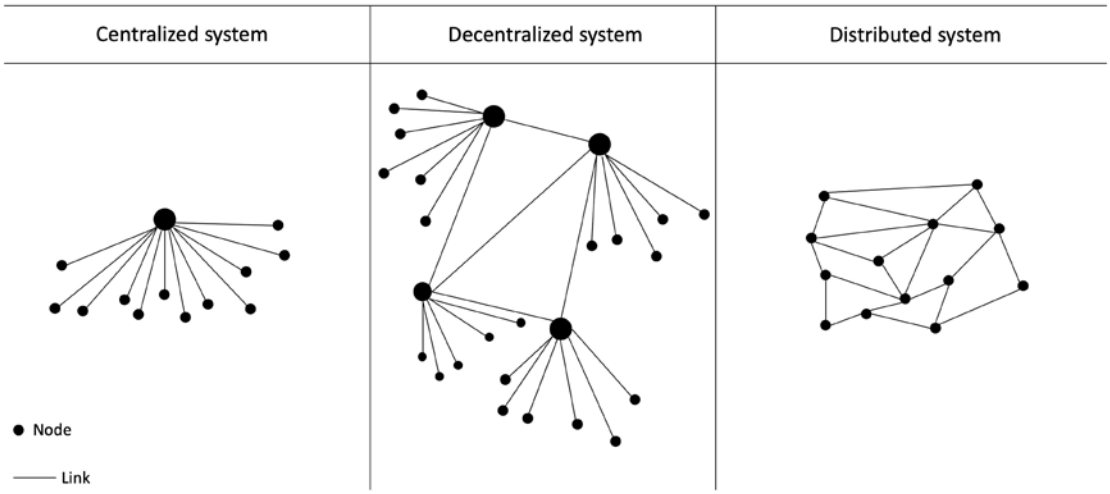


Figure 2.1: Different types of networks/systems

Centralized systems are conventional (client-server) IT systems in which there is a single authority that controls and is solely in charge of all operations on the system. All users of a centralized system are dependent on a single source of service. Most online service providers, including Google, Amazon, eBay, Yahoo!, and Apple's App Store, use this conventional model to deliver services.

In a **distributed system**, data and computation are replicated across multiple nodes in a network in what users view as a single, coherent system.



Sometimes, this term is confused with *parallel computing*. Variations of both models are used to achieve fault tolerance and speed. While there is some overlap in the definition, the main difference between these systems is that in a parallel computing system, computation is performed by all nodes simultaneously to achieve a single result; for example, parallel computing platforms are used in weather research and forecasting, simulation, and financial modeling. In the parallel system model, there is still a central authority that has control over all nodes and governs processing. This means that the system is still centralized in nature.

A **decentralized system** is a type of network where nodes are not dependent on a single master node; instead, control is distributed between many nodes. This is analogous to a model where each department in an organization is in charge of its own database server, thus taking away the power from the central server and distributing it to the sub-departments, who manage their own databases.

A significant innovation in the decentralized paradigm is **decentralized consensus**. This mechanism came into play with Bitcoin, and it enables a user to agree on something via a consensus algorithm without the need for a central, trusted third party, intermediary, or service provider.

We can also now view the different types of networks from a different perspective, where we highlight the controlling authority of these networks as a symbolic hand, as shown in the following diagram:

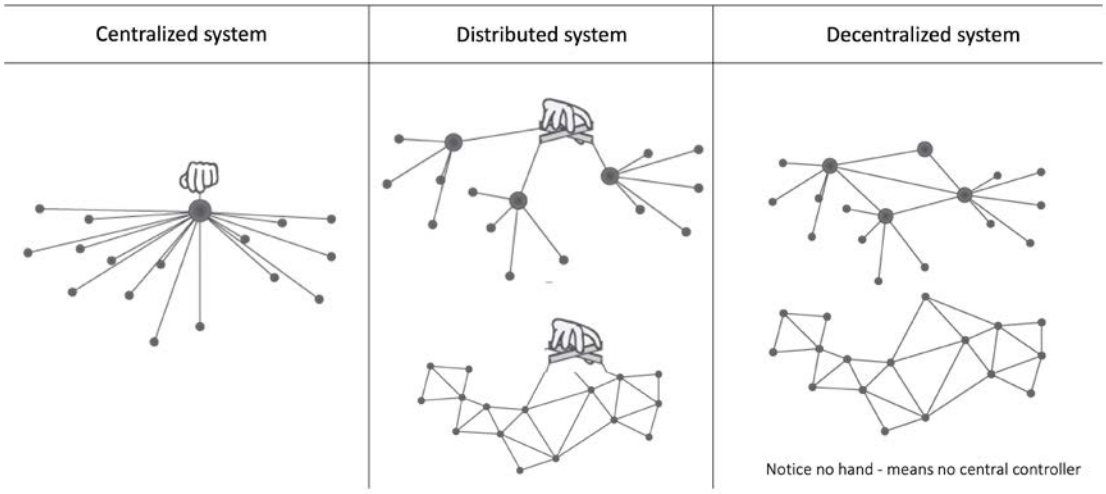


Figure 2.2: Different types of networks/systems depicting decentralization from a modern perspective

The preceding diagram shows the traditional centralized model with a central controller, which represents the usual client/server model. In the middle, we have distributed systems, where we still have a central controller, but the system comprises many dispersed nodes. On the right-hand side, notice that there is no hand/controller controlling the networks. This is the key difference between decentralized and distributed networks. A decentralized system may look like a distributed system from a topological point of view, but it doesn't have a central authority that controls the network.

The differences between distributed and decentralized systems can also be viewed at a practical level in the following diagram:

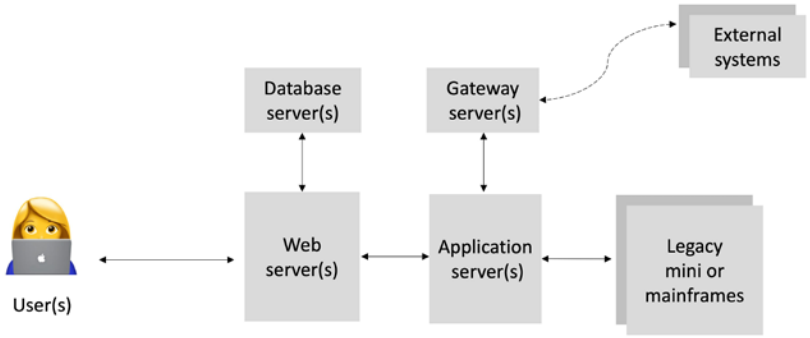


Figure 2.3: A traditional distributed system comprises many servers performing different roles

The following diagram shows a decentralized system (based on blockchain) where an exact replica of the applications and data is maintained across the entire network on each participating node:

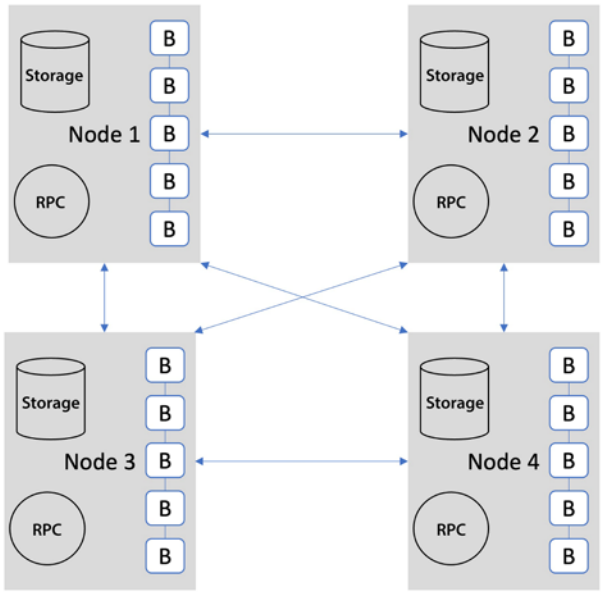


Figure 2.4: A blockchain-based decentralized system (notice the direct P2P connections and the exact replicas of blocks (data))

A comparison between centralized and decentralized systems is shown in the following table:

Feature	Centralized	Decentralized
Ownership	Service provider	All users
Architecture	Client/server	Distributed, different topologies
Security	Basic	More secure
High availability	No	Yes
Fault tolerance	Limited, single point of failure	Highly tolerant, as service is replicated
Collusion resistance	Basic, because it's under the control of a group or even a single individual	Highly resistant, as consensus algorithms ensure defense against adversaries
Application architecture	Single application	Application replicated across all nodes on the network
Trust	Consumers must trust the service provider, i.e., a trusted third party	No mutual trust required
Cost for consumer	High	Low

The comparison in the table only covers some main features and is not an exhaustive list, but this list should provide a good level of comparison.

Note that fault tolerance (the ability of a system to continue operating even if some of its components fail) in centralized systems is also improved by data replication. However, in the case of a decentralized system, fault tolerance is higher because, first, it's a distributed system, and secondly, it's decentralized, so no single participant could single-handedly game the system and gain a disproportionate advantage. If the architecture is a basic client/server with only one central server or perhaps just a primary and the backup server providing services, it will be considerably less fault-tolerant than a decentralized distributed blockchain system, as blockchains are replicated across usually hundreds and thousands of replicas (participants) worldwide in different geographic locations.

Now we will discuss what methods can be used to achieve decentralization.

Methods of decentralization

Two methods can be used to achieve decentralization: disintermediation and competition. These methods will be discussed in detail in the sections that follow.

Disintermediation

The concept of **disintermediation** can be explained with an example. Imagine that you want to send money to a friend in another country. You go to a bank, which, for a fee, will transfer your money to the bank in that country. In this case, the bank maintains a central database that is updated, confirming that you have sent the money.

With blockchain technology, it is possible to send this money directly without the need for a bank. All you need is an address on the blockchain. This way, the intermediary (that is, the bank) is no longer required, and decentralization is achieved by disintermediation. It is debatable, however, how practical decentralization through disintermediation is in the financial sector due to the massive regulatory and compliance requirements.



Central banks and monetary authorities have recognized that they can use blockchain to issue a regulated digital currency called **central bank digital currency** (**CBDC**), which can simplify the implementation and execution of monetary and fiscal policies. While this is a significant insight and could result in a safer, more efficient, and more financially inclusive financial ecosystem, it is expected to be centralized with a central bank or a nation's monetary authority regulating and issuing the currency.

Nevertheless, this model can be used not only in finance but in many other industries as well, such as health, law, and the public sector. In the health industry, instead of relying on a trusted third party (such as a hospital record system), patients can be in full control of their own identity and their data that they can share directly with only those entities that they trust. As a general solution, blockchain can serve as a decentralized health record management system where health records can be exchanged securely and directly between different entities (hospitals, pharmaceutical companies, patients) globally without any central authority. While interoperability between different standards for recording and categorizing health-related data is not easy, blockchain can at least provide a platform to share data between different health providers.

Contest-driven decentralization

In a method involving **competition**, different service providers compete to be selected for the provision of services by the system. This paradigm does not achieve complete decentralization. However, to a certain degree, it ensures that an intermediary or service provider is not monopolizing the service. In the context of blockchain technology, a system can be envisioned in which smart contracts can choose an external data provider from many providers based on their reputation, previous score, reviews, and quality of service. This method will not result in full decentralization, but it allows smart contracts to make a free choice based on various criteria. This way, an environment of competition is cultivated among service providers where they compete to become the service provider of choice.

Quantifying decentralization

In the following diagram, varying levels of decentralization are shown. On the left side, the conventional approach is shown where a central system is in control; on the right side, complete disintermediation is achieved, as intermediaries are entirely removed. Competing intermediaries or service providers are shown in the center.

At that level, intermediaries or service providers are selected based on reputation or voting, thus achieving partial decentralization:



Figure 2.5: Scale of decentralization

We can think about the decentralization spectrum from another angle, where the attainable decentralization ranges from **minimum achievable decentralization** (**MAD**) to **maximum feasible decentralization** (**MFD**). Moreover, we can focus on finding an **optimal decentralization point** (**ODP**) on the spectrum for a given use case, that is, where the maximum decentralization is achieved along with as little centralization as possible, which is most favorable for the specific use case under consideration.

A question arises here regarding how we can measure the level of decentralization. An answer to this question is the "Nakamoto coefficient." This metric is calculated using several factors. It represents the number of entities that are required to be controlled to compromise a blockchain network. The higher the value of the Nakamoto coefficient, the more decentralized the network is.



You can track the Nakamoto coefficient here: `https://nakaflow.io/`.

Any decentralized system is composed of several decentralized subsystems. If any subsystem is centralized, the overall system is considered centralized. For example, a blockchain can be composed of several subsystems, including miners, clients, developers, exchanges, nodes, and ownership. We can say that due to mining pools' monopoly on Bitcoin mining, Bitcoin can be considered centralized. The key insight behind this metric is to first enumerate the subsystems of a decentralized system, then figure out how many entities are required to be compromised to gain control of each subsystem, and then use the minimum of these values to get the overall effective decentralization of the system. For example, Ethereum could be considered centralized because only a handful of developers do most of the commits, hence resulting in developer centralization.



The Nakamoto coefficient was introduced by Balaji S. Srinivasan and Leland Lee in their article here: `https://news.earn.com/quantifying-decentralization-e39db233c28e`.

Benefits of decentralization

There are many benefits of decentralization, including transparency, efficiency, cost savings, the development of trusted ecosystems, and in some cases privacy and anonymity. Some challenges, such as security requirements, software bugs, and human error, need to be examined thoroughly.

For example, in a decentralized system such as Bitcoin or Ethereum where security is normally provided by private keys, how can we ensure that an asset or a token associated with these private keys cannot be rendered useless due to negligence or bugs in the code? What if the private keys are lost due to user negligence? What if due to a bug in the smart contract code the decentralized application becomes vulnerable to attack?

Before embarking on a journey to decentralize everything using blockchain and decentralized applications, it is essential that we understand that not everything can or needs to be decentralized.

This view raises some fundamental questions. Is a blockchain really needed? When is a blockchain required? In what circumstances is blockchain preferable to traditional databases? To answer these questions, go through the simple set of questions presented below:

Question	Yes/No	Recommended solution
Is high data throughput required?	Yes	Use a traditional database.
	No	A central database might still be useful if other requirements are met. For example, if users trust each other, then perhaps there is no need for a blockchain. However, if they don't or trust cannot be established for any reason, blockchain can be helpful.
Are updates centrally controlled?	Yes	Use a traditional database.
	No	You may investigate how a public/private blockchain can help.
Do users trust each other?	Yes	Use a traditional database.
	No	Use a public blockchain.
Are users anonymous?	Yes	Use a public blockchain.
	No	Use a private blockchain.
Is consensus required to be maintained within a consortium?	Yes	Use a private blockchain.
	No	Use a public blockchain.
Is strict data immutability required?	Yes	Use a blockchain.
	No	Use a central/traditional database.

Answering all these questions can help you decide whether a blockchain is required or suitable for solving a problem. Beyond the questions posed in this model, there are many other issues to consider, such as latency, the choice of consensus mechanisms, whether consensus is required or not, and where consensus is going to be achieved.

If consensus is maintained internally by a consortium, then a private blockchain should be used; otherwise, if consensus is required publicly among multiple entities, then a public blockchain solution should be considered. Other aspects, such as immutability, should also be considered when deciding whether to use a blockchain or a traditional database. If strict data immutability is required, then a public blockchain should be used; otherwise, a central database may be an option.

As blockchain technology matures, there will be more questions raised regarding this selection model. For now, however, this set of questions is sufficient for deciding whether a blockchain-based solution is suitable or not.

Now we understand different methods of decentralization and have looked at how to decide whether a blockchain is required or not in a particular scenario. Let's now look at the process of decentralization, that is, how we can take an existing system and decentralize it.

Evaluating requirements

There are systems that pre-date blockchain and Bitcoin, including BitTorrent and the Gnutella file-sharing system, which to a certain degree could be classified as decentralized. However, due to a lack of any incentivization mechanism, participation from the community gradually decreased. With the advent of blockchain technology, many initiatives are being taken to leverage this new technology to achieve decentralization.

The Bitcoin blockchain has been typically the first choice for many, as it has proven to be the most resilient and secure blockchain. However, Ethereum has become a more prominent choice because of the flexibility it allows for programming any business logic into the blockchain by creating **smart contracts**. Moreover, newer chains like Polkadot, Solana, and Cardano are also used as platforms for decentralization by many developers.

The Nakamoto coefficient varies from chain to chain and should be a deciding factor during the evaluation of blockchain platforms for a use case. While it is best to be as decentralized as possible, in some cases depending on the use case, some decentralization can be given up. Bitcoin has the highest Nakamoto coefficient whereas some chains have a very low Nakamoto coefficient. The choice of blockchain platform is governed by the use case and user requirements and in some cases, even giving up some level of decentralization is acceptable.

Arvind Narayanan et al. have proposed a framework in their book *Bitcoin and Cryptocurrency Technologies* that can be used to evaluate the decentralization requirements of a variety of issues in the context of blockchain technology. The framework raises four questions whose answers provide a clear understanding of how a system can be decentralized:

- *What is being decentralized?:* This can be any system, such as an identity system or a trading system.
- *What level of decentralization is required?:* This can be full disintermediation or partial disintermediation.
- *What blockchain is used?:* It can be the Bitcoin blockchain, Ethereum blockchain, or any other blockchain that is deemed fit for the specific application.

- *What security mechanism is used?:* For example, the security mechanism can be atomicity-based, where either the transaction executes in full or does not execute at all. This deterministic approach ensures the integrity of the system. Other mechanisms include those based on reputation, which allows for varying degrees of trust in a system.

Next, let's evaluate a money transfer system as an example of an application selected to be decentralized. The four questions discussed previously are used to evaluate the decentralization requirements of this application. The answers to these questions are as follows:

- *What is being decentralized?:* A money transfer system.
- *What level of decentralization is required?:* Disintermediation.
- *What blockchain is used?:* Bitcoin.
- *What security mechanism is used?:* Atomicity.

The responses indicate that the money transfer system can be decentralized by removing the intermediary, implemented on the Bitcoin blockchain, and that a security guarantee can be provided via atomicity. Atomicity will ensure that transactions execute successfully in full or do not execute at all. We have chosen the Bitcoin blockchain because it is the longest-established blockchain and has stood the test of time.

Similarly, this framework can be used for any other system that needs to be evaluated in terms of decentralization. The answers to these four simple questions help clarify what approach to take to decentralize the system.

To achieve complete decentralization, it is necessary that the environment around the blockchain also be decentralized. We'll look at the full ecosystem of decentralization next.

Full-ecosystem decentralization

The blockchain is a distributed ledger that runs on top of conventional systems. These elements include storage, communication, and computation.

There are other factors, such as identity and wealth, which are traditionally based on centralized paradigms, and there's a need to decentralize these aspects as well to achieve a sufficiently decentralized ecosystem.

Storage

Data can be stored directly on a blockchain, and with this fact it achieves decentralization. However, a significant disadvantage of this approach is that a blockchain is not suitable for storing large amounts of data by design. It can store simple transactions and some arbitrary data, but it is certainly not suitable for storing images or large blobs of data, as is the case with traditional database systems.

A better alternative for storing data is to use **distributed hash tables** (**DHTs**). DHTs were used initially in peer-to-peer file-sharing software, such as BitTorrent, Napster, Kazaa, and Gnutella. DHT research was made popular by the CAN, Chord, Pastry, and Tapestry projects. BitTorrent is the most scalable and fastest network, but the issue with BitTorrent and the others is that there is no incentive for users to keep the files indefinitely.

Users generally don't keep files permanently, and if nodes that have data still required by someone leave the network, there is no way to retrieve that data except by having the required nodes rejoin the network so that the files once again become available.

Two primary requirements here are high availability, which means that data should be available when required, and link stability, meaning that network links also should always be accessible. **Inter-Planetary File System** (**IPFS**) possesses both properties, and its vision is to provide a decentralized World Wide Web by replacing the HTTP protocol. IPFS uses the Kademlia DHT and Merkle **Directed Acyclic Graphs** (**DAGs**) to provide its storage and searching functionality, respectively. The concept of DHTs and DAGs will be introduced in detail in *Chapter 4*, *Asymmetric Cryptography* and *Chapter 17*, *Scalability*, respectively.

The incentive mechanism for storing data is based on a protocol known as Filecoin, which pays incentives to nodes that store data using the Bitswap mechanism. The Bitswap mechanism lets nodes keep a simple ledger of bytes sent or bytes received in a one-to-one relationship. Also, a Git-based version control mechanism is used in IPFS to provide structure and control over the versioning of data.

There are other alternatives for data storage, such as Ethereum Swarm, Storj, and MaidSafe. Ethereum has its own decentralized and distributed ecosystem that uses Swarm for storage and the Whisper protocol for communication. MaidSafe aims to provide a decentralized World Wide Web. All these projects are discussed later in this book in greater detail.

BigChainDB is another storage layer decentralization project aimed at providing a scalable, fast, and linearly scalable decentralized database as opposed to a traditional filesystem. BigChainDB complements decentralized processing platforms and filesystems such as Ethereum and IPFS.

Communication

The Internet (the communication layer in blockchain) appears to be decentralized. This belief is correct to some extent, as the original vision of the Internet was to develop a decentralized communications system. Services such as email and online storage are now all based on a paradigm where the service provider is in control, and users trust such providers to grant them access to the service as requested. This model is based on the unconditional trust of a central authority (the service provider) where users are not in control of their data. Even user passwords are stored on trusted third-party systems.

Thus, there is a need to provide control to individual users in such a way that access to their data is guaranteed and is not dependent on a single third party. Access to the Internet is based on **Internet Service Providers** (**ISPs**) who act as a central hub for Internet users. If the ISP is shut down for any reason, then no communication is possible with this model.

An alternative is to use mesh networks. Mesh networks use wireless technologies such as **Bluetooth Low Energy** (**BLE**) to form communication networks that do not need Internet connectivity. Even though they are limited in functionality compared to the Internet, they still provide a decentralized alternative where nodes in relative proximity can talk directly to each other without requiring the Internet or a central hub such as an ISP. Now imagine a network that allows users to be in control of their communication; no one can shut it down for any reason. This type of network is very advantageous in situations like natural disasters or war zones. Another use could be to arrange protests against oppressive regimes that might have blocked the Internet. One example of such an offline messaging app is **bridgefy**. This could be the next step toward decentralizing communication networks in the blockchain ecosystem.

As mentioned earlier, the original vision of the Internet was to build a decentralized network; however, over the years, with the arrival of large-scale service providers such as Google, Amazon, and eBay, control has shifted toward these big players. For example, email is a decentralized system at its core; that is, anyone can run an email server with minimal effort and can start sending and receiving emails. However, there are better alternatives available. For example, Gmail and Outlook already provide managed services for end users, so there is a natural inclination toward selecting one of these large, centralized services as they are more convenient, secure, and above all free to use. This is one example that shows how the Internet has moved toward centralization.

Free services, however, are offered at the cost of exposing valuable personal data, and many users are unaware of this fact. Blockchain has revived the vision of decentralization across the world, and now concerted efforts are being made to harness this technology and take advantage of the benefits that it can provide.

Computing power

The decentralization of computing or processing power is achieved by a blockchain technology such as Ethereum, where smart contracts with embedded business logic can run on the blockchain network. Other blockchain technologies also provide similar processing-layer platforms, where business logic can run over the network in a decentralized manner.

The following diagram shows an overview of a decentralized ecosystem:

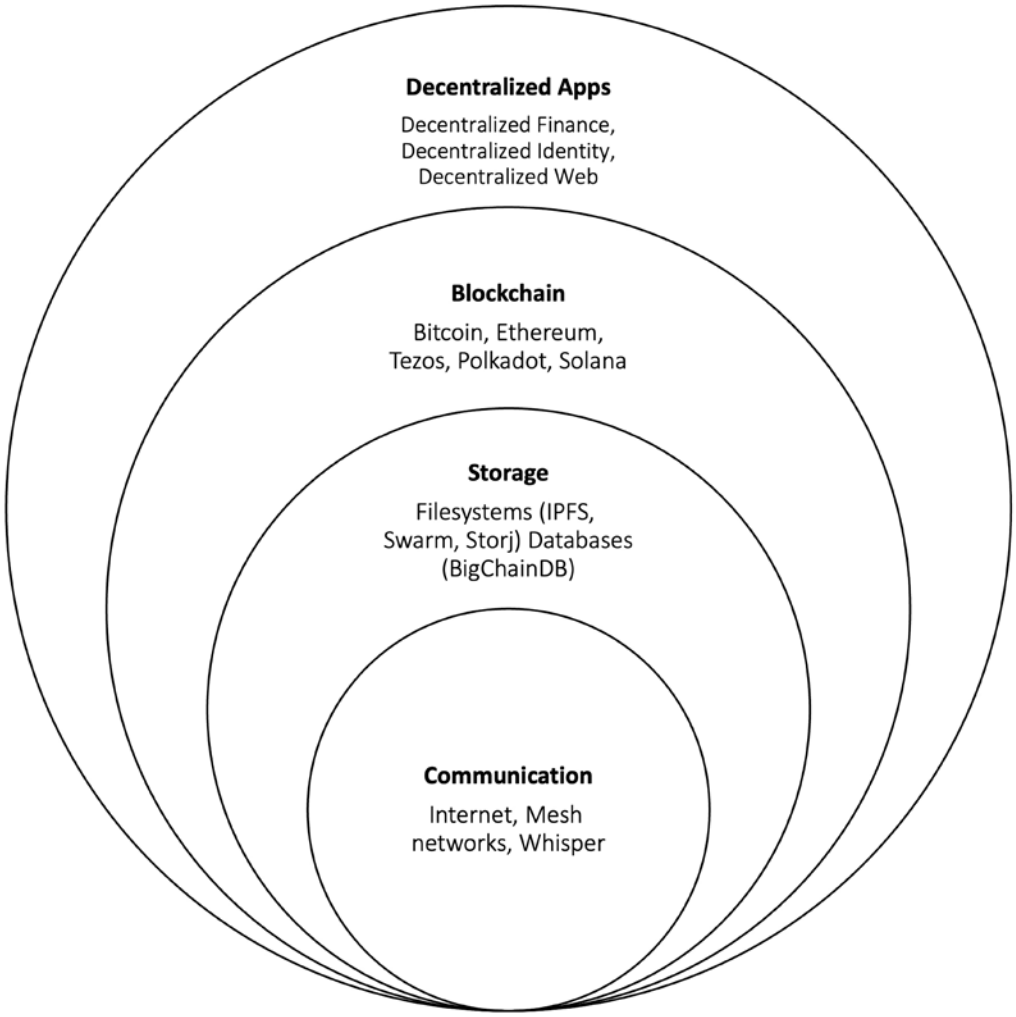


Figure 2.6: Decentralized ecosystem

In the bottom layer, the Internet or mesh networks provide a decentralized communication layer. In the next layer up, a storage layer uses technologies such as IPFS and BigChainDB to enable decentralization. Finally, in the next level up, you can see that the blockchain serves as a decentralized processing (computation) layer. Blockchain can, in a limited way, provide a storage layer too, but that severely hampers the speed and capacity of the system. Therefore, other solutions such as IPFS and BigChainDB are more suitable for storing large amounts of data in a decentralized way. The identity, finance, and web layers are shown at the top level.

The blockchain can provide solutions to various issues relating to decentralization. A concept relevant to identity known as **Zooko's triangle** requires that the naming system in a network protocol is secure, decentralized, and able to provide human-meaningful and memorable names to the users. Conjecture has it that a system can have only two of these properties simultaneously.

Nevertheless, this problem was resolved with the advent of **Namecoin**, which makes it possible to achieve security, decentralization, and human-meaningful names. However, this is not a panacea, and it comes with many challenges, such as reliance on users to store and maintain private keys securely. This raises other general questions about the suitability of decentralization to a particular problem.

Decentralization may not be appropriate for every scenario. Centralized systems with well-established reputations tend to work better in many cases. For example, email platforms from reputable companies such as Google or Microsoft would provide a better service than a scenario where individual email servers were hosted privately by users on the Internet.

There are many projects underway that are developing solutions for a more comprehensive distributed blockchain system. For example, Swarm and Whisper are being developed to provide decentralized storage and communication for Ethereum.

With the advent of blockchain technology, it is now possible to build software versions of traditional physical organizations in the form of **Decentralized Organizations** (**DOs**) and other similar constructs, which we will examine in detail shortly.

Moreover, with the emergence of the decentralization paradigm, different terminology and buzzwords are now appearing in the media and academic literature, which we will explore in the next section.

Decentralization in practice

The following concepts are worth citing in the context of decentralization. The terminology introduced here is often used in the literature concerning decentralization and its applications.

Smart contracts

A **smart contract** is a software program that usually runs on a blockchain. Smart contracts do not necessarily need a blockchain to run; however, due to the security benefits that blockchain technology provides, blockchain has become a standard decentralized execution platform for smart contracts.

A smart contract usually contains some business logic and a limited amount of data. The business logic is executed if specific criteria are met. Actors or participants in the blockchain use these smart contracts, or they run autonomously on behalf of the network participants.

More information on smart contracts will be provided in *Chapter 8, Smart Contracts*.

Autonomous agents

An **Autonomous Agent** (**AA**) is a software entity (artificially intelligent or traditionally programmed) that acts on the behalf of its owner to achieve some desirable goals without requiring any or minimal intervention from its owner.

Decentralized organizations

DOs are software programs that run on a blockchain and are based on the model of real-life organizations with people and protocols. Once a DO is added to the blockchain in the form of a smart contract or a set of smart contracts, it becomes decentralized, and parties interact with each other based on the code defined within the DO software.

Decentralized autonomous organizations

Just like DOs, a **decentralized autonomous organization** (**DAO**) is also a computer program that runs on top of a blockchain, and embedded within it are governance and business logic rules. DAOs and DOs are fundamentally the same thing. The main difference, however, is that DAOs are autonomous, which means that they are fully automated and contain artificially intelligent logic. DOs, on the other hand, lack this feature and rely on human input to execute business logic.

The Ethereum blockchain led the way with the introduction of DAOs. In a DAO, the code is considered the governing entity rather than people or paper contracts. However, a human curator maintains this code and acts as a proposal evaluator for the community. DAOs can hire external contractors if enough input is received from the token holders (participants).

The most famous DAO project is **the DAO**, which raised $168 million in its crowdfunding phase. The DAO project was designed to be a venture capital fund aimed at providing a decentralized business model with no single entity as owner. Unfortunately, this project was hacked due to a bug in the DAO code, and millions of dollars' worth of **ether** currency (**ETH**) was siphoned out of the project and into a child DAO created by hackers. A major network change (hard fork) was required on the Ethereum blockchain to reverse the impact of the hack and initiate the recovery of the funds. This incident opened the debate on the security, quality, and need for thorough testing of the code in smart contracts to ensure their integrity and adequate control. There are other projects underway, especially in academia, that seek to formalize smart contract coding and testing.

Currently, DAOs do not have any legal status, even though they may contain some intelligent code that enforces certain protocols and conditions. However, these rules have no value in the real-world legal system at present. One day, perhaps an AA (that is, a piece of code that runs without human intervention) commissioned by a law enforcement agency or regulator will contain rules and regulations that could be embedded in a DAO for the purpose of ensuring its integrity from a legalistic and compliance perspective. The fact that DAOs are purely decentralized entities enables them to run in any jurisdiction. Thus, they raise a big question as to how the current legal system could be applied to such a varied mix of jurisdictions and geographies.

Decentralized autonomous corporations

A **decentralized autonomous corporation** (**DAC**) is like a DAO in concept, though considered to be a subset of them. The definitions of DACs and DAOs may sometimes overlap, but the general distinction is that DAOs are usually considered to be nonprofit, whereas DACs can earn a profit via shares offered to the participants and to whom they can pay dividends. DACs can run a business automatically without human intervention based on the logic programmed into them.

Decentralized autonomous societies

A **decentralized autonomous society** (**DAS**) is an entire society that can function on a blockchain with the help of multiple, complex smart contracts and a combination of DAOs and **decentralized applications** (**DApps**) running autonomously. This model does not necessarily translate to a free-for-all approach, nor is it based on an entirely libertarian ideology; instead, many services that a government commonly offers can be delivered via blockchains, such as government identity card systems, passports, and records of deeds, marriages, and births. Another theory is that, if a government is corrupt and central systems do not provide the levels of trust that a society needs, then that society can start its own virtual one on a blockchain that is driven by decentralized consensus and transparency. This concept might look like a libertarian's or cypherpunk's dream, but it is entirely possible on a blockchain.

This concept also goes into the realm of algocracy, an alternative form of governance and social system where computer algorithms maintain, control, and automate public services such as law, legal system, regulation, governance, economics, policies, and public decision-making. Blockchain and DApps are well-suited means to enable algocracy, especially when combined with AI. Initially, AI (or even traditionally programmed software) was seen to allow algorithmic governance; now, blockchain combined with AI can offer a more elegant approach.

However, there are both opportunities and threats associated with Algocracy. Increasing reliance on governance by algorithms is seen as a threat to active human participation and real-life decision-making. This is true in traditional algocracy without blockchain. However, when combined with blockchain, the situation improves. Due to the blockchain's decentralized and community-governed model, governance algorithms are also subject to approval and scrutiny by the community (society) operating on the blockchain. Therefore, blockchain can be seen as a solution to this threat of losing control of the decision-making process. We could call this variation of algocracy "blockcracy", i.e., government by blockchain, after the vision of a blockchain running artificially intelligent smart contracts (algorithms) responsible for governance.

Decentralized applications

All the ideas mentioned up to this point come under the broader umbrella of decentralized applications, abbreviated to DApps (pronounced Dee-App, or now more commonly rhyming with app). DAOs, DACs, and DOs are DApps that run on top of a blockchain in a peer-to-peer network. They represent the latest advancement in decentralization technology.

DApps at a fundamental level are software programs that execute using either of the following methods. They are categorized as Type 1, Type 2, or Type 3 DApps:

- **Type 1**: These run on their own dedicated blockchain, for example, standard smart contract-based DApps running on Ethereum. If required, they make use of a native token, for example, ETH on the Ethereum blockchain. For example, **Ethlance** is a DApp that makes use of ETH to provide a job market.



More information about Ethlance can be found at `https://ethlance.com`.

- **Type 2**: These use an existing established blockchain. That is, they make use of Type 1 blockchain and bear custom protocols and tokens, for example, smart contract-based tokenization DApps running on the Ethereum blockchain. An example is **DAI**, which is built on top of the Ethereum blockchain, but contains its own stablecoins and mechanism of distribution and control. Another example is **Golem**, which has its own token GNT and a transaction framework built on top of the Ethereum blockchain to provide a **decentralized marketplace** for computing power where users share their computing power with each other in a peer-to-peer network. An example of Type 2 DApps is the OMNI network, which is a software layer built on top of Bitcoin to support the trading of custom digital assets and digital currencies.



More information on the OMNI network can be found at `https://www.omnilayer.org`. More information on the Golem network is available at `https://golem.network`. More information on DAI is available at `https://makerdao.com/en/`.

- **Type 3**: Use the protocols of Type 2 DApps; for example, the SAFE Network uses the OMNI network protocol.



More information on the SAFE Network can be found at `https://safenetwork.tech`.

Another example to understand the difference between different types of DApps is the USDT token (Tether). The original USDT uses the OMNI layer (a Type 2 DApp) on top of the Bitcoin network. USDT is also available on Ethereum using ERC-20 tokens. This example shows that USDT can be considered a Type 3 DApp, where the OMNI layer protocol (a Type 2 DApp) is used, which is itself built on Bitcoin (a Type 1 DApp). Also, from the perspective of Ethereum, USDT can also be considered a Type 3 DApp in that it makes use of the Type 1 DApp Ethereum blockchain using the ERC-20 standard, which was built to operate on Ethereum.



More information can be found about Tether at `https://tether.to`.

In the last few years, the expression DApp has been increasingly used to refer to any end-to-end decentralized blockchain application, including a user interface (usually a web interface), smart contract(s), and the host blockchain. The clear distinction between different types of DApps is now not commonly referred to, but it does exist. Often, no reference to their type is made and they are all called just DApps.

There are thousands of different DApps running on various platforms (blockchains) now. There are various categories of these DApps covering media, social, finance, games, insurance, and health. There are various decentralized platforms (or blockchains), such as Ethereum, Solana, Avalanche, Polkadot, and EOS. Some DApps stats are available here: `https://thedapplist.com`.

Criteria for a DApp

For an application to be considered decentralized, it should meet the following criteria:

- **Decentralized**: The DApp should be fully decentralized. In other words, no single entity should be in control of its operations. All changes to the application must be consensus-driven based on the view given by the community.
- **Opensource**: It must be open source for public scrutiny and transparency.
- **Cryptographically secure**: State transition and the data of the application must be cryptographically secured and stored on a blockchain to avoid any central points of failure. Note that data does not necessarily need to be encrypted to provide confidentiality but should be protected against unauthorized manipulation. Data integrity, authentication, and non-repudiation services should be provided.
- **Incentive availability**: A cryptographic token must be used by the application to provide access and incentives for those who contribute value to the applications, for example, miners in Bitcoin. This requirement can be relaxed in a consortium chain where a token can still be used for value transfer, but not as a cryptocurrency.
- **Proof of value**: The tokens (if applicable) must be generated by the decentralized application using consensus and an applicable cryptographic algorithm. This generation of tokens acts as a proof of the value to contributors (for example, miners).

Generally, DApps now provide all sorts of different services, including but not limited to financial applications, gaming, social media, and supply chain management.

Operations of a DApp

Establishment of consensus by a DApp can be achieved using consensus algorithms such as **Proof of Work** (**PoW**) and **Proof of Stake** (**PoS**). So far, only PoW has been found to be incredibly resistant to attacks, as is evident from the trust people have put in the Bitcoin network, along with its success. Furthermore, a DApp can distribute tokens (coins) via **mining**, **fundraising**, and **development**.

Design of a DApp

A DApp is a software application that runs on a decentralized network such as a distributed ledger. They have recently become very popular due to the development of various decentralized platforms such as Ethereum, Solana, EOS, and Tezos.

Traditional apps commonly consist of a user interface and usually a web server or an application server and a backend database. This is a common client/server architecture. This is visualized in the following diagram:

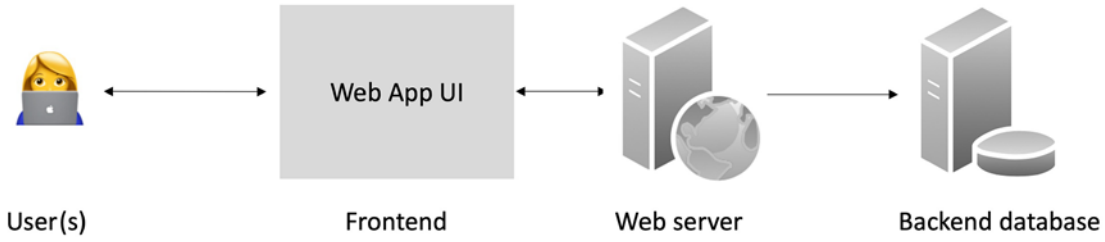


Figure 2.7: Traditional application architecture (generic client/server)

A DApp on the other hand has a blockchain as a backend and can be visualized as depicted in the following diagram. The key element that plays a vital role in the creation of a DApp is a smart contract that runs on the blockchain and has business logic embedded within it:

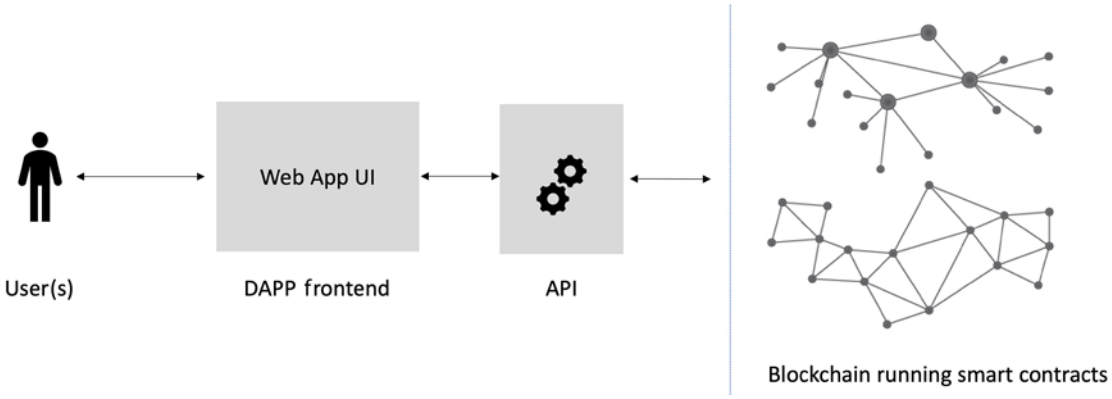


Figure 2.8: Generic DApp architecture

Note that the frontend in a DApp can either be a thick client, a mobile app, or a web frontend (a web user interface). However, it is usually a web frontend commonly written using a JavaScript framework such as React or Angular.

The following comparison table highlights the key properties of and differences between these different types of decentralized entities:

Entity	**Autonomous?**	**Software?**	**Owned?**	**Capital?**	**Legal status?**	**Cost**
DO	No	No	Yes	Yes	Yes	High
DAO	Yes	Yes	No	Yes	Some work has begun	Low
DAC	Yes	Yes	Yes	Yes	Unsettled	Low

DAS	Yes	Yes	No	Possible	Unsettled	Low
DApp	Yes	Yes	Yes	Optional tokens	Unsettled	Use case dependent

It is expected that all these entities will be regulated and will have some legal standing in the future while remaining decentralized.



For example, see the following article regarding DAO legality: `https://fedsoc.org/commentary/fedsoc-blog/the-legal-status-of-decentralized-autonomous-organizations-do-daos-require-new-business-structures-some-states-think-so`.

Any blockchain network, such as Bitcoin, Ethereum, Solana, Hyperledger Fabric, or Quorum, can serve as a decentralization platform on which DApps can be built and hosted.

Due to fast-paced innovation and the evolution of blockchain, many innovative trends have emerged, which we explore in the next section.

Innovative trends

With the growth of blockchain, several ideas have emerged that make use of the decentralization aspect of blockchain to provide more user-centric and fully decentralized services. Some of the key ideas in this space are the decentralized web, decentralized identity, and decentralized finance.

Decentralized web

Decentralized web is a term that's used to describe a vision of the web where no central authority or set of authorities will be in control. The original intention of the Internet was indeed decentralized, and the development of open protocols such HTTP, SMTP, and DNS meant that any individual could use these protocols freely, and immediately become part of the Internet. This is still true; however, with the emergence of a layer above these protocols called the **Web layer** a more service-oriented infrastructure was introduced, which inevitably led to large profit-seeking companies taking over. This is evident from the rise of Facebook, Google, Twitter, and Amazon, which of course provide excellent user services but at the cost of a more controlled, centralized, and closed system.

Once intended and developed as decentralized, open and free protocols are now being dominated by powerful commercial entities around the world, which has resulted in major concerns around privacy and data protection. These types of business models do work well and are quite popular due to the high level of standardization and services provided, but they pose a threat to privacy and decentralization due to the dominance of only a handful of entities on the entire Internet.

With blockchain, it is envisioned that this situation will change as it will allow development of the decentralized Internet, or the decentralized web, or Web 3 for short, which was the original intention of the Internet.

We can review the evolution of the Web over the last few decades by dividing the major developments into three key stages, Web 1, Web 2, and Web 3.

Web 1

This is the original World Wide Web, which was developed in 1989. This was the era when static web pages were hosted on servers and usually only allowed read actions from a user's point of view.

Web 2

This is the era when more service-oriented and web-hosted applications started to emerge around 2003. E-commerce websites, social networking, social media, blogs, multimedia sharing, mashups, and web applications are the main features of this period. The current web is Web 2, and even though we have a richer and more interactive Internet, all these services are still centralized. Web 2 has generated massive economic value and provides services that are essential for day-to-day business, personal use, social interactions, and almost every walk of life, but privacy concerns, the need for trusted third parties, and data breaches are genuine issues that need to be addressed. Common examples of centralized Web 2 services include Twitter, Facebook, Google Docs, and email services such Gmail and Hotmail.

Web 3

This is the vision of the decentralized internet or web that will revolutionize the way we use the internet today. This is the era that will be fully user-centric and decentralized without any single authority, large organization, or internet company in control. Some examples of Web 3 are as follows:

- **Steemit**: This is a social media platform based on the Steem blockchain and STEEM cryptocurrency. This cryptocurrency is awarded to contributors for the content they have shared, and the more votes they get, the more tokens they earn. More information is available at `https://steemit.com`.
- **Status**: This is a decentralized multipurpose communication platform providing secure and private communication. More information is available at `https://status.im`.
- **IPFS**: This is a peer-to-peer hypermedia/storage protocol that allows the storage and sharing of data in a decentralized fashion across a peer-to-peer network. More information is available at `https://ipfs.io`.

Other examples include OpenSea, a marketplace for trading NFTs, UniSwap, a decentralized cryptocurrency exchange, and Augur, a decentralized exchange. In Web 3, 3D virtual worlds called metaverses are likely to be extensively used.

Other fast-growing and exciting applications include **decentralized identity and decentralized finance** (**DeFi**), which we introduce in later chapters of this book.

Summary

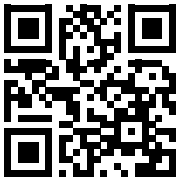
This chapter introduced the concept of decentralization, which is the core service offered by blockchain technology. Although the concept of decentralization is not new, it has gained renewed significance in the world of blockchain. As such, various applications based on a decentralized architecture have recently been introduced.

The chapter began with an introduction to the concept of decentralization. Next, decentralization from the blockchain perspective was discussed. Moreover, ideas relating to the different layers of decentralization in the blockchain ecosystem were introduced. Several new concepts and terms have emerged with the advent of blockchain technology and decentralization from the blockchain perspective, including DAOs, DACs, and DApps. Finally, some innovative trends relating to DApps were presented. We also touched upon the concepts of algocracy and blockcracy.

In the next chapter, fundamental concepts necessary to understand blockchain technology will be presented—principally cryptography, which provides a crucial foundation for blockchain technology.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

3

Symmetric Cryptography

In this chapter, you will be introduced to the concepts, theory, and practical aspects of **symmetric cryptography**. More focus will be given to the elements that are specifically relevant in the context of blockchain technology. This chapter will provide the concepts required to understand the material covered in later chapters.

You will also be introduced to applications of cryptographic algorithms so that you can gain hands-on experience in the practical implementation of cryptographic functions. For this, the OpenSSL command-line tool is used.

This chapter will cover the following topics:

- Introducing cryptography
- Cryptographic primitives
- Advanced Encryption Standard



For the exercises in this chapter, the installation of OpenSSL is necessary. On the Ubuntu Linux distribution, OpenSSL is usually already available. On macOS, LibreSSL is already available. However, it can be installed using the following command:

```
$ sudo apt-get install openssl
```

Examples in this chapter have been developed using OpenSSL 3.0.1. You are encouraged to use this specific version, as all examples in the chapter have been developed and tested with it. If you are running a version other than version 3, the examples may still work but that is not guaranteed.

Introducing cryptography

Cryptography is the science of making information secure in the presence of adversaries. **Ciphers** are algorithms used to encrypt or decrypt data so that if intercepted by an adversary, the data is meaningless to them without **decryption**, which requires a secret key.

Cryptography is primarily used to provide a confidentiality service. On its own, it cannot be considered a complete solution; rather, it serves as a crucial building block within a more extensive security system to address a security problem. For example, securing a blockchain ecosystem requires many different cryptographic primitives, such as hash functions, symmetric key cryptography, digital signatures, and public key cryptography, which we will discuss in the next chapter.

In addition to a confidentiality service, cryptography also provides other security services such as integrity, authentication (entity authentication and data origin authentication), and non-repudiation. Additionally, accountability is provided, which is a requirement in many security systems.

A generic cryptographic model is shown in the following diagram:

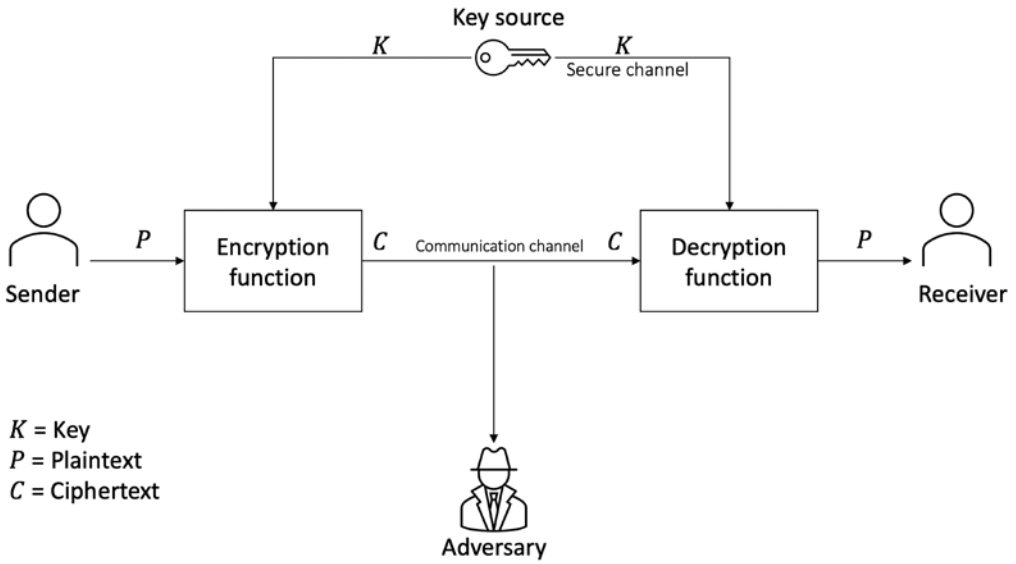


Figure 3.1: A generic encryption and decryption model

In the preceding diagram, *P*, *C*, and *K* represent plaintext, ciphertext, and key (data that is used to encrypt plaintext and decrypt ciphertext) respectively.

Services provided by cryptography

We mentioned the fact that cryptography provides various services. In the following section, we'll introduce these services.

Confidentiality is the assurance that information is only available to authorized entities.

Integrity is the assurance that information is modifiable only by authorized entities.

Authentication provides assurance about the identity of an entity or the validity of a message. There are two types of authentication mechanisms, namely, entity authentication and data origin authentication, which are discussed in the following sections.

Entity authentication is the assurance that an entity is currently involved and active in a communication session. Traditionally, users are issued a username and password that is used to gain access to the various platforms with which they are working. This practice is known as **single-factor authentication**, as there is only one factor involved, namely, something you know—that is, the password and username.

This type of authentication is not very secure for a variety of reasons, for example, password leakage; therefore, additional factors are now commonly used to provide better security. The use of additional techniques for user identification is known as **multi-factor authentication**:

- The first method uses *something you have*, such as a hardware token or a smart card. In this case, a user can use a hardware token in addition to login credentials to gain access to a system. A user who has access to the hardware token and knows the login credentials will be able to access the system. If the hardware token is inaccessible (for example, lost or stolen) or the login password is forgotten by the user, access to the system will be denied. The hardware token won't be of any use on its own unless the login password (*something you know*) is also known and used in conjunction with the hardware token.
- The second method uses *something you are*, which uses biometric features to identify the user. With this method, a user's fingerprint, retina, iris, or hand geometry is used to ensure that the user was indeed present during the authentication process. However, careful implementation is required to guarantee a high level of security, as some research has suggested that biometric systems can be circumvented under specific conditions.

Data origin authentication is another type of authentication, which is also known as **message authentication**. It is an assurance that the source of the information is indeed verified. Data origin authentication guarantees data integrity because, if a source is corroborated, then the data must not have been altered. Various methods are used for this type of authentication, such as **message authentication codes** (**MACs**) and digital signatures, which we'll cover later in this chapter and the next chapter.

Another important assurance provided by cryptography is **non-repudiation**. It is the assurance that an entity cannot deny a previous commitment or action by providing incontrovertible cryptographic evidence. This property is essential in debatable situations whereby an entity has denied the actions performed, for example, the placement of an order on an e-commerce system. Non-repudiation has been an active research area for many years. Disputes in electronic transactions are a common issue, and there is a need to address them to increase consumers' confidence levels in such services.

The non-repudiation protocol usually runs in a communications network, and it is used to provide evidence that an action has been taken by an entity (originator or recipient) on the network. In this context, two communications models can be used to transfer messages from originator *A* to recipient *B*:

- A message is sent directly from originator *A* to recipient *B*
- A message is sent to a delivery agent from originator *A*, which then delivers the message to recipient *B*

The primary requirements of a non-repudiation protocol are fairness, effectiveness, and timeliness. In many scenarios, there are multiple participants involved in a transaction. For example, in electronic trading systems, there can be clearing agents, brokers, traders, and other entities who can be involved in a single transaction. To address this problem, **multi-party non-repudiation** (**MPNR**) protocols have been developed.

Accountability is the assurance that actions affecting security can be traced back to the responsible party. This is usually provided by logging and audit mechanisms in systems where a detailed audit is required due to the nature of the business, for example, in electronic trading systems. Detailed logs are vital to trace an entity's actions, such as when a trade is placed in an audit record with the date and timestamp and the entity's identity is generated and saved in the log file. This log file can optionally be encrypted and be part of the database or a standalone ASCII text log file on a system.

To provide all of the services discussed in this section, different cryptographic primitives are used, which are presented in the next section.

Cryptographic primitives

Cryptographic primitives are the basic building blocks of a security protocol or system. A **security protocol** is a set of steps taken to achieve the required security goals by utilizing appropriate security mechanisms. Various types of security protocols are in use, such as authentication protocols, non-repudiation protocols, and key management protocols.

The taxonomy of cryptographic primitives can be visualized as shown here:

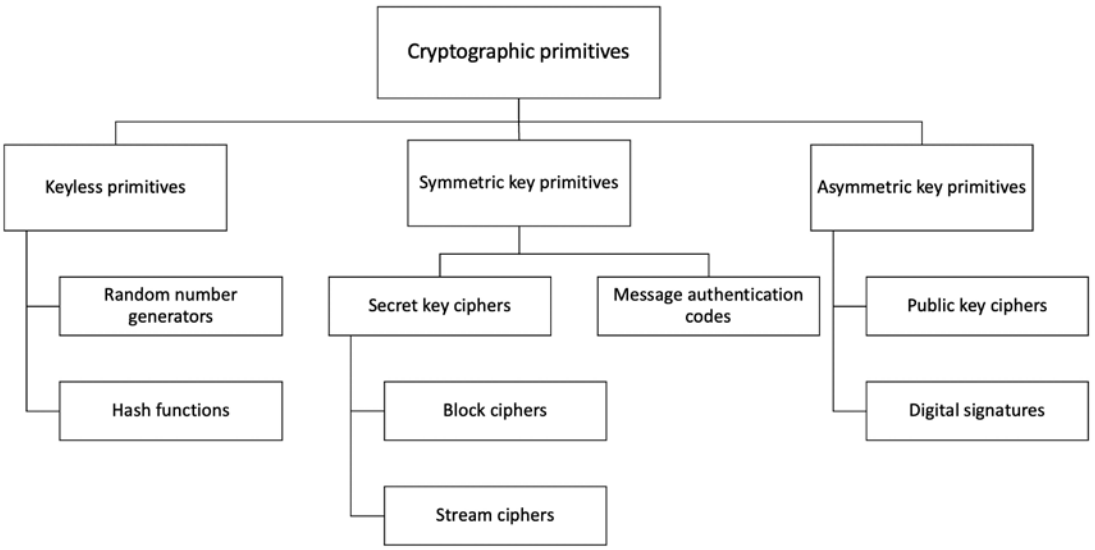


Figure 3.2: Cryptographic primitives

As shown in the preceding cryptographic primitive taxonomy diagram, cryptography is mainly divided into three categories: **keyless primitives, symmetric key primitives, and asymmetric key primitives**.

Keyless primitives and symmetric cryptography are discussed further in the next section, and we will cover asymmetric cryptography, or public key cryptography, in *Chapter 4*, *Asymmetric Cryptography*.

Keyless primitives

In this section, we will introduce two keyless primitives, namely, random numbers and hash functions.

Random numbers

Randomness provides an indispensable element for the security of cryptographic protocols. It is used for the generation of keys and in encryption algorithms. Randomness ensures that the operations of a cryptographic algorithm do not become predictable enough to allow cryptanalysts to infer the outputs and operations of the algorithm, which will make the algorithm insecure. It is quite a feat to generate suitable randomness with a high degree of uncertainty, but there are methods that ensure an adequate level of randomness is generated for use in cryptographic algorithms.

There are two categories of the source of randomness, namely, **random number generators** and **pseudorandom number generators**.

Random number generators

Random number generators (**RNGs**) are software or hardware systems that make use of the randomness available in the real world, called **real randomness**. This can be temperature variations, thermal noises from various electronic components, or acoustic noise. Other sources are based on physical phenomena such as keystrokes, mouse cursor movements, or disk movements of a running computer system. These types of sources of randomness are not very practical due to the difficulty of acquiring this data or not having enough entropy. Also, these sources are not always available or could be available only for a limited time.

Pseudorandom number generators

Pseudorandom number generators (**PRNGs**) are deterministic functions that work on the principle of using a random initial value called a **seed** to produce a random-looking set of elements. PRNGs are commonly used to generate keys for encryption algorithms. A common example of a PRNG is **Blum-Blum-Shub** (**BBS**). PRNGs are a better alternative to RNGs due to their reliability and deterministic nature.



More information on BBS is available in the following original research paper, *Blum, L., Blum, M., and Shub, M., 1986. A simple unpredictable pseudo-random number generator. SIAM Journal on computing, 15(2), pp.364–383*:

`https://shub.ccny.cuny.edu/articles/1986-A_simple_unpredictable_pseudo-random_number_generator.pdf`

Generating random strings

The following command can be used to generate a pseudorandom string using OpenSSL:

```
$ openssl rand -hex 16
```

It will produce the output random 16-byte string encoded in hex.

```
06532852b5da8a5616dfade354a9f270
```

There are other variations that you can explore more with the OpenSSL command-line tool. Further information and help can be retrieved with the following command:

```
$ openssl help
```

In the next section, we will look at hash functions, which play a crucial role in the development of blockchain.

Hash functions

Hash functions are used to create fixed-length digests of arbitrarily long input strings. Hash functions are **keyless**, and they provide a **data integrity service**. They are usually built using iterated and dedicated hash function construction techniques.

Various families of hash functions are available, such as MD, SHA1, SHA-2, SHA-3, RIPEMD, and Whirlpool. Hash functions are commonly used for digital signatures and MACs, such as HMACs.

Hash functions are also typically used to provide data integrity services. These can be used both as one-way functions and to construct other cryptographic primitives, such as MACs and digital signatures. Some applications use hash functions as a means of generating PRNGs. There are two practical properties of hash functions that must be met depending on the level of integrity required:

- **Compression of arbitrary messages into fixed-length digests**: This property relates to the fact that a hash function must be able to take an input text of any length and output a fixed-length compressed message. Hash functions produce a compressed output in various bit sizes, usually between 128-bit and 512-bit.
- **Easy to compute**: Hash functions are efficient and fast one-way functions. It is required that hash functions be very quick to compute regardless of the message size. The efficiency may decrease if the message is too big, but the function should still be fast enough for practical use.

There are also three security properties that must be met, depending on the level of integrity:

- **Pre-image resistance**: This property states that if given a value y, it is computationally infeasible (almost impossible) to find a value x such that $h(x)= y$. Here, h is the hash function, x is the input, and y is the hash. The first security property requires that y cannot be reverse computed to x. x is considered a pre-image of y, hence the name **pre-image resistance**. This is also called a one-way property.
- **Second pre-image resistance**: The **second pre-image resistance** property states that given x it is computationally infeasible to find another value x' such that $x' \neq x$ and $h(x') = h(x)$. This property is also known as **weak collision resistance**.
- **Collision resistance**: The **collision resistance** property states that it is computationally infeasible to find two distinct values x' and x such that $h(x') = h(x)$. In other words, two different input messages should not hash to the same output. This property is also known as strong collision resistance.

These security properties are depicted in the following diagram:

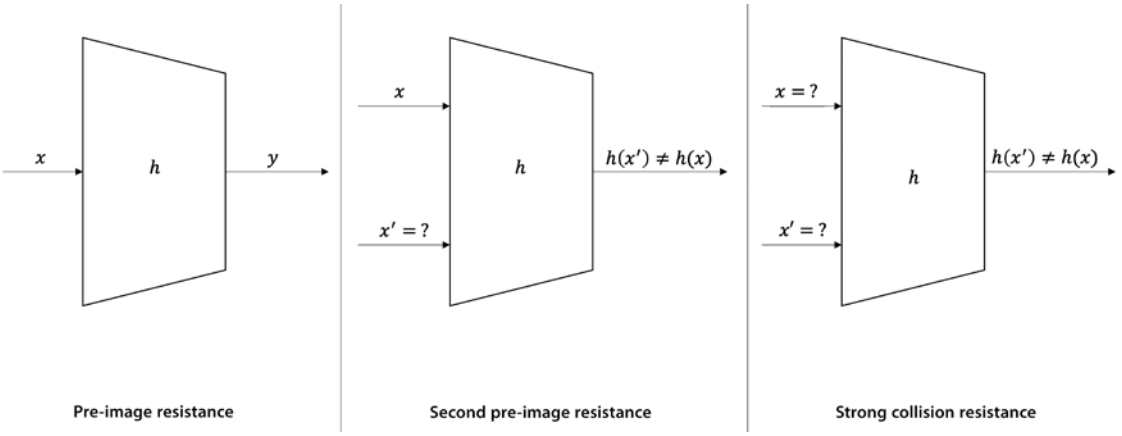


Figure 3.3: Three security properties of hash functions

Due to their very nature, hash functions will always have some collisions. This is a situation where two different messages hash to the same output. However, they should be computationally impractical to find. A concept known as the **avalanche effect** is desirable in all cryptographic hash functions. The avalanche effect specifies that a small change, even a single character change in the input text, will result in an entirely different hash output.

Hash functions are usually designed by following an iterated hash functions approach. With this method, the input message is compressed in multiple rounds on a block-by-block basis to produce the compressed output. A popular type of iterated hash function is the **Merkle-Damgard construction**. This construction is based on the idea of dividing the input data into equal block sizes and then feeding them through the compression functions in an iterative manner. The collision resistance of the property of compression functions ensures that the hash output is also collision-resistant. In addition to Merkle-Damgard, there are various other constructions of compression functions proposed by researchers, for example, Miyaguchi-Preneel and Davies-Meyer.

Multiple categories of hash functions are introduced in the following section.

Message digest functions

Message digest (**MD**) functions were prevalent in the early 1990s. MD4 and MD5 fall into this category. Both MD functions were found to be insecure due to message collisions found and are not recommended for use anymore. MD5 is a 128-bit hash function that was commonly used for file integrity checks.

Secure Hash Algorithms

The following list describes the most common **secure hash algorithms** (**SHAs**):

- **SHA-0**: This is a 160-bit function introduced by the U.S. **National Institute of Standards and Technology** (**NIST**) in 1993.

- **SHA-1**: SHA-1 was introduced in 1995 by NIST as a replacement for SHA-0. This is also a 160-bit hash function. SHA-1 is used commonly in SSL and TLS implementations. It should be noted that SHA-1 is now considered insecure, and it is being deprecated by certificate authorities. Its usage is discouraged in any new implementations.
- **SHA-2**: This category includes four functions defined by the number of bits of the hash: SHA-224, SHA-256, SHA-384, and SHA-512.
- **SHA-3**: This is the latest family of SHA functions. SHA3-224, SHA3-256, SHA3-384, and SHA3-512 are members of this family. SHA-3 is a NIST-standardized version of Keccak.
- **RIPEMD**: RIPEMD is the acronym for **RACE Integrity Primitives Evaluation Message Digest**. It is based on the design ideas used to build MD4. There are multiple versions of RIPEMD, including 128-bit, 160-bit, 256-bit, and 320-bit.
- **Whirlpool**: This is based on a modified version of the Rijndael cipher known as *W*. It uses the Miyaguchi-Preneel compression function, which is a type of one-way function used for the compression of two fixed-length inputs into a single fixed-length output. It is a single-block length compression function.

Hash functions have many practical applications ranging from simple file integrity checks and password storage to use in cryptographic protocols and algorithms. They are used in **Peer to Peer** (**P2P**) networks, P2P file sharing, virus fingerprinting, bloom filters, Merkle trees, Patricia trees, and distributed hash tables.

In the following section, you will be introduced to the design of SHA-256 and SHA-3. Both of these are used in Bitcoin and Ethereum, respectively. Ethereum uses Keccak, which is the original algorithm presented to NIST, rather than NIST standard SHA-3. NIST, after some modifications, such as an increase in the number of rounds and simpler message padding, standardized Keccak as SHA-3.

SHA-256

SHA-256 has an input message size limit of 2^{64} - 1 bits. The block size is 512 bits, and it has a word size of 32 bits. The output is a 256-bit digest. The compression function processes a 512-bit message block and a 256-bit intermediate hash value. There are two main components of this function: the compression function and a message schedule. The algorithm works as follows, in nine steps:

Pre-processing:

a. Padding of the message is used to adjust the length of a block to 512 bits if it is smaller than the required block size of 512 bits.
b. Parsing the message into message blocks, which ensures that the message and its padding are divided into equal blocks of 512 bits.

c. Setting up the initial hash value, which consists of the eight 32-bit words obtained by taking the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. These initial values are fixed and chosen to initialize the process. They provide a level of confidence that no backdoor exists in the algorithm.

Hash computation:

a. Each message block is then processed in a sequence, and it requires 64 rounds to compute the full hash output. Each round uses slightly different constants to ensure that no two rounds are the same.
b. The message schedule is prepared.
c. Eight working variables are initialized.
d. The compression function runs 64 times.
e. The intermediate hash value is calculated.
f. Finally, after repeating steps 5 through 8 until all blocks (chunks of data) in the input message are processed, the output hash is produced by concatenating intermediate hash values.

At a high level, SHA-256 can be visualized in the following diagram:

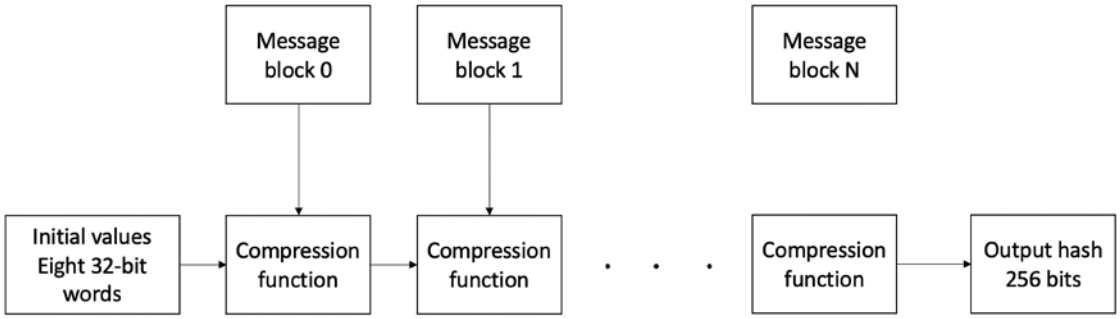


Figure 3.4: SHA-256 high-level overview

As shown in the preceding diagram, SHA-256 is a Merkle-Damgard construction that takes the input message and divides it into equal blocks (chunks of data) of 512 bits. Initial values (or initial hash values) or the initialization vector are composed of eight 32-bit constant words (*a, b, c, d, e, f, g* —256 bits each) that are fed into the compression function with the first message block. Subsequent blocks are fed into the compression function until all blocks are processed, and finally, the output hash is produced.

The compression function of SHA-256 is shown in the following diagram:

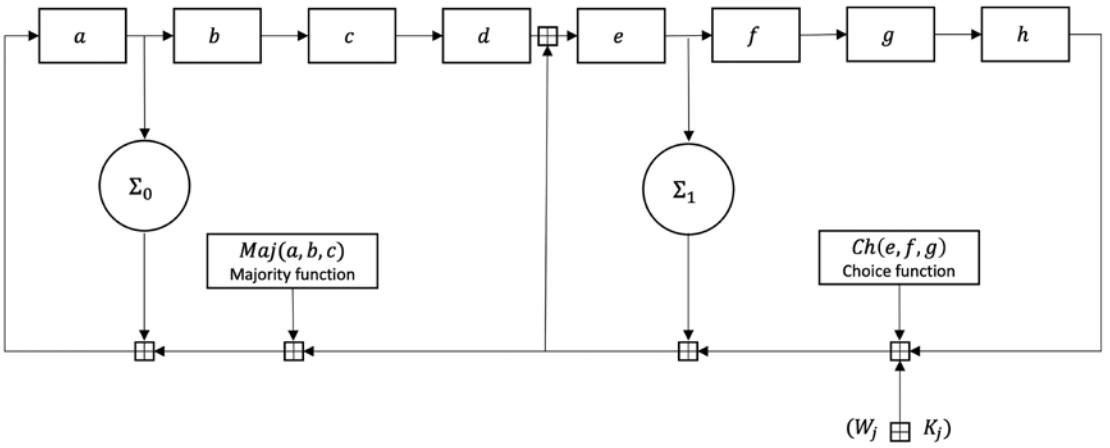


Figure 3.5: SHA-2 compression function

In the preceding diagram, *a, b, c, d, e, f, g* and *h* are the registers for eight initial pre-determined constants and then for intermediate hash values for the next blocks. *Maj* and *Ch* functions are defined as the formulae shown below:

$$Maj(a,b,c) = (a \wedge b) \oplus (a \wedge c) \oplus (b \wedge c)$$

$$Ch(e,f,g) = (e \wedge f) \oplus (\neg e \wedge g)$$

where $\wedge$ is bitwise AND, $\oplus$ is bitwise XOR, and $\neg$ is bitwise NOT. XOR can be replaced with bitwise OR without any change in the output. The functions operate on vectors of 32 bits.

Maj is the "majority" function where the output produced is based on the majority of the inputs. In other words, if most of the inputs are 1 then the output is 1; otherwise, 0. Here 3 input bits x, y and z produce the output.

Ch is the choice function where a choice is made between the inputs depending upon the value of one of the inputs. It is like an "if .. then .. else" construct in programming, or a data selector, or input selector. It works on the following principle:

$$Ch(e,f,g) = \begin{cases} f, & if\ e = 1 \\ g, & if\ e = 0 \end{cases}$$

Here *e*, *f*, *g* are inputs, and depending on whether $e = 1$ or $e = 0$, either *f* or *g* is selected.

As shown in *Figure 3.5*, *Maj* and *Ch* functions are applied bitwise. Σ_0 and Σ_1 perform bitwise rotation. $\Sigma_0(a)$ returns $(a \ggg 2) \oplus (a \ggg 13) \oplus (a \ggg 22)$, and $\Sigma_1(e)$ returns $(e \ggg 6) \oplus (e \ggg 11) \oplus (e \ggg 25)$. $\boxplus$ means addition modulo 2^{32} for SHA-256.

The mixing constants are W_i and K_j, which are added in the main loop (compressor function) of the hash function, which runs 64 times.

With this, our introduction to SHA-256 is complete, and next we explore a newer class of hash functions known as the SHA-3 algorithm.

SHA-3 (Keccak)

The structure of **SHA-3** is very different from that of SHA-1 and SHA-2. The key idea behind SHA-3 is based on unkeyed permutations, as opposed to other typical hash function constructions that used keyed permutations. **Keccak** also does not make use of the Merkle-Damgard transformation that is commonly used to handle arbitrary-length input messages in hash functions. A newer approach, called **sponge and squeeze construction**, is used in Keccak. It is a random permutation model. Different variants of SHA-3 have been standardized, such as SHA3-224, SHA3-256, SHA3-384, SHA3-512, SHAKE128, and SHAKE256. SHAKE128 and SHAKE256 are **extendable-output functions** (**XOFs**), which allow the output to be extended to any desired length.

The following diagram shows the sponge and squeeze model, which is the basis of SHA-3 or Keccak. Analogous to a sponge, the data (*m* input data) is first absorbed into the sponge after applying padding. It is then changed into a subset of the permutation state using **XOR** (**exclusive OR**), and then the output is squeezed out of the sponge function that represents the transformed state. The rate *r* is the input block size of the sponge function, while capacity *c* determines the security level:

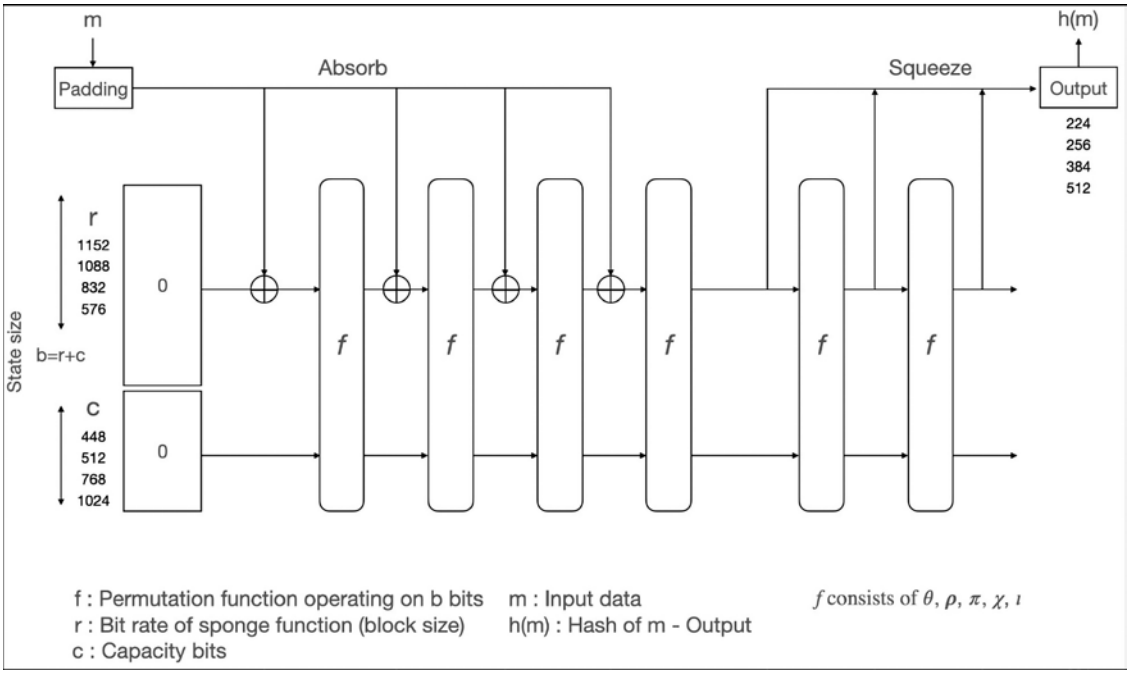


Figure 3.6: The SHA-3 absorbing and squeezing function in SHA3

In the preceding diagram, state size *b* is calculated by adding bit rate *r* and capacity bits *c*. *r* and *c* can be any values as long as sizes of *r* + *c* are 25, 50, 100, 200, 400, 800, or 1,600. The state is a 3-dimensional bit matrix. The initial state is set to 0.

The data *m* is entered into the absorb phase block by block via XOR ⊕ after applying padding.

The following table shows the value of bit rate *r* (block size) and capacity *c* required to achieve the desired output hash size under the most efficient setting of *r* + *c* = 1600:

r (block size)	c (capacity)	Output hash size
1152	448	224
1088	512	256
832	768	384
576	1024	512

The function *f* is a permutation function. It contains five transformation operations:

- θ(Theta): XOR bits in the state, used for mixing
- ρ (Rho): Diffusion function performing rotation of bits
- π (Pi): Diffusion function
- χ (XOR): Each bit, bitwise combine
- ι (Iota): Combination with round constants

The details of each transformation operation are beyond the scope of this book. The key idea is to apply these transformations to achieve the **avalanche effect**, which we introduced earlier in this chapter. These five operations combined form a **round.** In the SHA-3 standard, the number of rounds is 24 to achieve the desired level of security.

We will see some applications and examples of constructions built using hash functions such as Merkle trees in *Chapter 9*, *Ethereum Architecture*.

Encrypting messages with SHA-256

The following command will produce a hash of 256 bits of `Hello` messages using the SHA-256 algorithm:

```
$ echo -n 'Hello' | openssl dgst -sha256
(stdin)= 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
```

Now we run the following command to see the avalanche effect in action:

```
$ echo -n 'hello' | openssl dgst -sha256
(stdin)= 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824
```

Note that even a small change in the text, such as changing the case of the letter `H`, results in a big change in the output hash:

```
Hello:
18:5f:8d:b3:22:71:fe:25:f5:61:a6:fc:93:8b:2e:26:43:06:ec:30:4e:da:51:80:07:
d1:76:48:26:38:19:69
hello:
2c:f2:4d:ba:5f:b0:a3:0e:26:e8:3b:2a:c5:b9:e2:9e:1b:16:1e:5c:1f:a7:42:5e:73:
04:33:62:93:8b:98:24
```

Usually, hash functions do not use a key. Nevertheless, if they are used with a key, then they can be used to create **MACs**.

Applications of cryptographic hash functions

There are various constructs that have been built using basic cryptographic parameters to solve different problems in computing. These constructs are also used in blockchains to provide various protocol-specific services. For example, hash functions are used to build Merkle trees, which are used to efficiently and securely verify large amounts of data in distributed systems. Some other applications of hash functions in blockchains are to provide several security services.

These services are listed here:

- Hash functions are used in cryptographic puzzles such as the **proof of work** (**PoW**) mechanism in Bitcoin. Bitcoin's PoW makes use of the SHA-256 cryptographic hash function.
- The generation of addresses in blockchains. For example, in Ethereum, blockchain accounts are represented as addresses. These addresses are obtained by hashing the public key with the Keccak-256 hash algorithm and then using the last 20 bytes of this hashed value.
- Message digests in digital signatures.
- The creation of Merkle trees to guarantee the integrity of transaction structure in the blockchain. Specifically, this structure is used to quickly verify whether a transaction is included in a block or not. However, note that Merkle trees on their own are not a new idea; it has just been made more popular with the advent of blockchain technology.

Merkle trees are the core building blocks of all blockchains, for example, Bitcoin and Ethereum. We will explore Merkle trees in detail now.

Merkle tree

The Merkle tree was introduced by Ralph Merkle. **Merkle trees** enable the secure and efficient verification of large datasets. A diagram of a Merkle tree is shown below:

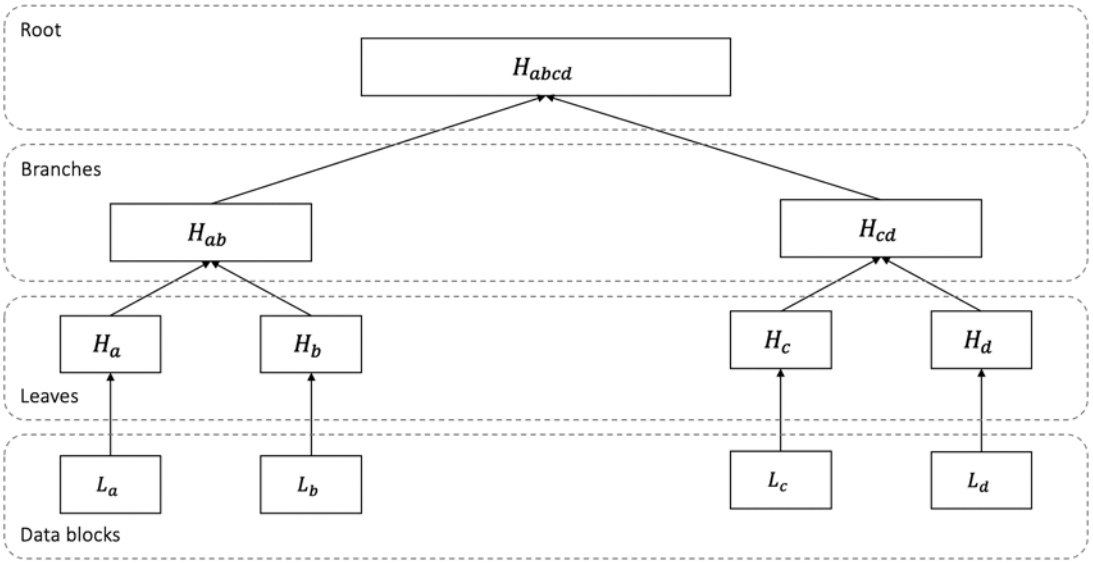


Figure 3.7: A Merkle tree

A Merkle tree is a binary tree in which the inputs are first placed at the leaves (nodes with no children), and then the values of pairs of child nodes are hashed together to produce a value for the parent node (internal node), until a single hash value known as a **Merkle root** is achieved. This structure helps to quickly verify the integrity of the entire tree (entire dataset), but just by verifying the Merkle root on top of the Merkle tree, because if any change occurs in any of the hashes in the tree, the Merkle root will also change.

A Merkle tree is used to prove that a data block B_i is a member of a set of N data blocks, formally $B \in \{B_1, .., B_N\}$. With a Merkle root and a candidate data block, using Merkle trees, we can prove that a data block exists within a set. This proof is called Merkle proof and involves obtaining a set of hashes called "Merkle path" for a given data block and Merkle root. In other words, the Merkle path for a data element (or block) is the minimum chain of hashes required to recreate the Merkle root by repeatedly hashing and concatenating until the Merkle root is created. For example, in *Figure 3.7*, if the existence of L_a needs to be proven, then the Merkle path would be $H_a \rightarrow H_{ab}$ up to the root H_{abcd}. We can prove the existence of data block L_a by recreating the Merkle root through this path and comparing it with the provided Merkle root; if the Merkle roots don't match, then we can say that the data element in question is not in the Merkle tree or vice versa. The Merkle path is also known as the authentication path.

This is the reason why the integrity of the system can be verified quickly by just looking at the Merkle root. Another advantage of Merkle trees is that there is no requirement to store large amounts of data, only the hashes of the data, which are fixed-length digests of the large dataset. Due to this property, the storage and management of Merkle trees are easy and efficient as they take up a very small amount of space for storage. Also, since the tree is storage efficient, the relevant proofs for integrity are also smaller in size and quick to transmit over the network, thus making them bandwidth-efficient over the network.

Merkle Patricia trie

To understand Patricia trees, we need to understand **trie** first. A trie, or a **digital tree**, is an ordered tree data structure used to store a dataset. The **Practical Algorithm to Retrieve Information Coded in Alphanumeric** (**PATRICIA**) tree, also known as a **Radix tree**, is a compact representation of a trie in which a node that is the only child of a parent is merged with its parent. The keys represent the path to reach a node. The nodes that share the same key can share the same path, thus making it an efficient way of finding common prefixes while utilizing a small amount of memory.

A **Merkle-Patricia tree** is a tree that has a root node that contains the hash value of the entire data structure. The Merkle-Patricia tree combines Merkle and Patricia trees where Patricia is used for efficient storage and Merkle enables tamper-proof data validation. Patricia tree is also modified to store hexadecimal strings instead of bits and support 16 branches. Merkle Patricia tree has four types of nodes:

- **Null** nodes, which are non-existent nodes represented as empty strings.
- **Branch** nodes, which are 17-item nodes used for branching. They have 16 possible branches from 0–F, and the 17th item (value) stores a value only if the node is terminating.
- **Extension** nodes, which are 2-item nodes containing a path and a value. They are used for compression, where a common part of multiple keys is stored, i.e., a shared nibble.
- **Leaf** nodes, which are 2-item node containing a path and a value. The leaf node is the terminating node in the path with no children and groups common key suffixes in the path as the compression technique.



Leaf nodes and extension nodes are distinguished by a hex prefix value. A leaf node with an even number of nibbles has a prefix of 2; for an odd number of nibbles, 3 is used. Extension nodes with an even number of nibbles have the prefix 0, and for an odd number the prefix is 1.

The figure below shows how a Merkle-Patricia tree is used to store keys and values (accounts and balances in Ethereum):

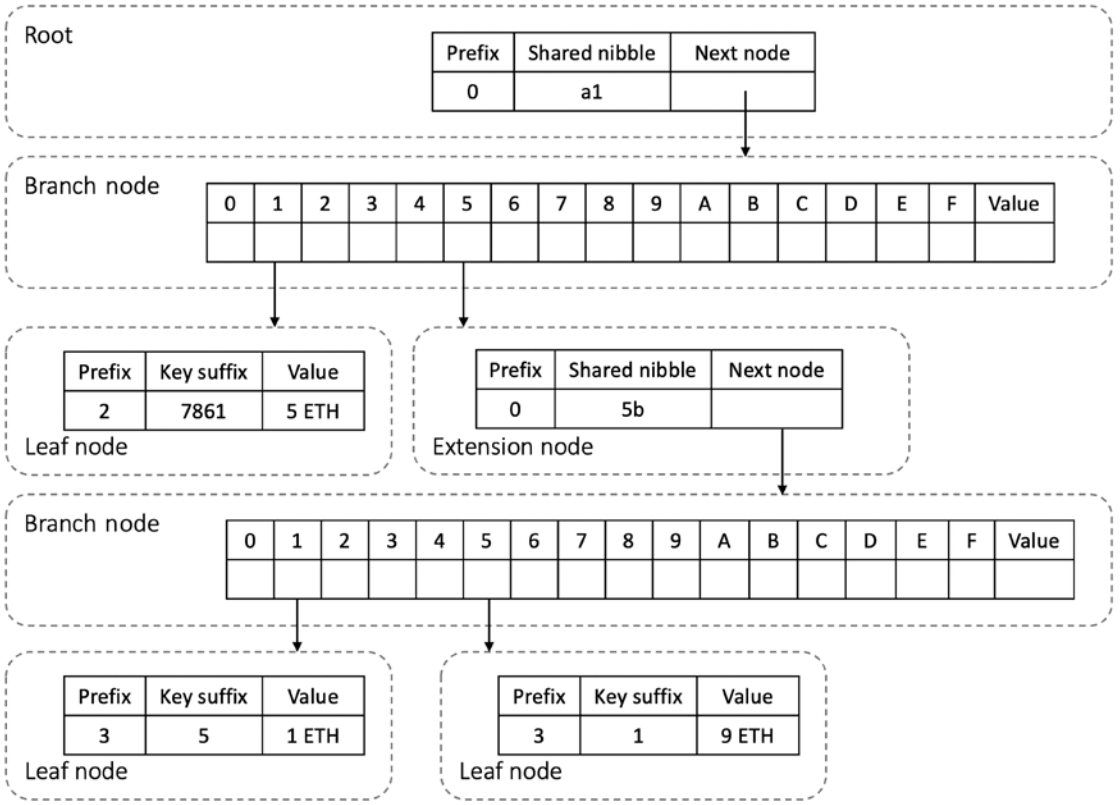


Figure 3.8: Merkle-Patricia tree

In the diagram above, key a117861 has a value of 5 ETH, key a155b15 has a value of 1 ETH, and key a155b51 has a value of 9 ETH. Notice that a1 is a common prefix (a shared nibble) and is grouped at the root extension node. Then a leaf node (with no child) with the key suffix 7861 branches out from the branch node. Further, another extension node, with the shared nibble (prefix) 5b branches out and leads to another branch node, finally ending with two leaf nodes with no further children with the key suffixes 5 and 1. Merkle-Patricia trees are used in the Ethereum blockchain to store key-value pairs where the root is hashed using SHA3 and included in the header of the block.

Distributed hash tables

A hash table is a data structure that is used to map keys to values. Internally, a hash function is used to calculate an index into an array of buckets from which the required value can be found. Buckets have records stored in them using a hash key and are organized into a particular order.

With the definition provided above in mind, we can think of a **distributed hash table** (**DHT**) as a data structure where data is spread across various nodes, and nodes are equivalent to buckets in a P2P network. The following diagram shows how a DHT works:

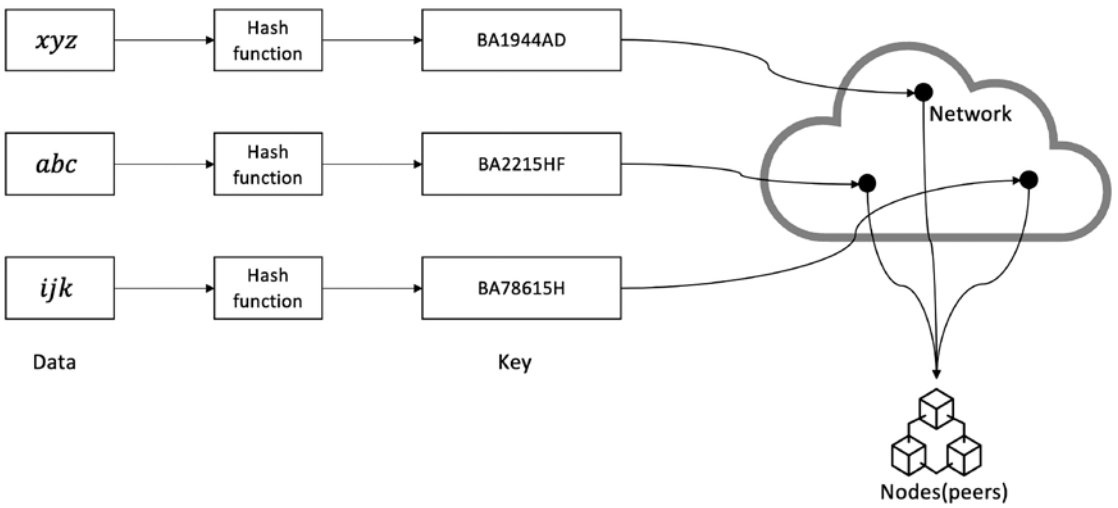


Figure 3.9: Distributed hash table

In the preceding diagram, data is passed through a hash function, which then generates a compact key. This key is then linked with the data (values) on the P2P network. When users on the network request the data (via the filename), the filename can be hashed again to produce the same key, and any node on the network can then be requested to find the corresponding data. A DHT provides decentralization, fault tolerance, and scalability.

In this section, we covered various applications of cryptographic hash functions. Hash functions are of particular importance in blockchains, as they are key to constructing Merkle trees, which are used in blockchains for the efficient and fast verification of large datasets. We will see how MACs and HMACs work after we've introduced symmetric key cryptography, because these constructions make use of keys for encryption.

Symmetric key primitives

Symmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is the same one that is used for decrypting the data. Thus, it is also known as **shared key cryptography**. The key must be established or agreed upon before the data exchange occurs between the communicating parties. This is the reason it is also called **secret key cryptography**.

Other types of keys are **public keys** and **private keys**, which are generated in pairs for public key cryptography or asymmetric cryptography. Public keys are used for encrypting the plaintext, whereas private keys are used for decryption and are expected to be kept secret by the receiver.

Keys can also be **ephemeral** (temporary) or **static**. Ephemeral keys are intended to be used only for a short period of time, such as in a single session between the participants, whereas static keys are intended for long-term usage. Another type of key is called the **master key**, which is used for the protection, encryption, decryption, and generation of other keys.

There are different methods to generate keys. These methods are listed as follows:

- Random, where a random number generator is used to generate a random set of bytes that can be used as a key.
- Key derivation-based, where a single key or multiple keys are derived from a password. A **key derivation function** (**KDF**) is used for this purpose, taking the password as input, and converting that into a key. Commonly used key derivation functions are **Password-Based Key Derivation Function 1** (**PBKDF1**), PBKDF2, Argon 2, and Scrypt.



KDFs are used in Ethereum wallets (keystore files) to generate an AES symmetric key that is used to encrypt the wallets. The KDF function used in Ethereum wallets is **Scrypt**. We will explain all this in detail in *Chapter 9*, *Ethereum Architecture*. More information about PBKDFs can be found at `https://tools.ietf.org/html/rfc8018`.

- Key agreement protocol, where two or more participants run a protocol that produces a key, and that key is then shared between participants. In key agreement schemes, all participants contribute equally in the effort to generate the shared secret key. The most used key agreement protocol is the Diffie-Hellman key exchange protocol.

Within the encryption schemes, there are also some random numbers that play a vital role during the operation of the encryption process. These random numbers are explained as follows:

- The **nonce** is a number that can be used only once in a cryptographic protocol. It must not be reused. Nonces can be generated from a large pool of random numbers or they can also be sequential. The most common use of nonces is to prevent replay attacks in cryptographic protocols.
- The **initial value** or **initialization vector** (**IV**) is a random number, which is basically a nonce, but it must be chosen in an unpredictable manner. This means that it cannot be sequential. IVs are used extensively in encryption algorithms to provide increased security.
- The **salt** is a cryptographically strong random value that is typically used in hash functions to provide defense against dictionary or rainbow attacks. Using dictionary attacks, hashing-based password schemes can be broken by trying hashes of millions of words from a dictionary in a brute-force manner and matching it with the hashed password. If a salt is used, then a dictionary attack becomes difficult to run because a random salt makes each password unique, and secondly, the attacker will then have to run a separate dictionary attack for random salts, which is quite unfeasible.

Now that we've introduced symmetric cryptography, we're ready to look at MACs and HMACs.

Message authentication codes

Message authentication codes (**MACs**) are sometimes called **keyed hash functions**, and they can be used to provide message integrity and authentication. More specifically, they are used to provide data origin authentication. These are symmetric cryptographic primitives that use a shared key between the sender and the receiver. MACs can be constructed using block ciphers or hash functions.

Hash-based message authentication codes

Like the hash function, **hash-based MACs** (**HMACs**) produce a fixed-length output and take an arbitrarily long message as the input. In this scheme, the sender signs a message using the MAC and the receiver verifies it using the shared key. The key is hashed with the message using either of the two methods known as **secret prefix** or **secret suffix**. With the secret prefix method, the key is concatenated with the message; that is, the key comes first, and the message comes afterward, whereas with the secret suffix method, the key comes after the message, as shown in the following equations:

$$Secret\ prefix: M\ =\ MACk(x)\ =\ h(k||x)$$

$$Secret\ suffix: M = MACk(x)\ =\ h(x||k_{)}$$

There are pros and cons to both methods. Some attacks on both schemes have occurred. There are HMAC constructions schemes that use various techniques, such as **ipad** (**inner padding**) and **opad** (**outer padding**) that have been proposed by cryptographic researchers. These are considered secure with some assumptions:

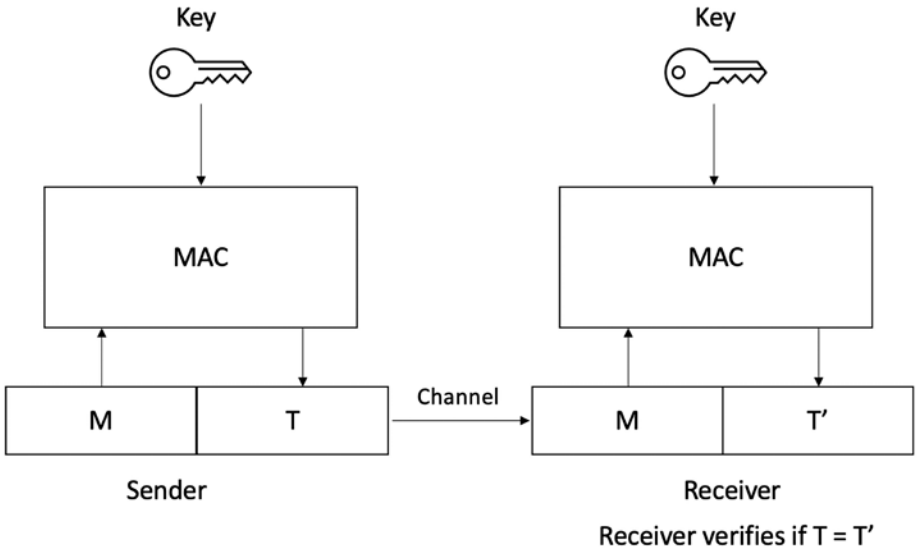


Figure 3.10: Operation of a MAC function

Secret key ciphers

There are two types of secret key ciphers or symmetric ciphers: **stream ciphers** and **block ciphers**.

Stream ciphers

Stream ciphers are encryption algorithms that apply encryption algorithms on a bit-by-bit basis (one bit at a time) to plaintext using a keystream.

In stream ciphers, encryption and decryption are the same functions because they are simple modulo 2 additions or **XOR** operations. The fundamental requirement in stream ciphers is the security and randomness of keystreams. Various techniques ranging from PRNGs to true hardware RNGs have been developed to generate random numbers, and it is vital that all key generators be cryptographically secure:

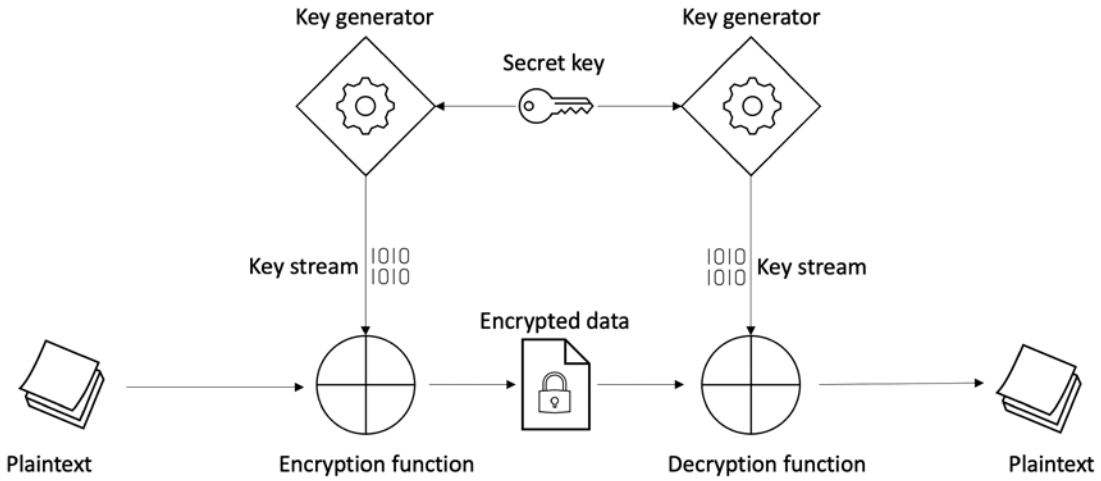


Figure 3.11: Operation of a stream cipher

There are two types of stream ciphers:

- Synchronous stream ciphers are those where the keystream is dependent only on the key.
- Asynchronous stream ciphers have a keystream that is also dependent on the encrypted data.

This concept can be visualized in *Figure 3.12* below.

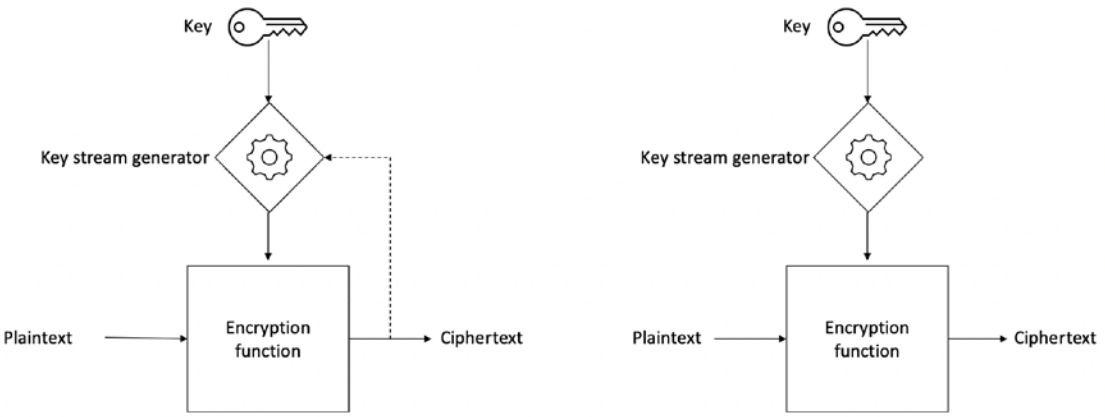


Figure 3.12: Asynchronous stream cipher (left) vs synchronous stream cipher (right)

RC4 and A5 are commonly used stream ciphers.

Block ciphers

Block ciphers are encryption algorithms that break up the text to be encrypted (plaintext) into blocks of a fixed length and apply the encryption block by block. Block ciphers are generally built using a design strategy known as a **Feistel cipher**. Recent block ciphers such as AES (Rijndael) have been built using a combination of substitution and permutation called a **substitution-permutation network** (**SPN**). **Data Encryption Standard** (**DES**) and **Advanced Encryption Standard** (**AES**) are typical examples of block ciphers.

Feistel ciphers are based on the Feistel network, which is a structure developed by Horst Feistel. This structure is based on the idea of combining multiple rounds of repeated operations to achieve desirable cryptographic properties known as **confusion** and **diffusion**. Feistel networks operate by dividing data into two blocks (left and right) and processing these blocks via keyed **round functions** in iterations to provide sufficient pseudorandom permutations.

Confusion adds complexity to the relationship between the encrypted text and plaintext. This is achieved by substitution. In practice, *A* in plaintext is replaced by *X* in encrypted text. In modern cryptographic algorithms, substitution is performed using lookup tables called **S-boxes**. The **diffusion** property spreads the plaintext statistically over the encrypted data. This ensures that even if a single bit is changed in the input text, it results in changing at least half (on average) of the bits in the ciphertext. Confusion is required to make finding the encryption key very difficult, even if many encrypted and decrypted data pairs are created using the same key. In practice, this is achieved by transposition or permutation.

A key advantage of using a Feistel cipher is that encryption and decryption operations are almost identical and only require a reversal of the encryption process to achieve decryption. DES is a prime example of Feistel-based ciphers:

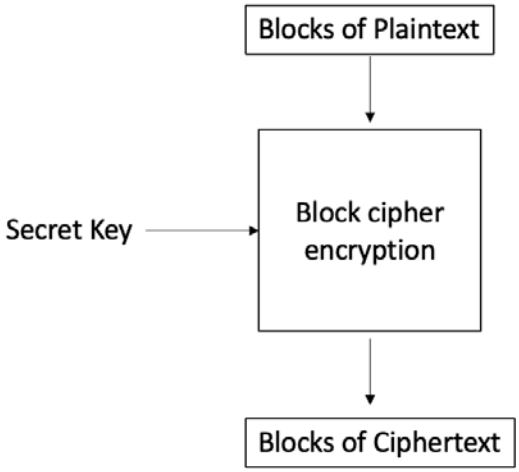


Figure 3.13: Simplified operation of a block cipher

Various modes of operation for block ciphers are used to specify the way in which an encryption function is applied to the plaintext. Some of these modes of block cipher encryption are introduced here.

Block encryption modes

In **block encryption mode**, the plaintext is divided into blocks of fixed length depending on the type of cipher used. Then the encryption function is applied to each block. The most common block encryption modes are ECB, CBC, and CTR.

Electronic code book (**ECB**) is a basic mode of operation in which the encrypted data is produced as a result of applying the encryption algorithm to each block of plaintext, one by one. This is the most straightforward mode, but it should not be used in practice as it is insecure and can reveal information:

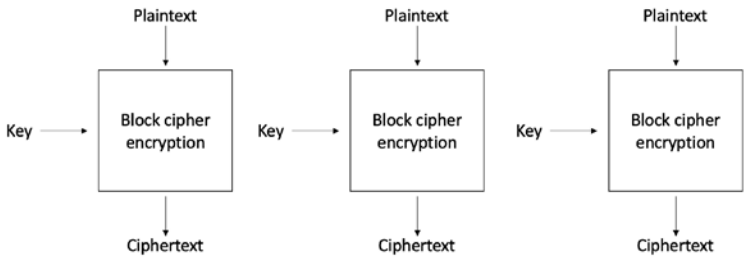


Figure 3.14: Electronic codebook mode for block ciphers

The preceding diagram shows that we have plaintext **P** provided as an input to the block cipher encryption function along with a key, and ciphertext **C** is produced as output.

In **cipher block chaining** (**CBC**) mode, each block of plaintext is XORed with the previously encrypted block. CBC mode uses the IV to encrypt the first block. It is recommended that the IV be randomly chosen:

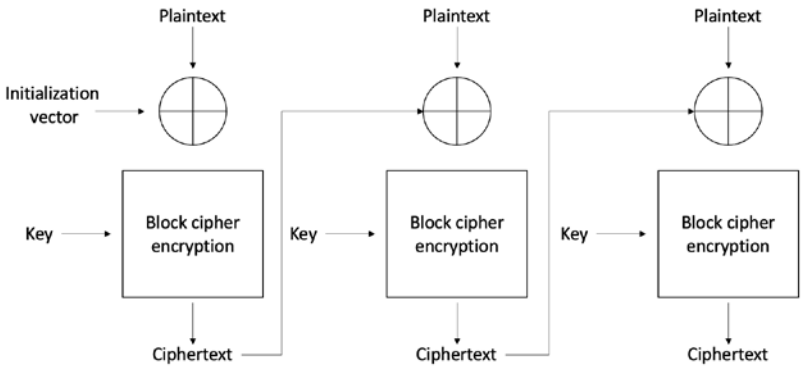


Figure 3.15: Cipher block chaining mode

The **counter (CTR) mode** effectively uses a block cipher as a stream cipher. In this case, a unique nonce is supplied that is concatenated with the counter value to produce a **keystream**:

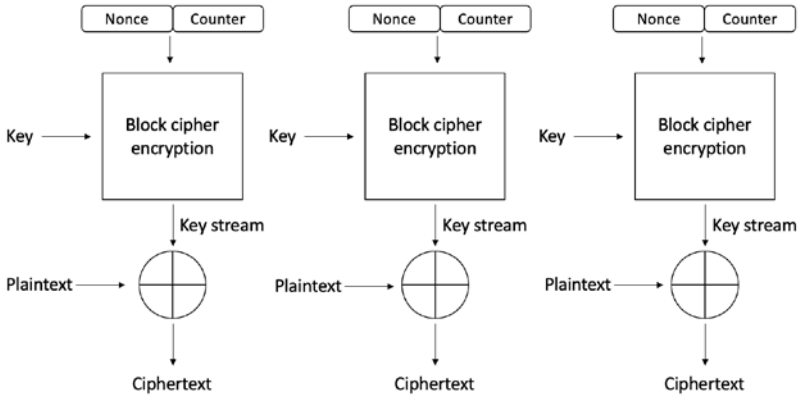


Figure 3.16: Counter mode

CTR mode, as shown in the preceding diagram, works by utilizing a **nonce** (**N**) and a **counter** (**C**) that feed into the **block cipher encryption** function. The block cipher encryption function takes the **secret key** (**KEY**) as input and produces a keystream (a stream of pseudorandom or random characters), which, when XORed with the **plaintext** (**P**), produces the **ciphertext** (**C**).

So far, we have discussed modes that are used to produce ciphertexts (encrypted data). However, there are other modes that can be used for different purposes as listed below:

- In **keystream generation mode**, the encryption function generates a keystream that is used in stream ciphers. Specifically, the keystream is usually XORed with the plaintext stream to produce encrypted text.
- In **message authentication mode**, a MAC is produced from an encryption function. A MAC is a cryptographic checksum that provides an *integrity service*. The most common method to generate a MAC using block ciphers is CBC-MAC, where a part of the last block of the chain is used as a MAC. In other words, block ciphers are used in the **cipher block chaining mode** (**CBC mode**) to generate a MAC. A MAC can be used to check if a message has been modified by an unauthorized entity. This can be achieved by encrypting the message with a key using the MAC function. The resulting message and the MAC of the message, once received by the receiver, are checked by encrypting the message received, again with the key, and comparing it with the MAC received from the sender. If they both match, then it means that the message has not been modified by some unauthorized entity, thus an integrity service is provided. If they don't match, then it means that the message has been altered by some unauthorized entity during transmission.
- Any block cipher, for example, AES in CBC mode, can be used to generate a MAC. The MAC of the message is, in fact, the output of the last round of the CBC operation. The length of the MAC output is the same as the block length of the block cipher used to generate the MAC. It should also be noted that MACs work like **digital signatures**. However, they cannot provide a non-repudiation service due to their symmetric nature.
- Hash functions are primarily used to compress a message to a fixed-length digest. In **cryptographic hash mode**, block ciphers are used as a compression function to produce a hash of plaintext.

There are also other modes, such as **cipher feedback** (**CFB**) mode, **Galois counter** (**GCM**) mode, and **output feedback** (**OFB**) mode, which are also used in various scenarios. Discussion of all these different modes is beyond the scope of this book. Interested readers may refer to any standard textbook on cryptography for further details.

We now introduce some concepts that are relevant to cryptography and are extensively used in many applications.

Advanced Encryption Standard

In this section, we will introduce the design and mechanism of a currently market-dominant block cipher known as AES.

Before discussing AES, let's review some history of DES that led to the development of the new AES standard.

Data Encryption Standard

DES was introduced by **NIST** as a standard algorithm for encryption, and it was in widespread use during the 1980s and 1990s. However, it did not prove to be very resistant to brute-force attacks, due to advances in technology and cryptography research. In July 1998, for example, the **Electronic Frontier Foundation** (**EFF**) broke DES using a special-purpose machine called the EFF DES cracker (or **Deep Crack**).

DES uses a key of only 56 bits, which raised some concerns. This problem was addressed with the introduction of **Triple DES** (**3DES**), which proposed applying a DES cipher three times to each block, thus making the key size 168 bits. This technique makes brute-force attacks almost impossible. However, other limitations, such as slow performance and a small 64-bit block size, were not desirable.

How AES works

In 2001, after an open competition, an encryption algorithm named Rijndael invented by cryptographers Joan Daemen and Vincent Rijmen was standardized as **AES** with minor modifications by NIST. So far, no attack has been found against AES that is more effective than the brute-force method. The original version of Rijndael permits different key and block sizes of 128 bits, 192 bits, and 256 bits. In the AES standard, however, only a 128-bit block size is allowed. However, key sizes of 128 bits, 192 bits, and 256 bits are permissible.

During AES algorithm processing, a 4×4 array of bytes known as the **state** is modified using multiple rounds. Full encryption requires 10 to 14 rounds, depending on the size of the key. The following table shows the key sizes and the required number of rounds:

Key size	Number of rounds required
128-bit	10 rounds
192-bit	12 rounds
256-bit	14 rounds

Once the state is initialized with the input to the cipher, the following four operations are performed sequentially step by step to encrypt the input:

a. **AddRoundKey**: In this step, the state array is XORed with a **subkey**, which is derived from the master key.
b. **SubBytes**: This is the substitution step where a lookup table (S-box) is used to replace all bytes of the state array.
c. **ShiftRows**: This step is used to shift each row to the left, except for the first one, in the state array in a cyclic and incremental manner.
d. **MixColumns**: Finally, all bytes are mixed in a linear fashion (linear transformation), column-wise.

This is one round of AES. In the final round (either the 10th, 12th, or 14th round, depending on the key size), stage 4 is replaced with **AddRoundKey** to ensure that the first three steps cannot be simply reversed, as shown in the following diagram:

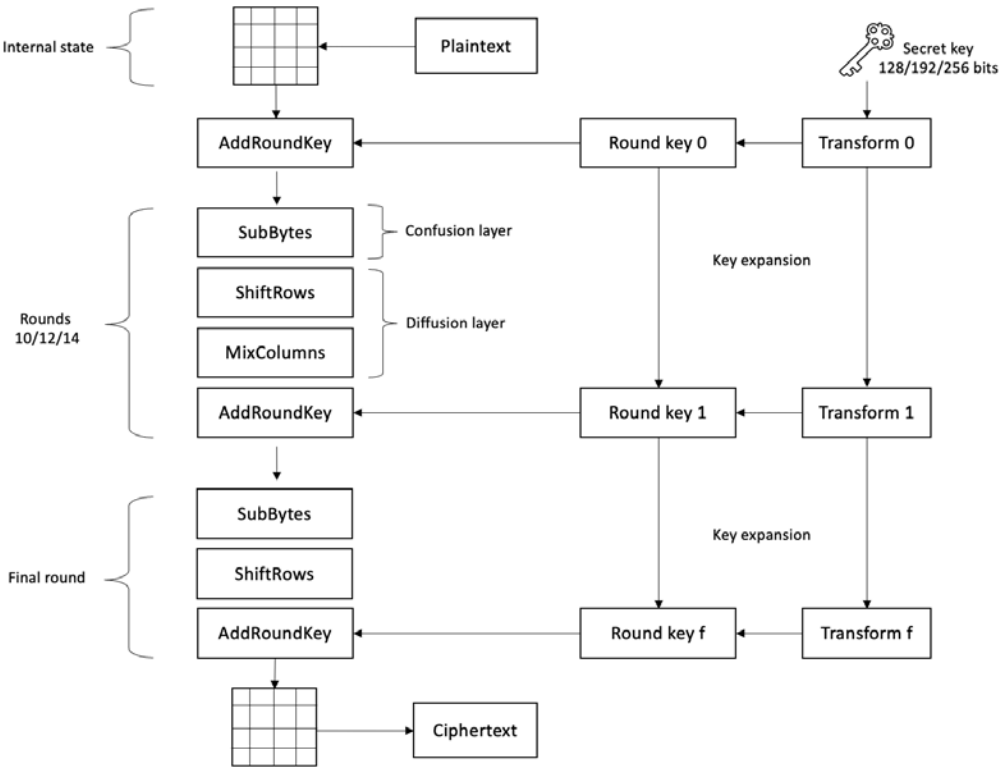


Figure 3.17: AES block diagram, showing the first round of AES encryption. In the final round, the mixing step is not performed

Various cryptocurrency wallets use AES encryption to encrypt locally stored data. Bitcoin wallets use AES-256 in CBC mode to encrypt the private keys. In Ethereum wallets, AES-128-CTR is used; that is, AES 128-bit in counter mode is used to encrypt the private key. Peers in Ethereum also use AES in counter mode (AES CTR) to encrypt their **P2P** communication.

Encrypting and decrypting using AES

We can use the OpenSSL command-line tool to perform encryption and decryption operations. An example is given here.

First, we create a plaintext file to be encrypted:

```
$ echo Datatoencrypt > message.txt
```

Now the file is created, we can run the OpenSSL tool with appropriate parameters to encrypt the file `message.txt` using 256-bit AES in CBC mode:

```
$ openssl enc -aes-256-cbc -in message.txt -out message.bin
```

It will prompt for the password:

```
enter aes-256-cbc encryption password:
Verifying - enter aes-256-cbc encryption password:
```

Once the operation completes, it will produce a `message.bin` file containing the encrypted data from the `message.txt` file. We can view this file, which shows the encrypted contents of the `message.txt` file:

```
$ cat message.bin
```

This shows the encrypted output below:

```
Salted__?W?~?;??G+??"f??%
```

Note that `message.bin` is a binary file. Sometimes, it is desirable to encode this binary file in a text format for compatibility/interoperability reasons. A common text encoding format is `base64`. The following commands can be used to create a base64-encoded message:

```
$ openssl enc -base64 -in message.bin -out message.b64
```

Enter the command below to view the file:

```
$ cat message.b64
```

This shows the base64-encoded output below:

```
U2FsdGVkX1/tEFeZfszXiB47pOt/RyuN/CJm/x/KBBw=
```

To decrypt an AES-encrypted file, the following commands can be used. An example of `message.bin` from a previous example is used:

```
$ openssl enc -d -aes-256-cbc -in message.bin -out message.dec
```

It will ask for the password:

```
enter aes-256-cbc decryption password:
```

Once entered, execute the following command:

```
$ cat message.dec
```

This shows the output below, the original plaintext:

```
Datatoencrypt
```

Readers may have noticed that no IV has been provided, even though it's required in all block encryption modes of operation except ECB. The reason for this is that OpenSSL automatically derives the IV from the given password. Users can specify the IV using the `-iv` switch as shown below:

```
-iv val
```

Here, `val` is IV in hex. In order to decode from base64, the following commands are used. Follow the `message.b64` file from the previous example:

```
$ openssl enc -d -base64 -in message.b64 -out message.ptx
$ cat message.ptx
```

This shows the output below:

```
Salted__?W?~??;??G+??"f??%
```

There are many types of ciphers that are supported in OpenSSL. You can explore these options based on the examples provided thus far. Further information and help can be retrieved with the following command:

```
$ openssl help
```

We will also use OpenSSL in the next chapter to demonstrate various public key cryptographic primitives and protocols.

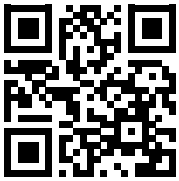
Summary

This chapter introduced symmetric key cryptography. We started with basic mathematical definitions and cryptographic primitives. After this, we introduced the concepts of stream and block ciphers along with the working modes of block ciphers. Moreover, we introduced some practical exercises using OpenSSL to complement the theoretical concepts covered.

In the next chapter, we will introduce public key cryptography, which is used extensively in blockchain technology and has very interesting properties.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

4

Asymmetric Cryptography

In this chapter, you will be introduced to the concepts and practical aspects of **public key cryptography**, also called **asymmetric cryptography** or **asymmetric key cryptography**. We will start with the theoretical foundations of public key cryptography and will gradually build on the concepts, with relevant practical exercises with OpenSSL. Please refer to the previous chapter for an OpenSSL installation guide, if you have not already. After this, we will introduce some new and advanced cryptography constructs.

Along the way, we will discuss the following topics:

- Foundational mathematics
- Asymmetric cryptography
- Introducing RSA
- Introducing **elliptic curve cryptography** (**ECC**)
- Digital signatures
- Cryptographic constructs and blockchain technology

Before discussing cryptography further, some mathematical terms and concepts need to be explained in order to build a foundation for fully understanding the material provided later in this chapter.

Foundational mathematics

As the subject of cryptography is based on mathematics, this section will introduce some basic concepts that will help you understand the concepts presented later. An explanation with proofs and relevant background for all of these terms would require somewhat complex mathematics, which is beyond the scope of this book. More details on these topics can be found in any standard number theory, algebra, or cryptography book:

- **Modular arithmetic**: Also known as clock arithmetic, numbers in modular arithmetic wrap around when they reach a certain fixed number. This fixed number is a positive number called **modulus** (sometimes abbreviated to **mod**), and all operations are performed concerning this fixed number. In other words, this type of arithmetic deals with the remainders after the division operation. For example, 50 mod 11 is 6 because 50/11 leaves a remainder of 6.

- **Sets**: These are collections of distinct objects, for example, X = {1, 2, 3, 4, 5}.
- **Groups**: A group is a commutative set with an operation that combines two elements of the set. The group operation is closed and associated with a defined identity element. Additionally, each element in the set has an inverse. **Closure** (closed) means that if, for example, elements **A** and **B** are in the set, then the resultant element after performing an operation on the elements is also in the set. **Associative** means that the grouping of elements does not affect the result of the operation. Four group axioms must be satisfied for a set to qualify as a group. These group axioms include closure, associativity, an identity element, and an inverse element:
    - **Cyclic group**: A type of group that can be generated by a single element called the group generator.
- **Fields**: A field is a set in which all its elements form an additive or multiplicative group. It satisfies specific axioms for addition and multiplication. For all group operations, the **distributive law** is also applied. The law dictates that the same sum or product will be produced, even if any of the terms or factors are reordered:
    - A **finite field** is one with a finite set of elements. Also known as **Galois fields**, these structures are of particular importance in cryptography as they can be used to produce accurate and error-free results of arithmetic operations.
    - A **prime field** is a finite one with a prime number of elements. It has specific rules for addition and multiplication, and each non-zero element in the field has an inverse. Addition and multiplication operations are performed mod *p*, that is, modulo of a prime number.
    - **Order**: The number of elements in a field. It is also known as the cardinality of the field.

This completes a basic introduction to some mathematical concepts involved in cryptography. In the next section, you will be introduced to cryptography concepts.

Asymmetric cryptography

Asymmetric cryptography refers to a type of cryptography where the key that is used to encrypt the data is different from the key that is used to decrypt the data. These keys are called public and private keys, respectively, which is why asymmetric cryptography is also known as **public key cryptography**. It uses both public and private keys to encrypt and decrypt data, respectively. Various asymmetric cryptography schemes are in use, including **RSA** and **ElGamal** encryption.

Public and private keys

A **private key**, as the name suggests, is a randomly generated number that is kept secret and held privately by its users. Private keys need to be protected and no unauthorized access should be granted to those keys; otherwise, the whole scheme of public key cryptography is jeopardized, as this is the key that is used to decrypt messages. Private keys can be of various lengths, depending on the type and class of algorithms used. For example, in RSA, typically, a key of 1,024 bits or 2,048 bits is used. The 1,024-bit key size is no longer considered secure, and at least a 2,048-bit key size is recommended.

A **public key** is freely available and published by the private key owner. Anyone who would then like to send the publisher of the public key an encrypted message can do so, by encrypting the message using the published public key and sending it to the holder of the private key. No one else can decrypt the message because the corresponding private key is held securely by the intended recipient. Once the public key-encrypted message is received, the recipient can decrypt the message using the private key. There are a few concerns, however, regarding public keys. These include authenticity and identification of the publisher of the public keys.

A generic depiction of public - key cryptography is shown in the following diagram:

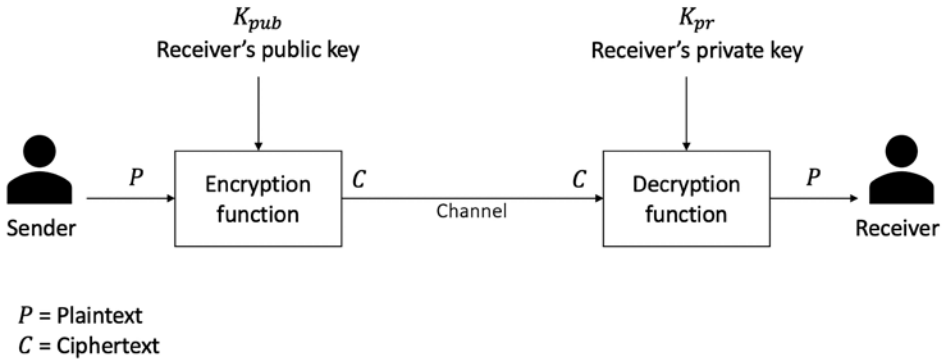


Figure 4.1: Encryption/decryption using public/private keys

The preceding diagram illustrates how a sender encrypts data **P** using the recipient's public key and encryption function, and produces an output of encrypted data **C**, which is then transmitted over the network to the receiver. Once it reaches the receiver, it can be decrypted using the receiver's private key by feeding the **C**-encrypted data into the decryption function, which will output plaintext **P**. This way, the private key remains on the receiver's side, and there is no need to share keys to perform encryption and decryption, which is the case with symmetric encryption.

The following diagram shows how the receiver uses public key cryptography to verify the integrity of the received message. In this model, the sender signs the data using their private key and transmits the message across to the receiver. Once the message is received, it is verified for integrity by the sender's public key.

It's worth noting that there is no encryption being performed in this model. It is simply presented here to help you thoroughly understand the material on message authentication and validation that will be provided later in this chapter:

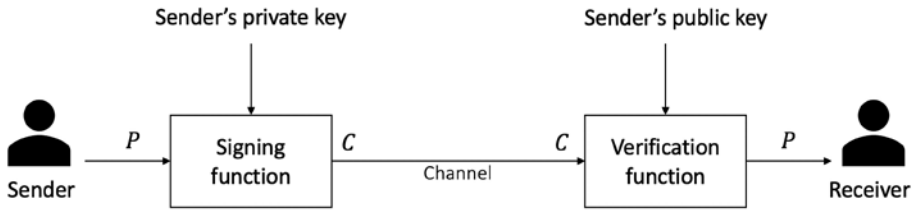


Figure 4.2: Model of a public-key cryptography digital signature scheme

The preceding diagram shows that the sender digitally signs the plaintext **P** with their private key using signing function **S**, and produces data **C**, which is sent to the receiver, who verifies **C** using the sender's public key and function **V** to ensure the message has indeed come from the sender.

Security mechanisms offered by public key cryptosystems include key establishment, digital signatures, identification, encryption, and decryption.

Key establishment mechanisms are concerned with the design of protocols that allow the setting up of keys over an insecure channel. Non-repudiation (defined in *Chapter 3*, *Symmetric Cryptography*, as the assurance that an action, once taken by someone, cannot be denied later) services, a very desirable option in many scenarios, can be provided using **digital signatures**. Sometimes, it is important not only to authenticate a user but also to identify the entity involved in a transaction. This can also be achieved by a combination of digital signatures and **challenge-response protocols**. Finally, the encryption mechanism to provide confidentiality can also be obtained using public key cryptosystems, such as RSA, ECC, and ElGamal.

Asymmetric cryptography algorithms

Public key algorithms are slower in terms of computation than symmetric key algorithms. Therefore, they are not commonly used in the encryption of large files or the actual data that requires encryption. They are usually used to exchange keys for symmetric algorithms. Once the keys are established securely, symmetric key algorithms can be used to encrypt the data.

Public key cryptography algorithms are based on various underlying mathematical functions. The three main **categories of asymmetric algorithms** are described here.

Integer factorization

Integer factorization schemes are based on the hard problem that large integers are extremely hard to factor. RSA is a prime example of this type of algorithm.

Discrete logarithm

A **discrete logarithm scheme** is based on a problem in modular arithmetic. It is easy to calculate the result of a modulo function, but it is computationally impractical to find the exponent of the generator. In other words, it is extremely difficult to find the input from the result (output). This is called a one-way function.

For example, consider the following equation:

$$3^2 \ mod \ 10 = 9$$

Now, given 9, the result of the preceding equation, finding 2, which is the exponent of the generator (3) in the equation, is extremely hard to determine. This difficult problem is commonly used in **Diffie-Hellman** key exchange and digital signature algorithms.

Elliptic curves

The **elliptic curve algorithm** is based on the discrete logarithm problem discussed previously, but in the context of elliptic curves. An **elliptic curve** is an algebraic cubic curve over a field, which can be defined by an equation, as shown here. The curve is non-singular, which means that it has no cusps or self-intersections. It has two variables *a* and *b*, as well as a point of infinity:

$$y^2 = x^3 + ax + b$$

Here, *a* and *b* are integers whose values are elements of the field on which the elliptic curve is defined. Elliptic curves can be defined over reals, rational numbers, complex numbers, or finite fields. For cryptographic purposes, an elliptic curve over prime finite fields is used instead of real numbers. Additionally, the prime should be greater than 3. Different curves can be generated by varying the values of *a* and/or *b*.

The most prominently used cryptosystems based on elliptic curves are the **Elliptic Curve Digital Signatures Algorithm** (**ECDSA**) and the **Elliptic Curve Diffie-Hellman** (**ECDH**) key exchange.

Integrated encryption scheme

An **integrated encryption scheme** (**IES**) is a hybrid encryption mechanism that combines public-key schemes with symmetric key schemes to achieve convenience and efficiency. Public-key schemes are convenient as there is no need to follow the cumbersome process of secret key sharing. On the other hand, symmetric-key schemes are more efficient than public-key schemes for data encryption. So, hybrid encryption schemes combine the best of both worlds to achieve efficiency and convenience.

Hybrid schemes are composed of two mechanisms, firstly a key encapsulation mechanism, which is a public key cryptosystem, and secondly a data encapsulation mechanism, which is a symmetric key encryption mechanism. Protocols such as TLS and SSH utilize a hybrid encryption scheme. An IES has two variants: a **discrete logarithm integrated encryption scheme** (**DLIES**) and an **elliptic curve integrated encryption scheme** (**ECIES**).

In the following sections, we will introduce two examples of asymmetric key cryptography: **RSA** and **ECC**. RSA is the first implementation of public key cryptography, whereas ECC is used extensively in blockchain technology.

Introducing RSA

RSA was invented in 1977 by Ron **R**ivest, Adi **S**hamir, and Leonard **A**delman, hence the name RSA. This type of public key cryptography is based on the integer factorization problem, where the multiplication of two large prime numbers is easy, but it is difficult to factor the product (the result of the multiplication) back into the two original numbers.

The crux of the work involved with the RSA algorithm happens during the key generation process. An RSA key pair is generated by performing the following steps:

1. Modulus generation:
    - Select *p* and *q* , which are very large prime numbers.

- Multiply *p* and *q*, *n=p.q* to generate modulus *n*.

2. Generate the co-prime:
   - Assume a number called *e*.
   - *e* should satisfy a certain condition; that is, it should be greater than *1* and less than *(p-1) (q-1)*. In other words, *e* must be a number such that no number other than *1* can be divided into *e* and *(p-1) (q-1)*. This is called a **co-prime**, that is, *e* is the co-prime of *(p-1)(q-1)*.
3. Generate the public key:
   - The modulus generated in *step 1* and co-prime *e* generated in *step 2* is a pair that is a public key. This part is the public part that can be shared with anyone; however, *p* and *q* need to be kept secret.
4. Generate the private key:
   - The private key is called *d* here and is calculated from *p*, *q*, and *e*. The private key is basically the inverse of *e* modulo *(p-1)(q-1)*. As an equation, it is as follows:

$$ed = 1 \ mod (p-1)(q-1)$$

Usually, an extended Euclidean algorithm is used to take *p*, *q*, and *e* and calculate *d*. The key idea in this scheme is that anyone who knows *p* and *q* can easily calculate private key *d* by applying the extended Euclidean algorithm. However, someone who does not know the value of *p* and *q* cannot generate *d*. This also implies that *p* and *q* should be large enough for the modulus *n* to become extremely difficult (computationally impractical) to factor.



The original RSA paper is available here: `http://web.mit.edu/6.857/OldStuff/Fall03/ref/rivest78method.pdf`.

Rivest, R.L., Shamir, A., and Adleman, L., 1978. A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM, 21(2), pp.120–126.

Now, let's see how encryption and decryption operations are performed using RSA. RSA uses the following equation to produce ciphertext:

$$C = P^e \ mod \ n$$

This means that plaintext *P* is raised to the power of *e* and then reduced to modulo *n*. Decryption in RSA is provided in the following equation:

$$P = C^d \ mod \ n$$

This means that the receiver who has a public key pair (*n*, *e*) can decipher the data by raising *C* to the value of the private key *d*, and then reducing to modulo *n*.

Encrypting and decrypting with RSA

The following example illustrates how RSA public and private key pairs can be generated using the OpenSSL command line.

First, the RSA private key can be generated using OpenSSL as follows:

```
$ openssl genpkey -algorithm RSA -out privatekey.pem -pkeyopt rsa_keygen_
bits:1024
..............................++++++
...................++++++
```

After executing the command, a file named `privatekey.pem` is produced, which contains the generated private key, as follows:

```
$ cat privatekey.pem
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBAKJOFBzPy2vOd6em Bk/
UGrzDy7TvgDYnYxBfiEJId/r+EyMt/F14k2fDTOVwxXaXTxiQgD+BKuiey/69
9itnrqW/xy/pocDMvobj8QCngEntOdNoVSaN+t0f9nRM3iVM94mz3/C/v4vXvoac
PyPkr/0jhIV0woCurXGTghgqIbHRAgMBAAECgYEAlB3s/N4lJh0l1TkOSYunWtzT
6isnNkR7g1WrY9H+rG9xx4kP5b1DyE3SvxBLJA6xgBle8JVQMzm3sKJrJPFZzzT5
NNNnugCxairxcF1mPzJAP3aqpcSjxKpTv4qgqYevwgW1A0R3xKQZzBKU+bTO2hXV
D1oHxu75mDY3xCwqSAECQQDUYV04wNSEjEy9tYJ0zaryDAcvd/VG2/U/6qiQGajB
eSpSqoEESigbusKku+wVtRYgWWEomL/X58t+K01eMMZZAkEAw6PUR9YLebsm/
Sji iOShV4AKuFdi7t7DYWE5Ulb1uqP/i28zN/ytt4BXKIs/KcFykQGeAC6LDHZyycyc
ntDIOQJAVqrE1/wYvV5jkqcXbYLgV5YA+KYDOb9Y/ZRM5UETVKCVXNanf5CjfW1h
MMhfNxyGwvy2YVK0Nu8oY3xYPi+5QQJAUGcmORe4w6Cs12JUJ5p+zG0s+rG/URhw
B7djTXm7p6b6wR1EWYAZDM9MArenj8uXAA1AGCcIsmiDqHfU7lgz0QJAe9mOdNGW
7qRppgmOE5nuEbxkDSQI7OqHYbOLuwfCjHzJBrSgqyi6pj9/9CbXJrZPgNDwdLEb
GgpDKtZs9gLv3A==
-----END PRIVATE KEY-----
```



It is OK to reveal the private key here because we are simply using it for our example. However, in production systems, safeguarding the private key is of utmost importance. Make sure that the private key is always kept secret. Also, remember that the preceding key is just being used here as an example; do not reuse it.

As the private key is mathematically linked to the public key, it is also possible to generate or derive the public key from the private key. Using the example of the preceding private key, the public key can be generated as follows:

```
$ openssl rsa -pubout -in privatekey.pem -out publickey.pem
 writing RSA key
```

The public key can be viewed using a file reader or any text viewer:

```
$ cat publickey.pem
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
74A2J2MQX4hCSHf6/hMjLfxdeJNnw0zlcMV2l08YkIA/gSronsv+vfYrZ66lv8cv 6aHAzL6G4/
EAp4BJ7TnTaFUmjfrdH/Z0TN4lTPeJs9/wv7+L176GnD8j5K/9I4SF dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
```

In order to see more details of the various components, such as the modulus, prime numbers that are used in the encryption process, or exponents and coefficients of the generated private key, the following command can be used (only part of the output is shown here as the actual output is very long):

```
$ openssl rsa -text -in privatekey.pem
Private-Key: (1024 bit)
modulus:
```

Similarly, the public key can be explored using the following commands. Public and private keys are base64-encoded:

```
$ openssl pkey -in publickey.pem -pubin -text
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCiThQcz8trznenpgZP1Bq8w8u0
74A2J2MQX4hCSHf6/hMjLfxdeJNnw0zlcMV2l08YkIA/gSronsv+vfYrZ66lv8cv 6aHAzL6G4/
EAp4BJ7TnTaFUmjfrdH/Z0TN4lTPeJs9/wv7+L176GnD8j5K/9I4SF dMKArq1xk4IYKiGx0QIDAQAB
-----END PUBLIC KEY-----
Public-Key: (1024 bit) Modulus:
00:a2:4e:14:1c:cf:cb:6b:ce:77:a7:a6:06:4f:d4:
1a:bc:c3:cb:b4:ef:80:36:27:63:10:5f:88:42:48:
77:fa:fe:13:23:2d:fc:5d:78:93:67:c3:4c:e5:70:
c5:76:97:4f:18:90:80:3f:81:2a:e8:9e:cb:fe:bd:
f6:2b:67:ae:a5:bf:c7:2f:e9:a1:c0:cc:be:86:e3:
f1:00:a7:80:49:ed:39:d3:68:55:26:8d:fa:dd:1f:
f6:74:4c:de:25:4c:f7:89:b3:df:f0:bf:bf:8b:d7:
be:86:9c:3f:23:e4:af:fd:23:84:85:74:c2:80:ae:
ad:71:93:82:18:2a:21:b1:d1
Exponent: 65537 (0x10001)
```

Now, the public key can be shared openly, and anyone who wants to send you a message can use the public key to encrypt the message and send it to you.

Taking the private key we generated in the previous example, the command to encrypt a text file `message.txt` can be constructed, as shown here:

```
$ echo datatoencrypt > message.txt
```

```
$ openssl rsautl -encrypt -inkey publickey.pem -pubin -in message.txt -out
message.rsa
```

This will produce a file named `message.rsa`, which is in binary format. If you display `message.rsa`, it will show some scrambled data, as shown below:

```
$ cat message.rsa
s???c?ngJ[Lt!?LgC\[f?L?1?^q?r?
                              a??????Da?=??m??_?P?Y???KE
```

In order to decrypt the RSA-encrypted file, the following command will employ the corresponding private key to decrypt the file:

```
$ openssl rsautl -decrypt -inkey privatekey.pem -in message.rsa -out message.
dec
```

Now, if the file is read using `cat`, decrypted plaintext can be seen, as shown here:

```
$ cat message.dec
datatoencrypt
```

With this example, we complete our introduction to RSA. Next, let's explore elliptic curve cryptography.

Introducing ECC

ECC is based on the discrete logarithm problem, founded upon elliptic curves over finite fields (Galois fields). The main benefit of ECC over other types of public key algorithms is that it requires a smaller key size, while providing the same level of security as, for example, RSA. Two notable schemes that originate from ECC are ECDH for key exchange and ECDSA for digital signatures.

ECC can also be used for encryption, but it is not usually used for this purpose in practice. Instead, key exchange and digital signatures are used more commonly. As ECC needs less space to operate, it is becoming very popular on embedded platforms and in systems where storage resources are limited. By comparison, the same level of security can be achieved with ECC when only using 256-bit operands as compared to 3,072 bits in RSA.

Mathematics behind ECC

To understand ECC, a basic introduction to the underlying mathematics is necessary. An elliptic curve is basically a type of polynomial equation known as the **Weierstrass equation**, which generates a curve over a finite field. The most commonly used field is where all arithmetic operations are performed modulo a prime number *p*. Elliptic curve groups consist of points on the curve over a finite field.

An elliptic curve is defined in the following equation:

$$y^2 = x^3 + Ax + B \ mod \ P$$

Here, *A* and *B* belong to a finite field *Zp* or *Fp* (a prime finite field), along with a special value called the point of infinity. The point of infinity, ∞, is used to provide identity operations for points on the curve.

Furthermore, a condition also needs to be met that ensures that the equation mentioned earlier has no repeated roots. This means that the curve is non-singular.

The condition is described in the following equation, which is a standard requirement that needs to be met. More precisely, this ensures that the curve is non-singular:

$$4a^3 + 27b^2 \neq 0 \ mod \ p$$

To construct the discrete logarithm problem based on elliptic curves, a large enough cyclic group is required. First, the group elements are identified as a set of points that satisfy the previous equation. After this, group operations need to be defined on these points.

Basic group operations on elliptic curves are point addition and point doubling. **Point addition** is a process where two different points are added, and **point doubling** means that the same point is added to itself.

Point addition

Point addition is shown in the following diagram. This is a geometric representation of point addition on elliptic curves. In this method, a diagonal line is drawn through the curve that intersects the curve at two points shown below *P* and *Q*, which yields a third point between the curve and the line.

This point is mirrored as *P*+*Q*, which represents the result of the addition as *R*. This is shown as *P*+*Q* in the following diagram:

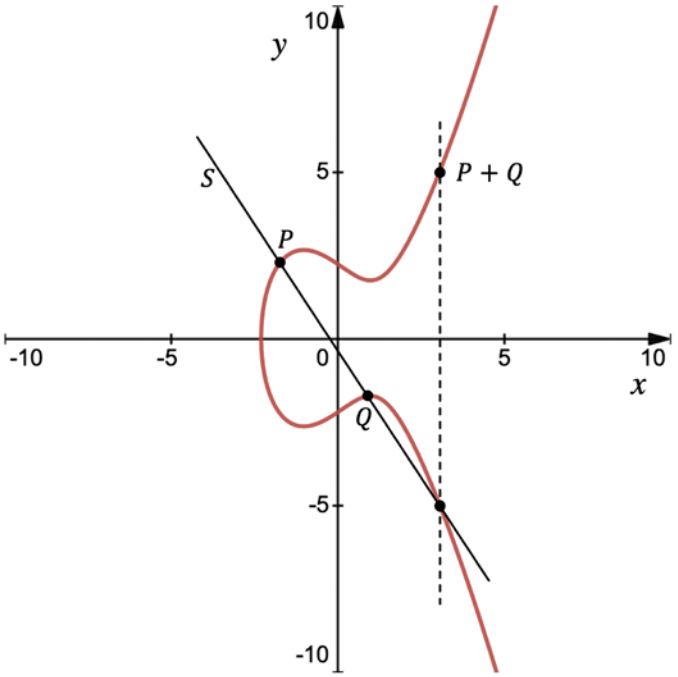


Figure 4.3: Point addition over R

The group operation denoted by the + sign for addition yields the following equation:

$$P + Q = R$$

More precisely, this means that coordinates are added, as shown in the following equation:

$$(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$$

So, if $P \neq Q$, then it means point addition. To add these points, see as follows:

$$S = \frac{(y_1 - y_2)}{(x_1 - x_2)} mod\ p$$

S depicts the line going through P and Q.

$$x_3 = s^2 - x_1 - x_2\ mod\ p$$

$$y_3 = s(x_1 - x_3) - y_1\ mod\ p$$

So far, we've learned how the point addition group operation works and seen the analytical expressions of the addition group operation.

Next, let's look at a complete example of point addition. This example shows the addition and solution for the equation over a finite field F_{23}, which contains 23 elements.

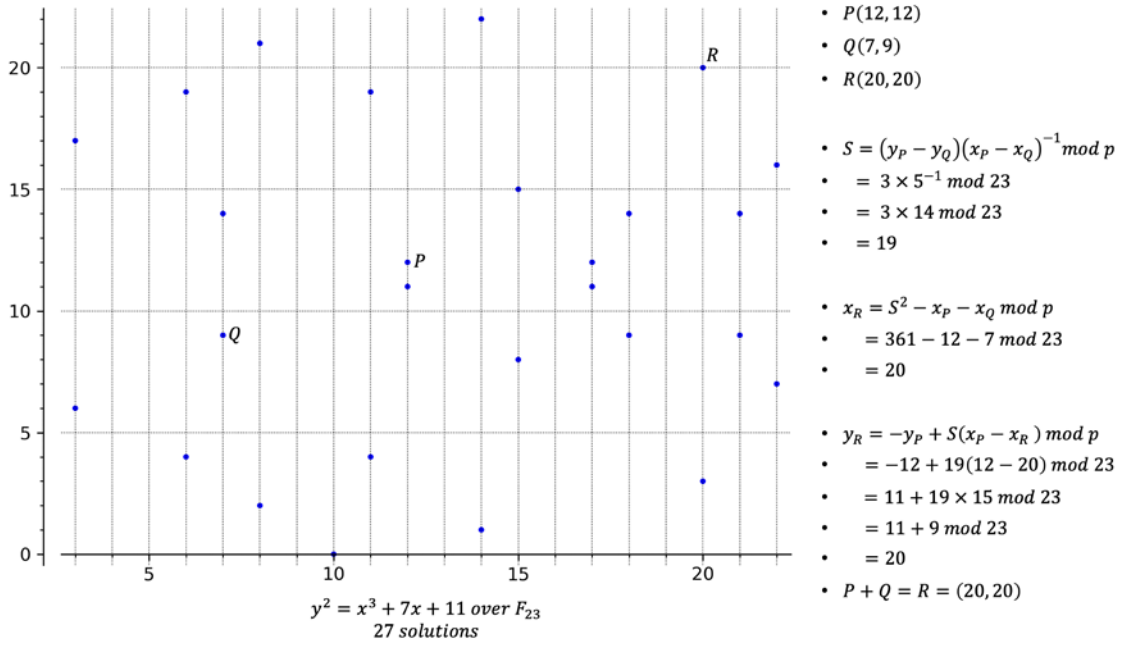


Figure 4.4: Example of point addition

In the preceding example, the graph on the left-hand side shows the points that satisfy this equation:

$$y^2 = x^3 + 7x + 11$$

There are 27 solutions to the equation shown earlier over the finite field F_{23}. P and Q are chosen to be added to produce point R. Calculations are shown on the right-hand side, which calculates the third point R. Note that in the preceding graph, in the calculation shown on the right-hand side, l is used to depict the line going through P and Q.

As an example, to show how the equation is satisfied by the points shown in the graph, a point (x, y) is picked up where $x = 3$ and $y = 6$. This point can be visualized in the graph shown here. Notice the point at coordinate (3, 6), indicated by an arrow:

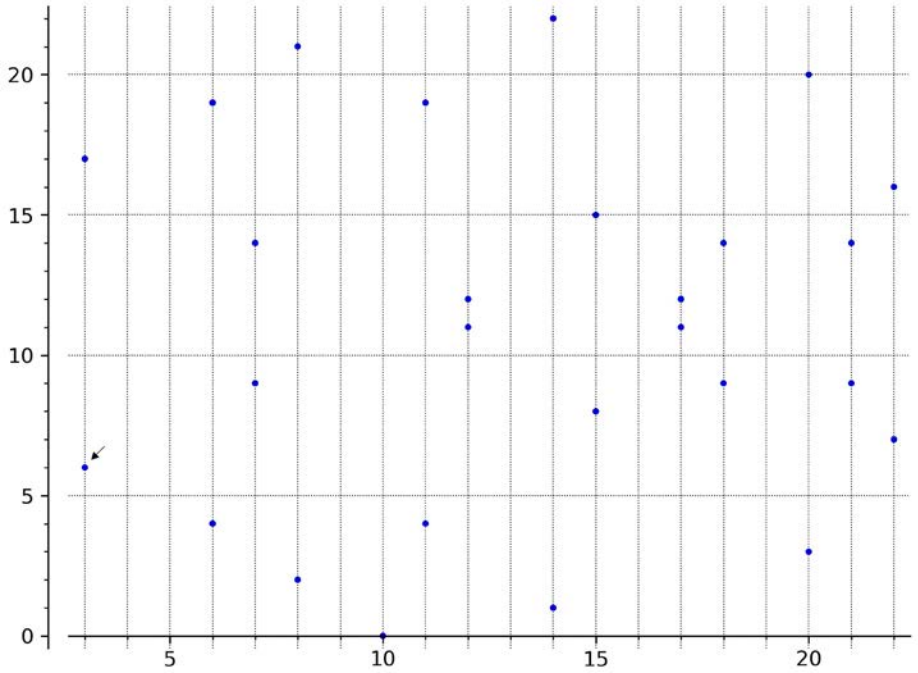


Figure 4.5: Point at (3, 6) shown with an arrow

Using these values shows that the equation is indeed satisfied:

$$y^2 \ mod\ 23 = x^3 + 7x + 11 \ mod\ 23$$

$$6^2 \ mod\ 23 = 3^3 + 7(3) + 11 \ mod\ 23$$

$$36 \ mod\ 23 = 59 \ mod\ 23$$

$$13 = 13$$

The next section introduces the concept of point doubling, which is another operation that can be performed on elliptic curves.

Point doubling

The other group operation on elliptic curves is called **point doubling**. This is a process where P is added to itself. This is the case when P and Q are at the same point, so effectively the operation becomes adding the point to itself and is therefore called point doubling. In this method, a tangent line is drawn through the curve, as shown in the following graph. The second point is obtained, which is at the intersection of the tangent line drawn and the curve. This point is then mirrored to yield the result, which is shown as $2P = P + P$:

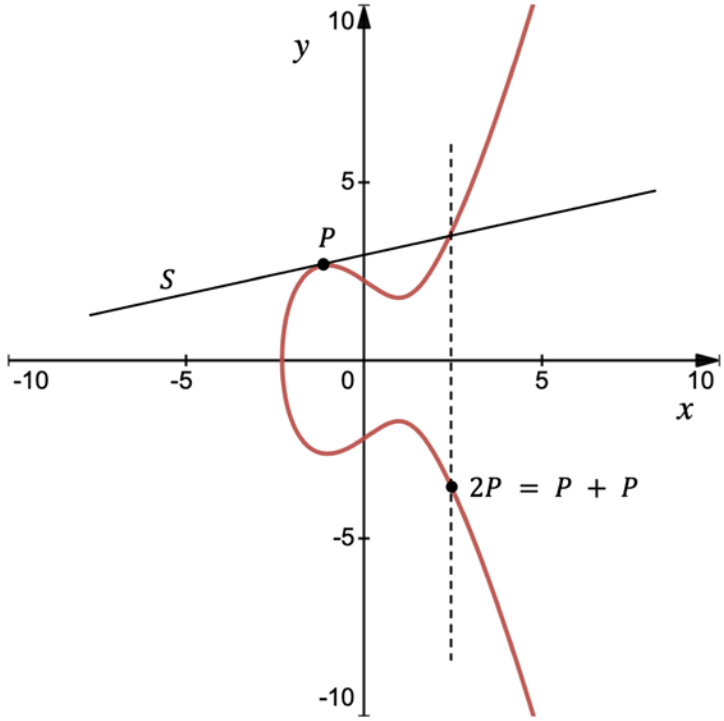


Figure 4.6: Graph representing point doubling

In the case of point doubling, the equation becomes:

$$S = \frac{3x_1^2 + a}{2y_1}$$

$$x_3 = s^2 - x_1 - x_2 \; mod \; p$$

$$y_3 = s(x_1 - x_3) - y_1 \; mod \; p$$

Here, S is the slope of the tangent line going through P, which is the line shown at the top in the preceding graph. In the preceding example, the curve is plotted as a simple example, and no solution to the equation is shown.

The following example shows the solutions and point doubling of elliptic curves over the finite field F_{23}.

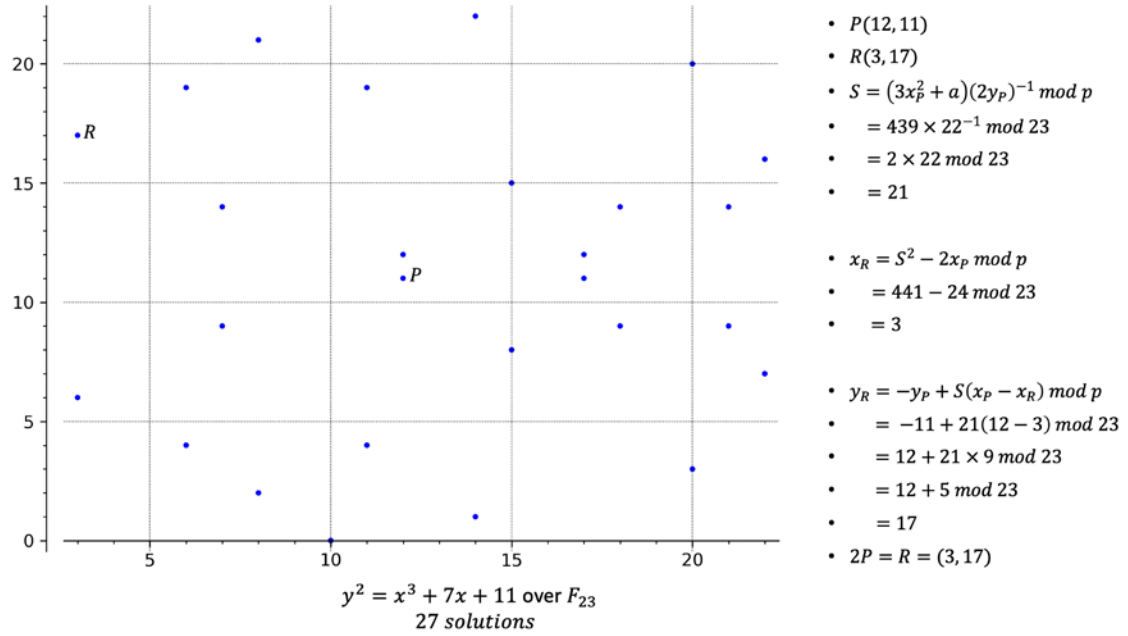


Figure 4.7: Example of point doubling

The graph on the left-hand side shows the points that satisfy the equation:

$$y^2 = x^3 + 7x + 11$$

As shown on the right-hand side of the preceding graph, the calculation finds R, after P is added to itself (point doubling). There is no Q shown here because the same point P is used for doubling.

Using addition and doubling, we can construct another operation, called point multiplication, which we introduce next.

Point multiplication

This operation is also called scalar point multiplication. It is used to multiply points on elliptic curves by a given integer. Let's call this integer d and the point P. We get dP by repeatedly adding P, d times. This operation can be described as follows:

$$P + P + \ldots + P = dP, \text{ where } P \text{ is added } d \text{ times.}$$

Any point on the curve can be added multiple times to itself. The result of this operation is always another point on the curve.

The addition operation is not efficient for large values of d. However, point multiplication operation can be made efficient by utilizing the **double and add algorithm**, which combines point addition and doubling operations to achieve exponential performance gain.

For example, if using addition only, to get 9*P*, we must do *P* + *P* + *P* + *P* + *P* + *P* + *P* + *P* + *P*, which can become infeasible quickly if the number of *P*s increases. To address this, we can use the double-and-add mechanism where we first convert nine into binary, then starting from the most significant bit, for each bit that is 1 (high) perform a double-and-add operation, and for each 0, perform only the double operation. We do not perform any operation on the most significant bit. As 9 is 1001 in binary, we get for each bit, starting from left to right, *P*, 2*P*, 4*P*, and 8*P*+*P*. This process produces 9*P* only with three double operations and one addition operation, instead of nine addition operations.

In this example, point doubling and addition have been used to construct an efficient operation of scalar multiplication. Now consider that *dP* results in producing another point on the curve; let's call that point *T*. We can say that with all these doublings and additions we have computed a point called *T*. We multiplied a point *P* on the curve with a number *d* to compute another point *T*.

Here's the key idea now: even if we know points *P* and *T*, it is computationally infeasible to reconstruct the sequence of all the double-and-add operations that we did to figure out the number *d*. In other words, even if someone knows *P* and *T*, it is almost impossible for them to find *d*. This means that it is a one-way function, and it is the basis of the **elliptic curve discrete logarithm problem** (**ECDLP**). We describe the discrete logarithm problem in more detail next.

The discrete logarithm problem

The discrete logarithm problem in ECC is based on the idea that, under certain conditions, all points on an elliptic curve form a cyclic group.

On an elliptic curve, the public key is a random multiple of the generator point, whereas the private key is a randomly chosen integer used to generate the multiple. In other words, a private key is a randomly selected integer, whereas the public key is a point on the curve. The discrete logarithm problem is used to find the private key (an integer) where that integer falls within all points on the elliptic curve. The following equation shows this concept more precisely.

Consider an elliptic curve *E*, with two elements *P* and *T*. The discrete logarithmic problem is to find the integer *d*, where *1* <= *d* <= *#E*, such that:

$$P + P + \ldots + P = dP = T$$

Here, *T* is the public key (a point on the curve, (*x*, *y*)), and *d* is the private key. In other words, the public key is a random multiple of the generator *P*, whereas the private key is the integer that is used to generate the multiple. *#E* represents the order of the elliptic curve, which means the number of points that are present in the cyclic group of the elliptic curve. A **cyclic group** is formed by a combination of points on the elliptic curve and the point of infinity.

The initial starting point *P* is a public parameter, and the public key *T* is also published, whereas *d*, the private key, is kept secret. If *d* is not known, it is impractical to calculate it by only having the knowledge of *T* and *P*, thus creating the hard problem on which ECDLP is built.

A key pair is linked with the specific domain parameters of an elliptic curve. Domain parameters include the field size, the field representation, two elements from the fields *a* and *b*, two field elements X_g and Y_g, order *n* of point *G*, which is calculated as $G = (X_g, Y_g)$, and the cofactor $h = \#E(F_q)/n$. A practical example using OpenSSL will be described later in this section.

Various parameters are recommended and standardized so they can be used as curves with ECC. An example of the secp256k1 specification is shown here. The following figure is an excerpt taken from the standard specification from `http://www.secg.org/sec2-v2.pdf`. It is used in Bitcoin.

The elliptic curve domain parameters over F_p, associated with a Koblitz curve secp256k1, are specified by the sextuple $T = (p,a,b,G,n,h)$, where the finite field F_p is defined by:

$$P = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ FFFFFC2F$$

$$= 2256 - 232 - 29 - 28 - 27 - 26 - 24 - 1$$

The curve E: $y^2 = x^3+ax+b$ over F_p is defined by:

$$a = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000$$

$$b = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000007$$

The base point G in compressed form, this is:

$$G = 02\ 79BE667E\ F9DCBBAC\ 55A06295\ CE870B07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798$$

In the uncompressed form is:

$$G = 04\ 79BE667E\ F9DCBBAC\ 55A06295\ CE870B07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798$$
$$483ADA77\ 26A3C465\ 5DA4FBFC\ 0E1108A8\ FD17B448\ A6855419\ 9C47D08F\ FB10D4B8$$

Finally, the order n of G and the cofactor are:

$$n = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ BAAEDCE6\ AF48A03B\ BFD25E8C\ D0364141$$

$$h = 01$$

Here, we see:

- P is the prime p that specifies the size of the finite field.
- a and b are the coefficients of the elliptic curve equation.
- G is the base point that generates the required subgroup, also known as the generator. The base point can be represented in either compressed or uncompressed form. There is no need to store all points on the curve in a practical implementation. The compressed generator works because the points on the curve can be identified using only the x coordinate and the least significant bit of the y coordinate.
- n is the order of the subgroup.
- h is the cofactor of the subgroup.

We can also use the OpenSSL command line to view these parameters of secp256k1. This can be seen here:

```
$ openssl ecparam -param_enc explicit -text -noout -name secp256k1
```

This command will show the output as follows:

```
Field Type: prime-field
```

```
Prime:
  00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
  ff:fc:2f
A:  0
B:  7 (0x7)
Generator (uncompressed):
  04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
  0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
  f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
  0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
  8f:fb:10:d4:b8
Order:
  00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
  ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
  36:41:41
Cofactor: 1 (0x1)
```

This output can be readily compared and verified using the SECP256K1 specification shown earlier.



Note that there are different types of curves, and they must be chosen carefully in order to ensure cryptographic security guarantees. There are also curves that have been standardized by the NIST in the US and published in the FIPS 186 document.

You can find more about safe curves and their selection criteria at `https://safecurves.cr.yp.to`.

As a summary, a quick comparison of RSA and ECC cryptography is shown below.

Feature	**ECC**	**RSA**
Key size	Smaller	Larger
Key generation speed	Faster	Slower
Encryption speed	Faster	slower
Decryption speed	Slower	faster
Quantum-resistant	No	No

In the following section, examples of using OpenSSL will be shown to help you understand the practical aspects.

Generating keys with ECC

OpenSSL provides a very rich library of functions to perform ECC. In this section, first, an example will be presented that demonstrates the creation of a private key using the ECC functions available in the OpenSSL library.

ECC is based on domain parameters defined by various standards. You can see the list of all available standards defined and the recommended curves available in OpenSSL using the following command (once again, only part of the output is shown here):

```
$ openssl ecparam -list_curves
.
.
.
brainpoolP384r1: RFC 5639 curve over a 384 bit prime field brainpoolP384t1: RFC
5639 curve over a 384 bit prime field brainpoolP512r1: RFC 5639 curve over a
512 bit prime field brainpoolP512t1: RFC 5639 curve over a 512 bit prime field
```

Here, a private key based on SECP256K1 will be generated:

```
$ openssl ecparam -name secp256k1 -genkey -noout -out ec-privatekey.pem
```

This command will produce a file named `ec-privatekey.pem`, which we can view using the command shown here:

```
$ cat ec-privatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIJHUIm9NZAgfpUrSxUk/iINq1ghM/ewn/RLNreuR52h/oAcGBSuBBAAK
oUQDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8JdcGXYGxHdzr0Jt1NInaYE0GG
ChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END EC PRIVATE KEY-----
```

The file named `ec-privatekey.pem` now contains the elliptic curve private key that is generated based on the SECP256K1 curve. In order to generate a public key from a private key, issue the following command:

```
$ openssl ec -in ec-privatekey.pem -pubout -out ec-pubkey.pem
read EC key
writing EC key
```

Reading the file produces the following output, displaying the generated public key:

```
$ cat ec-pubkey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAE0G33mCZ4PKbg5EtwQjk6ucv9Qc9DTr8J
dcGXYGxHdzr0Jt1NInaYE0GGChFMT5pK+wfvSLkYl5ul0oczwWKjng==
-----END PUBLIC KEY-----
```

Now, the `ec-pubkey.pem` file contains the public key derived from `ec-privatekey.pem`. The private key can be further explored using the following command:

```
$ openssl ec -in ec-privatekey.pem -text -noout
read EC key
Private-Key: (256 bit) priv:
```

```
00:91:d4:22:6f:4d:64:08:1f:a5:4a:d2:c5:49:3f:
88:83:6a:d6:08:4c:fd:ec:27:fd:12:cd:ad:eb:91: e7:68:7f
pub:

04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
33:c1:62:a3:9e
ASN1 OID: secp256k1
```

Similarly, the public key can be further explored with the following command:

```
$ openssl ec -in ec-pubkey.pem -pubin - text -noout
read EC key
Private-Key: (256 bit) pub:
04:d0:6d:f7:98:26:78:3c:a6:e0:e4:4b:70:42:39:
3a:b9:cb:fd:41:cf:43:4e:bf:09:75:c1:97:60:6c:
47:77:3a:f4:26:dd:4d:22:76:98:13:41:86:0a:11:
4c:4f:9a:4a:fb:07:ef:48:b9:18:97:9b:a5:d2:87:
33:c1:62:a3:9e
ASN1 OID: secp256k1
```

It is also possible to generate a file with the required parameters—in this case, SECP256K1—and then explore it further to understand the underlying parameters:

```
$ openssl ecparam -name secp256k1 -out secp256k1.pem
$ cat secp256k1.pem
-----BEGIN EC PARAMETERS-----
BgUrgQQACg==
-----END EC PARAMETERS-----
```

The file now contains all the SECP256K1 parameters, and it can be analyzed using the following command:

```
$ openssl ecparam -in secp256k1.pem -text -param_enc explicit -noout
```

This command will produce output similar to the one shown here:

```
Field Type: prime-field Prime:
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff: ff:fc:2f
A: 0
B: 7 (0x7)
Generator (uncompressed): 04:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
```

```
0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
f8:17:98:48:3a:da:77:26:a3:c4:65:5d:a4:fb:fc:
0e:11:08:a8:fd:17:b4:48:a6:85:54:19:9c:47:d0:
8f:fb:10:d4:b8 Order:
00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
ff:fe:ba:ae:dc:e6:af:48:a0:3b:bf:d2:5e:8c:d0:
36:41:41
Cofactor: 1 (0x1)
```

The preceding example shows the prime number used and the values of `A` and `B`, along with the generator, order, and cofactor of the SECP256K1 curve domain parameters. With this, our introduction to public key cryptography from an encryption and decryption perspective is complete. Next, we introduce digital signatures.

Digital signatures

Digital signatures provide a means of associating a message with an entity from which the message has originated. Digital signatures are used to provide data origin authentication and non-repudiation.

Various schemes, such as RSA, DSA, and ECDSA-based digital signature schemes, are used in practice. RSA is the most common; however, with the traction of ECC, ECDSA-based schemes have become quite popular. This is beneficial in blockchains because ECC provides the same level of security that RSA does, but it uses less space. Also, the generation of keys is much faster in ECC compared to RSA, so it helps with the overall performance of the blockchain. The following table shows that ECC can provide the same level of cryptographic strength as an RSA-based system with a smaller key size:

RSA key size (bits)	Elliptic curve key size (bits)
1,024	160
2,048	224
3,072	256
7,680	384
15,360	521

Digital signatures are calculated in two steps. As an example, the high-level steps of the RSA digital signature scheme are as follows.

RSA digital signature algorithms

RSA-based digital signature algorithms are calculated using the two steps listed here. Fundamentally, the idea is to first compute the hash of the data and then sign it with the private key:

- **Calculate the hash value of the data packet**: This will provide the data integrity guarantee, as the hash can be computed at the receiver's end again and matched with the original hash to check whether the data has been modified in transit. Technically, message signing can work without hashing the data first, but that is not considered secure.

- **Sign the hash value with the signer's private key**: As only the signer has the private key, the authenticity of the signature and the signed data is ensured.

Digital signatures have some important properties, such as authenticity, unforgeability, and non-reusability:

- **Authenticity** means that digital signatures are verifiable by a receiving party.
- The **unforgeability** property ensures that only the sender of the message can use the signing functionality using the private key. Digital signatures must provide protection against **forgery**. Forgery means an adversary fabricating a valid signature for a message without any access to the legitimate signer's private key. In other words, unforgeability means that no one else can produce the signed message produced by a legitimate sender. This is also called the property of **non-repudiation**.
- **Non-reusability** means that the digital signature cannot be separated from a message and used again for another message. In other words, the digital signature is firmly bound to the corresponding message and cannot be simply cut from its original message and attached to another.

The operation of a generic digital signature function is shown in the following diagram:

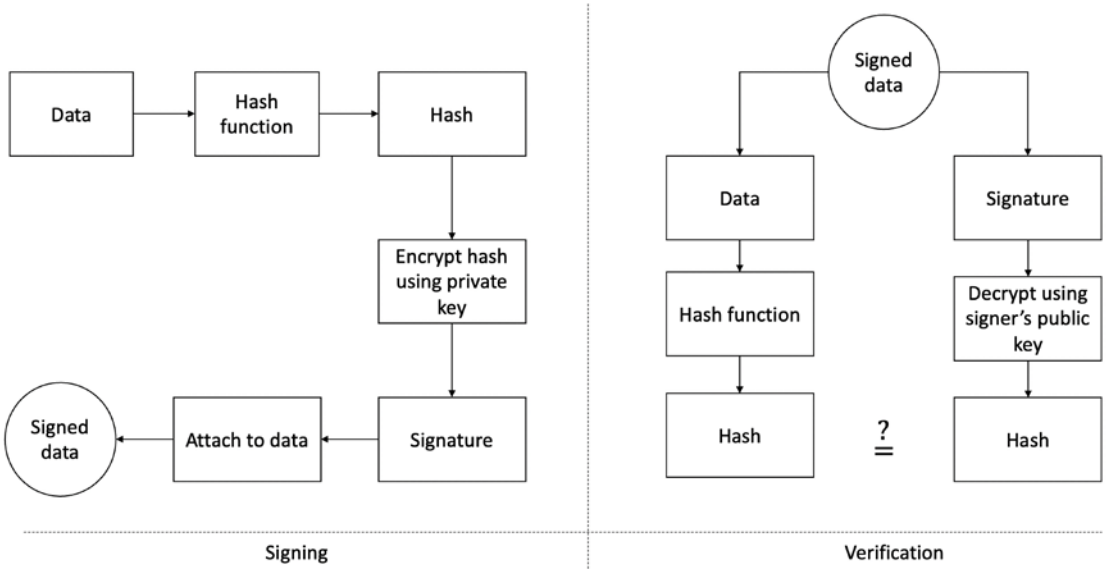


Figure 4.8: Digital signing (left) and verification process (right) (example of RSA digital signatures)

If a sender wants to send an authenticated message to a receiver, there are two methods that can be used:

- **Sign then encrypt**: With this approach, the sender digitally signs the data using the private key, appends the signature to the data, and then encrypts the data and the digital signature using the receiver's public key. This is considered a more secure scheme compared to the **encrypt then sign** scheme described next.
- **Encrypt then sign**: With this method, the sender encrypts the data using the receiver's public key and then digitally signs the encrypted data.



In practice, a digital certificate that contains the digital signature is issued by a **certificate authority** (**CA**) that associates a public key with an identity.

Various practical examples will now be shown that demonstrate how the RSA digital signature can be generated, used, and verified using OpenSSL.

Generating RSA digital signatures

The first step is to generate a hash of the message file. Note that the `SHA-256` hash is just chosen as an example:

```
$ openssl dgst -sha256 message.txt
SHA256(message.txt)=
eb96d1f89812bf4967d9fb4ead128c3b787272b7be21dd2529278db1128d559c
```

Both hash generation and signing can be done in a single step, as shown here. Note that `privatekey.pem` was generated in the steps provided previously:

```
$ openssl dgst -sha256 -sign privatekey.pem -out signature.bin message.txt
```

Now, let's display the directory showing the relevant files:

```
$ cat signature.bin
```

To verify the signature, the following operation can be performed:

```
$ openssl dgst -sha256 -verify publickey.pem -signature signature.bin  message.
txt
Verified OK
```

Similarly, if some other signature file that is not valid is used, the verification will fail, as shown here:

```
$ openssl dgst -sha256 -verify publickey.pem –signature someothersignature.bin
message.txt
Verification Failure
```

Now that we understand how the RSA digital signature scheme works, in the next section, we'll introduce ECDSA, which is another popular digital signature scheme.

The elliptic curve digital signature algorithm

The **ECDSA** is a DSA based on elliptic curves. The DSA is a standard for digital signatures. It is based on modular exponentiation and the discrete logarithm problem. It is used on the Bitcoin and Ethereum blockchain platforms to validate messages and provide data integrity services. Now, we'll describe how the ECDSA works.

To sign and verify using the ECDSA scheme, the first key pair needs to be generated:

1. First, define an elliptic curve E with the following:
    - Modulus P
    - Coefficients a and b
    - Generator point A that forms a cyclic group of prime order q
2. An integer d is chosen randomly so that $0 < d < q$.
3. Calculate public key **B** so that **B = d A**:
    - The public key is a sextuple in the form shown here:

$$Kpb = (p, a, b, q, A, B)$$

    - The private key is a randomly chosen integer d in *step 2*:

$$Kpr = d$$

    - Now, the signature can be generated using the private and public keys.
    - An ephemeral key K_e is chosen, where $0 < K_e < q$. It should be ensured that K_e is truly random and that no two signatures have the same key; otherwise, the private key can be calculated.
    - Another value R is calculated using $R = K_e A$—that is, by multiplying A (the generator point) and the random ephemeral key.
    - Initialize a variable r with the x coordinate value of point R so that $r = xR$.
    - The signature can be calculated as follows:

$$S = (h(m)+dr)K_e^{-1} \bmod q$$

4. Here, m is the message for which the signature is being computed, and $h(m)$ is the hash of the message m.
5. Signature verification is carried out by following this process:
    - Auxiliary value w is calculated as $w = s^{-1} \bmod q$
    - Auxiliary value $u_1 = w.\ h(m) \bmod q$
    - Auxiliary value $u_2 = w.\ r \bmod q$
    - Calculate point P, $P = u_1A + u_2B$
6. Verification is carried out as follows:
    - r, s is accepted as a valid signature if the x coordinate of point P calculated in *step 4* has the same value as the signature parameter $r \bmod q$—that is:

$$X_p = r \bmod q \text{ means valid signature}$$

$$X_p \neq r \bmod q \text{ means valid signature}$$

Next, an example will be presented that shows how OpenSSL can be used to perform ECDSA-related operations based on ECC.

Generating ECDSA digital signatures

In this example we'll see how an ECDSA digital signature can be generated using OpenSSL. First, the private key is generated using the following commands:

```
$ openssl ecparam -genkey -name secp256k1 -noout -out eccprivatekey.pem
$ cat eccprivatekey.pem
-----BEGIN EC PRIVATE KEY-----
MHQCAQEEIMVmyrnEDOs7SYxS/AbXoIwqZqJ+gND9Z2/nQyzcpaPBoAcGBSuBBAAK
oUQDQgAEEKKS4E4+TATIeBX8o2J6PxKkjcoWrXPwNRo/k4Y/CZA4pXvlyTgH5LYm
QbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END EC PRIVATE KEY-----
```

Next, the public key is generated from the private key:

```
$ openssl ec -in eccprivatekey.pem -pubout -out eccpublickey.pem
read EC key
writing EC key
$ cat eccpublickey.pem
-----BEGIN PUBLIC KEY-----
MFYwEAYHKoZIzj0CAQYFK4EEAAoDQgAEEKKS4E4+TATIeBX8o2J6PxKkjcoWrXPw
NRo/k4Y/CZA4pXvlyTgH5LYmQbU0qUtPM7dAEzOsaoXmetqB+6cM+Q==
-----END PUBLIC KEY-----
```

Now, suppose a file named `testsign.txt` needs to be signed and verified. This can be achieved as follows:

1. Create a test file:

   ```
   $ echo testing > testsign.txt
   $ cat testsign.txt
   testing
   ```

2. Run the following command to generate a signature using a private key for the `testsign.txt` file:

   ```
   $ openssl dgst -ecdsa-with-SHA1 -sign eccprivatekey.pem testsign.txt >
   ecsign.bin
   ```

3. Finally, the command for verification can be run, as shown here:

   ```
   $ openssl dgst -ecdsa-with-SHA1 -verify eccpublickey.pem -signature
   ecsign.bin testsign.txt
   Verified OK
   ```

A certificate can also be produced by using the private key generated earlier by using the command shown here:

```
$ openssl req -new -key eccprivatekey.pem -x509 -nodes -days 365 - out
ecccertificate.pem
```

Enter the appropriate parameters to generate the certificate:

```
Country Name (2 letter code) []:GB
State or Province Name (full name) []:Montgomery
Locality Name (eg, city) []:London
Organization Name (eg, company) []:BAK
Organizational Unit Name (eg, section) []:Research
Common Name (eg, fully qualified host name) []:masteringblockchain.com
Email Address []:sonic@xmail.com
```

Once generated, the certificate can be explored using the following command:

```
$ openssl x509 -in ecccertificate.pem -text -noout
```

The output shows the certificate:

```
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 9248556263013038870 (0x805978eb961b3b16)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=GB, ST=Montgomery, L=London, O=BAK, OU=Research,
CN=masteringblockchain.com/emailAddress=sonic@xmail.com
        Validity
            Not Before: Apr 23 21:21:40 2022 GMT
            Not After : Apr 23 21:21:40 2023 GMT
        Subject: C=GB, ST=Montgomery, L=London, O=BAK, OU=Research,
CN=masteringblockchain.com/emailAddress=sonic@xmail.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:d3:11:d6:9d:6a:f6:bf:0c:11:3c:32:05:66:49:
                    c0:b0:06:ec:0f:1a:9f:28:47:19:06:61:b7:7e:d5:
                    e8:df:26:85:07:7a:e1:f5:a3:f9:9a:ea:61:52:b3:
                    4e:d7:bd:4b:ab:21:12:db:9e:4f:41:cf:b4:85:50:
                    2a:d2:08:3c:57
```

```
                ASN1 OID: secp256k1
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:be:ba:b3:65:74:12:33:86:1e:1c:8b:cf:82:
         19:5e:34:b9:f1:28:26:bf:84:3b:ba:34:8c:64:87:f2:16:29:
         a2:02:21:00:ec:98:8f:fd:37:d5:38:4b:e2:15:f1:87:06:f4:
         d9:4a:fd:b4:fd:a1:d7:ff:9b:38:6d:e0:7a:73:f0:05:f8:75
```

In this section, we covered digital signature schemes and some practical examples. Next, we'll introduce different types of digital signatures.

Different types of digital signatures

Now, we'll provide an overview of different types of digital signatures and their relevance and applications in blockchain technology. There are different types of digital signature schemes based on the underlying mathematics they use. Families of signatures schemes are:

- RSA-based signature and public keys
- Discrete log-based digital signatures

RSA signatures and public keys are at least 256 bytes in size, which is not suitable for blockchains as they require more space. They are, however, fast to verify but due to the unnecessary space requirements, they are not used in blockchains.

Discrete log signatures are commonly used in blockchains. They are small, usually 48 or 64 bytes with 32-byte public keys. This small size makes them useful for blockchains, and they are used in Bitcoin, Ethereum, and quite a few other blockchains. This family includes Schnorr and ECDSA signatures.

Digital signatures are used in blockchains, where transactions are digitally signed by senders using their private key before the sender broadcasts the transaction to the network. This digital signing proves that the sender is the rightful owner of the asset, for example, of bitcoins, and is authorized. These transactions are verified again by other nodes on the network to ensure that the funds indeed belong to the node (user) who claims to be the owner. They are also used in consensus protocols to sign votes, for example, in BFT consensus schemes.

Some other types based on the difference in their application and mechanics are described below.

Blind signatures

The **blind signature scheme** was invented by David Chaum in 1982. It is based on public key digital signature schemes, such as RSA. The key idea behind blind signatures is to get the message signed by the signer, without actually revealing the message. This is achieved by disguising or blinding the message before signing it, hence the name **blind signatures**.

This blind signature can then be verified against the original message, just like a normal digital signature. Blind signatures were introduced as a mechanism to allow the development of digital cash schemes. Specifically, a blind signature can provide unlinkability and anonymity services in a distributed system, such as a blockchain.



The original paper from David Chaum on blind signatures is available at `http://blog.koehntopp.de/uploads/Chaum.BlindSigForPayment.1982.PDF`

Chaum, David. "Blind signatures for untraceable payments." In Advances in cryptology, pp. 199–203. Springer, Boston, MA, 1983.

The blind signature scheme is also used to build Blindcoin, which is used in the Mixcoin protocol for Bitcoin to hide user addresses from the mixing service. The Mixcoin protocol allows anonymous payments in the Bitcoin network, but the user addresses are still visible to the mixing service. Blindcoin is a protocol that addresses this limitation.



More information on this scheme is available in the following paper:

Valenta, L. and Rowan, B., January. "Blindcoin: Blinded, accountable mixes for bitcoin." In International Conference on Financial Cryptography and Data Security, pp. 112–126. Springer, Berlin, Heidelberg, 2015.

Multisignatures

In the **multisignature scheme**, a group of entities signs a single message. In other words, multiple unique keys held by their respective owners are used to sign a single message.



This scheme was introduced in 1983 by Itakura et al. in their paper "*A Public-key Cryptosystem Suitable for Digital Multisignatures, vol. 71." Nec Research & Development, pp. 474–480*, 1983.

Multisignatures are also sometimes called multiparty signatures in literature. In blockchain implementations, multisignatures provide the ability to allow transactions to be signed by multiple users, which results in increased security. This is also called multi-sig and has been implemented in Bitcoin. These schemes can be used in such a way that the requirement of a number of signatures can be set in order to authorize transactions. For example, a 1-of-2 multisignature can represent a joint account where either one of the joint account holders is required to authorize a transaction by signing it.

In another variation, for example, a 2-of-2 multisignature can be used in a scenario where both joint account holders' signatures are required to sign the transaction. This concept is also generalized as *m* of *n* signatures, where *m* is the minimum number of required signatures and *n* is the total number of signatures.

This concept can be visualized with the following diagram:

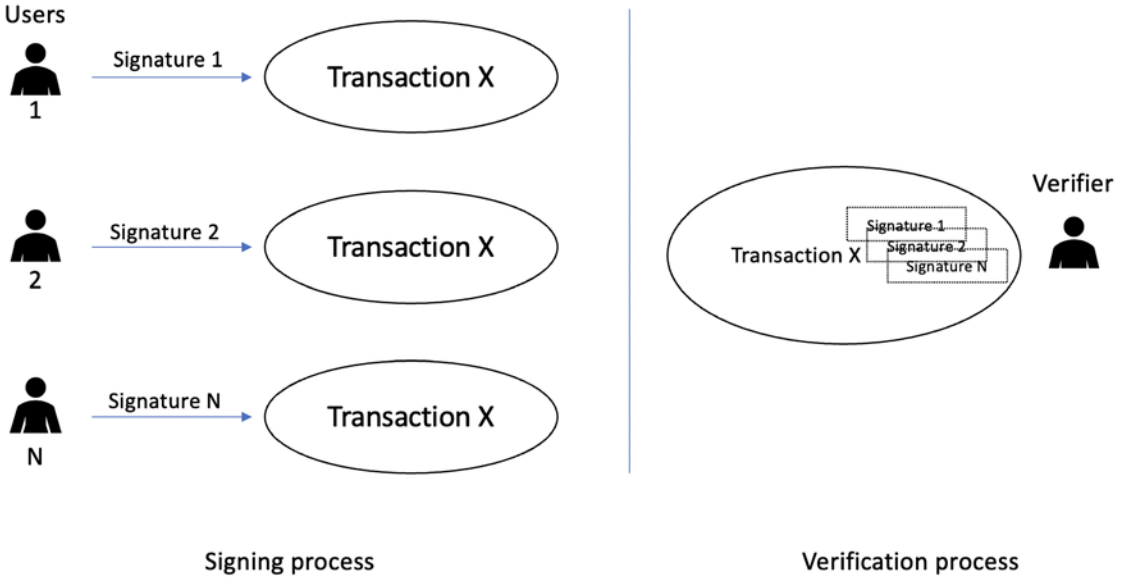


Figure 4.9: Multisignature scheme

The preceding diagram shows the signing process on the left-hand side, where *m* is the number of different users, and holding *m* unique signatures signs a single transaction. When the validator or verifier receives it, all the signatures in it need to be individually verified.



The Openchain and Multichain blockchains make use of multisignature schemes.

More information on Openchain is available at `https://docs.openchain.org/en/latest/general/overview.html`.

More information regarding Multichain's multisignature scheme is available at `https://www.multichain.com/developers/multisignature-transactions/`.

Threshold signatures

This scheme is a specific type of multisignature. It does not rely on users to sign the message with their own unique keys; instead, it requires only one public key and one private key, and results in only one digital signature. In multisignature, the resultant message contains digital signatures from all signers and requires verification individually by the verification party, but in threshold signatures, the verifier verifies only one digital signature. The key idea of the scheme is to split the private key into multiple parts, and each signer keeps their own share of the private key. The signing process requires each user to use their respective share of the private key to sign the message. In this scheme, only a subset of signers can produce the signature with their share, and there is no need for all members to collaborate to produce the signature.

The communication between signers is governed by a specific communication protocol. In contrast with multisignatures, the threshold signatures result in a smaller transaction size and are quicker to verify. A threshold signature increases the availability and resilience of the scheme as shares of the signature are stored by distinct signers (servers).

A slight limitation, however, is that for threshold signatures to work, all signers involved in the signing process must remain online to participate in an interactive protocol to generate the signature, whereas in multisignatures, the signature can be provided asynchronously; that is, users can provide signatures whenever they are available.

There are also variants that are non-interactive, where each user computes its share of the signature without any online communication with other users (servers). From another angle, there could be a scenario in multisignature schemes where a user could withhold their signature maliciously, which could result in denial of service. With threshold signatures, it is possible to construct the required signature only with any authorized subset of the entire group, i.e., $t + 1 \leq n$, but at least those $t + 1$ participants must be online.

Threshold signatures can also be used to provide anonymity services in a blockchain network. This is so because threshold signature schemes do not reveal the members in the threshold group that have signed to produce the signature. This is in contrast with multisignature schemes, which do reveal the identities of all the signers. Moreover, in permissioned networks, threshold signatures can be used in scenarios where a threshold of nodes (users) is required to agree on an operation.

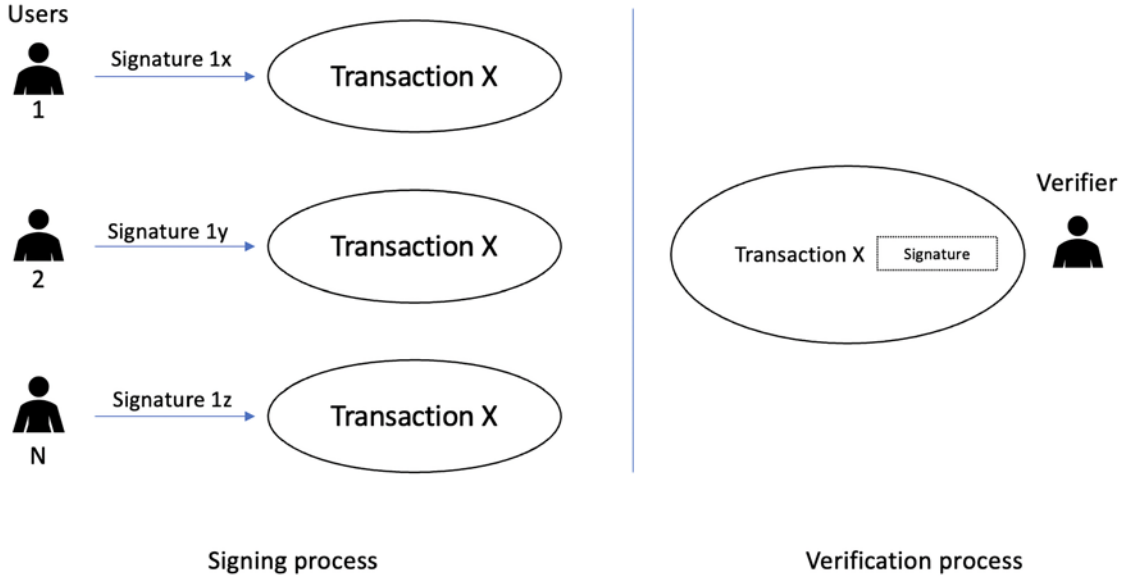


Figure 4.10: Threshold signature scheme

The preceding diagram shows the signing process on the left-hand side, where m number of different users, holding different parts (shares) of a digital signature, sign a single transaction. When the validator or verifier receives it, only one signature needs to be verified.

Aggregate signatures

Aggregate signatures are used to reduce the size of digital signatures. This scheme is particularly useful in scenarios where multiple digital signatures are in use. The core idea is to aggregate multiple signatures into a single signature, without increasing the size of the signature of a single message.It is simply a type of digital signature that supports aggregation. The small aggregate signature is enough to provide verification to the verifier that all users did sign their original messages. Aggregate signatures are commonly used to reduce the size of messages in network and security protocols. For example, the size of digital certificate chains in **public key infrastructure** (**PKI**) can be reduced significantly by compressing all signatures in the chain into a single signature. **Boneh–Lynn–Shacham** (**BLS**) aggregate signatures are a popular example of an aggregate signature.



More information regarding aggregate BLS signatures is available in the paper here: `https://crypto.stanford.edu/~dabo/papers/aggreg.pdf`.

Boneh, D., Gentry, C., Lynn, B. and Shacham, H., Aggregate and Verifiably Encrypted Signatures from Bilinear Maps.

More information on BLS and specifically its usage in blockchains to reduce the size of the blockchain is available in an excellent paper here: `https://eprint.iacr.org/2018/483.pdf`

Boneh, D., Drijvers, M. and Neven, G., 2018, December. Compact multi-signatures for smaller blockchains. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 435–464). Springer, Cham.

Aggregation allows you to compress many signatures from many different users into a single 48-byte signature. Instead of writing x number of signatures for x number of transactions, you can compress and write only one signature on the blockchain. This property makes BLS signatures very useful to blockchains due to the space-saving benefit. Moreover, it's quick to verify, which makes it even more suitable for blockchains. BLS signatures allow you to build m out of n type threshold schemes quite easily, where a threshold of m users is required to sign a transaction out of n users.

BLS signatures are used in Ethereum and a few other blockchains. BLS signatures are 48 bytes in length; they can be aggregated and allow you to build threshold schemes. Most new blockchains use BLS signatures due to their smaller size and aggregation feature.

Ring signatures

Ring signatures were introduced in 2001 by Ron Rivest, Adi Shamir, and Yael Tauman.



The original paper is available at: `https://link.springer.com/content/pdf/10.1007/3-540-45682-1_32.pdf`

Rivest, R.L., Shamir, A. and Tauman, Y., 2001, December. How to leak a secret. In International Conference on the Theory and Application of Cryptology and Information Security (pp. 552–565). Springer, Berlin, Heidelberg.

Ring signature schemes allow a mechanism where any member of a group of signers can sign a message on behalf of an entire group. The key requirement here is that the identity of the actual signer who signed the message must remain unknown (computationally infeasible to determine) to an outside observer. It looks equally likely that anyone in the trusted group of signers could have signed the message, but it is not possible to figure out who actually signed the message. Each member of the ring group keeps a public key and a private key. Ring signatures can be used to provide privacy (anonymity)-preserving services. A variation of ring signatures called traceable ring signatures so that double spending can be prevented is used in **CryptoNote**. **Monero** cryptocurrency also uses ring signatures.

In this section, we covered different types of digital signatures, their operations, and their relevance to blockchains. Next, we'll present some more advanced cryptography topics.

Cryptographic constructs and blockchain technology

Now, we'll present some advanced topics in cryptography that not only are important on their own but are also relevant to blockchain technology due to their various applications in this space.

Homomorphic encryption

An encryption algorithm is homomorphic if it can apply some operation on encrypted data without decrypting it. Public key cryptosystems, such as RSA, are multiplicative homomorphic or additive homomorphic, such as the Paillier cryptosystem, and are called **partially homomorphic** systems. Additive **partially homomorphic encryptions** (**PHEs**) are suitable for e-voting and banking applications due to their ability to perform additional operations on encrypted data. For example, in voting, they can sum up votes even if they are encrypted.

Until recently, there has been no system that supported both operations, but in 2009, a **fully homomorphic** system was discovered by Craig Gentry. As these schemes enable the processing of encrypted data without the need for decryption, they have many different potential applications, especially in scenarios where maintaining privacy is required, but data is also mandated to be processed by potentially untrusted parties, for example, cloud computing and online search engines.

Recent developments in homomorphic encryption have been very promising, and researchers are actively working to make it efficient and more practical. This is of particular interest in blockchain technology, as described later in this book, as it can solve the problem of confidentiality and privacy in the blockchain.

Secret sharing

Secret sharing is the mechanism of distributing a secret among a set of entities. All entities within a set get a unique part of the secret after it is split into multiple parts. The secret can be reconstructed by combining all or some parts (a certain number or threshold) of the secret. The individual secret shares/parts, on their own, do not reveal anything about the secret.

This concept can be visualized through the following diagram:

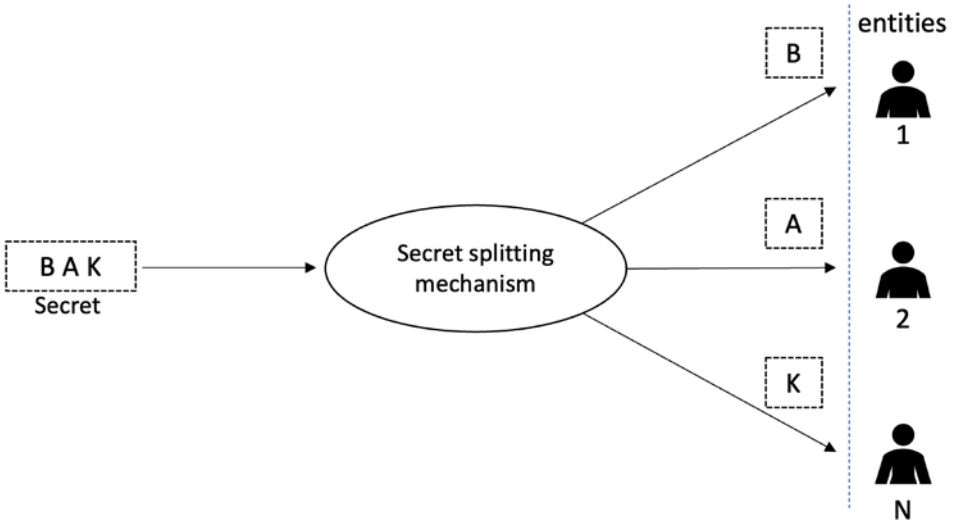


Figure 4.11: Secret sharing scheme

Commitment schemes

Commitment schemes are usually described as a digital cryptographic equivalent of a sealed envelope. A commitment itself does not reveal any information about the actual value inside it. This scheme runs in two phases, namely:

- **Commit phase**: The commit phase provides two security features—that is, **hiding** and **binding**:
  - The **hiding** property ensures that an adversary cannot find any information or reveal the committed value before the open phase.
  - The **binding** property ensures that once the sender has committed to a value, that value or message cannot be changed.
- **Open phase**: The open phase (also called the unveil phase) is where the receiver receives some additional information from the sender. This allows the receiver to establish the value hidden (concealed) by the commitment with proof that the sender has not modified the value after the commitment— that is, that the sender has not deceived during the commit phase.

Commitment schemes play a vital role in building privacy-preserving blockchains. They have applications in building zero-knowledge protocols and range-proof protocols. Blockchains such as **Monero** and **Zcash** make use of commitment schemes to introduce privacy.

The key idea behind commitment schemes is that they allow someone to secretly commit to a value with the ability to reveal it later. These schemes prevent parties from changing their committed value later after they've committed to a value. A prime example of a commitment scheme is the **Pedersen commitment scheme**.



The original paper on the Pedersen commitment scheme is available here: `https://link.springer.com/content/pdf/10.1007/3-540-46766-1_9.pdf`.

Pedersen, T.P., 1991, August. Non-interactive and information-theoretic secure verifiable secret sharing. In Annual international cryptology conference (pp. 129–140). Springer, Berlin, Heidelberg.

In the next section, we'll introduce **Zero-Knowledge Proofs** (**ZKPs**), which is an exciting subject and a very ripe area for research. Already, various major blockchains and cryptocurrencies make use of zero-knowledge proofs to provide privacy. A prime example is **Zcash** (`https://z.cash`).

Zero-knowledge proofs

ZKPs were introduced by Goldwasser, Micali, and Rackoff in 1985. These proofs are used to prove the validity of an assertion without revealing any information whatsoever about the assertion. There are three properties of ZKPs that are required: completeness, soundness, and the zero-knowledge property:

- **Completeness** ensures that if a certain assertion is true, then the verifier will be convinced of this claim by the prover.
- The **soundness** property makes sure that if an assertion is false, then no dishonest prover can convince the verifier otherwise.
- The **zero-knowledge property**, as the name implies, is the key property of ZKPs, whereby it is ensured that absolutely nothing is revealed about the assertion except whether it is true or false.

ZKPs are probabilistic instead of deterministic. The probabilistic nature of ZKPs is because these protocols require several rounds to achieve a higher level of assurance gradually, with each round, which allows the verifier to accept that the prover indeed knows the secret.

In literature, usually, an analogy known as **Ali Baba's Cave** is used to explain ZKPs. The following diagram shows a variant of this analogy of how the zero-knowledge protocol works. It shows two characters called **Peggy** and **Victor** whose roles are *Prover* and *Verifier*, respectively. Peggy knows a secret magic word to open the door in a cave, which is shaped like a ring. It has a split entrance and a magic door in the middle of the ring that blocks the other side. We have also labeled the left and right entrances as **A** and **B**.

Victor wants to know the secret, but Peggy does not want to reveal it. However, she would like to prove to Victor that she does know the secret:

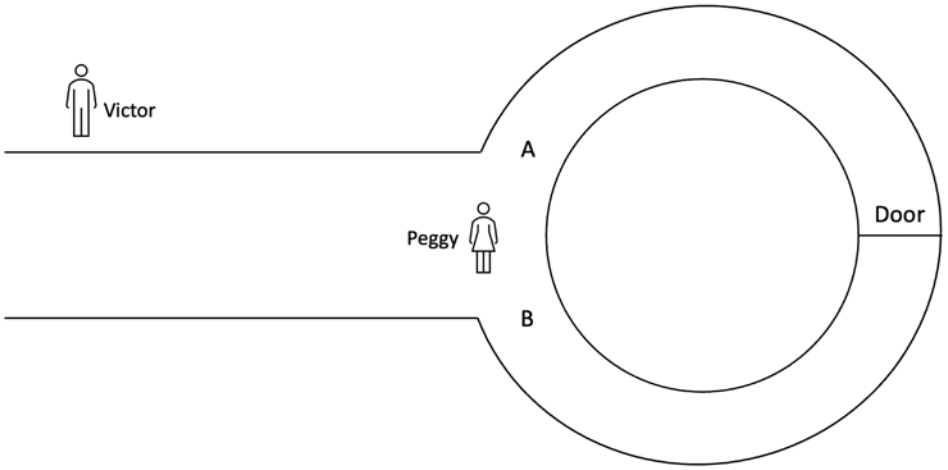


Figure 4.12: Analogy of the zero-knowledge mechanism

Now, there are several steps that are taken by both Peggy and Victor to reach a conclusion regarding whether Peggy knows the secret or not:

- First, Victor waits outside the main cave entrance and Peggy goes in the cave.
- Peggy randomly chooses either the **A** or **B** entrance to the cave.
- Now, Victor enters the cave and shouts either **A** or **B** randomly, asking Peggy to come out of the exit he named.
- Victor records which exit Peggy comes out from.

Now, suppose Victor asked Peggy to come out from exit A and she came out from exit B. Victor then knows that Peggy does not know the secret. If Peggy comes out of exit A, then there is a 50% chance that she does know the secret, but this also means that she may have gotten lucky and chosen A to enter the cave in the first place, and now has just returned without needing to go through the magic door at all. However, if this routine is performed several times, and given that Victor is choosing A or B at random, with each run (round) of this routine (protocol), the chances of Peggy getting lucky diminish. If Peggy repeatedly manages to emerge from the entrance that Victor has named, then it is highly probable that Peggy does know the secret to open the magic door.



Remember, we said earlier that zero-knowledge protocols are probabilistic. Peggy and Victor repeatedly performing this routine proves knowledge of the secret with high probability, as the probability of guessing or getting lucky reduces to the point of being negligible after *some* rounds.

Note that during this process, Peggy has not revealed the secret at all, but still managed to convince Victor with high probability that she does know the secret.

A ZKP comprises the following phases:

- **Witness phase:** In this phase, the prover sends proof of the statement and sends it to the verifier.
- **Challenge phase:** In this phase, the verifier chooses a question (challenge) and sends it to the prover.
- **Response phase:** In this phase, the prover generates an answer and sends it as a response to the verifier. The verifier then checks the answer to ascertain whether the prover really knows the statement.

This scheme can also be visualized using the following diagram, which shows how the zero-knowledge protocol generally works and what steps the prover and verifier take to successfully run the protocol:



Figure 4.13: A ZKP scheme

Even though **Zero Knowledge** (**ZK**) is not a new concept, these protocols have sparked a special interest among researchers in the blockchain space due to their privacy properties, which are very much desirable in finance and many other fields, including law and medicine. A prime example of the successful implementation of a ZKP mechanism is the **Zcash** cryptocurrency. In Zcash, the **zero-knowledge Succinct Non-interactive ARgument of Knowledge** (**zk-SNARK**) is implemented to provide anonymity and confidentiality.

Applications of ZKP include proof of ownership, that is, proof that the prover is the owner of some secret or a private key without revealing the nature of it. ZKPs can also be used to prove that the prover is a member of some organization or group, without revealing their identity. Another use case could be proof of age, where the prover proves that they are, for example, older than 25 years, but does not want to reveal their exact age. Another application could be where citizens can pay taxes without revealing their income.

zk-SNARKs

Zero-knowledge protocols are usually interactive as they require repeated interactions between the prover and the verifier, but there are protocols that do not require any interaction between a prover and a verifier. These types of ZKPs are called non-interactive types of proofs. A prominent example is the zk-SNARK.

Before the introduction of zk-SNARKs, ZKPs were considered not very practical because of the complexity that arises from the requirements of repeated interaction between the prover and the verifier and a large proof size.

There has been some very influential work published on non-interactive zero-knowledge proofs over the years. The most prominent of these, and the one that proposed the first universal design, is provided by the paper by Eli Ben-Sasson et al., which is available at `https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-ben-sasson.pdf`

Ben-Sasson, E., Chiesa, A., Tromer, E., and Virza, M., 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In 23rd {USENIX} Security Symposium ({USENIX} Security 14) (pp. 781–796).



Other previous notable works that contributed to the development of zk-SNARKs include:

Gennaro, R., Gentry, C., Parno, B., and Raykova, M., "Quadratic span programs and succinct NIZKs without PCPs." In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 626–645, Springer, Berlin, Heidelberg, 2013.

(`https://link.springer.com/content/pdf/10.1007/978-3-642-38348-9_37.pdf`)

Groth, J. and Sahai, A., "Efficient non-interactive proof systems for bilinear groups." In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 415–432, Springer, Berlin, Heidelberg , 2008.

(`https://link.springer.com/content/pdf/10.1007/978-3-540-78967-3_24.pdf`)

zk-SNARKs have several properties, which are described here:

- **zk**: This property allows a prover to convince a verifier that an assertion (statement) is true without revealing any information about it. A statement in this context is any computer program that terminates and does not take too long to run, that is, not too many cycles.
- **Succinct** (**S**): This property allows for a small-size proof that is quick to verify.
- **Non-interactive** (**N**): This property allows the prover to prove a statement by only sending just a single message (proof) to the verifier.
- **ARguments** (**AR**): These are the arguments to convince the verifier that the prover's assertion is true. Remember that we discussed soundness and completeness earlier. In the case of arguments, the prover is computationally bounded, and it is computationally infeasible for it to cheat the verifier.
- **Knowledge** (**K**): This property means that the prover has the evidence that they indeed know the statement they claim knowledge of.

zk-SNARKs have been implemented in different blockchains. The most prominent example is Zcash, which uses them for its shielded transactions feature to provide confidentiality and anonymity. Support for cryptographic primitives required for zk-SNARKs has also been introduced in the Ethereum blockchain with the Byzantium release.

zk-SNARK proofs are generated by following several different steps. Generating SNARKs for a program (a statement) is not a simple process as it requires the program to be converted into a circuit with a very specific format. This specific form is called the **quadratic arithmetic program** (**QAP**). The process of proof generation is as follows:



Figure 4.14: zk-SNARK construction

The process takes the following steps:

- **Arithmetic circuit**: The first step in zk-SNARK construction is to convert a logical step of a **computation** into the smallest possible units comprised of basic arithmetic operations. Arithmetic operations include addition, subtraction, multiplication, and division. In the arithmetic circuit, the computation is presented as wires and gates, representing the flow of inputs and arithmetic operations performed on the inputs. It is a **directed acyclic graph** (**DAG**) that evaluates a polynomial by taking inputs and performing arithmetic operations on it.
- **R1CS**: R1CS is the abbreviation of **Rank 1 Constraint System**. Basically, this system is a set of constraints that allows all the steps in the arithmetic circuit to be verified and confirms that, at the end of the process, the output is as expected.
- **QAP**: The prover makes use of QAPs to construct a proof of the statement. In R1CS, the verifier must check many different constraints, but with a QAP representation of the circuit, all the different constraints can be bundled into only a single constraint.
- Finally, QAP is used in the **zk-SNARK** protocol to prove the assertion through the prover and verifier.

zk-SNARKs are not a silver bullet to privacy problems. Instead, like many other technologies, there are pros and cons.

The biggest criticism of zk-SNARKs is the initial trusted setup. This trusted setup can be influenced and compromised. However, if done correctly, it does work, but there is, however, always a chance that the initial setup was compromised, and no one will ever be able to find out. This is the issue that has been addressed by zk-STARKs.

zk-STARKs

Zero-Knowledge Scalable Transparent ARguments of Knowledge (**zk-STARKs**) are a new type of ZKP that has addressed several limitations in zk-SNARKs.



This scheme was designed by Eli-Ben Sasson et al. The original paper on the subject is available here:

`https://eprint.iacr.org/2018/046.pdf`

Ben-Sasson, E., Bentov, I., Horesh, Y., and Riabzev, M., "Scalable, transparent, and post-quantum secure computational integrity. "IACR Cryptology ePrint Archive, pp. 46, 2018

The key differences between the zk-STARK and zk-SNARK schemes are listed in the following table:

Properties	**zk-SNARKs**	**zk-STARKs**
Scalability	Less scalable	More scalable
Initial trusted setup	Required	Not required—has a publicly verifiable mechanism
Post-quantum-resistant	Not resistant to attacks from quantum computers	Resistant to attacks from quantum computers
Construction techniques	Rely on elliptic curves and pairings	Based on hash functions and concepts from information theory

Even though this scheme is more efficient, post-quantum-resistant, scalable, and transparent, the biggest limitation is its size of proof, which is a few hundred kilobytes compared to zk-SNARKs' 288 bytes. This is seen as a problem in public blockchains where large proofs may pose a problem to scalability and performance.

Zero-knowledge range proofs

Zero-knowledge range proofs (**ZKRPs**) are used to prove that a given number is within a certain range. This can be useful when, for example, someone does not want to reveal their salary but is willing to only reveal the range between which their salary lies. Range proofs can also be used for age checks without requiring the prover to reveal their exact age.

In this section, we have covered the very interesting topic of ZKPs and their relevant techniques and developments. ZKPs is a very active area of research, especially in the context of blockchains, where this technology can provide much needed privacy and confidentiality services on public blockchain networks.

In the next section, we will introduce encoding schemes and some relevant examples.

Encoding schemes

Other than cryptographic primitives, binary-to-text **encoding schemes** are also used in various scenarios. The most common use is to convert binary data into text so that it can either be processed, saved, or transmitted via a protocol that does not support the processing of binary data. For example, images can be stored in a database encoded in base64, which allows a text field to be able to store a picture. Another encoding, named base58, was popularized by its use in Bitcoin.

Base64

The base64-encoding scheme is used to encode binary data into printable characters.

We can do a quick experiment using the OpenSSL command line:

```
$ openssl rand 16 -base64
4ULD5sJtGeoSnogIniHp7g==
```

The preceding command has generated a random sequence of 16 bits and then, using the `-base64` switch, it has converted that into a base64 text string. The base64 converts from 8 bits to a 6-bit ASCII representation. This is useful for storage and transmission, especially in cases where binary data handling could lead to incompatibility between systems. This mechanism is a flexible way to represent binary data as ASCII, which can be easily and universally stored and transmitted.

base58

The base58 scheme was first introduced with Bitcoin and is used to encode integers into alphanumeric strings. The key idea behind this encoding scheme is to avoid non-alphanumeric characters and also those characters that look similar and could lead to ambiguity; for example, a lowercase L (l) may look like the number one (1). This feature is especially useful because Bitcoin addresses must not have any confusion about the character representation; otherwise, it could lead to wrongly sending bitcoins to some non-existent or incorrect address, which is clearly a financial loss. This ingenious encoding scheme avoids this type of situation by ignoring similar-looking characters.

We will explore base58 and its role in generating Bitcoin addresses in detail in *Chapter 6*, *Bitcoin Architecture.*

Verifiable random functions

A **verifiable random function** (**VRF**) is a keyed hash function that uses public cryptography instead of symmetric key cryptography, as in MACs. It is a public key variation of a keyed hash function. In this scheme, the hash is computed by the holder of the private key, which is publicly verifiable, with the public key used to check if the hash is correct.



VRF was introduced by Micali et.al in:

Micali, Silvio, Michael Rabin, and Salil Vadhan. "Verifiable random functions." In Proceedings of the 40th Annual Symposium on the Foundations of Computer Science (FOCS '99), pp. 120–130. New York: IEEE Computer Society Press, 1999.

The original paper is available here: `http://nrs.harvard.edu/urn-3:HUL.InstRepos:5028196`.

A VRF is a random number generator. It takes an arbitrary input and produces a random output and a proof. The proof is used to verify the correctness of the output. More formally, it comprises three algorithms, KeyGen, Evaluate, and Verify. KeyGen takes a random input and generates a verification key and a secret key.

The Evaluate function takes the private key, a message, and produces a random output and a proof. This output is unique and pseudorandom. Finally, the Verify algorithm takes the verification key, the message, the output, and the proof and verifies if the output produced by the evaluating algorithm is indeed correct.

VRFs are used in blockchains for block proposer selection in proof-of-stake consensus protocols. Some examples of such protocols include **Algorand**, **Cardano Ouroboros**, and **Polkadot BABE**.

Cryptography is a vast field, and this section has introduced some of the main concepts that are essential to understanding cryptography in general, specifically from a blockchain and cryptocurrency point of view.

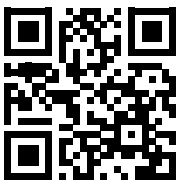
Summary

This chapter started with an introduction to some basic mathematics concepts and asymmetric key cryptography. We discussed various constructs such as RSA and ECC. We also performed some experiments using OpenSSL to see how theoretical concepts could be implemented practically. We also looked at some advanced and modern concepts such as zero-knowledge proofs and relevant constructions, along with different types of digital signatures.

In the next chapter, we will explore the captivating world of distributed consensus, which is central to the integrity of any blockchain and is a very active area of research.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

5

Consensus Algorithms

Consensus is a fundamental problem in distributed systems. Since the 1970s, this problem has been researched in the context of distributed systems, but a renewed interest has arisen in developing distributed consensus algorithms that are suitable for blockchain networks. In this chapter, we will explore the underlying techniques behind distributed consensus algorithms, their inner workings, and new algorithms that have been specifically developed for blockchain networks.

In addition, we will introduce various well-known algorithms in a traditional distributed systems arena that can also be implemented in blockchain networks with some modifications, such as Paxos, Raft, and PBFT. We will also explore other mechanisms that have been introduced specifically for blockchain networks such as **proof of work** (**PoW**), **proof of stake** (**PoS**), and modified versions of traditional consensus such as **Istanbul Byzantine Fault Tolerant** (**IBFT**), which is a modified, '**block-chained**' version of the **Practical Byzantine Fault Tolerant** (**PBFT**) algorithm. Along the way, we'll cover the following topics:

- Introducing consensus
- Analysis and design
- Classification
- Algorithms
- Choosing an algorithm

Before we delve into specific algorithms, we first need to understand some fundamental concepts and an overview of the consensus problem.

Introducing consensus

The distributed consensus problem has been studied extensively in distributed systems research since the late 1970s. Distributed systems are classified into two main categories, namely, **message passing** and **shared memory**. In the context of blockchain, we are concerned with the message-passing type of distributed systems, where participants on the network communicate with each other via passing messages to each other. Consensus is the process that allows all processes in a network to agree on some specific value in the presence of faults.

As we saw in *Chapter 1*, *Blockchain 101*, there are different types of blockchain networks. In particular, two types, permissioned and public (permissionless), were discussed. The consensus problem can also be classified based on these two paradigms. For example, Bitcoin is a public blockchain. It runs PoW, also called **Nakamoto consensus**. In contrast, many permissioned blockchains tend to run variants of traditional or classical distributed consensus. A prime example is IBFT, which is a **blockchained** version of PBFT. Other examples include Tendermint, Casper FFG, and many variants of PBFT.

A common research area is to convert traditional (classical) distributed consensus mechanisms into their blockchain variants. Another area of interest is to analyze existing and new consensus protocols.

Fault tolerance

A fundamental requirement in a consensus mechanism is fault tolerance in a network, and it should continue to work even in the presence of faults. This naturally means that there must be some limit to the number of faults a network can handle since no network can operate correctly if many of its nodes fail. Based on the fault-tolerance requirement, consensus algorithms are also called fault-tolerant algorithms, and there are two types of fault-tolerant algorithms:

- **Crash fault-tolerance** (**CFT**), which covers only crash faults or, in other words, benign faults.
- **Byzantine fault-tolerance** (**BFT**), which deals with the type of faults that are arbitrary and can even be malicious.

Replication is a standard approach to improve the fault tolerance and availability of a network. Replication results in a synchronized copy of data across all nodes in a network. This means that even if some of the nodes become faulty, the overall system/network remains available due to the data being available on multiple nodes. There are two main types of replication techniques:

- **Active replication**, which is a type where each replica becomes a copy of the original state machine replica.
- **Passive replication**, which is a type where there is only a single copy of the state machine in the system kept by the primary node, and the rest of the nodes/replicas only maintain the state.

In the context of fault-tolerant consensus mechanisms, replication plays a vital role by introducing resiliency into the system.

State machine replication (**SMR**) is a de facto technique that is used to provide deterministic replication services to achieve fault tolerance in a distributed system. At an abstract level, a state machine is a mathematical model that is used to describe a machine that can be in different states but occupies one state at a time. A state machine stores a state of the system and transitions it to the next state as a result of the input received. As a result of state transition, an output is produced along with an updated state. The fundamental idea behind SMR can be summarized as follows:

1. All servers always start with the same initial state.
2. All servers receive requests in a **totally ordered** fashion (sequenced as generated from clients).
3. All servers produce the same deterministic output for the same input.

State machine replication is implemented under a primary/backup paradigm, where a primary node is responsible for receiving and broadcasting client requests. This broadcast mechanism is called **total order broadcast** or **atomic broadcast**, which ensures that backup or replica nodes receive and execute the same requests in the same sequence as the primary.

Consequently, this means that all replicas will eventually have the same state as the primary, thus resulting in achieving consensus. In other words, this means that total order broadcast and distributed consensus are equivalent problems; if you solve one, the other is solved too.

Now that we understand the basics of replication and fault tolerance, it is important to understand that fault tolerance works up to a certain threshold. For example, if a network has a vast majority of constantly failing nodes and communication links, it is not hard to understand that this type of network may not be as fault tolerant as we might like it to be. In other words, even in the presence of fault-tolerant measures, if there is a lack of resources on a network, the network may still not be able to provide the required level of fault tolerance. In some scenarios, it might be impossible to provide the required services due to a lack of resources in a system. In distributed computing, such impossible scenarios are researched and reported as impossibility results.

FLP impossibility

Impossibility results provide an understanding of whether a problem is solvable, and the minimum resources required to do so. If the problem is unsolvable, then these results give a clear understanding that a specific task cannot be accomplished, and no further research is necessary. From another angle, we can say that impossibility results (sometimes called unsolvability results) show that certain problems are not computable under insufficient resources. Impossibility results unfold deep aspects of distributed computing and enable us to understand why certain problems are difficult to solve and under what conditions a previously unsolved problem might be solved.

The problems that are not solvable under any conditions are known as **unsolvability results**. This result is known as the **FLP impossibility** result, which is a fundamental unsolvability result that states that in an asynchronous environment, the deterministic consensus is impossible, even if only one process is faulty.



FLP is named after the authors' names, Fischer, Lynch, and Patterson. Their result was presented in their paper:

Fischer, M.J., Lynch, N.A., and Paterson, M.S., 1982. *Impossibility of distributed consensus with one faulty process (No. MIT/LCS/TR-282)*. *Massachusetts Inst of Tech Cambridge lab for Computer Science*.

The paper is available at `https://apps.dtic.mil/dtic/tr/fulltext/u2/a132503.pdf`.

To circumvent **FLP impossibility**, several techniques have been introduced:

- **Failure detectors** can be seen as **oracles** associated with processors to detect failures. In practice, usually, this is implemented as a timeout mechanism.

- **Randomized algorithms** have been introduced to provide a probabilistic termination guarantee. The core idea behind the randomized protocols is that the processors in such protocols can make a random choice of decision value if the processor does not receive the required quorum of trusted messages. Eventually, with a very high probability, the entire network flips to a decision value.
- **Synchrony assumptions**, where additional synchrony and timing assumptions are made to ensure that the consensus algorithm gets adequate time to run so that it can make progress and terminate.

Now that we understand a fundamental impossibility result, let's look at another relevant result that highlights the unsolvability of consensus due to a lack of resources: that is, a **lower bound result**. We can think of lower bound as a minimum number of resources, for example, the number of processors or communication links required to solve a problem. The most common and fundamental of these results is the minimum number of processors required for consensus. These results are listed below, where *f* represents the number of failed nodes:

- In the case of **CFT**, at least $2f + 1$ number of nodes is required to achieve consensus.
- In the case of **BFT**, at least $3f + 1$ number of nodes is required to achieve consensus.

We have now covered the fundamentals of distributed consensus theory. Now, we'll delve a little bit deeper into the analysis and design of consensus algorithms.

Analysis and design

To analyze and understand a consensus algorithm, we need to define a model under which our algorithm will run. This model provides some assumptions about the operating environment of the algorithm and provides a way to intuitively study and reason about the various properties of the algorithm.

In the following sections, we'll describe a model that is useful for describing and analyzing consensus mechanisms.

Model

Distributed computing systems represent different entities in the system under a computational model. This computational model is a beneficial way of describing the system under some system assumptions. A computational model represents processes, network conditions, timing assumptions, and how all these entities interact and work together. We will now look at this model in detail and introduce all objects one by one.

Processes

Processes communicate with each other by passing messages to each other. Therefore, these systems are called message-passing distributed systems. There is another class, called shared memory, which we will not discuss here, as all blockchain systems are message-passing systems.

Timing assumptions

There are also some timing assumptions that are made when designing consensus algorithms:

- **Synchrony**: In synchronous systems, there is a known upper bound on communication and processor delays. Synchronous algorithms are designed to be run on synchronous networks. At a fundamental level, in a synchronous system, a message sent by a processor to another is received by the receiver in the same communication round as it is sent.
- **Asynchrony**: In asynchronous systems, there is no upper bound on communication and processor delays. In other words, it is impossible to define an upper bound for communication and processor delays in asynchronous systems. Asynchronous algorithms are designed to run on asynchronous networks without any timing assumptions. These systems are characterized by the unpredictability of message transfer (communication) delays and processing delays. This scenario is common in large-scale geographically dispersed distributed systems and systems where the input load is unpredictable.
- **Partial synchrony**: In this model, there is an upper bound on the communication and processor delays, however, this upper bound is not known to the processors. An eventually synchronous system is a type of partial synchrony, which means that the system becomes synchronous after an instance of time called **global stabilization time** (**GST**). GST is not known to the processors. Generally, partial synchrony captures the fact that, usually, the systems are synchronous, but there are arbitrary but bounded asynchronous periods. Also, the system at some point is synchronous for long enough that processors can decide (achieve agreement) and terminate during that period.

Now that we understand the fundamentals of the distributed consensus theory, let's look at two main classes of consensus algorithms. This categorization emerged after the invention of Bitcoin. Prior to Bitcoin, there was a long history of research in distributed consensus protocols.

Classification

The consensus algorithms can be classified into two broad categories:

- **Traditional**: Voting-based consensus. Traditional voting-based consensus has been researched in distributed systems for many decades. Many fundamental results and a lot of ground-breaking work have already been produced in this space. Algorithms like **Paxos** and **PBFT** are prime examples of such types of algorithms. Traditional consensus can also be called fault-tolerant distributed consensus.
- **Lottery-based**: Nakamoto and post-Nakamoto consensus. Lottery-based or Nakamoto-type consensus was first introduced with Bitcoin. This class can also be simply called blockchain consensus.



Note that they are distinguished by the time period in which they've been invented; traditional protocols were developed before the introduction of Bitcoin, and lottery-based protocols were introduced with Bitcoin.

The fundamental requirements of consensus algorithms boil down to **Safety** and **Liveness** conditions. The **Safety** requirement generally means that nothing bad happens, and is usually based on some safety requirements of the algorithms, such as agreement, validity, and integrity. There are usually three properties within this class of requirements, which are listed as follows:

- **Agreement**. The agreement property requires that no two processes decide on different values.
- **Validity**. Validity states that if a process has decided on a value, that value must have been proposed by a process. In other words, the decided value is always proposed by an honest process and has not been created out of thin air.
- **Integrity**. A process must decide only once.

The **Liveness** requirement generally means that something good eventually happens. It means that the protocol can make progress even if the network conditions are not ideal. It usually has one requirement:

- **Termination**. This liveness property states that each honest node must eventually decide on a value.

With this, we have covered the classification and requirements of consensus algorithms. In the next section, we'll introduce various consensus algorithms that we can evaluate using the consensus models covered in this section.

Algorithms

In this section, we will discuss the key algorithms in detail. We'll be looking at the two main types of fault-tolerant algorithms, which are classified based on the fault tolerance level they provide: CFT or BFT.

CFT algorithms

We'll begin by looking at some algorithms that solve the consensus problem with crash fault tolerance. One of the most fundamental algorithms in this space is Paxos.

Paxos

Leslie Lamport developed Paxos. It is the most fundamental distributed consensus algorithm, allowing consensus over a value under unreliable communications. In other words, Paxos is used to build a reliable system that works correctly, even in the presence of faults. There are many other protocols that have since emerged from basic Paxos, such as Multi-Paxos, Fast Paxos, and Cheap Paxos.



Paxos was proposed first in 1989 and then later, more formally, in 1998, in the following paper:

Lamport, L., 1998. "The part-time parliament", *ACM Transactions on Computer Systems (TOCS)*, 16(2): pp. 133-169.

The paper is available here: `https://lamport.azurewebsites.net/pubs/lamport-paxos.pdf`.

Paxos works under an asynchronous network model and supports the handling of only benign failures. This is not a Byzantine fault-tolerant protocol. However, later, a variant of Paxos was developed that provides BFT.



The paper in which a Byzantine fault-tolerant version of Paxos is described is available here:

Lamport, L., 2011, September. *Byzantizing Paxos by refinement*. *International Symposium on Distributed Computing* (pp. 211-224). Springer, Berlin, Heidelberg.

A link to the paper is available here: `http://lamport.azurewebsites.net/pubs/web-byzpaxos.pdf`.

Paxos makes use of *2f + 1* processes to ensure fault tolerance in a network where processes can crash fault, that is, experience benign failures. Benign failure means either the loss of a message or a process stops. In other words, Paxos can tolerate one crash failure in a three-node network.

Paxos is a two-phase protocol. The first phase is called the *prepare* phase, and the next phase is called the *accept* phase. Paxos has a proposer and acceptors as participants, where the proposer is the replicas or nodes that propose the values, and acceptors are the nodes that accept the values.

The Paxos protocol assumes an asynchronous message-passing network with less than 50% of crash faults. As usual, the critical properties of the Paxos consensus algorithm are safety and liveness. Under safety, we have:

- **Agreement**, which specifies that no two different values are agreed on.
- **Validity**, which means that only the proposed values are decided.

Under liveness, we have:

- **Termination**, which means that, eventually, the protocol is able to decide and terminate.

Processes can assume different roles, which are listed as follows:

- **Proposers** are elected leader(s) that can propose a new value to be decided.
- **Acceptors** participate in the protocol as a means to provide a majority decision.
- **Learners** are nodes that just observe the decision process and value.

A single process in a Paxos network can assume all three roles.

The key idea behind Paxos is that the proposer node proposes a value, which is considered final only if a majority of the acceptor nodes accept it. The learner nodes also learn this final decision.

Paxos can be seen as a protocol that is quite similar to the two-phase commit protocol. **Two-phase commit** (**2PC**) is a standard atomic commitment protocol to ensure that transactions are committed in distributed databases only if *all* participants agree to commit. In contrast with the two-phase commit, Paxos introduced ordering (sequencing to achieve total order) of the proposals and majority-based acceptance of the proposals instead of expecting all nodes to agree (to allow progress even if some nodes fail). Both improvements contribute toward ensuring the safety and liveness of the Paxos algorithm.

We'll now describe how the Paxos protocol works step by step:

1. The proposer proposes a value by broadcasting a message, `<prepare(n)>`, to all acceptors.
2. Acceptors respond with an acknowledgment message if proposal *n* is the highest that the acceptor has responded to so far. The acknowledgment message `<ack(n, v, s)>` consists of three variables, where *n* is the proposal number, *v* is the proposal value of the highest numbered proposal the acceptor has accepted so far, and *s* is the sequence number of the highest proposal accepted by the acceptor so far. This is where acceptors agree to commit the proposed value. The proposer now waits to receive acknowledgment messages from the majority of the acceptors indicating the **chosen** value.
3. If the majority is received, the proposer sends out the "accept" message `<accept(n, v)>` to the acceptors.
4. If the majority of the acceptors accept the proposed value (now the "accept" message), then it is decided: that is, an agreement is achieved.
5. Finally, in the learning phase, acceptors broadcast the "accepted" message `<accepted(n, v)>` to the proposer. This phase is necessary to disseminate which proposal has been finally accepted. The proposer then informs all other learners of the decided value. Alternatively, learners can learn the decided value via a message that contains the accepted value (decision value) multicast by acceptors.

We can visualize this process in the following diagram:

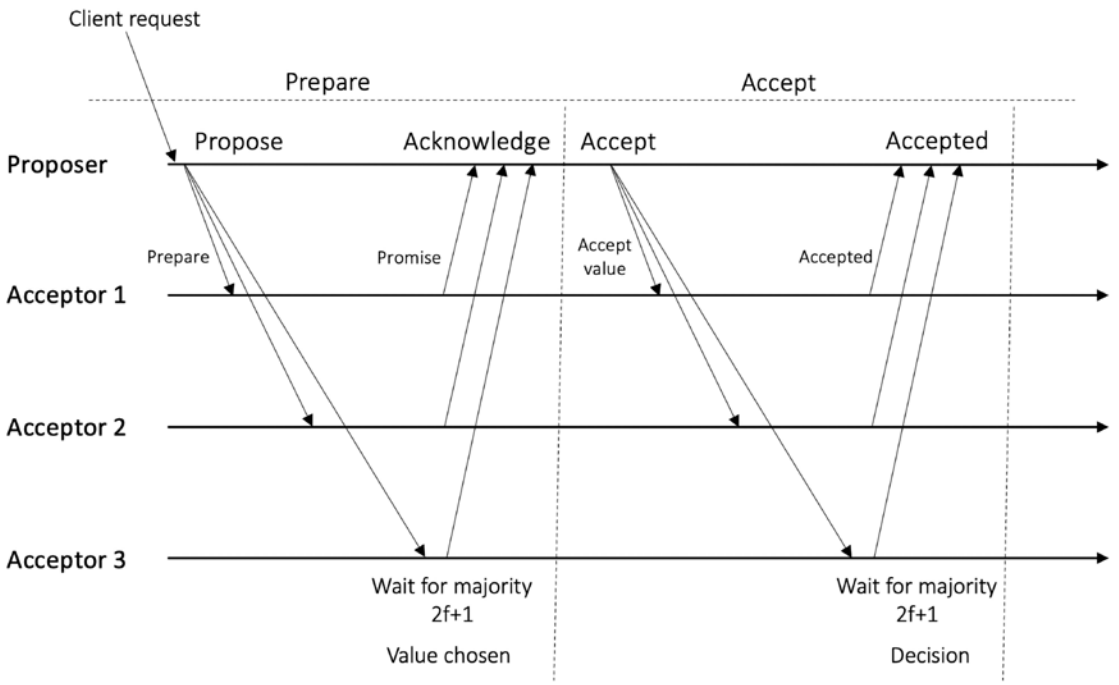


Figure 5.1: How Paxos works

A natural question arises about how Paxos ensures its safety and liveness guarantees. Paxos, at its core, is quite simple, yet it achieves all these properties efficiently. The actual proofs for the correctness of Paxos are quite in-depth and are not the subject of this chapter. However, the rationale behind each property is presented as follows:

- **Agreement** is ensured by enforcing that only one proposal can win votes from a majority of the acceptors.
- **Validity** is ensured by enforcing that only genuine proposals are decided. In other words, no value is committed unless it is proposed in the proposal message first.
- **Liveness,** or termination, is guaranteed by ensuring that at some point during the protocol execution, eventually, there is a period during which there is only one fault-free proposer.

Even though the Paxos algorithm is quite simple at its core, it is seen as challenging to understand, and many academic papers have been written to explain it. This slight problem, however, has not prevented it from being implemented in many production networks, such as Google's Spanner, as it has proven to be the most efficient protocol to solve the consensus problem. Nevertheless, there have been attempts to create alternative easy-to-understand algorithms. Raft is such an attempt to create an easy-to-understand CFT algorithm.

Raft

The Raft protocol is a CFT consensus mechanism developed by Diego Ongaro and John Ousterhout at Stanford University. In Raft, the leader is always assumed to be honest.

At a conceptual level, it is a replicated log for a **replicated state machine** (**RSM**) where a unique leader is elected every "term" (time division) whose log is replicated to all follower nodes.

Raft is composed of three subproblems:

- **Leader election** (a new leader election in case the existing one fails)
- **Log replication** (leader-to-follower log sync)
- **Safety** (no conflicting log entries (index) between servers)

The Raft protocol ensures **election safety** (that is, only one winner each election term), **leader append-only**, **log matching**, **leader completeness**, **liveness** (that is, some candidate must eventually win), and **state machine safety**.

Each server in Raft can have either a **follower**, **leader**, or **candidate** state. At a fundamental level, the protocol is quite simple and can be described simply by the following sequence:

1. First, the node starts up.
2. After this, the leader election process starts. Once a node is elected as leader, all changes go through that leader.
3. Each change is entered into the node's log.
4. Log entry remains uncommitted until the entry is replicated to follower nodes and the leader receives write confirmation votes from a majority of the nodes; then it is committed locally.

5. The leader notifies the followers regarding the committed entry.
6. Once this process ends, agreement is achieved.

The state transition of the Raft algorithm can be visualized in the following diagram:

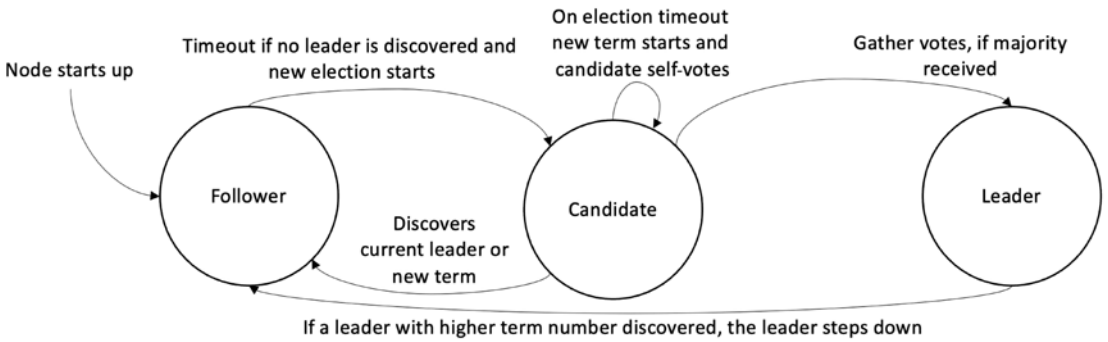


Figure 5.2: Raft state transition

We saw earlier that data is eventually replicated across all nodes in a consensus mechanism. In Raft, the log (data) is eventually replicated across all nodes. The replication logic can be visualized in the following diagram. The aim of log replication is to synchronize nodes with each other:

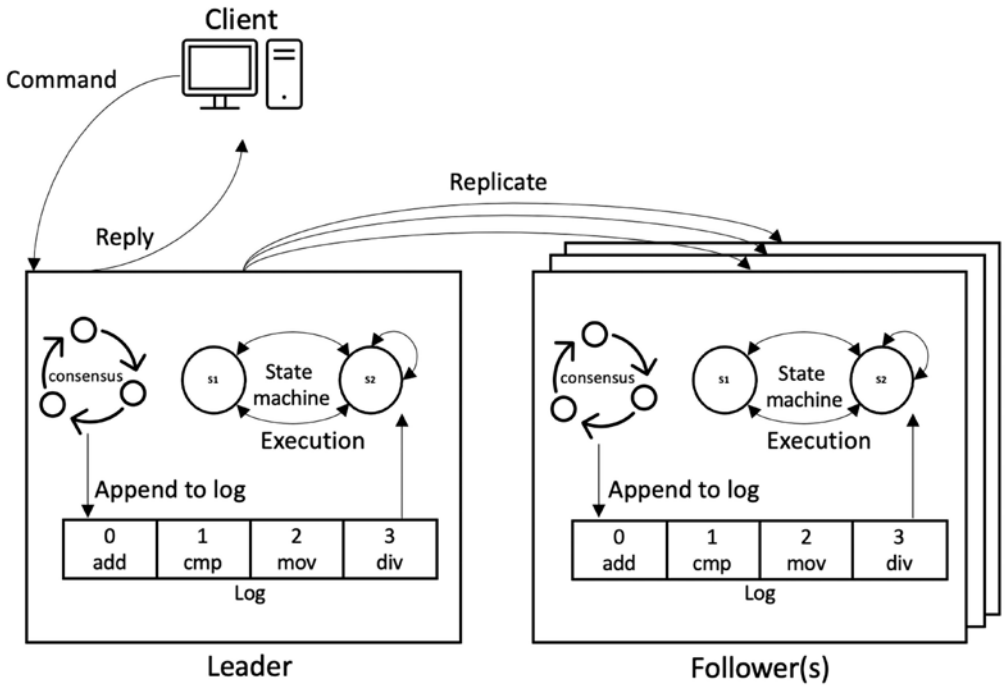


Figure 5.3: Log replication mechanism

Log replication is a simple mechanism. As shown in the preceding diagram, the leader is responsible for log replication. Once the leader has a new entry in its log, it sends out the requests to replicate to the follower nodes. When the leader receives enough confirmation votes back from the follower nodes indicating that the replicate request has been accepted and processed by the followers, the leader commits that entry to its local state machine. At this stage, the entry is considered committed.

With this, our discussion on CFT algorithms is complete. Now we'll introduce Byzantine fault-tolerant algorithms, which have been a research area for many years in distributed computing.

BFT algorithms

In this section, we'll introduce the mechanisms that were developed to solve the Byzantine generals (consensus in the presence of faults) problem.

Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (**PBFT**) was developed in 1999 by Miguel Castro and Barbara Liskov. PBFT, as the name suggests, is a protocol developed to provide consensus in the presence of Byzantine faults. This algorithm demonstrated for the first time that PBFT is possible.

PBFT comprises three subprotocols:

- **Normal operation** subprotocol refers to a scheme that is executed when everything is running normally, and no errors are in the system.
- **View change** is a subprotocol that runs when a faulty leader node is detected in the system.
- **Checkpointing** is another subprotocol, which is used to discard old data from the system.

The protocol runs in rounds where, in each round, an elected leader, or primary, node handles the communication with the client. In each round, the protocol progresses through three phases or steps. These phases are pre-prepare, prepare, and commit.

The participants in the PBFT protocol are called replicas, where one of the replicas becomes primary as a leader in each round, and the rest of the nodes act as backups. Each replica maintains a local state comprising three main elements: a service state, a message log, and a number representing the replica's current view.

PBFT is based on the SMR protocol introduced earlier. Here, each node maintains a local log, and the logs are kept in sync with each other via the consensus protocol: that is, PBFT.

As we saw earlier, in order to tolerate Byzantine faults, the minimum number of nodes required is $n = 3f + 1$, where n is the number of nodes and f is the number of faulty nodes. PBFT ensures BFT as long as the number of nodes in a system stays $n >= 3f + 1$.

Now we'll discuss the phases mentioned above one by one, starting with pre-prepare.

Pre-prepare: The main purpose of this phase is to assign a unique sequence number to the request. We can think of it as an orderer. This is the first phase in the protocol, where the primary node, or **primary**, receives (accepts) a request from the client. The primary node assigns a sequence number to the request. It then sends the pre-prepare message with the request to all backup replicas.

When the pre-prepare message is received by the backup replicas, it checks a number of things to ensure the validity of the message:

1. First, whether the digital signature is valid.
2. After this, whether the current view number is valid.
3. Then, whether the sequence number of the operation's request message is valid.
4. Finally, whether the digest / hash of the operation's request message is valid.

If all of these elements are valid, then the backup replica accepts the message. After accepting the message, it updates its local state and progresses toward the prepare phase.

Prepare: This phase ensures that honest replicas/nodes in the network agree on the total order of requests within a view. Note that the pre-prepare and prepare phases together provide the total order to the messages. A prepare message is sent by each backup to all other replicas in the system. Each backup waits for at least *2f+1* prepare messages to be received from other replicas. They also check whether the prepare message contains the same view number, sequence number, and message digest values. If all these checks pass, then the replica updates its local state and progresses toward the commit phase.

Commit: This phase ensures that honest replicas/nodes in the network agree on the total order of requests across views. In the commit phase, each replica sends a commit message to all other replicas in the network. The same as the prepare phase, replicas wait for *2f+1* commit messages to arrive from other replicas. The replicas also check the view number, sequence number, and message digest values. If they are valid for *2f+1* commit messages received from other replicas, then the replica executes the request, produces a result, and finally, updates its state to reflect a commit. If there are already some messages queued up, the replica will execute those requests first before processing the latest sequence numbers. Finally, the replica sends the result to the client in a reply message. The client accepts the result only after receiving *2f+1* reply messages containing the same result.

In summary, the protocol works as follows:

1. A client sends a request to invoke a service operation in the primary.
2. The primary multicasts the request to the backups.
3. Replicas execute the request and send a reply to the client.
4. The client waits for replies from different replicas with the same result; this is the result of the operation.

The primary purpose of these phases is to achieve consensus, where each phase is responsible for a critical part of the consensus mechanism, which, after passing through all phases, eventually ends up achieving consensus.

The normal view of the PBFT protocol can be visualized as follows:

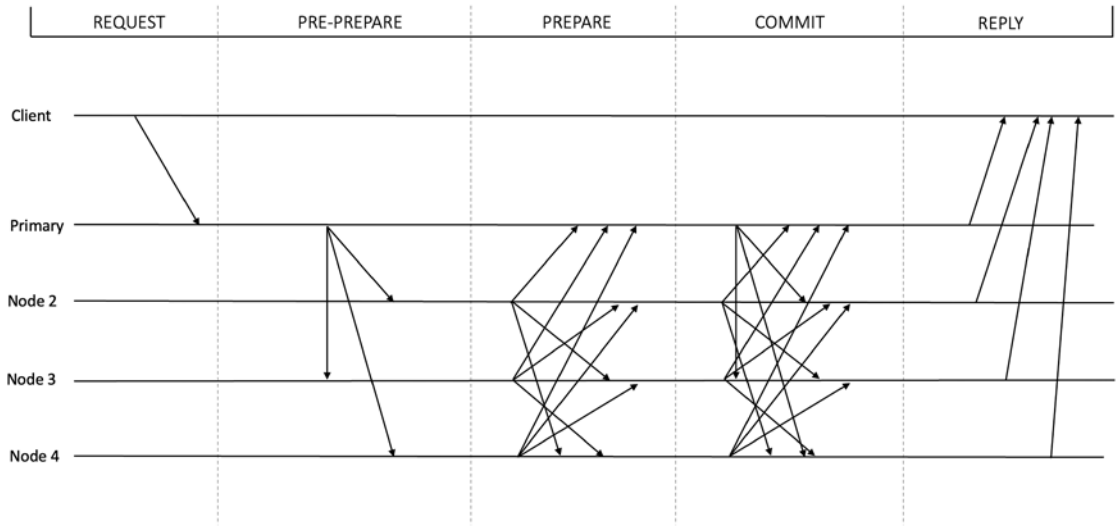


Figure 5.4: PBFT protocol

During the execution of the protocol, the integrity of the messages and protocol operations must be maintained to provide an adequate level of security and assurance. This is maintained using digital signatures. In addition, certificates are used to ensure the adequate majority of participants (nodes).



Do not confuse these certificates with digital certificates commonly used in **public key infrastructure** (**PKI**), or on websites and IT infrastructures to secure assets such as servers.

Certificates

Certificates in PBFT protocols are used to demonstrate that at least *2f+1* nodes have stored the required information. For example, if a node has collected *2f+1* prepare messages, then combining it with the corresponding pre-prepare message with the same view, sequence, and request represents a certificate, called a prepared certificate. Similarly, a collection of *2f+1* commit messages is called a commit certificate. There are also a number of variables that the PBFT protocol maintains to execute the algorithm. These variables and their meanings are listed as follows:

State variable	Explanation
`v`	View number
`m`	Latest request message
`n`	Sequence number of the message
`h`	Hash of the message
`i`	Index number
`C`	Set of all checkpoints

P	Set of all pre-prepare and corresponding prepare messages
O	Set of pre-prepare messages without corresponding request messages

We can now look at the types of messages and their formats, which becomes quite easy to understand if we refer to the preceding variables table.

Types of messages

The PBFT protocol works by exchanging several messages. A list of these messages is presented as follows with their format and direction.

The following table contains message types and relevant details:

Message	From	To	Format	Signed by
Request	Client	Primary	`<REQUEST, m>`	Client
Pre-prepare	Primary	Backups	`<PRE-PREPARE, v, n, h>`	Client
Prepare	Replica	Replicas	`<PREPARE, v, n, h, i>`	Replica
Commit	Replica	Replicas	`<COMMIT, v, n, h, i>`	Replica
Reply	Replica	Client	`<REPLY, r, i>`	Replica
View change	Replica	Replicas	`<VIEWCHANGE, v+1, n, C, P, i>`	Replica
New view	Primary replica	Replicas	`<NEWVIEW, v + 1, v, O>`	Replica
Checkpoint	Replica	Replicas	`<CHECKPOINT, n, h, i>`	Replica

Note that all these messages are signed. Let's look at some specific message types that are exchanged during the PBFT protocol.

View change occurs when a primary is suspected to be faulty. This phase is required to ensure protocol progress. With the view change subprotocol, a new primary is selected, which then starts normal mode operation again. The new primary is selected in a round-robin fashion.

When a backup replica receives a request, it tries to execute it after validating the message, but for some reason, if it does not execute it for a while, the replica times out and initiates the view change subprotocol.

In the view change protocol, the replica stops accepting messages related to the current view and updates its state to `VIEW-CHANGE`. The only messages it can receive in this state are `CHECKPOINT` messages, `VIEW-CHANGE` messages, and `NEW-VIEW` messages. After that, it sends a `VIEW-CHANGE` message with the next view number to all replicas.

When this message arrives at the new primary, the primary waits for at least *2f* `VIEW-CHANGE` messages for the next view. If at least *2f* `VIEW-CHANGE` messages are received it broadcasts a new view message to all replicas and progresses toward running normal operation mode once again.

When other replicas receive a `NEW-VIEW` message, they update their local state accordingly and start normal operation mode.

The algorithm for the view change protocol is shown as follows:

1. Stop accepting pre-prepare, prepare, and commit messages for the current view.
2. Create a set of all the certificates prepared so far.
3. Broadcast a `VIEW-CHANGE` message with the next view number and a set of all the prepared certificates to all replicas.

The view change protocol can be visualized in the following diagram:

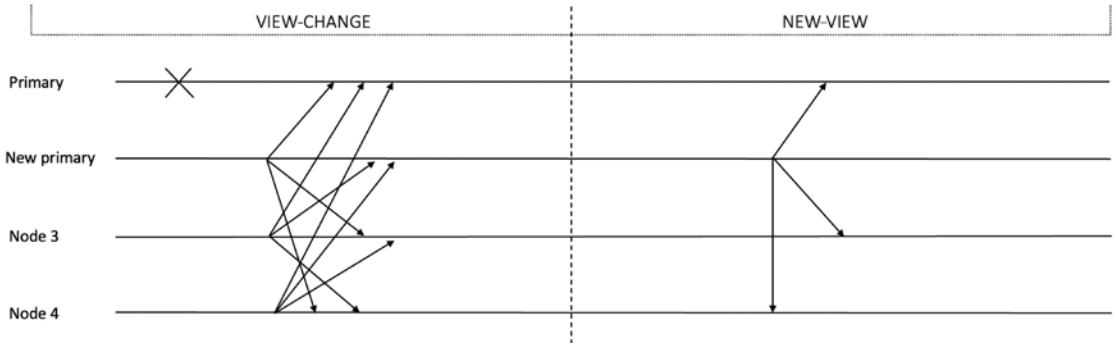


Figure 5.5: View change subprotocol

The view change subprotocol is a mechanism to achieve liveness. Three smart techniques are used in this subprotocol to ensure that, eventually, there is a time when the requested operation executes:

1. A replica that has broadcast the `VIEW-CHANGE` message waits for *2f+1* `VIEW-CHANGE` messages and then starts its timer. If the timer expires before the node receives a `NEW-VIEW` message for the next view, the node will start the view change for the next sequence but will increase its timeout value. This will also occur if the replica times out before executing the new unique request in the new view.
2. As soon as the replica receives *f+1* `VIEW-CHANGE` messages for a view number greater than its current view, the replica will send the `VIEW-CHANGE` message for the smallest view it knows of in the set so that the next view change does not occur too late. This is also the case even if the timer has not expired; it will still send the view change for the smallest view.
3. As the view change will only occur if at least *f+1* replicas have sent the `VIEW-CHANGE` message, this mechanism ensures that a faulty primary cannot indefinitely stop progress by successively requesting view changes.

Checkpointing is another crucial subprotocol. It is used to discard old messages in the log of all replicas. With this, the replicas agree on a stable checkpoint that provides a snapshot of the global state at a certain point in time. This is a periodic process carried out by each replica after executing the request and marking that as a checkpoint in its log. A variable called **low watermark** (in PBFT terminology) is used to record the sequence number of the last stable checkpoint. This checkpoint is then broadcast to other nodes. As soon as a replica has at least *2f+1* checkpoint messages, it saves these messages as proof of a stable checkpoint. It discards all previous **pre-prepare, prepare**, and **commit** messages from its logs.

PBFT is indeed a revolutionary protocol that has opened a new research field of PBFT protocols. The original PBFT does have many strengths, but it also has some limitations. We discuss most of the commonly cited strengths and limitations in the following table.

Strengths	Limitations
PBFT provides immediate and deterministic transaction finality. This is in contrast with the PoW protocol, where several confirmations are required to finalize a transaction with high probability.	Node scalability is quite low. Usually, smaller networks of few nodes (10s) are suitable for use with PBFT due to its high communication complexity. This is the reason it is more suitable for consortium networks instead of public blockchains.
PBFT is energy efficient as compared to PoW, which consumes a tremendous amount of electricity.	Sybil attacks can be carried out on a PBFT network, where a single entity can control many identities to influence the voting and, subsequently, the decision. However, the fix is trivial, and, in fact, this attack is not very practical in consortium networks because all identities are known on the network. This problem can be addressed simply by increasing the number of nodes in the network.

In the traditional client-server model, PBFT works well; however, in the case of blockchain, directly implementing PBFT in its original state may not work correctly. This is because PBFT's original design was not developed for blockchain. This research resulted in IBFT and PBFT implementation in Hyperledger Sawtooth, Hyperledger Fabric, and other blockchains. In all these scenarios, some changes have been made in the core protocol to ensure that they're compatible with the blockchain environment.



More details on PBFT in Hyperledger Sawtooth can be found here: `https://github.com/hyperledger/sawtooth-rfcs/blob/master/text/0019-pbft-consensus.md`.

Istanbul Byzantine Fault Tolerance

IBFT was developed by AMIS Technologies as a variant of PBFT suitable for blockchain networks. It was presented in EIP 650 for the Ethereum blockchain.

Let's first discuss the primary differences between the PBFT and IBFT protocols. They are as follows:

- In IBFT, validator membership can be changed dynamically, and validators voted in and out as required. It is in contrast with the original PBFT, where the validator nodes are static.
- There are two types of nodes in an IBFT network: nodes and validators. Nodes are synchronized with the blockchain without participating in the IBFT consensus process.
- IBFT relies on a more straightforward structure of view change (round change) messages as compared to PBFT.

- In contrast with PBFT, in IBFT, there is no concrete concept of checkpoints. However, each block can be considered an indicator of the progress so far (the chain height).
- There is no garbage collection in IBFT.

Now we've covered the main points of comparison between the two BFT algorithms, we'll examine how the IBFT protocol runs, and its various phases.

IBFT assumes a network model under which it is supposed to run. The model is composed of at least *3f+1* processes (standard BFT assumption), a partially synchronous message-passing network, and secure cryptography. The protocol runs in rounds. It has three phases: *pre-prepare*, *prepare*, and *commit*. In each round, usually, a new leader is elected based on a round-robin mechanism. The following flowchart shows how the IBFT protocol works:

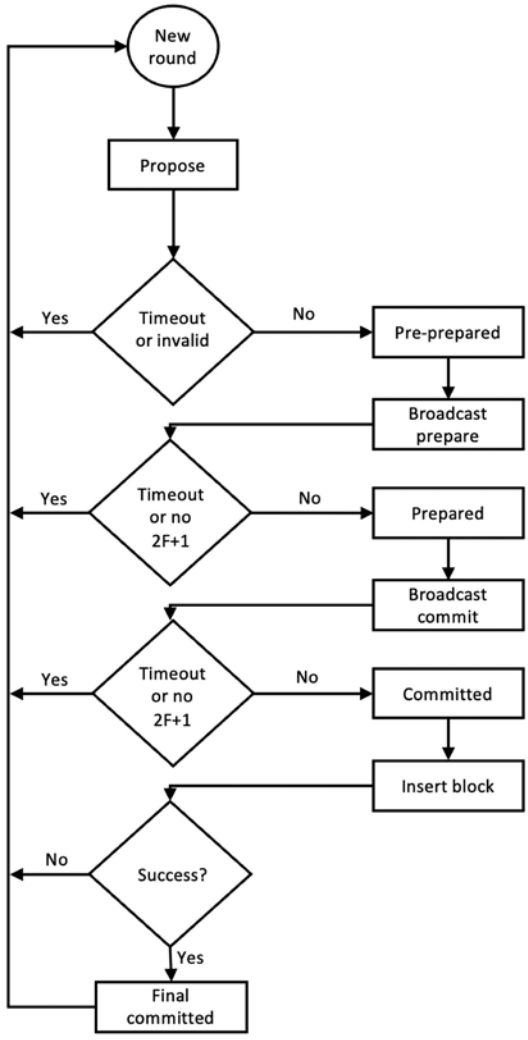


Figure 5.6: IBFT flowchart

We'll discuss this process step by step as follows:

1. The protocol starts with a new round. In the new round, the selected proposer broadcasts a proposal (block) as a pre-prepare message.
2. The nodes that receive this pre-prepare message validate the message and accept it if it is a valid message. The nodes also then set their state to pre-prepared.
3. At this stage, if a timeout occurs, or a proposal is seen as invalid by the nodes, they will initiate a round change. The normal process then begins again with a proposer, proposing a block.
4. Nodes then broadcast the prepare message and wait for *2f+1* prepare messages to be received from other nodes. If the nodes do not receive *2f+1* messages in time, then they time out, and the round change process starts. The nodes then set their state to prepared after receiving *2f+1* messages from other nodes.
5. Finally, the nodes broadcast a commit message and wait for *2f+1* messages to arrive from other nodes. If they are received, then the state is set to committed, otherwise, a timeout occurs, and the round change process starts.
6. Once committed, block insertion is tried. If it succeeds, the protocol proceeds to the final committed state and, eventually, a new round starts. If insertion fails for some reason, the round change process triggers. Again, nodes wait for *2f+1* round change messages, and if the threshold of the messages is received, then round change occurs.

Now that we've understood the flow of IBFT, let's now further explore which states it maintains and how.

Consensus states

IBFT is an SMR algorithm. Each validator maintains a state machine replica in order to reach block consensus, that is, agreement. These states are listed as follows with an explanation:

- **New round**: In this state, a new round of the consensus mechanism starts, and the selected proposer sends a new block proposal to other validators. In this state, all other validators wait for the `PRE-PREPARE` message.
- **Pre-prepared**: A validator transitions to this state when it has received a `PRE-PREPARE` message and broadcasts a `PREPARE` message to other validators. The validator then waits for *2f + 1* `PREPARE` or `COMMIT` messages.
- **Prepared**: This state is achieved by a validator when it has received *2f+1* prepare messages and has broadcast the commit messages. The validator then awaits *2f+1* `COMMIT` messages to arrive from other validators.
- **Committed**: The state indicates that a validator has received *2f+1* `COMMIT` messages. The validator at this stage can insert the proposed block into the blockchain.
- **Final committed**: This state is achieved by a validator when the newly committed block is inserted successfully into the blockchain. At this state, the validator is also ready for the next round of consensus.
- **Round change**: This state indicates that the validators are waiting for *2f+1* round change messages to arrive for the newly proposed new round number.

The IBFT protocol can be visualized using a diagram, similar to PBFT, as follows:

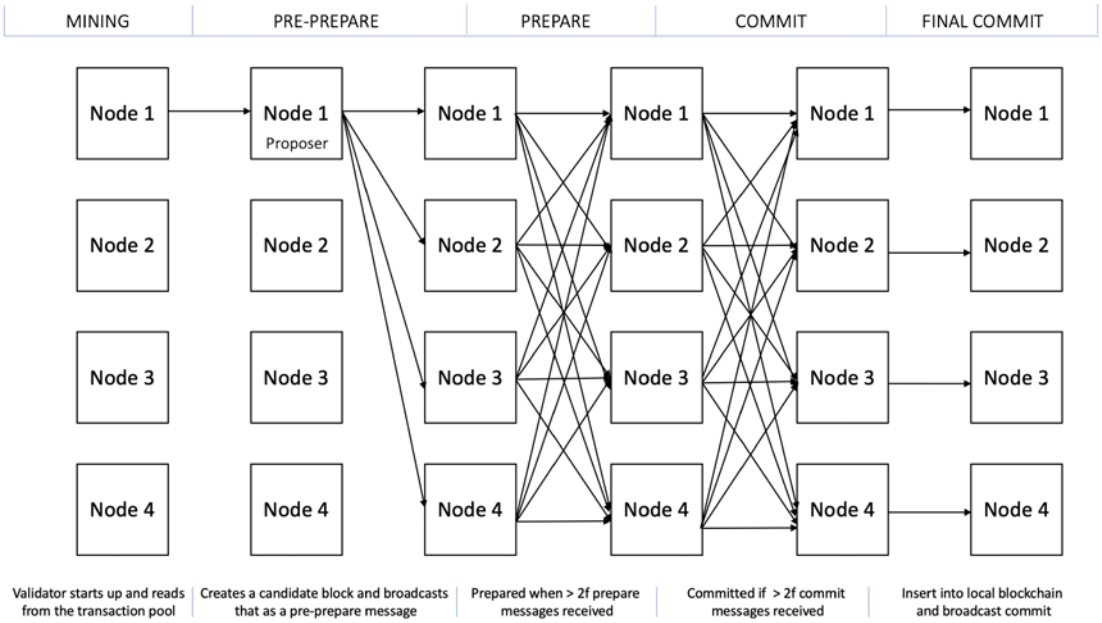


Figure 5.7: IBFT

An additional mechanism that makes IBFT quite appealing is its validator management mechanism. By using this mechanism, validators can be added or removed by voting between members of the network. This is quite a useful feature and provides the right level of flexibility when it comes to managing validators efficiently, instead of manually adding or removing validators from the validator set.



IBFT has been implemented in several blockchains. Sample implementations include Quorum and Celo. A Quorum implementation is available at the following link: `https://github.com/ConsenSys/quorum`.

IBFT, with some modifications, has also been implemented in the Celo blockchain, which is available at `https://github.com/celo-org/celo-blockchain`.

Several other consensus algorithms have been inspired by PBFT and have emerged as a result of deep interest in blockchain research. One such algorithm is Tendermint.

Tendermint

Tendermint is another variant of PBFT. It is inspired by both the DLS and PBFT protocols. Tendermint also makes use of the SMR approach to provide fault-tolerant replication. As we saw before, state machine replication is a mechanism that allows synchronization between replicas/nodes of the network.



The DLS protocol is a consensus mechanism named after its authors (Dwork, Lynch, Stockmeyer). This protocol consists of two initial rounds. This process of rounds with appropriate message exchange ensures that agreement is eventually achieved on a proposed or default value. As long as the number of nodes in the system is more than *3f*, this protocol achieves consensus.

Traditionally, a consensus mechanism was used to run with a small number of participants, and thus performance and scalability were not a big concern. However, with the advent of blockchain, there is a need to develop algorithms that can work on wide-area networks and in asynchronous environments. Research into these areas of distributed computing is not new and especially now, due to the rise of cryptocurrencies and blockchain, the interest in these research topics has grown significantly in the last few years.

The Tendermint protocol works by running rounds. In each round, a leader is elected, which proposes the next block. Also note that in Tendermint, the round change or view change process is part of the normal operation, as opposed to PBFT, where view change only occurs in the event of errors, that is, a suspected faulty leader. Tendermint works similarly to PBFT, where three phases are required to achieve a decision. Once a round is complete, a new round starts with three phases and terminates when a decision is reached. A key innovation in Tendermint is the design of a new termination mechanism. As opposed to other PBFT-like protocols, Tendermint has developed a more straightforward mechanism, which is similar to PBFT-style normal operation. Instead of having two subprotocols for normal mode and view change mode (recovery in event of errors), Tendermint terminates without any additional communication costs.

We saw earlier, in the introduction, that each consensus model is studied, developed, and run under a system model with some assumptions about the system. Tendermint is also designed with a system model in mind. Now we'll define and introduce each element of the system model:

- **Processes**: A process is the fundamental key participant of the protocol. It is also called a replica (in PBFT traditional literature), a node, or merely a process. Processes can be correct or honest. Processes can also be faulty or Byzantine. Each process possesses some voting power. Also, note that processes are not necessarily connected directly; they are only required to connect loosely or just with their immediate subset of processes/nodes. Processes have a local timer that they use to measure timeout.
- **Network model**: The network model is a network of processes that communicate using messages. In other words, the network model is a set of processes that communicate using message passing. Particularly, the gossip protocol is used for communication between processes. The standard assumption of $n >= 3f + 1$ BFT is also taken into consideration. This means that the protocol operates correctly as long as the number of nodes in the network is more than *3f*, where *f* is the number of faulty nodes. This implies that, at a minimum, there have to be four nodes in a network to tolerate Byzantine faults.

- **Timing assumptions**: Under the network model, Tendermint assumes a partially synchronous network. This means that there is an unknown bound on the communication delay, but it only applies after an unknown instance of time called GST. Practically, this means that the network is eventually synchronous, meaning the network becomes synchronous after an unknown finite point in time.
- **Security and cryptography**: It is assumed that the public key cryptography used in the system is secure and the impersonation or spoofing of accounts/identities is not possible. The messages on the network are authenticated and verified via digital signatures. The protocol ignores any messages with an invalid digital signature.
- **State machine replication**: To achieve replication among the nodes, the standard SMR mechanism is used. One key observation that is fundamental to the protocol is that in SMR, it is ensured that all replicas on the network receive and process the same sequence of requests. As noted in the Tendermint paper, agreement and order are two properties that ensure that all requests are received by replicas and order ensures that the sequence in which the replicas have received requests is the same. Both requirements ensure total order in the system. Also, Tendermint ensures that requests themselves are valid and have been proposed by the clients. In other words, only valid transactions are accepted and executed on the network.

There are three fundamental consensus properties that Tendermint solved. As we discussed earlier in the chapter, generally in a consensus problem, there are safety and liveness properties that are required to be met. Similarly, in Tendermint, these safety and liveness properties consist of agreement, termination, and validity:

- **Agreement**: No two correct processes decide on different values.
- **Termination**: All correct processes eventually decide on a value.
- **Validity**: A decided-upon value is valid, that is, it satisfies the predefined predicate denoted `valid()`.

State transition in Tendermint is dependent on the messages received and timeouts. In other words, the state is changed in response to messages received by a processor or in the event of timeouts. The timeout mechanism ensures liveness and prevents endless waiting. It is assumed that, eventually, after a period of asynchrony, there will be a round or communication period during which all processes can communicate in a timely fashion, which will ensure that processes eventually decide on a value.

The following diagram depicts the Tendermint protocol at a high level:

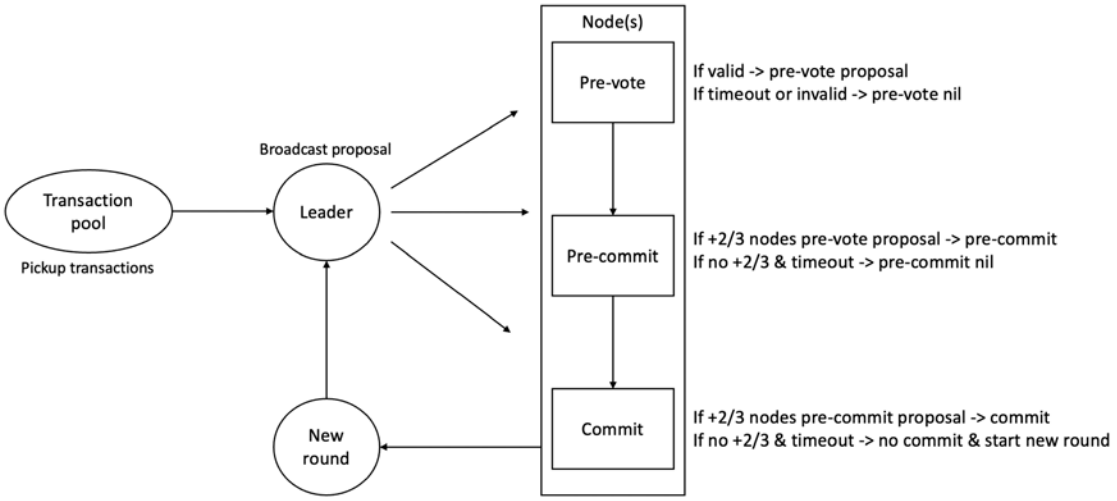


Figure 5.8: Tendermint high-level overview

Tendermint works in rounds and each round comprises phases: **propose**, **pre-vote**, **pre-commit**, and **commit**:

1. Every round starts with a proposal value being proposed by a proposer. The proposer can propose any new value at the start of the first round for each height.
2. After the first round, any subsequent rounds will have the proposer, which proposes a new value only if there is no valid value present, that is, null. Otherwise, the possible decision value is proposed, which has already been locked in a previous round. The proposal message also includes a value of a valid round, which denotes the last round in which there was a valid value updated.

3. The proposal is accepted by a correct process only if:
    - The proposed value is valid
    - The process has not locked on a round
    - Or the process has a value locked
4. If the preceding conditions are met, then the correct process will accept the proposal and send a `PRE-VOTE` message.
5. If the preceding conditions are not met, then the process will send a `PRE-VOTE` message with a `nil` value.
6. In addition, there is also a timeout mechanism associated with the proposal phase, which initiates timeout if a process has not sent a `PRE-VOTE` message in the current round, or the timer expires in the proposal stage.
7. If a correct process receives a proposal message with a value and $2f + 1$ pre-vote messages, then it sends the `PRE-COMMIT` message.
8. Otherwise, it sends out a `nil` pre-commit.
9. A timeout mechanism associated with the pre-commit will initialize if the associated timer expires or if the process has not sent a `PRE-COMMIT` message after receiving a proposal message and $2f + 1$ `PRE-COMMIT` messages.
10. A correct process decides on a value if it has received the proposal message in some round and $2f + 1$ `PRE-COMMIT` messages for the ID of the proposed value.
11. There is also an associated timeout mechanism with this step, which ensures that the processor does not wait indefinitely to receive $2f + 1$ messages. If the timer expires before the processor can decide, the processor starts the next round.
12. When a processor eventually decides, it triggers the next consensus instance for the next block proposal and the entire process of proposal, pre-vote, and pre-commit starts again.

We can visualize the protocol flow with the diagram shown here:

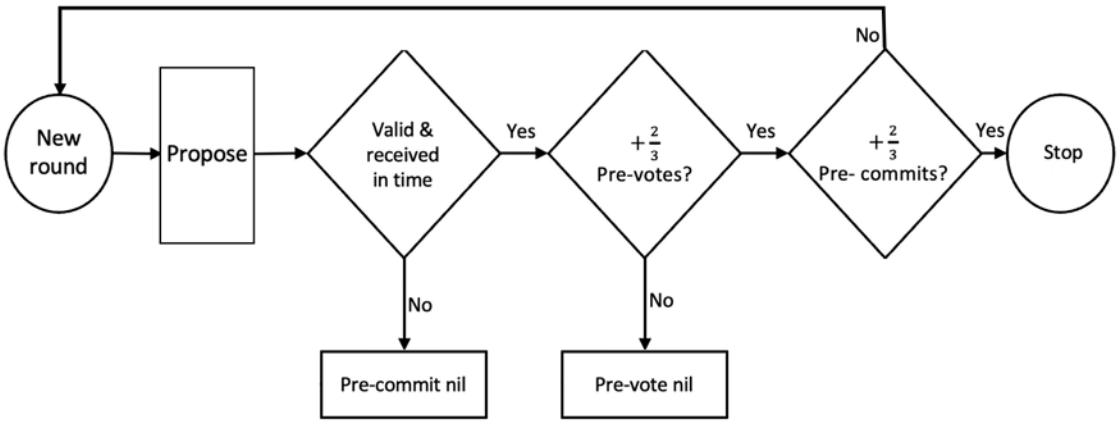


Figure 5.9: Tendermint flowchart

Types of messages

There are three types of messages in Tendermint:

- **Proposal**: As the name suggests, this is used by the leader of the current round to propose a value or block.
- **Pre-vote**: This message is used to vote on a proposed value.
- **Pre-commit**: This message is also used to vote on a proposed value.

These messages can be considered somewhat equivalent to PBFT's `PRE-PREPARE`, `PREPARE`, and `COMMIT` messages.



Note that in Tendermint, only the proposal message carries the original value, and the other two messages, pre-vote and pre-commit, operate on a value identifier, representing the original proposal.

All of the aforementioned messages also have a corresponding timeout mechanism, which ensures that processes do not end up waiting indefinitely for some conditions to meet. If a processor cannot decide in an expected amount of time, it will time out and trigger a round change.

Each type of message has an associated timeout. As such, there are three timeouts in Tendermint, corresponding to each message type:

- `Timeout-propose`
- `Timeout-prevote`
- `Timeout-precommit`

These timeout mechanisms prevent the algorithm from waiting infinitely for a condition to be met. They also ensure that processes progress through the rounds. A clever mechanism to increase timeout with every new round ensures that after reaching GST, eventually, the communication between correct processes becomes reliable and a decision can be reached.

State variables

All processes in Tendermint maintain a set of variables, which helps with the execution of the algorithm. Each of these variables holds critical values, which ensure the correct execution of the algorithm.

These variables are listed and discussed as follows:

- **Step**: The `step` variable holds information about the current state of the algorithm, that is, the current state of the Tendermint state machine in the current round.
- **lockedValue**: The `lockedValue` variable stores the most recent value (with respect to a round number) for which a pre-commit message has been sent.
- **lockedRound**: The `lockedRound` variable contains information about the last round in which the process sent a non-nil `PRE-COMMIT` message. This is the round where a possible decision value has been locked. This means that if a proposal message and corresponding *2f + 1* messages have been received for a value in a round, then due to the reason that *2f + 1* pre-votes have already been received for this value, this is a possible decision value.
- **validValue**: The role of the `validValue` variable is to store the most recent possible decision value.
- **validRound**: The `validRound` variable is the last round in which `validValue` was updated.



`lockedValue`, `lockedRound`, `validValue`, and `validRound` are reset to the initial values every time a decision is reached.

Apart from the preceding variables, a process also stores the current consensus instance (called height in Tendermint), and the current round number. These variables are attached to every message. A process also stores an array of decisions. Tendermint assumes a sequence of consensus instances, one for each height.

Now we'll explore the new **termination** mechanism. For this purpose, there are two variables, namely `validValue` and `validRound`, which are used by the proposal message. Both variables are updated by a correct process when the process receives a valid proposal message and subsequent/corresponding *2f + 1* `PRE-VOTE` messages.

This process works by utilizing the gossip protocol, which ensures that if a correct process has locked a value in a round, all correct processes will then update their `validValue` and `validRound` variables with the locked values by the end of the round during which they have been locked. The key idea is that once these values have been locked by a correct processor, they will be propagated to other nodes within the same round, and each processor will know the locked value and round, that is, the valid values.

Now, when the next proposal is made, the locked values will be picked up by the proposer, and they will have already been locked as a result of the valid proposal and corresponding *2f + 1* `PRE-VOTE` messages. This way, it can be ensured that the value that processes eventually decide upon is acceptable as specified by the validity conditions described above.



Tendermint is developed in Go. It can be implemented in various scenarios. The source code is available at `https://github.com/tendermint/tendermint`. It also is released as Tendermint Core, which is a language-agnostic programming middleware that takes a state transition machine and replicates it on many machines. Tendermint Core is available here: `https://tendermint.com/core/`.

The algorithms discussed so far are variants of traditional Byzantine fault-tolerant algorithms. Now, we'll introduce the protocols that are specifically developed for blockchain protocols.

Nakamoto consensus

Nakamoto consensus, or **PoW**, was first introduced with Bitcoin in 2009. Since then, it has stood the test of time and is the longest-running blockchain network. Contrary to common belief, the PoW mechanism is not a consensus algorithm but a Sybil attack defense mechanism. The consensus is achieved by applying the fork choice rule to choose the longest/heaviest chain to finalize the canonical chain. In this sense, we can say that PoW is a consensus facilitation algorithm that facilitates consensus and mitigates Sybil attacks, thus maintaining the blockchain's consistency and network's security.



A Sybil attack is a type of attack that aims to gain a majority influence on the network to control the network. Once a network is under the control of an adversary, any malicious activity could occur. A Sybil attack is usually conducted by a node generating and using multiple identities on the network. If there are enough multiple identities held by an entity, then that entity can influence the network by skewing majority-based network decisions. The majority in this case is held by the adversary.

The key idea behind PoW as a solution to the Byzantine generals problem is that all honest generals (miners in the Bitcoin world) achieve agreement on the same state (decision value). As long as honest participants control the majority of the network, PoW solves the Byzantine generals problem. Note that this is a probabilistic solution and not deterministic.



The original posting by Satoshi Nakamoto can be found here: `https://satoshi.nakamotoinstitute.org/emails/cryptography/11/`.

It is quite easy to obtain multiple identities and try to influence the network. However, in Bitcoin, due to the hashing power requirements, this attack is mitigated.

In a nutshell, PoW works as follows:

- PoW makes use of hash puzzles.
- A node that proposes a block has to find a nonce such that *H* (*nonce* || *previous hash* || *Tx* || ... || *Tx*) < *Threshold value*. This threshold value or the target value is based on the mining difficulty of the network, which is set by adding or reducing the number of zeroes in front of the target hash value. More zeroes mean more difficulty and a smaller number of zeroes means less difficult to solve. If there are more miners in the network, then the mining difficulty increases, otherwise, it reduces as the number of miners reduces.

The process can be summarized as follows:

1. New transactions are broadcast to the network.
2. Each node collects the transactions into a candidate block.
3. Miners propose new blocks.
4. Miners concatenate and hash with the header of the previous block.
5. The resultant hash is checked against the target value, that is, the network difficulty target value.
6. If the resultant hash is less than the threshold value (the target value), then PoW is solved, otherwise, the nonce is incremented and the node tries again. This process continues until a resultant hash is found that is less than the threshold value.

We can visualize how PoW works with the diagram shown here:

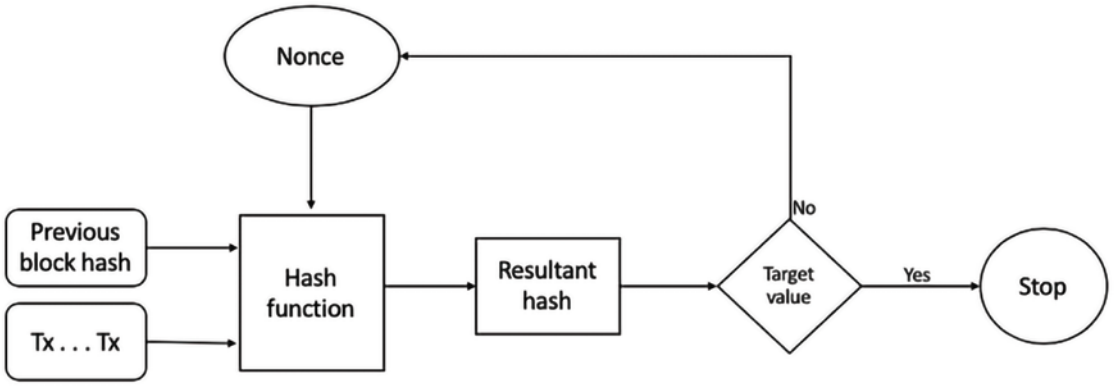


Figure 5.10: PoW

Nakamoto versus traditional consensus

The Nakamoto consensus was the first of its kind. It solved the consensus problem, which had been traditionally solved using pre-Bitcoin protocols like PBFT. However, we can map the properties of PoW consensus to traditional Byzantine consensus.

In traditional consensus algorithms, we have **agreement**, **validity**, and **liveness** properties, which can be mapped to Nakamoto-specific properties of the **common prefix**, **chain quality**, and **chain growth** properties respectively:

- The **common prefix** property means that the blockchain hosted by honest nodes will share the same large common prefix. If that is not the case, then the **agreement** property of the protocol cannot be guaranteed, meaning that the processors will not be able to decide and agree on the same value.
- The **chain quality** property means that the blockchain contains a certain required level of correct blocks created by honest nodes (miners). If chain quality is compromised, then the **validity** property of the protocol cannot be guaranteed. This means that there is a possibility that a value will be decided that is not proposed by a correct process, resulting in safety violations.
- The **chain growth** property simply means that new correct blocks are continuously added to the blockchain. If chain growth is impacted, then the **liveness** property of the protocol cannot be guaranteed. This means that the system can deadlock or fail to decide on a value.



PoW chain quality and common prefix properties are introduced and discussed in the following paper:

Garay, J., Kiayias, A. and Leonardos, N., 2015, April. *The bitcoin backbone protocol: Analysis and applications*. *In Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 281-310). Springer, Berlin, Heidelberg.

`https://eprint.iacr.org/2014/765.pdf`

Variants of PoW

There are two main variants of PoW algorithms, based on the type of hardware used for their processing:

- **CPU-bound PoW** refers to a type of PoW where the processing required to find the solution to the cryptographic hash puzzle is directly proportional to the calculation speed of the CPU or hardware such as ASICs. Because ASICs have dominated the Bitcoin PoW and provide somewhat undue advantage to the miners who can afford to use ASICs, this CPU-bound PoW is seen as shifting toward centralization. Moreover, mining pools with extraordinary hashing power can shift the balance of power toward them. Therefore, memory-bound PoW algorithms have been introduced, which are ASIC-resistant and are based on memory-oriented design instead of CPU.
- **Memory-bound PoW** algorithms rely on system RAM to provide PoW. Here, the performance is bound by the access speed of the memory or the size of the memory. This reliance on memory also makes these PoW algorithms ASIC-resistant. Equihash is one of the most prominent memory-bound PoW algorithms.

PoW consumes tremendous amounts of energy; as such, there are several alternatives suggested by researchers. One of the first alternatives proposed is PoS.

Alternatives to PoW

PoW does have various drawbacks, and the biggest of all is energy consumption—the total electricity consumed by Bitcoin miners is more than many countries! This is huge, and all that power is, in a way, wasted; no useful purpose is served except mining. Environmentalists have raised real concerns about this situation. In addition to electricity consumption, the carbon footprint is also very high, and is only expected to grow.



For recent data on Bitcoin's carbon footprint, visit `https://digiconomist.net/bitcoin-energy-consumption`.

It has been proposed that PoW puzzles can be designed in such a way that they serve two purposes. First, their primary purpose is in consensus mechanisms, and second, they serve to perform some useful scientific computation. This way not only can the schemes be used in mining, but they can also help to solve other scientific problems.

Another drawback of the PoW scheme, especially the one used in Bitcoin (Double SHA-256), is that since the introduction of ASICs, the power is shifting toward miners or mining pools who can afford to operate large-scale ASIC farms. This power shift challenges the core philosophy of the decentralization of Bitcoin. There are a few alternatives that have been proposed, such as ASIC-resistant puzzles, which are designed in such a way that building ASICs for solving this puzzle is unfeasible and does not result in a major performance gain over commodity hardware. A common technique used for this purpose is to apply a class of computationally hard problems called **memory-hard computational puzzles**. The core idea behind this method is that as puzzle solving requires a large amount of memory, it is not feasible to be implemented on ASIC-based systems.

This technique was initially used in Litecoin and Tenebrix, where the Scrypt hash function was used as an ASIC-resistant PoW scheme. Even though this scheme was initially advertised as ASIC resistant, Scrypt ASICs became available a few years ago, disproving the original claim by Litecoin. This happened because even if Scrypt is a memory-intensive mechanism, initially, it was thought that building ASICs with large memories is difficult due to technical and cost limitations. This is no longer the case because memory hardware is increasingly becoming cheaper. Also, with the ability to produce nanometer-scale circuits, it is possible to build ASICs that can run the Scrypt algorithm.

Another approach to ASIC resistance is where multiple hash functions are required to be calculated to provide PoW. This is also called a **chained hashing scheme**. The rationale behind this idea is that designing multiple hash functions on an ASIC is not very feasible. The most common example is the X11 memory hard function, implemented in Dash. X11 comprises 11 SHA-3 contestants, where one algorithm outputs the calculated hash to the next algorithm until all 11 algorithms are used in a sequence. These algorithms include BLAKE, BMW, Groestl, JH, Keccak, Skein, Luffa, CubeHash, SHAvite, SIMD, and ECHO. This approach did provide some resistance to ASIC development initially, but now ASIC miners are available commercially and support mining of X11 and similar schemes.

Perhaps another approach could be to design self-mutating puzzles that intelligently or randomly change the PoW scheme or its requirements as a function of time. This strategy will make it almost impossible to be implemented in ASICs as it will require multiple ASICs to be designed for each function. Also, randomly changing schemes would be practically impossible to handle in ASICs. At the moment, it is unclear how this can be achieved practically.

In the following sections, we consider some alternative consensus mechanisms to PoW.

Proof of Storage

Also known as **proof of retrievability**, this is another type of proof of useful work that requires storage of a large amount of data. Introduced by Microsoft Research, this scheme provides a useful benefit of distributing the storage of archival data. Miners are required to store a pseudorandomly selected subset of large data to perform mining.

Proof of Stake

Proof of stake (**PoS**) is also called **virtual mining**. This is another type of mining puzzle that has been proposed as an alternative to traditional PoW schemes. It was first proposed in Peercoin in August 2012, and now, prominent blockchains such as EOS, NxT, Steem, Ethereum 2.0, Polkadot, and Tezos are using variants of PoS algorithms. Ethereum, with its Serenity release, will soon transition to a PoS-based consensus mechanism.

In this scheme, the idea is that users are required to demonstrate the possession of a certain amount of currency (coins), thus proving that they have a stake in the coin. The simplest form of this stake is where mining is made comparatively easier for those users who demonstrably own larger amounts of digital currency. The benefits of this scheme are twofold; first, acquiring large amounts of digital currency is relatively difficult as compared to buying high-end ASIC devices, and second, it results in saving computational resources. Various forms of stake have been proposed:

- **Proof of coinage**: The age of a coin is the time since the coins were last used or held. This is a different approach from the usual form of PoS, where mining is made easier for users who have the highest stake in the altcoin. In the coin-age-based approach, the age of the coin (coin-age) is reset every time a block is mined. The miner is rewarded for holding and not spending coins for a period of time. This mechanism has been implemented in Peercoin combined with PoW in a creative way. The difficulty of mining puzzles (PoW) is inversely proportional to the coinage, meaning that if miners consume some coinage using coin-stake transactions, then the PoW requirements are relieved.
- **Proof of Deposit**: This is a type of PoS. The core idea behind this scheme is that newly minted blocks by miners are made unspendable for a certain period. More precisely, the coins get locked for a set number of blocks during the mining operation. The scheme works by allowing miners to perform mining at the cost of freezing a certain number of coins for some time.
- **Proof of Burn**: As an alternate expenditure to computing power, this method destroys a certain number of Bitcoins to get equivalent altcoins. This is commonly used when starting up a new coin projects as a means to provide a fair initial distribution. This can be considered an alternative mining scheme where the value of the new coins comes from the fact that, previously, a certain number of coins have been destroyed.

The stake represents the number of coins (money) in the consensus protocol staked by a blockchain participant. The key idea is that if someone has a stake in the system, then they will not try to sabotage the system. The chance of proposing the next block is directly proportional to the value staked by the participant.

In PoS systems, there is no concept of mining as in the traditional Nakamoto consensus sense. However, the process related to earning revenue is sometimes called virtual mining. A PoS miner is called either a validator, minter, or stakeholder.

The right to win the next proposer role is usually assigned randomly. Proposers are rewarded either with transaction fees or block rewards. Similar to PoW, control over the majority of the network in the form of the control of a large portion of the stake is required to attack and control the network.

PoS mechanisms generally select a stakeholder and grant appropriate rights to it based on its staked assets. The stake calculation is application-specific, but generally, is based on balance, deposit value, or voting among the validators. Once the stake is calculated and a stakeholder is selected to propose a block, the block proposed by the proposer is readily accepted. The probability of selection increases with a higher stake. In other words, the higher the stake, the better the chances of winning the right to propose the next block.

The diagram below shows how a generic PoS mechanism works:

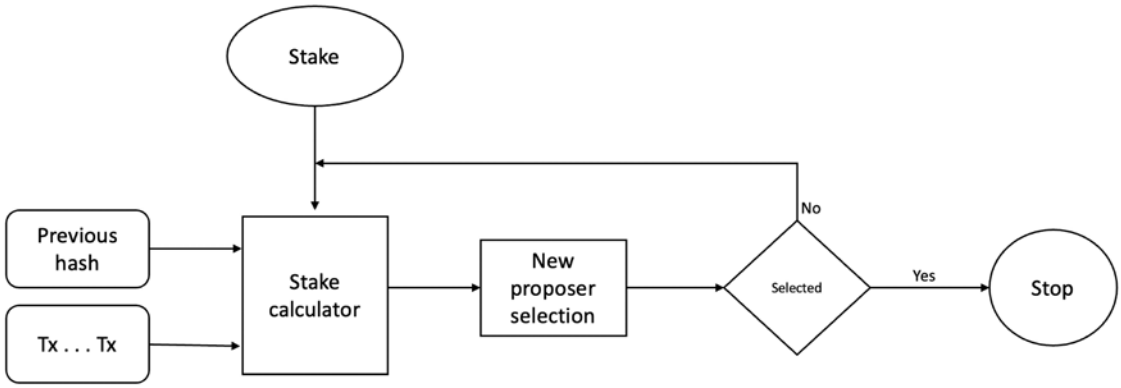


Figure 5.11: PoS

In the preceding diagram, a stake calculator function is used to calculate the amount of staked funds, and based on that, a new proposer is selected. If for some reason the proposer selection fails, then the stake calculator and new proposer selection function run again to choose a new block proposer (leader).

Chain-based PoS is very similar to PoW. The only change from the PoW mechanism is the block generation method. A block is generated in two steps by following a simple protocol:

- Transactions are picked up from the memory pool and a candidate block is created.
- A clock is set up with a constant tick interval, and at each clock tick, whether the hash of the block header concatenated with the clock time is less than the product of the target value and stake value is checked.

This process can be shown in a simple formula:

$$Hash\ (B\ ||\ clock\ time) < target\ x\ stake\ value$$

The stake value is dependent on the way the algorithm is designed. In some systems, it is directly proportional to the amount of stake, and in others, it is based on the amount of time the stake has been held by the participant (also called coinage). The target is the mining difficulty per unit of the value of the stake.

This mechanism still uses hashing puzzles, as in PoW. But, instead of competing to solve the hashing puzzle by consuming a high amount of electricity and specialized hardware, the hashing puzzle in PoS is solved at regular intervals based on the clock tick. A hashing puzzle becomes easier to solve if the stake value of the minter is high.



Peercoin was the first blockchain to implement PoS. Nxt (`https://www.jelurida.com/nxt`) and Peercoin (`https://www.peercoin.net`) are two examples of blockchains where chain-based PoS is implemented.

In **committee-based PoS**, a group of stakeholders is chosen randomly, usually by using a **verifiable random function** (**VRF**). This VRF, once invoked, produces a random set of stakeholders based on their stake and the current state of the blockchain. The chosen group of stakeholders becomes responsible for proposing blocks in sequential order.



This mechanism is used in the Ouroboros PoS consensus mechanism, which is used in Cardano. More details on this are available here: `https://www.cardano.org/en/ouroboros/`.

VRFs were introduced in *Chapter 4*, *Asymmetric Cryptography*. More information on VRF can be found here: `https://datatracker.ietf.org/doc/html/draft-goldbe-vrf-01`.

Delegated PoS is very similar to committee-based PoS, but with one crucial difference. Instead of using a random function to derive the group of stakeholders, the group is chosen by stake delegation. The group selected is a fixed number of minters that create blocks in a round-robin fashion. Delegates are chosen via voting by network users. Votes are proportional to the amount of stake that participants have in the network. This technique is used in Lisk, Cosmos, and EOS. DPoS is not decentralized as a small number of known users are made responsible for proposing and generating blocks.

Proof of Activity (PoA)

This scheme is a hybrid of PoW and PoS. In this scheme, blocks are initially produced using PoW, but then each block randomly assigns three stakeholders that are required to digitally sign it. The validity of subsequent blocks is dependent on the successful signing of previously randomly chosen blocks.

There is, however, a possible issue known of the **nothing at stake** problem, where it would be trivial to create a fork of the blockchain. This is possible because in PoW, appropriate computational resources are required to mine, whereas in PoS, there is no such requirement; as a result, an attacker can try to mine on multiple chains using the same coin.

Non-outsourceable puzzles

The key motivation behind this puzzle is to develop resistance against the development of mining pools. Mining pools, as previously discussed, offer rewards to all participants in proportion to the computing power they consume. However, in this model, the mining pool operator is a central authority to whom all the rewards go and who can enforce specific rules. Also, in this model, all miners only trust each other because they are working toward a common goal, in the hope of the pool manager getting the reward. Non-outsourceable puzzles are a scheme that allows miners to claim rewards for themselves; consequently, pool formation becomes unlikely due to inherent mistrust between anonymous miners.

Next, we consider HotStuff, the latest class of BFT protocol with several optimizations.

HotStuff

HotStuff is the latest class of BFT protocol with several optimizations. There are several changes in HotStuff that make it a different and, in some ways, better protocol than traditional PBFT. HotStuff was introduced by VMware Research in 2018. Later, it was presented at the Symposium on Principles of Distributed Computing.



The paper reference is as follows:

Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G. and Abraham, I., 2019, July. HotStuff: *BFT consensus with linearity and responsiveness*. *In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (pp. 347-356). ACM.

The three key optimizations of HotStuff are listed below:

- **Linear view change** results in reduced communication complexity. It is achieved by the algorithm where after GST is reached, a correct designated leader will send only **O(n) authenticators** (**either a partial signature or signature**) to reach a consensus. In the worst case, where leaders fail successively, the communication cost is $O(n^2)$ —quadratic. In simpler words, quadratic complexity means that the performance of the algorithm is proportional to the squared size of the input.
- **Optimistic responsiveness** ensures that any correct leader after GST is reached only requires the first $n-f$ responses to ensure progress.
- **Chain quality** ensures fairness and liveness in the system by allowing fast and frequent leader rotation.

Another innovation in HotStuff is the separation of safety and liveness mechanisms. The separation of concerns allows for better modularity, cleaner architecture, and control over the development of these features independently. Safety is ensured through voting and commit rules for participant nodes in the network. Liveness, on the other hand, is the responsibility of a separate module, called **Pacemaker**, which ensures a new, correct, and unique leader is elected.

In comparison with traditional PBFT, HotStuff has introduced several changes, which result in improved performance. Firstly, PBFT-style protocols work using a **mesh communication topology**, where each message is required to be broadcast to other nodes on the network. In HotStuff, the communication has been changed to the star topology, which means that nodes do not communicate with each other directly, but all consensus messages are collected by a leader and then broadcast to other nodes. This immediately results in reduced communication complexity.

A question arises here: what happens if the leader somehow is corrupt or compromised? This issue is solved by the same BFT tolerance rules where, if a leader proposes a malicious block, it will be rejected by other honest validators and a new leader will be chosen. This scenario can slow down the network for a limited time (until a new honest leader is chosen), but eventually (as long as a majority of the network is honest), an honest leader will be chosen, which will propose a valid block. Also, for further protection, usually, the leader role is frequently (usually, with each block) rotated between validators, which can neutralize any malicious attacks targeting the network. This property ensures **fairness**, which helps to achieve **chain quality**, introduced previously.

However, note that there is one problem in this scheme where, if the leader becomes too overloaded, then the processing may become slow, impacting the whole network. Mesh and star topologies can be visualized in the following diagram:

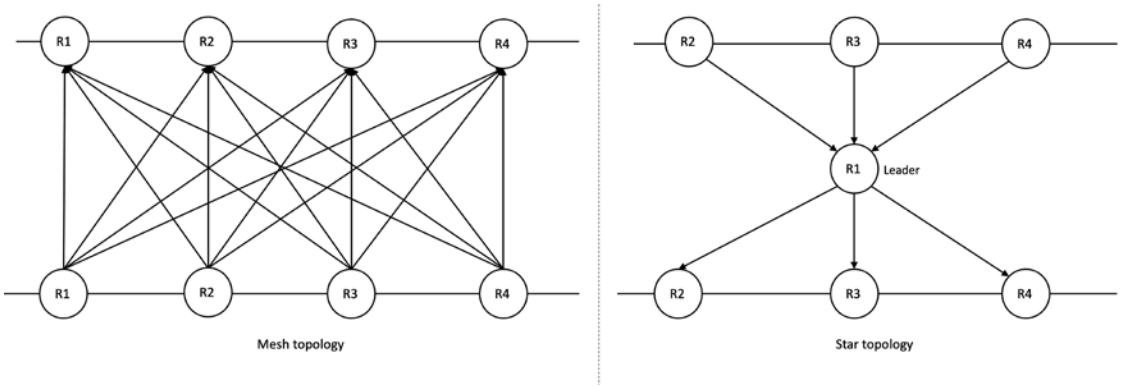


Figure 5.12: Star topology

A second change is regarding PBFT's two main subprotocols, namely, normal mode and view change mode. In this model, the view change mode is triggered when a leader is suspected of being faulty. This approach does work to provide a liveness guarantee to PBFT but increases communication complexity significantly. HotStuff addresses this by merging the view change process with normal mode. This means that nodes can switch to a new view directly without waiting for a threshold of view change messages to be received by other nodes. In PBFT, nodes wait for *2f+1* messages before the view change can occur, but in HotStuff, view change can occur directly without requiring a new subprotocol. Instead, the checking of the threshold of the messages to change the view becomes part of the normal view.

When compared with Tendermint, HotStuff is improved in two aspects. First, it can be chained, which means that it can be pipelined where a single quorum certificate can service in different phases at the same time, resulting in better performance. Second, it is optimistically responsive, as introduced above.

Just like PBFT, HotStuff also solves the SMR problem. Now, we'll describe how this protocol works. As we saw earlier, consensus algorithms are described and work under a system model.

The system is based on a standard BFT assumption of $n=3f+1$ nodes in the system, where f is a faulty node and N is the number of nodes in the network. Nodes communicate with each other via point-to-point message-passing, utilizing reliable and authenticated communication links. The network is supposed to be partially synchronous.

HotStuff makes use of threshold signatures where a single public key is used by all nodes, but a unique private key is used by each replica. The use of threshold signatures results in the decreased communication complexity of the protocol. In addition, cryptographic hash functions are used to provide unique identifiers for messages.



We discussed threshold signatures in *Chapter 4*, *Asymmetric Cryptography*. You can refer to the details there if required.

HotStuff works in phases, namely the prepare phase, pre-commit phase, commit phase, and decide phase:



A common term that we will see in the following section is **quorum certificate** (**QC**). It is a data structure that represents a collection of signatures produced by $N–F$ replicas to demonstrate that the required threshold of messages has been achieved. In other words, it is simply a set of votes from $n–f$ nodes.

- **Prepare:** Once a new leader has collected `NEW-VIEW` messages from $n–f$ nodes, the protocol for the new leader starts. The leader collects and processes these messages to figure out the latest branch in which the highest quorum certificate of prepare messages was formed.
- **Pre-commit:** As soon as a leader receives $n–f$ prepare votes, it creates a quorum certificate called "prepare quorum certificate." This "prepare quorum certificate" is broadcast to other nodes as a `PRE-COMMIT` message. When a replica receives the `PRE-COMMIT` message, it responds with a pre-commit vote. The quorum certificate is the indication that the required threshold of nodes has confirmed the request.
- **Commit:** When the leader receives $n–f$ pre-commit votes, it creates a `PRE-COMMIT` quorum certificate and broadcasts it to other nodes as the `COMMIT` message. When replicas receive this `COMMIT` message, they respond with their commit vote. At this stage, replicas lock the `PRE-COMMIT` quorum certificate to ensure the safety of the algorithm even if view change occurs.
- **Decide:** When the leader receives $n–f$ commit votes, it creates a `COMMIT` quorum certificate. This `COMMIT` quorum certificate is broadcast to other nodes in the `DECIDE` message. When replicas receive this `DECIDE` message, replicas execute the request, because this message contains an already committed certificate/value. Once the state transition occurs as a result of the `DECIDE` message being processed by a replica, the new view starts.

This algorithm can be visualized in the following diagram:

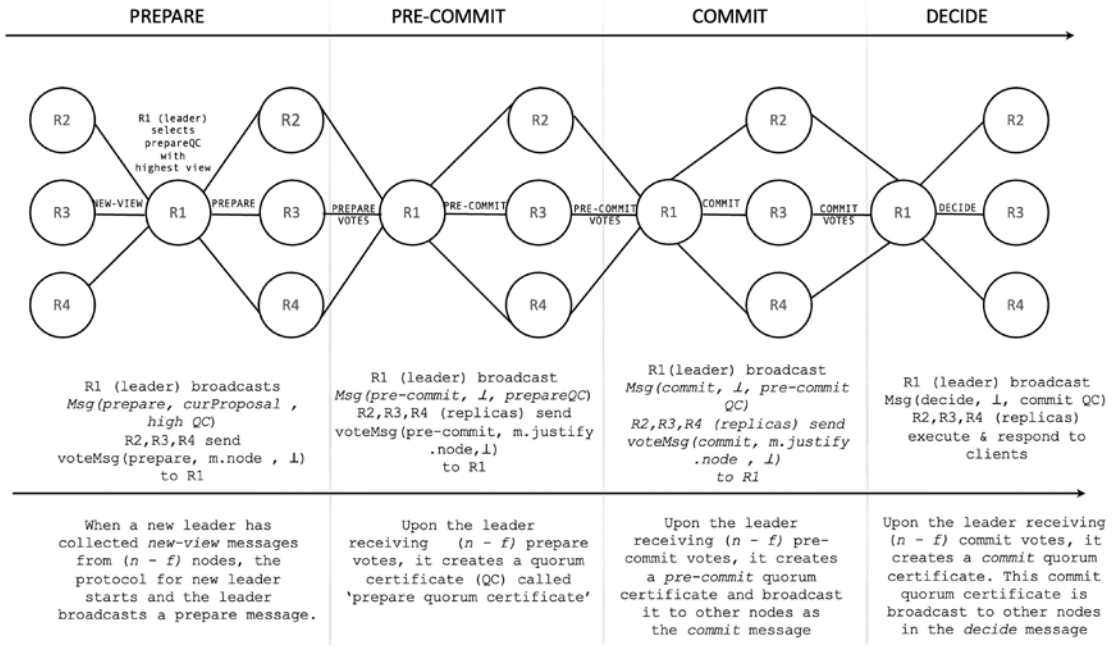


Figure 5.13: HotStuff protocol

HotStuff, with some variations, is also used in DiemBFT, which is a distributed database designed to support a global currency proposed by Facebook.



More details on the Diem protocol are available here: `https://developers.diem.com/papers/diem-consensus-state-machine-replication-in-the-diem-blockchain/2021-08-17.pdf`.

HotStuff guarantees **liveness** (progress) by using **Pacemaker**, which ensures progress after GST within a bounded time interval. This component has two elements:

- There are time intervals during which all replicas stay at a height for sufficiently long periods. This property can be achieved by progressively increasing the time until progress is made (a decision is made).
- A unique and correct leader is elected for the height. New leaders can be elected deterministically by using a rotating leader scheme or pseudo-random functions.

Safety in HotStuff is guaranteed by voting and relevant commit rules. **Liveness** and **safety** are separate mechanisms that allow independent development, modularity, and separation of concerns.

HotStuff is a simple yet powerful protocol that provides linearity and responsiveness properties. It allows consensus without any additional latency, at the actual speed of the network.

Moreover, it is a protocol that manifests linear communication complexity, thus reducing the cost of communication compared to PBFT-style protocols. It is also a framework in which other protocols, such as DLS, PBFT, and Tendermint can be expressed.

In this section, we have explored various consensus algorithms in detail. We discussed pre-Bitcoin distributed consensus algorithms and post-Bitcoin, PoW-style algorithms. Now, a question arises naturally: with the availability of all these different algorithms, which algorithm is best, and which one should you choose for a particular use case? We'll try to answer this question in the next section.

Choosing an algorithm

Choosing a consensus algorithm depends on several factors. It is not only use-case-dependent, but some trade-offs may also have to be made to create a system that meets all the requirements without compromising the core safety and liveness properties of the system. We will discuss some of the main factors that can influence the choice of consensus algorithm. Note that these factors are different from the core safety and liveness properties discussed earlier, which are the fundamental requirements needed to be fulfilled by any consensus mechanism. Other factors that we are going to introduce here are use-case-specific and impact the choice of consensus algorithm. These factors include finality, speed, performance, and scalability.

Finality

Finality refers to a concept where once a transaction has been completed, it cannot be reverted. In other words, if a transaction has been committed to the blockchain, it won't be revoked or rolled back. This feature is especially important in financial networks, where once a transaction has gone through, the consumer can be confident that the transaction is irrevocable and final. There are two types of finality, probabilistic and deterministic.

Probabilistic finality, as the name suggests, provides a probabilistic guarantee of finality. The assurance that a transaction, once committed to the blockchain, cannot be rolled back builds over time instead of immediately. For example, in Nakamoto consensus, the probability that a transaction will not be rolled back increases as the number of blocks after the transaction commits increases. As the chain grows, the block containing the transaction goes deeper, which increasingly ensures that the transaction won't be rolled back. While this method worked and stood the test of time for many years, it is quite slow. For example, in the Bitcoin network, users usually have to wait for six blocks, which is equivalent to an hour, to get a high level of confidence that a transaction is final. This type of finality might be acceptable in public blockchains. Still, such a delay is not acceptable in financial transactions in a consortium blockchain. In consortium networks, we need immediate finality.

Deterministic finality or immediate finality provides an absolute finality guarantee for a transaction as soon as it is committed in a block. There are no forks or rollbacks, which could result in a transaction rollback. The confidence level of transaction finality is 100 percent as soon as the block that contains the transaction is finalized. This type of finality is provided by fault-tolerant algorithms such as PBFT.

Speed, performance, and scalability

Performance is a significant factor that impacts the choice of consensus algorithms. PoW chains are slower than BFT-based chains. If performance is a crucial requirement, then it is advisable to use voting-based algorithms for permissioned blockchains such as PBFT, which will provide better performance in terms of quicker transaction processing. There is, however, a caveat that needs to be kept in mind here—PBFT-type blockchains do not scale well but provide better performance. On the other hand, PoW-type chains can scale well but are quite slow and do not meet enterprise-grade performance requirements.

Summary

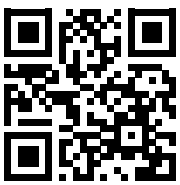
In this chapter, we explained some of the most prominent protocols in blockchain and traditional distributed system consensus. We covered several algorithms, including PoW, proof of stake, traditional BFT protocols, and the latest protocols, such as HotStuff. We did not cover newer algorithms such as BABE, GRANDPA, proof of history, and Casper which we will discuss later in this book.

Distributed consensus is a very interesting area of research, and academics and industry researchers are participating in this exciting subject. It is also a deep and vast area of research; therefore, it is impossible to cover every protocol and all dimensions of this gripping discipline in a single chapter. Nevertheless, the protocols and ideas presented in this chapter provide solid ground for further research and more in-depth exploration.

In the next chapter, we'll introduce Bitcoin, which is the first blockchain and cryptocurrency, invented in 2008.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

6

Bitcoin Architecture

Bitcoin was the first application of blockchain technology and has started a revolution with the introduction of the very first fully decentralized digital currency. It has proven to be remarkably secure and valuable as a digital currency, despite being highly volatile. The invention of Bitcoin has also sparked great interest within academia and industry and opened many new research areas. In this chapter, we shall introduce Bitcoin in detail.

Specifically, in this chapter, we will focus our attention upon the fundamentals of Bitcoin, how transactions are constructed and used, transaction structures, addresses, accounts, and mining, over the following topic areas:

- Introducing Bitcoin
- Cryptographic keys
- Addresses
- Transactions
- Blockchain
- Miners
- Network
- Wallets

Let's begin with an overview of Bitcoin.

Introducing Bitcoin

Since its introduction in 2008 by Satoshi Nakamoto, **Bitcoin** has gained immense popularity and is currently the most successful digital currency in the world, with billions of dollars invested in it.

Its popularity is also evident from the high number of users and investors, daily news related to Bitcoin, and the many start-ups and companies that are offering Bitcoin-based online exchanges. It is now also traded as Bitcoin futures on the **Chicago Mercantile Exchange** (**CME**).



Interested readers can read more about Bitcoin futures at `http://www.cmegroup.com/trading/bitcoin-futures.html`.

In 2008, Bitcoin was introduced in a paper called *Bitcoin: A Peer-to-Peer Electronic Cash System*. This paper is available at `https://bitcoin.org/en/bitcoin-paper`. The name of the author, Satoshi Nakamoto, is believed to be a pseudonym, as the true identity of the inventor of Bitcoin is unknown and is the subject of much speculation.

The first key idea introduced in the paper was of purely P2P electronic cash that does not need an intermediary bank to transfer payments between peers. However, Bitcoin can be defined in various ways; it's a protocol, a digital currency, and a platform. It is a combination of a P2P network, protocols, and software that facilitates the creation and usage of the digital currency. Nodes in this P2P network talk to each other using the Bitcoin protocol.

Bitcoin is built on decades of research. Various ideas and techniques from cryptography and distributed computing such as Merkle trees, hash functions, and digital signatures were used to design Bitcoin. Other ideas such as BitGold, b-money, hashcash, and cryptographic time-stamping also provided some groundwork for the invention of Bitcoin. Ideas from many of these developments were ingeniously used in Bitcoin to create the first ever truly decentralized currency. Bitcoin solves several historically difficult problems related to electronic cash and distributed systems, including:

- The Byzantine generals problem
- Sybil attacks
- The double-spending problem

The double-spending problem arises when, for example, a user sends coins to two different users at the same time, and they are verified independently as valid transactions. The double-spending problem is resolved in Bitcoin by using a distributed ledger (the blockchain) where every transaction is recorded permanently, and by implementing a transaction validation and confirmation mechanism.

In this chapter, we will look at the various actors and components of the Bitcoin network, and how they interact to form it:

- Cryptographic keys
- Addresses
- Transactions
- Blockchain
- Miners
- Network
- Wallets

First, we will look at the keys and addresses that are used to represent the ownership and transfer of value on the Bitcoin network.

Cryptographic keys

On the Bitcoin network, possession of bitcoins and the transfer of value via transactions are reliant upon private keys, public keys, and addresses. **Elliptic Curve Cryptography** (**ECC**) is used to generate public and private key pairs in the Bitcoin network. We have already covered these concepts in *Chapter 4*, *Asymmetric Cryptography*, and here we will see how private and public keys are used in the Bitcoin network.

Private keys in Bitcoin

Private keys are required to be kept safe and normally reside only on the owner's side. Private keys are used to digitally sign transactions, proving ownership of bitcoins.

Private keys are fundamentally 256-bit numbers randomly chosen in the range specified by the `SECP256K1 ECDSA` curve recommendation. Any randomly chosen 256-bit number from `0x1` to `0xFFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFE BAAE DCE6 AF48 A03B BFD2 5E8C D036 4140` is a valid private key.

Private keys are usually encoded using **Wallet Import Format** (**WIF**) in order to make them easier to copy and use. It is a way to represent the full-size private key in a different format. `WIF` can be converted into a private key and vice versa. For example, consider the following private key:

```
A3ED7EC8A03667180D01FB4251A546C2B9F2FE33507C68B7D9D4E1FA5714195201
```

When converted into `WIF` format, it looks as shown here:

```
L2iN7umV7kbr6LuCmgM27rBnptGbDVc8g4ZBm6EbgTPQXnj1RCZP
```



Interested readers can do some experimentation using the online tool available at `http://gobittest.appspot.com/PrivateKey`.

Also, **mini private key format** is sometimes used to create a private key with a maximum of 30 characters to allow storage where physical space is limited. For example, etching on physical coins or encoding in damage resistant QR codes. The QR code is more damage resistant because more dots can be used for error correction and fewer for encoding the private key.



QR codes use **Reed-Solomon error correction**. Discussion on the error correction mechanism and its underlying details is out of the scope of this book but readers are encouraged to research QR code error correction if interested.

A private key encoded using mini private key format is also sometimes called a **minikey**. The first character of the mini private key is always the uppercase letter `S`. A mini private key can be converted into a normal-sized private key, but an existing normal-sized private key cannot be converted into a mini private key. This format was used in **Casascius** physical bitcoins. The Bitcoin core client also allows the encryption of the wallet that contains the private keys.

As we learned in *Chapter 4*, *Asymmetric Cryptography*, private keys have their own corresponding public keys. Public or private keys on their own are useless—pairs of public and private keys are required for the normal functioning of any public key cryptography-based systems such as the Bitcoin blockchain.

Public keys in Bitcoin

All network participants can see public keys on the blockchain. Public keys are derived from private keys due to their special mathematical relationship. Once a transaction signed with the private key is broadcast on the Bitcoin network, public keys are used by the nodes to verify that the transaction has indeed been signed with the corresponding private key. This process of verification proves the ownership of the Bitcoin.

Bitcoin uses ECC based on the SECP256K1 standard. More specifically, it makes use of an **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) to ensure that funds remain secure and can only be spent by the legitimate owner. If you need to refresh the relevant cryptography concepts, you can refer to *Chapter 4*, *Asymmetric Cryptography*, where ECC was explained. Public keys can be represented in uncompressed or compressed format and are fundamentally x and y coordinates on an elliptic curve. The compressed version of public keys includes only the x part since the y part can be derived from it.

The reason why the compressed version of public keys works is that if the ECC graph is visualized, it reveals that the y coordinate can be either below the x axis or above the x axis, and as the curve is symmetric, only the location in the prime field is required to be stored. If y is even, then its value lies above the x axis, and if it is odd, then it is below the x axis. This means that instead of storing both x and y as the public key, only x needs to be stored with the information about whether y is even or odd.

Initially, the Bitcoin client used uncompressed keys, but starting from Bitcoin Core client 0.6, compressed keys are used as standard. This resulted in an almost 50% reduction of space used to store public keys in the blockchain.

Keys are identified by various prefixes, described as follows:

- Uncompressed public keys use `0x04` as the prefix. Uncompressed public keys are 65 bytes long. They are encoded as 256-bit unsigned big-endian integers (32 bytes), which are concatenated together and finally prefixed with a byte `0x04`. This means 1 byte for the `0x04` prefix, 32 bytes for the x integer, and 32 bytes for the y integer, which makes it 65 bytes in total.
- Compressed public keys start with `0x03` if the y 32-byte (256-bit) part of the public key is odd. It is 33 bytes in length as 1 byte is used by the `0x03` prefix (depicting an odd y) and 32 bytes are used for storing the x coordinate.
- Compressed public keys start with `0x02` if the y 32-byte (256-bit) part of the public key is even. It is 33 bytes in length as 1 byte is used by the `0x02` prefix (depicting an even y) and 32 bytes are used for storing the x coordinate.

Having talked about private and public keys, let's now move on to another important aspect of Bitcoin: addresses derived from public keys.

Addresses

The following diagram shows how an address is generated, from generating the private key to the final output of the Bitcoin address:

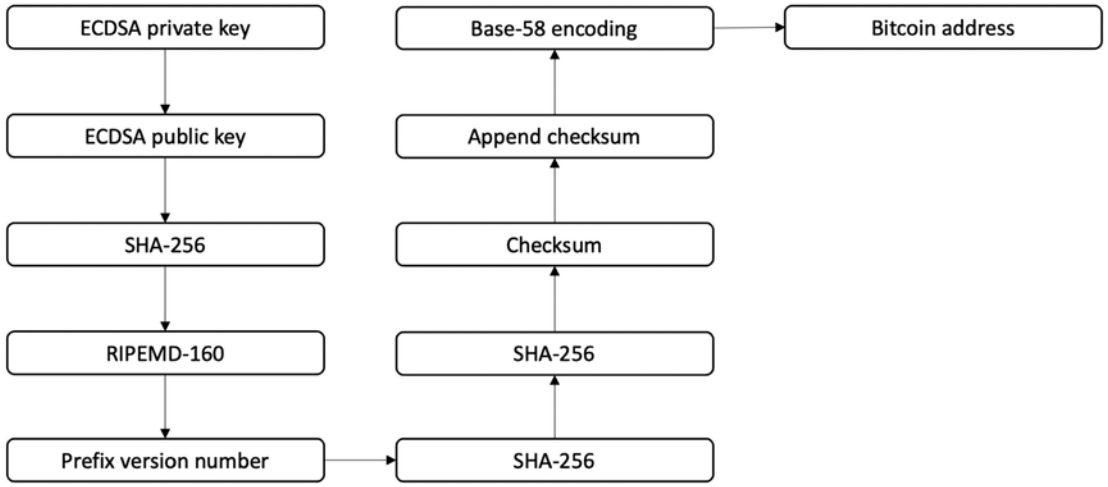


Figure 6.1: Address generation in Bitcoin

In the preceding diagram, there are a number of steps:

1. In the first step, we have a randomly generated ECDSA private key.
2. The public key is derived from the ECDSA private key.
3. The public key is hashed using the SHA-256 cryptographic hash function.
4. The hash generated in *step 3* is hashed using the RIPEMD-160 hash function.
5. The version number is prefixed to the RIPEMD-160 hash generated in *step 4*.
6. The result produced in *step 5* is hashed using the SHA-256 cryptographic hash function.
7. SHA-256 is applied again.
8. The first 4 bytes of the result produced from *step 7* are the address `checksum`.
9. This `checksum` is appended to the RIPEMD-160 hash generated in *step 4*.
10. The resultant byte string is encoded into a Base58-encoded string by applying the Base58 encoding function.
11. Finally, the result is a typical Bitcoin address.

Typical Bitcoin addresses

Bitcoin addresses are 26-35 characters long and begin with the digits 1 or 3. A typical Bitcoin address looks like the string shown here:

```
15ccPQG3PQXcj7fhgmWAHN7SQ7JBvfNFGb
```

Addresses are also commonly encoded in a QR code for easy distribution. The QR code of the preceding Bitcoin address is shown in the following image:



Figure 6.2: QR code of the Bitcoin address 15ccPQG3PQXcj7fhgmWAHN7SQ7JBvfNFGb

Currently, there are two types of addresses, the commonly used P2PKH and the P2SH type (both defined later in this chapter), starting with numbers 1 and 3, respectively. In the early days, Bitcoin used direct Pay-to-Pubkey, which has now been superseded by P2PKH. These types will be explained later in the chapter. However, direct Pay-to-Pubkey is still used in Bitcoin for coinbase addresses. Addresses should not be used more than once; otherwise, privacy and security issues can arise.

Avoiding address reuse circumvents anonymity issues to an extent, but Bitcoin has some other security issues as well, such as transaction malleability, Sybil attacks, race attacks, and selfish mining, all of which require different approaches to resolve. Transaction malleability has been resolved with the so-called SegWit soft-fork upgrade of the Bitcoin protocol. This concept will be explained later in the chapter.



Figure 6.3: From bitaddress.org, a private key and Bitcoin address in a paper wallet

Bitcoin addresses are encoded using the Base58Check encoding. This encoding is used to limit the confusion between various characters, such as 0/O or I/l, as they can look the same in different fonts. The encoding basically takes the binary byte arrays and converts them into human-readable strings. This string is composed by utilizing a set of 58 alphanumeric symbols. More explanation and logic can be found in the `base58.h` source file in the Bitcoin source code:

```
/**
* Why base-58 instead of standard base-64 encoding?
* - Don't want 0OIl characters that look the same in some fonts and
*      could be used to create visually identical looking data.
* - A string with non-alphanumeric characters is not as easily accepted as input.
* - E-mail usually won't line-break if there's no punctuation to break at.
* - Double-clicking selects the whole string as one word if it's all alphanumeric.
*/
```



This file is present in the Bitcoin source code and can be viewed at `https://github.com/bitcoin/bitcoin/blob/c8971547d9c9460fcbec6f54888df83f002c3dfd/src/base58.h`.

Advanced Bitcoin addresses

In addition to common types of addresses in Bitcoin, there are some advanced types of addresses available in Bitcoin too:

- **Vanity addresses**: As Bitcoin addresses are based on Base58 encoding, it is possible to generate addresses that contain human-readable messages and are personalized. An example is `1BasHiry2VoCQCdX6X64oxvKRuf7fW6qGr`—note that the address contains the name `BasHir`. Vanity addresses are generated using a brute-force method. There are various online services that provide this service. More details on the mechanism and a link to a standalone command line program is available at `https://en.bitcoin.it/wiki/Vanitygen`.
- **Multi-signature addresses**: As the name implies, these addresses require multiple private keys. In practical terms, this means that in order to release the coins, a certain set number of signatures is required. This is also known as *M of N multisig*. Here, *M* represents the threshold or minimum number of signatures required from *N* number of keys to release the bitcoins. Remember that we discussed this concept in *Chapter 4*, *Asymmetric Cryptography*.

With this section, we've finished our introduction to addresses in Bitcoin. In the next section, we will introduce Bitcoin transactions, which are the most fundamental and important aspect of Bitcoin.

Transactions

Transactions are at the core of the Bitcoin ecosystem. Transactions can be as simple as just sending some bitcoins to a Bitcoin address, or can be quite complex, depending on the requirements.

A Bitcoin transaction is composed of several elements:

- **Transaction ID**: A 32 byte long unique transaction identifier
- **Size**: This is the size of the transaction in bytes.
- **Weight**: This is a metric given for the block and transaction sizes since the introduction of the **SegWit soft-fork** version of Bitcoin.
- **Time**: This is the time when the block containing this transaction was mined.
- **Included in block**: This shows the block number on the blockchain in which the transaction is included.
- **Confirmations**: This is the number of confirmations completed by miners for this transaction.
- **Total input**: This is the number of total inputs in the transaction.
- **Total output**: This is the number of total outputs from the transaction.
- **Fees**: This is the total fee charged.
- **Fee per byte**: This field represents the total fee divided by the number of bytes in the transaction; for example, 10 Satoshis per byte.
- **Fee per weight unit**: For legacy transactions, this is calculated using the total number of bytes * 4. For SegWit transactions, it is calculated by combining a SegWit marker, flag, and witness field as one weight unit, and each byte of the other fields as four weight units.
- **Input Index**: This is the sequence number of the input.
- **Output index**: This is the sequence number of the output.
- **Output address**: This is where the bitcoins are going to.
- **Previous transaction ID**: This is the transaction ID of the previous transaction whose output(s) is used as input(s) in this transaction.
- **Previous output index**: This is the index of the previous output showing which output has been used as input in this transaction.
- **Value**: This is the amount of bitcoins.
- **Input address**: This is the address where the input is from.
- **Pkscript**: This is the unlock script for the input(s).
- **SigScript**: This is the signature for unlocking the input.
- **Witness**: This is the witness for this transaction—used only in SegWit.

Each transaction is composed of at least one input and output. Inputs can be thought of as coins being spent that have been created in a previous transaction, and outputs as coins being created. If a transaction is minting (mining) new coins rather than spending previously created coins, then there is no input, and therefore no signature is needed. This transaction is called a coinbase transaction.

Coinbase transactions

A coinbase transaction or generation transaction is always created by a miner and is the first transaction in a block. It is used to create new coins. It includes a special field, also called the **coinbase**, which acts as an input to the coinbase transaction. This transaction also allows up to 100 bytes of arbitrary data storage.

A coinbase transaction input has the same number of fields as a usual transaction input, but the structure contains the coinbase data size and fields instead of the unlocking script size and fields. Also, it does not have a reference pointer to the previous transaction. This structure is shown in the following table:

Field	**Size**	**Description**
Transaction hash	32 bytes	Set to all zeroes as no hash reference is used
Output index	4 bytes	Set to `0xFFFFFFFF`
Coinbase data length	1-9 bytes	2-100 bytes
Data	Variable	Any data
Sequence number	4 bytes	Set to `0xFFFFFFFF`

On the other hand, if a transaction should send coins to some other user (a Bitcoin address), then it needs to be signed by the sender with their private key. In this case, a reference is also required to the previous transaction to show the origin of the coins. Coins are unspent transaction outputs represented in Satoshis.



Bitcoin, being a digital currency, has various denominations. The smallest Bitcoin denomination is the **Satoshi**, which is equivalent to 0.00000001 BTC.

The transaction lifecycle

Now, let's look at the lifecycle of a Bitcoin **transaction**. The steps of the process are as follows:

1. A user/sender sends a transaction using wallet software or some other interface.
2. The wallet software signs the transaction using the sender's private key.
3. The transaction is broadcast to the Bitcoin network using a flooding algorithm, which is an algorithm to distribute data to every node in the network.
4. Mining nodes (miners) who are listening for the transactions verify and include this transaction in the next block to be mined. Just before the transactions are placed in the block, they are placed in a special memory buffer called the transaction pool.
5. Next, the mining starts, which is the process through which the blockchain is secured and new coins are generated as a reward for the miners who spend appropriate computational resources. Once a miner solves the **Proof of Work** (**PoW**) problem, it broadcasts the newly mined block to the network. The nodes verify the block and propagate the block further, and confirmations start to generate.

6. Finally, the confirmations start to appear in the receiver's wallet and after approximately three confirmations, the transaction is considered finalized and confirmed. However, three to six is just the recommended number; the transaction can be considered final even after the first confirmation. The key idea behind waiting for six confirmations is that the probability of double spending is virtually eliminated after six confirmations.

When a transaction is created by a user and sent to the network, it ends up in a special area on each Bitcoin software client. This special area is called the transaction pool. Also known as memory pools, transaction pools are created in local memory (computer RAM) by nodes (Bitcoin clients) to maintain a temporary list of transactions that have not yet been added to a block. Miners pick up transactions from these memory pools to create candidate blocks. Miners select transactions from the pool after they pass the verification and validity checks. We introduce how Bitcoin transactions are validated in the next section.

Transaction validation

This verification process is performed by Bitcoin nodes. There are three main things that nodes check when verifying a transaction:

1. That transaction inputs are previously unspent. This validation step prevents double spending by verifying that the transaction inputs have not already been spent by someone else.
2. That the sum of the transaction outputs is not more than the total sum of the transaction inputs. However, both input and output sums can be the same, or the sum of the input (total value) could be more than the total value of the outputs. This check ensures that no new bitcoins are created out of thin air.
3. That the digital signatures are valid, which ensures that the script is valid.

To send transactions on the Bitcoin network, the sender needs to pay a fee to the miners. The selection of which transactions to choose is based on the fee and their place in the order of transactions in the pool. Miners prefer to pick up transactions with higher fees.

Transaction fees

Transaction fees are charged by the miners. The fee charged is dependent upon the size and weight of the transaction. Transaction fees are calculated by subtracting the sum of the inputs from the sum of the outputs:

$$fee = sum(inputs) - sum(outputs)$$

The fees are used as an incentive for miners to encourage them to include users' transactions in the block the miners are creating. All transactions end up in the memory pool, from which miners pick up transactions based on their priority to include them in the proposed block. The calculation of priority is introduced later in this chapter; however, from a transaction fee point of view, a transaction with a higher fee will be picked up sooner by the miners. There are different rules based on which fee is calculated for various types of actions, such as sending transactions, inclusion in blocks, and relaying by nodes.

Fees are not fixed by the Bitcoin protocol and are not mandatory; even a transaction with no fee will be processed in due course but may take a very long time. This is, however, no longer practical due to the high volume of transactions and competing investors on the Bitcoin network; therefore, it is advisable to always provide a fee. The time taken for transaction confirmation usually ranges from 10 minutes to over 12 hours in some cases. The transaction time is also dependent on network activity. If the network is very busy, then naturally, transactions will take longer to process.

At times in the past, Bitcoin fees were so high that even for smaller transactions, a high fee was charged. This was due to the fact that miners are free to choose which transactions they pick to verify and add to a block, and they naturally select the ones with higher fees. A high number of users creating thousands of transactions also played a role in causing this situation of high fees because transactions were competing to be picked up first and miners picked up the ones with the highest fees. This fee is also usually estimated and calculated by the Bitcoin wallet software automatically before sending the transaction.

Mining and miners are concepts that we will look at a bit later in this chapter in the section about *Mining*.

The transaction data structure

A transaction, at a high level, contains metadata, inputs, and outputs. Transactions are combined to create a block's body. The general transaction data structure is shown in the following table:

Field	Size	Description
Version number	4 bytes	Specifies the rules to be used by the miners and nodes for transaction processing. There are two versions of transactions, that is, 1 and 2.
Flag	2 bytes or none	Either always 0001 or none, used to indicate the presence of witness data.
Input counter	1–9 bytes	The number (a positive integer) of inputs included in the transaction.
List of inputs	Variable	Each input is composed of several fields. These include: • The previous transaction hash • The index of the previous transaction • Transaction script length • Transaction script • Sequence number
Output counter	1–9 bytes	A positive integer representing the number of outputs.
List of outputs	Variable	Outputs included in the transaction. This field depicts the target recipient(s) of the bitcoins.
List of witnesses	Based on how many or none	Witnesses—1 for each input. Absent if the *Flag* field is absent.

Lock time	4 bytes	This field defines the earliest time when a transaction becomes valid. It is either a Unix timestamp or the block height.

The following is a sample decoded transaction:


```
{
    "txid": "d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d",
    "hash": "d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d",
    "version": 1,
    "size": 226,
    "vsize": 226,
    "weight": 904,
    "locktime": 0,
    "vin": [
        {
            "txid": "40120e43f00ff96e098a9173f14f1371655b3478bc0a558d6dc17a4ab17
6387d",
            "vout": 139,
            "scriptSig": {
                "asm":
"3045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7afbf92e
74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71758d[ALL]
02c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12",
                "hex":
"483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01757c8e4fa7afbf92e742
57c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5ef971e3f98ccba2c71758d012
102c16942555f5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12"
            },
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 0.00033324,
            "n": 0,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160
c568ffeb46c6a9362e44a5a49deaa6eab05a619a OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a914c568ffeb46c6a9362e44a5a49deaa6eab05a619a88ac",
                "address": "1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3",
                "type": "pubkeyhash"
            }
```

```
        },
        {
            "value": 0.00093376,
            "n": 1,
            "scriptPubKey": {
                "asm": "OP_DUP OP_HASH160
9386c8c880488e80a6ce8f186f788f3585f74aee OP_EQUALVERIFY OP_CHECKSIG",
                "hex": "76a9149386c8c880488e80a6ce8f186f788f3585f74aee88ac",
                "address": "1ET3oBGf8JpunjytE7owyVtmBjmvcDycQe",
                "type": "pubkeyhash"
            }
        }
    ]
}
```

As shown in the preceding decoded transaction, there are several structures that make up a transaction. All these elements will be described now.

Metadata

This part of the transaction contains values such as the size of the transaction, the number of inputs and outputs, the hash of the transaction, and a `locktime` field. Every transaction has a prefix specifying the version number. These fields are shown in the preceding example as `locktime`, `size`, `weight`, and `version`.

Inputs

Generally, each input (`vin`) spends a previous output. Each output is considered an **Unspent Transaction Output** (**UTXO**) until an input consumes it. A UTXO can be spent as an input to a new transaction. The transaction input data structure is explained in the following table:

Field	Size	Description
Transaction hash	32 bytes	The hash of the previous transaction with UTXO
Output index	4 bytes	This is the previous transaction's output index, such as UTXO to be spent
Script length	1-9 bytes	The size of the unlocking script
Unlocking script	Variable	The input script (`ScriptSig`), which satisfies the requirements of the locking script
Sequence number	4 bytes	Usually disabled or contains lock time— it being disabled is represented by `0xFFFFFFFF`

In the previous sample decoded transaction, the inputs are defined under the `"inputs" : [` section.

Outputs

Outputs (`vout`) have three fields, and they contain instructions for sending bitcoins. The first field contains the amount of Satoshis, whereas the second field contains the size of the locking script. Finally, the third field contains a locking script that holds the conditions that need to be met for the output to be spent. More information on transaction spending using locking and unlocking scripts and producing outputs is discussed later in this section.

The transaction output data structure is explained in the following table:

Field	Size	Description
Value	8 bytes	The total number (in positive integers) of Satoshis to be transferred
Script size	1-9 bytes	Size of the locking script
Locking script	Variable	Output script (`ScriptPubKey`)

In the previous sample decoded transaction, two outputs are shown under the `"vout":[ section`.

Verification

Verification is performed using Bitcoin's scripting language where transactions' cryptographic signatures are checked for validity, all inputs and outputs are checked, and the sum of all inputs must be equal to or greater than the sum of all outputs.

Now that we've covered the transaction lifecycle and data structure, let's move on to talk about the scripts used to undertake these transactions.

The Script language

Bitcoin uses a simple stack-based language called **Script** to describe how bitcoins can be spent and transferred. It is not Turing-complete and has no loops to avoid any undesirable effects of long-running/hung scripts on the Bitcoin network. This scripting language is based on a Forth programming language-like syntax and uses a reverse polish notation in which every operand is followed by its operators. It is evaluated from left to right using a **Last in, First Out** (**LIFO**) stack.

Scripts are composed of two components, namely elements and operations. Scripts use various operations (**opcodes**) or instructions to define their operations. Elements simply represent data such as digital signatures. Opcodes are also known as words, commands, or functions. Earlier versions of the Bitcoin node software had a few opcodes that are no longer used due to bugs discovered in their design.

The various categories of scripting opcodes are constants, flow control, stack, bitwise logic, splice, arithmetic, cryptography, and lock time.

A transaction script is evaluated by combining `ScriptSig` and `ScriptPubKey`:

- `ScriptSig` is the unlocking script, provided by the user who wishes to unlock the transaction

- `ScriptPubKey` is the locking script, is part of the transaction output, and specifies the conditions that need to be fulfilled in order to unlock and spend the output

We will look at a script execution in detail shortly.

Opcodes

In a computer, an opcode is an instruction to perform some operation. For example, `ADD` is an opcode, which is used for integer addition in Intel CPUs and various other architectures. Similarly, in Bitcoin design, opcodes are introduced that perform several operations related to Bitcoin transaction verification.

Descriptions of some of the most commonly used opcodes are listed in the following table, extracted from the Bitcoin Developer's Guide:

Opcode	Description
`OP_CHECKSIG`	This takes a public key and signature and validates the signature of the hash of the transaction. If it matches, then `TRUE` is pushed onto the stack; otherwise, `FALSE` is pushed.
`OP_EQUAL`	This returns `1` if the inputs are exactly equal; otherwise, `0` is returned.
`OP_DUP`	This duplicates the top item in the stack.
`OP_HASH160`	The input is hashed twice, first with SHA-256 and then with RIPEMD-160.
`OP_VERIFY`	This marks the transaction as invalid if the top stack value is not true.
`OP_EQUALVERIFY`	This is the same as `OP_EQUAL`, but it runs `OP_VERIFY` afterward.
`OP_CHECKMULTISIG`	This instruction takes the first signature and compares it against each public key until a match is found and repeats this process until all signatures are checked. If all signatures turn out to be valid, then a value of `1` is returned as a result; otherwise, `0` is returned.
`OP_HASH256`	The input is hashed twice with SHA-256.
`OP_MAX`	This returns the larger value of two inputs.

There are many opcodes in the Bitcoin scripting language and covering all of them is out of the scope of this book. However, all opcodes are declared in the `script.h` file in the Bitcoin reference client source code, available at `https://github.com/Bitcoin/Bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/script/script.h#L53`.

There are various standard scripts available in Bitcoin to handle the verification and value transfer from the source to the destination. These scripts range from very simple to quite complex depending upon the requirements of the transaction.

Standard transaction scripts

Standard transactions are evaluated using the `IsStandard()` and `IsStandardTx()` tests and only those transactions that pass the tests are allowed to be broadcast or mined on the Bitcoin network.

However, nonstandard transactions are also allowed on the network, as long as they pass the validity checks:

- **Pay-to-Public-Key Hash (P2PKH):** P2PKH is the most commonly used transaction type and is used to send transactions to Bitcoin addresses. The format of this type of transaction is shown as follows:

```
ScriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
ScriptSig: <sig> <pubKey>
```

  The `ScriptPubKey` and `ScriptSig` parameters are concatenated together and executed. An example will follow shortly in this section, where this is explained in more detail.

- **Pay-to-Script Hash (P2SH):** P2SH is used in order to send transactions to a script hash (that is, addresses starting with 3) and was standardized in BIP16. In addition to passing the script, the redeem script is also evaluated and must be valid. The template is shown as follows:

```
ScriptPubKey: OP_HASH160 <redeemScriptHash> OP_EQUAL
ScriptSig: [<sig>…<sign>] <redeemScript>
```

- **MultiSig (Pay to MultiSig):** The *M* of *N* multisignature transaction script is a complex type of script where it is possible to construct a script that requires multiple signatures to be valid in order to redeem a transaction. Various complex transactions such as escrow and deposits can be built using this script. The template is shown here:

```
ScriptPubKey: <m> <pubKey> [<pubKey> . . . ] <n> OP_CHECKMULTISIG
ScriptSig: 0 [<sig > . . . <sign>]
```

  Raw `multisig` is obsolete, and `multisig` is now usually part of the P2SH redeem script, mentioned in the previous bullet point.

- **Null data/OP_RETURN:** This script is used to store arbitrary data on the blockchain for a fee. The limit of the message is 40 bytes. The output of this script is unredeemable because `OP_RETURN` will fail the validation in any case. `ScriptSig` is not required in this case. The template is very simple and is shown as follows:

```
OP_RETURN <data>
```

The P2PKH script execution is shown in the following diagram:

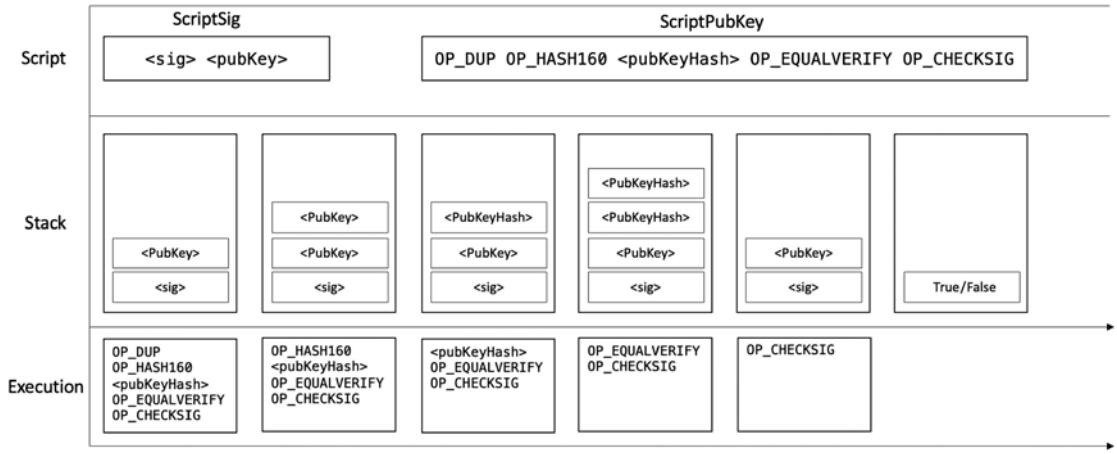


Figure 6.4: P2PKH script execution

In the preceding diagram, we have a standard P2PKH script presented at the top, which shows both the unlocking (`ScriptSig`) and locking (`ScriptPubKey`) parts of the script.

The unlocking script is composed of `<sig>` and `<pubkey>` elements, which are part of all transaction inputs. The unlocking script satisfies the conditions that are required to consume the output. The locking script defines the conditions that are required to be met to spend the bitcoins. Transactions are authorized by executing both parts collectively.

Now focusing again on *Figure 6.4*, we see that in the middle we have a visualization of the stack where the data elements are pushed and popped from. In the bottom section, we show the execution of the script. This diagram shows the step-by-step execution of the script with its outcome on the stack.

Now let's examine how this script is executed:

1. In the first step of the data elements, `<sig>` and `<pubkey>` are placed on the stack.
2. The stack item at the top, `<pubkey>`, is duplicated due to the `OP_DUP` instruction, which duplicates the top stack item.
3. After this, the instruction `OP_HASH160` executes, which produces the hash of `<pubkey>`, which is the top element in the stack.
4. `<pubkeyhash>` is then pushed on to the stack. At this stage, we have two hashes on the stack: the one that is produced as a result of executing `OP_HASH160` on `<pubkey>` from the unlocking script, and the other one provided by the locking script.
5. Now the `OP_EQUALVERIFY` opcode instruction executes and checks whether the top two elements (that is, the hashes) are equal or not. If they are equal, the script continues; otherwise, it fails.
6. Finally, `OP_CHECKSIG` executes to check the validity of the signatures of the top two elements of the stack. If the signature is valid then the stack will contain the value `True`, that is, `1`; otherwise, `False`, that is, `0`.

All transactions are encoded into hex format before being transmitted over the Bitcoin network. A sample transaction can be retrieved using `bitcoin-cli` on the Bitcoin node running on the main net as follows:

```
$ bitcoin-cli getrawtransaction
"d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d"
```

The output is shown in hex format:

```
{
"result":
"01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef90ff043
0e12408b0000006b483045022100de6fd8120d9f142a82d5da9389e271caa3a757b01
757c8e4fa7afbf92e74257c02202a78d4fbd52ae9f3a0083760d76f84643cf8ab80f5
ef971e3f98ccba2c71758d012102c16942555f5e633645895c9affcb994ea7910097b
7734a6c2d25468622f25e12ffffffff022c82000000000001976a914c568ffeb46c6
a9362e44a5a49deaa6eab05a619a88acc06c0100000000001976a9149386c8c880488
e80a6ce8f186f788f3585f74aee88ac00000000", "error": null, "id": null
}
```



As an alternative to client software that is locally installed, an online service can also be used, which is available here: `https://chainquery.com/bitcoin-cli/getrawtransaction`

Scripting is quite limited and can only be used to program one thing—the transfer of bitcoins from one address to other addresses. However, there is some flexibility possible when creating these scripts, which allows for certain conditions to be put on the spending of the bitcoins. This set of conditions can be considered a basic form of a financial contract.

Contracts

Contracts are Bitcoin scripts that use the Bitcoin blockchain to enforce a financial agreement. This is a simple definition but has far-reaching consequences as it allows users to programmatically create complex contracts that can be used in many real-world scenarios. Contracts allow the development of fully decentralized, independent, and reduced-risk platforms by programmatically enforcing different conditions for unlocking bitcoins. With the security guarantees provided by the Bitcoin blockchain, it is almost impossible to circumvent these conditions.



Note that these are not the same as **smart contracts**, which allow the writing of arbitrary programs on the blockchain. We will discuss these further in *Chapter 8*, *Smart Contracts*.

Various contracts, such as escrow, arbitration, and micropayment channels, can be built using the Bitcoin scripting language. The current implementation of the Script language is minimal, but it is still possible to develop various types of complex contracts. For example, we could set the release of funds to be allowed only when multiple parties sign the transaction or release the funds only after a specific amount of time has elapsed. Both scenarios can be realized using the `multisig` and transaction lock time options.

Even though the scripting language in Bitcoin can be used to create complex contracts, it is quite limited and is not at all on par with smart contracts, which are Turing-complete constructs and allow arbitrary program development. Recently, however, there has been some more advancement in this area. A new smart contract language for Bitcoin has been announced, called **Miniscript**, which allows for a more structured approach to writing Bitcoin scripts. Even though Bitcoin Script supports combinations of different spending conditions, such as time and hash locks, it is not easy to analyze existing scripts or build new complex scripts. Miniscript makes it easier to write complex spending rules. It also makes it easier to ascertain the correctness of the script. Currently, Miniscript supports P2WSH and P2SH-P2WSH, and offers limited support for P2SH.



More information on Miniscript is available at `http://bitcoin.sipa.be/miniscript/`.

We will now introduce some of Bitcoin's infamous shortcomings.

Transaction bugs

Even though transaction construction and validation are generally secure and sound processes, some vulnerabilities exist in Bitcoin. The following are two major Bitcoin vulnerabilities that have been infamously exploited:

- **Transaction malleability** is a Bitcoin attack that was introduced due to a bug in the Bitcoin implementation. Due to this bug, it became possible for an adversary to change the transaction ID of a transaction, thus resulting in a scenario where it appears that a certain transaction has not been executed. This can allow scenarios where double deposits or withdrawals can occur. This was fixed under BIP62, which fixed several causes of malleability. You can find more details here: `https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki`.
- **Value overflow**: This incident is one of the most well-known events in Bitcoin history. On August 15, 2010, a transaction was discovered that created roughly 184 billion bitcoins. This problem occurred due to the integer overflow bug where the amount field in the Bitcoin code was defined as a signed integer instead of an unsigned integer. This bug meant that the amount could also be negative and resulted in a situation where the outputs were so large that the total value resulted in an overflow. To the validation logic in the Bitcoin code, all appeared to be correct, and it looked like the fee was also positive (after the overflow). This bug was fixed via a soft fork (more on this in the *Blockchain* section) quickly after its discovery.



More information on this vulnerability is available at `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-5139`.

Security research in general, and specifically in the world of Bitcoin (cryptocurrency/blockchain), is a fascinating subject; perhaps some readers will look at the vulnerability examples at the following links for inspiration and embark on a journey to discover some more vulnerabilities.

An example of a recently resolved critical problem in Bitcoin, which remained undiscovered for quite some time, can be found at `https://Bitcoincore.org/en/2019/11/08/CVE-2017-18350/`.

Another example of a critical inflation and denial-of-service bug, which was discovered on September 17, 2018 and fixed rapidly, is detailed at `https://Bitcoincore.org/en/2018/09/20/notice/`.

Perhaps there are still some bugs that are yet to be discovered!

Let's now look at the Bitcoin blockchain, which is the foundation of the Bitcoin cryptocurrency.

Blockchain

The Bitcoin blockchain can be defined as a public, distributed ledger holding a timestamped, ordered, and immutable record of all transactions on the Bitcoin network. Transactions are picked up by miners and bundled into blocks for mining. Each block is identified by a hash and is linked to its previous block by referencing the previous block's hash in its header.

Structure

The data structure of a Bitcoin block is shown in the following table:

Field	Size	Description
Block size	4 bytes	The size of the block.
Block header	80 bytes	This includes fields from the block header described in the next section.
Transaction counter	Variable	The field contains the total number of transactions in the block, including the coinbase transaction. The size ranges from 1-9 bytes.
Transactions	Variable	All transactions in the block.

The block header mentioned in the previous table is a data structure that contains several fields. This is shown in the following table:

Field	Size	Description
Version	4 bytes	The block version number that dictates the block validation rules to follow.
Previous block's header hash	32 bytes	This is a double SHA-256 hash of the previous block's header.
Merkle root hash	32 bytes	This is a double SHA-256 hash of the Merkle tree of all transactions included in the block.
Timestamp	4 bytes	This field contains the approximate creation time of the block in the Unix epoch time format. More precisely, this is the time when the miner started hashing the header (the time from the miner's location).
Difficulty target	4 bytes	This is the current difficulty target of the network/block.
Nonce	4 bytes	This is a number that miners change repeatedly to produce a hash that is lower than the difficulty target.

As shown in the following diagram, a blockchain is a chain of blocks where each block is linked to the previous block by referencing the previous block header's hash. This linking makes sure that no transaction can be modified unless the block that records it and all blocks that follow it are also modified. The first block is not linked to any previous block and is known as the genesis block:

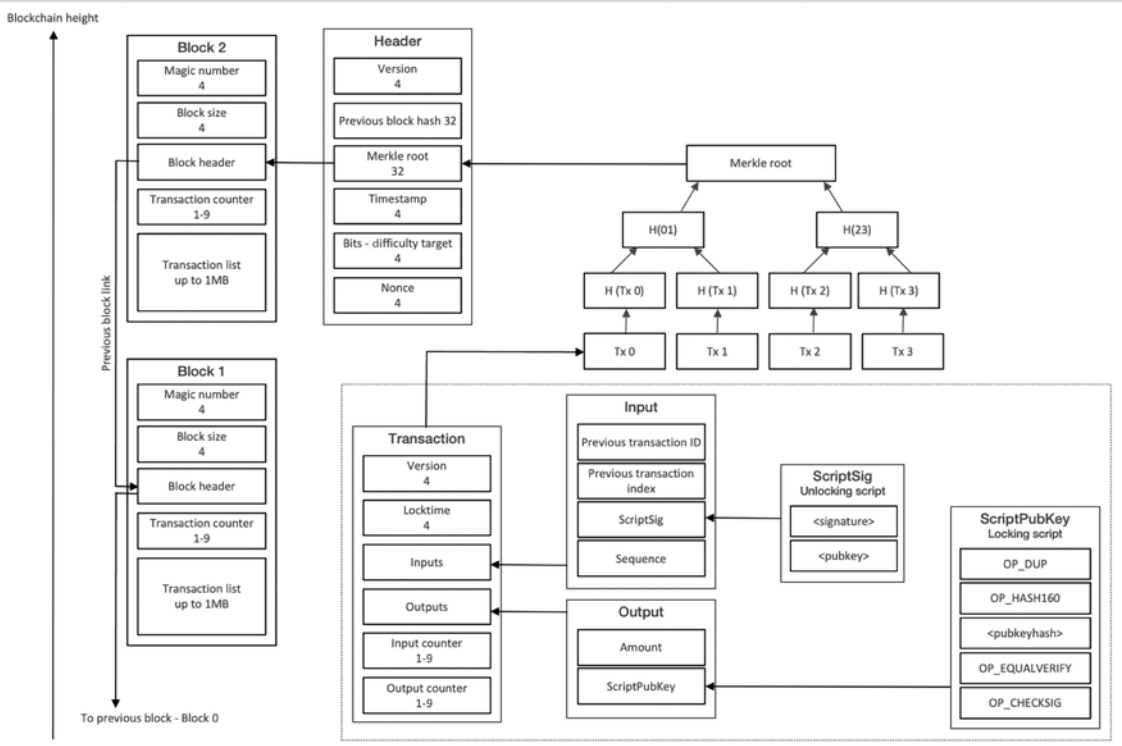


Figure 6.5: A visualization of a blockchain, block, block header, transactions, and scripts

On the left-hand side, blocks are shown starting from bottom to top. Each block contains transactions and block headers, which are further magnified on the right-hand side. At the top, first, the block header is enlarged to show various elements within the block header. Then on the right-hand side, the Merkle root element of the block header is shown in a magnified view, which shows how the Merkle root is constructed.



We have discussed Merkle trees in detail previously. You can refer to *Chapter 4*, *Asymmetric Cryptography*, if you need to revise the concept.

Further down the diagram, transactions are also magnified to show the structure of a transaction and the elements that it contains. Also, note that transactions are then further elaborated to show what locking and unlocking scripts look like. The size (in bytes) of each field of block, header, and transaction is also shown as a number under the name of the field.

Let's move on to discuss the first block in the Bitcoin blockchain: the genesis block.

The genesis block

This is the first block in the Bitcoin blockchain. The genesis block was hardcoded in the Bitcoin core software. In the genesis block, the coinbase transaction included a comment taken from *The Times* newspaper, proving that the Bitcoin genesis block was not mined earlier than January 3, 2009:

```
"The Times 03/Jan/2009 Chancellor on brink of second bailout for banks"
```

The following representation of the genesis block code can be found in the `chainparams.cpp` file available at `https://github.com/Bitcoin/Bitcoin/blob/master/src/chainparams.cpp`:

```
static CBlock CreateGenesisBlock(uint32_t nTime, uint32_t nNonce, uint32_t
nBits, int32_t nVersion, const CAmount& genesisReward)
{
    const char* pszTimestamp = "The Times 03/Jan/2009 Chancellor on brink of
second bailout for banks";
    const CScript genesisOutputScript = CScript() <<
ParseHex("04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0ea1f61
deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702b6bf11d5f") << OP_
CHECKSIG;
    return CreateGenesisBlock(pszTimestamp, genesisOutputScript, nTime, nNonce,
nBits, nVersion, genesisReward);
}
```

The block height is the number of blocks before a particular block in the blockchain. PoW is used to secure the blockchain. Each block contains one or more transactions, the first of which is the coinbase transaction. There is a special condition for coinbase transactions that prevents them from being spent until at least 100 blocks have passed to avoid a situation where the block may be declared stale later.

Stale and orphan blocks

Stale blocks are old blocks that have already been mined. Miners who keep working on these blocks due to a fork, where the longest chain (main chain) has already progressed beyond those blocks, are said to be working on a stale block. In other words, these blocks exist on a shorter chain, and will not provide any reward to their miners.

Orphan blocks are a slightly different concept. Their parent blocks are unknown. As their parents are unknown, they cannot be validated. This problem occurs when two or more miners discover a block at almost the same time. These are valid blocks and were correctly discovered at some point in the past but now they are no longer part of the main chain. The reason why this occurs is that if there are two blocks discovered at almost the same time, the one with a larger amount of **Proof of Work** (**PoW**) will be accepted and the one with a lower amount of work will be rejected. Similar to stale blocks, they do not provide any reward to their miners.

We can see this concept visually in the following diagram:

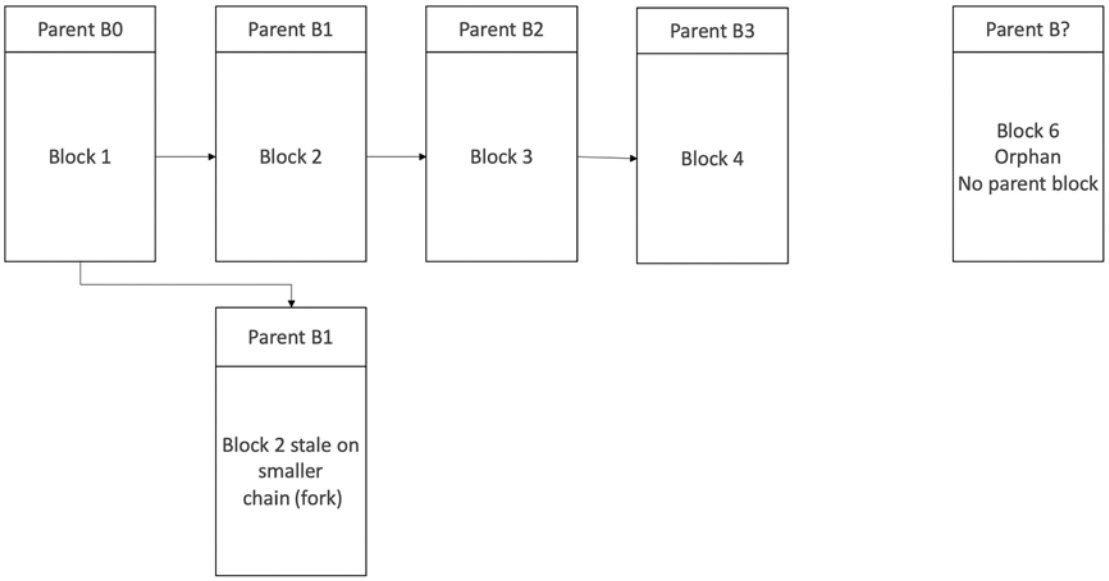


Figure 6.6: Orphan and stale blocks

Forks

In the preceding introduction to stale blocks, we introduced a new term, a **fork**. A fork is a condition that occurs when two different versions of the blockchain exist. It is acceptable in some conditions and detrimental in a few others.

Because of the distributed nature of Bitcoin, network forks can occur inherently. In cases where two nodes simultaneously announce a valid block, it can result in a situation where there are two blockchains with different transactions. This is an undesirable situation but can be addressed by the Bitcoin network only by accepting the longest chain.

In this case, the smaller chain will be considered orphaned. If an adversary manages to gain control of 51% of the network hash rate (computational power), then they can impose their own version of the transaction history.

There are different types of forks that can occur in a blockchain:

- Temporary forks
- Soft forks
- Hard forks

Temporary forks can occur naturally in Bitcoin protocol when two blocks are mined roughly at the same time and a tree-like structure starts to emerge with two or more branches. However, due to the chain choice rule, which enforces the selection of the longest chain among all the branches, the shorter chain is removed. Longest chain means the chain that has the most accumulated work (difficulty) behind it.

In the case of a **soft fork**, a client that chooses not to upgrade to the latest version supporting the updated protocol will still be able to work and operate normally. In this case, new and previous blocks are both acceptable, thus making a soft fork backward compatible. Miners are only required to upgrade to the new soft-fork client software to make use of the new protocol rules. Planned upgrades do not necessarily create forks because all users should have updated the software already.

A **hard fork**, on the other hand, invalidates previously valid blocks and requires all users to upgrade. New transaction types are sometimes added as a soft fork, and any changes such as block structure changes or major protocol changes result in a hard fork.

As Bitcoin evolves and new upgrades and innovations are introduced in it, the version associated with blocks also changes. These versions introduce various security parameters and new features.



Details of BIP0065 are available at `https://github.com/Bitcoin/bips/blob/master/bip-0065.mediawiki`.

Properties

Bitcoin is an ever-growing chain of blocks and is increasing in size. The current size of the Bitcoin blockchain stands at approximately 432 GB. The graph at `https://www.blockchain.com/charts/blocks-size` shows the current and historic size. As the chain grows and more miners are added to the network, the network difficulty also increases. Network difficulty refers to a measure of how difficult it is to find a new block, or in other words, how difficult it is to find a hash below the given target.

New blocks are added to the blockchain approximately every 10 minutes, and the network difficulty is adjusted dynamically every 2,016 blocks (roughly every two weeks) in order to maintain a steady addition of new blocks to the network.

Network difficulty is calculated using the following formula:

$$\text{Difficulty}_{\text{new}} = \frac{\text{Difficulty}_{\text{previous}} \times (2016 \times 10 \text{ minutes})}{\text{time taken to mine last 2016 blocks}}$$

The *previous difficulty* represents the old target value, and 2016 * 10 is the *total* time taken to generate the previous 2,016 blocks. Network difficulty essentially means how hard it is for miners to find a new block; that is, how difficult the hashing puzzle is now.

In the next section, mining is discussed, which will explain how the hashing puzzle is solved.

Miners

Mining is a process by which new blocks are added to the blockchain. This process is resource-intensive due to the requirements of PoW, where miners compete to find a number less than the difficulty target of the network. This difficulty in finding the correct value (also sometimes called the **mathematical puzzle**) is there to ensure that miners have spent the required resources before a new proposed block can be accepted. The miners mint new coins by solving the PoW problem, also known as the partial hash inversion problem. This process consumes a high amount of resources, including computing power and electricity. This process also secures the system against fraud and double-spending attacks while adding more virtual currency to the Bitcoin ecosystem.

Roughly one new block is created (mined) every 10 minutes to control the frequency of the generation of bitcoins. This frequency needs to be maintained by the Bitcoin network. It is encoded in the Bitcoin Core client to control the "money supply."

Approximately 144 blocks, that is, 1,728 bitcoins, are generated per day. The number of actual coins can vary per day; however, the number of blocks remains at an average of 144 per day. Bitcoin supply is also limited. In 2140, all 21 million bitcoins will have finally been created, and no new bitcoins will be able to be created after that. Bitcoin miners, however, will still be able to profit from the ecosystem by charging transaction fees.

Once a node connects to the Bitcoin network, there are several tasks that a Bitcoin miner performs:

1. **Syncing up with the network**: Once a new node joins the Bitcoin network, it downloads the blockchain by requesting historical blocks from other nodes. This is mentioned here in the context of the Bitcoin miner; however, this is not necessarily a task that only concerns miners. Other nodes, such as full nodes who are not necessarily mining, also sync with the network to update the local blockchain database.
2. **Transaction validation**: Transactions broadcast on the network are validated by full nodes by verifying and validating signatures and outputs.
3. **Block validation**: Miners and full nodes can start validating blocks received by them by evaluating them against certain rules. This includes the verification of each transaction in the block along with the verification of the nonce value.
4. **Creating a new block**: Miners propose a new block by combining transactions broadcast on the network after validating them.

5. **Performing PoW**: This task is the core of the mining process, and this is where miners find a valid block by solving a computational puzzle. The block header contains a 32-bit nonce field and miners are required to repeatedly vary the nonce until the resultant hash is less than a predetermined target.
6. **Fetch reward**: Once a node solves the hash puzzle (PoW), it immediately broadcasts the results, and other nodes verify it and accept the block. There is a slight chance that the newly minted block will not be accepted by other miners on the network due to a clash with another block found at roughly the same time, but once accepted, the miner is rewarded with bitcoins and any associated transaction fees.

Miners are rewarded with new coins if and when they discover new blocks by solving the PoW. Miners are paid transaction fees in return for the transactions in their proposed blocks. New blocks are created at an approximate fixed rate of every 10 minutes.

The rate of creation of new bitcoins decreases by 50% every 210,000 blocks, which is roughly every 4 years. When Bitcoin started in 2009, the mining reward used to be 50 bitcoins. After every 210,000 blocks, the block reward halves. In November 2012, it halved down to 25 bitcoins. Currently, since May 2020, it is 6.25 bitcoins per block. The next halving is expected to occur in early 2024, which will reduce the reward to 3.125 BTC. This mechanism is hardcoded in Bitcoin to regulate and control inflation and limit the supply of bitcoins. For miners to earn the reward, they must show that they have solved the computational puzzle. This is called the PoW.

Proof of Work (PoW)

This is proof that enough computational resources have been spent to build a valid block. PoW is based on the idea that a random node is selected every time to create a new block. In this model, nodes compete in proportion to their computing capacity to be selected. The following formula sums up the PoW requirement in Bitcoin:

$$H\ (N\ ||\ P_hash\ ||\ Tx\ ||\ Tx\ ||\ ...\ Tx) < Target$$

Here, *N* is a nonce, *P_hash* is a hash of the previous block, *Tx* represents the transactions in the block, and *Target* is the target network difficulty value. This means that the hash of the previously mentioned concatenated fields should be less than the target hash value.

The only way to find this nonce is the brute force method. Once a certain pattern of a certain number of zeroes is met by a miner, the block is immediately broadcast and accepted by other miners.

The mining algorithm consists of the following steps:

1. The previous block's header is retrieved from the Bitcoin network.
2. Assemble a set of transactions broadcast on the network into a block to be proposed.
3. Compute the double hash of the previous block's header, combined with a nonce and the newly proposed block, using the SHA-256 algorithm.
4. Check if the resulting hash is lower than the current difficulty level (the target). If so, then the PoW is solved. As a result of successful PoW, the discovered block is broadcast to the network and miners fetch the reward.

5. If the resultant hash is not less than the current difficulty level (target), then repeat the process after incrementing the nonce.

As the hash rate of the Bitcoin network increased, the total amount of the 32-bit nonce was exhausted too quickly. To address this issue, the extra nonce solution was implemented, whereby the coinbase transaction is used to provide a larger range of nonces to be searched by the miners. This process is visualized in the following flowchart:

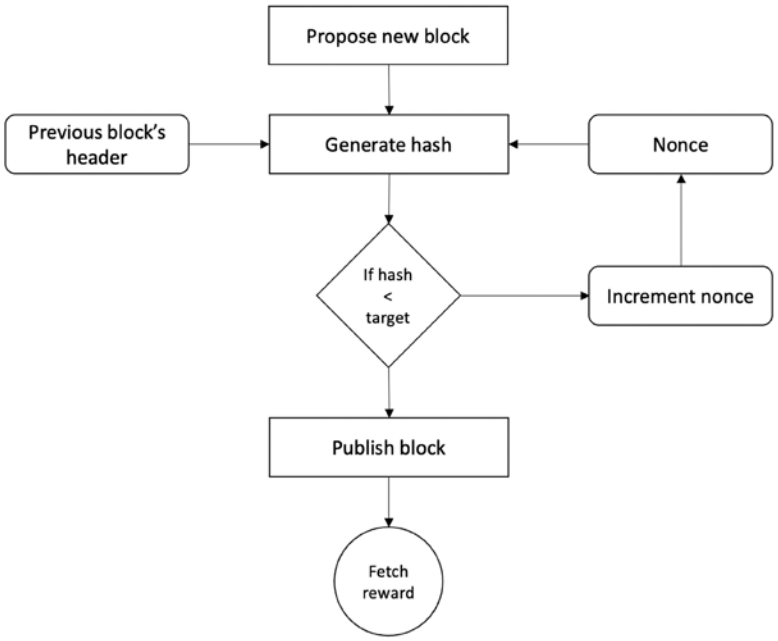


Figure 6.7: Mining process

Mining difficulty increases over time and bitcoins that could once be mined by a single-CPU laptop computer now require dedicated mining centers to solve the hash puzzle. The current difficulty level can be queried through the Bitcoin command-line interface using the following command:

```
$ bitcoin-cli getdifficulty
```

This generates something like the following:

```
36,835,682,546,787.98
```

This number represents the PoW difficulty level of the Bitcoin network. Recall from previous sections that miners compete to find a solution to a problem. This number, in fact, shows how difficult it is to find a hash lower than the network difficulty target. All successfully mined blocks must contain a hash that is less than this target number. This number is updated every 2 weeks or 2,016 blocks to ensure that on average, the 10-minute block generation time is maintained. Bitcoin network difficulty has increased in a roughly exponential fashion.



The difficulty of the Bitcoin network has increased quite significantly over the last few years. The latest difficulty graph is available here: `https://www.blockchain.com/charts/difficulty`.

The reason why mining difficulty increases is because, in Bitcoin, the block generation time always must be around ten minutes. This means that if blocks are being mined too quickly because of faster hardware, the difficulty increases accordingly so that the block generation time can remain at roughly ten minutes per block. This phenomenon is also true in reverse. If blocks take longer than ten minutes on average to mine, then the difficulty is decreased. The difficulty is calculated every 2,016 blocks (around two weeks) and adjusted accordingly. If the previous set of 2,016 blocks were mined in a period of less than two weeks, then the difficulty increases. Similarly, if 2,016 blocks were found in more than two weeks (bearing in mind that if blocks are mined every ten minutes, then 2,016 blocks take two weeks to be mined), then the difficulty is decreased.

The Bitcoin miners calculate hashes to solve the PoW algorithm. If the difficulty goes up then a higher hash rate is required to find the blocks. The difficulty increases accordingly if more hashing power is added due to more miners joining the network. The hash rate basically represents the rate of hash calculation per second. In other words, this is the speed at which miners in the Bitcoin network are calculating hashes to find a block.



The hash rate increases over time and is currently at 266 EH. To see the latest figure, a graph is available here: `https://www.blockchain.com/charts/hash-rate`

Mining systems

Over time, Bitcoin miners have used various methods to mine bitcoins. As the core principle behind mining is based on the double SHA-256 algorithm, over time, experts have developed sophisticated systems to calculate the hash faster and faster. The following is a review of the different types of mining methods used in Bitcoin and how they have evolved with time.

CPU

CPU mining was the first type of mining available in the original Bitcoin client. Users could even use laptop or desktop computers to mine bitcoins. CPU mining is no longer profitable and now more advanced mining methods such as ASIC-based mining are used. CPU mining only lasted for around a year from the introduction of Bitcoin, and soon other methods were explored and tried by miners.

GPU

Due to the increased difficulty of the Bitcoin network and the general tendency of finding faster methods to mine, miners started to use the GPUs or graphics cards available in PCs to perform mining. GPUs support faster and parallelized calculations, which are usually programmed using the OpenCL language.

This turned out to be a faster option as compared to CPUs. Users also used techniques such as overclocking to gain the maximum benefit of the GPU power. Also, the possibility of using multiple graphics cards in parallel increased the popularity of graphics cards' usage for Bitcoin mining. GPU mining, however, has some limitations, such as overheating and the requirement for specialized motherboards and extra hardware to house multiple graphics cards. From another angle, graphics cards have become quite expensive due to increased demand, and this has impacted gamers and graphics software users.

FPGAs

Even GPU mining did not last long, and soon miners found another way to perform mining using **Field Programmable Gate Arrays** (**FPGAs**).



It should be noted that GPU mining is still profitable for some other cryptocurrencies to some extent, because the network difficulty is much lower than that of Bitcoin.

An FPGA is basically an integrated circuit that can be programmed to perform specific operations. FPGAs are usually programmed in **Hardware Description Languages** (**HDLs**), such as Verilog and VHDL. Double SHA-256 quickly became an attractive programming task for FPGA programmers and several open-source projects were started too. FPGAs offered much better performance compared to GPUs; however, issues such as accessibility, programming difficulty, and the requirement for specialized knowledge to program and configure FPGAs resulted in a short life for the FPGA era of Bitcoin mining.

Mining hardware such as the X6500 miner, Ztex, and Icarus was developed during the time when FPGA mining was profitable. Various FPGA manufacturers, such as Xilinx and Altera, produce FPGA hardware and development boards that can be used to program mining algorithms.

ASICs

ASICs were designed to perform SHA-256 operations. These special chips were sold by various manufacturers and offered a very high hashing rate. The arrival of ASICs resulted in the quick phasing out of FPGA-based systems for mining.

This worked for some time, but due to the quickly increasing mining difficulty level, single-unit ASICs are no longer profitable. With the current difficulty factor (as of May 2022), if a miner manages to produce a hash rate of 100 trillion hashes per second (TH/s), they can hope to make 0.0004374 BTC (around $12) per day, and around $4,456.61 a year, which is very low compared to the investment required to source the equipment that can produce 100 TH/s, roughly 20,000 USD. Including running costs such as electricity, this turns out to be not very profitable at all.

Now, professional mining centers using thousands of ASIC units in parallel are offering mining contracts to users to perform mining on their behalf. There is no technical limitation—a single user can run thousands of ASICs in parallel—but it will require dedicated data centers and hardware; therefore, the cost for an individual can become prohibitive.

Mining pools

A mining pool forms when a group of miners work together to mine a block. The pool manager receives the coinbase transaction if the block is successfully mined and is then responsible for distributing the reward to the group of miners who invested resources to mine the block. This is more profitable than solo mining, where only one sole miner is trying to solve the partial hash inversion function (hash puzzle) because, in mining pools, the reward is paid to each member of the pool regardless of whether they (or more specifically, their individual node) solved the puzzle or not.

There are various models that a mining pool manager can use to pay miners, such as the pay-per-share model and the proportional model. In the pay-per-share model, the mining pool manager pays a flat fee to all miners who participated in the mining exercise, whereas in the proportional model, the share is calculated based on the amount of computing resources spent to solve the hash puzzle.



Many commercial pools now exist and provide mining service contracts via the cloud and easy-to-use web interfaces. The most used ones are AntPool, ViaBTC, F2Pool, and Poolin. A comparison of the hashing power of major mining pools is available here: `https://blockchain.info/pools`

Mining centralization can occur if a pool manages to control more than 51% of the network by generating more than 51% of the hash rate of the Bitcoin network. As discussed earlier in the introduction, a 51% attack can result in successful double-spending attacks, and it can impact consensus and in fact, even impose another version of the transaction history on the Bitcoin network. This event has happened once in Bitcoin history when GHash.io, a large mining pool, managed to acquire more than 51% of the network capacity. Theoretical solutions, such as two-phase PoW, have been proposed in academia to disincentivize large mining pools.



The following article describes a two-phase PoW proposal, in order to disincentivize large Bitcoin mining pools: `http://hackingdistributed.com/2014/06/18/how-to-disincentivize-large-bitcoin-mining-pools/`.

This scheme introduces a second cryptographic puzzle that results in mining pools either revealing their private keys or providing a considerable portion of the hash rate of their mining pool, thus reducing the overall hash rate of the pool.

The race to build efficient miners is on and is only expected to grow, though not infinitely. However, there is a limit to hardware acceleration and physical limitations. Soon, no room will be left for any on-chip optimizations. Or, it may become extremely challenging to achieve any further optimizations unless some fundamental change occurs in the way hardware silicone is developed.

Network

The Bitcoin network is a **P2P** network where nodes perform transactions. They verify and propagate transactions and blocks. As we have just seen, nodes called miners also produce blocks. A full Bitcoin node performs four functions. These are wallet, miner, blockchain, and network routing.

A Bitcoin network is identified by its magic value. Magic values are used to indicate the message's origin network. A list of these networks and values is shown in the following table:

Network	Magic value	Description
main	0xD9B4BEF9	Bitcoin main network
testnet	0xDAB5BFFA	First Bitcoin test network
testnet3	0x0709110B	Current Bitcoin test network
signet(default)	0x40CF030A	Bitcoin test network with an additional signature requirement for block validation

Before we examine how the Bitcoin discovery protocol and block synchronization work, we need to understand the different types of messages that the Bitcoin protocol uses.

Types of messages

There are 27 types of protocol messages in total, but that's likely to increase over time as the protocol grows. The most used protocol messages and an explanation of them are listed as follows:

- **Version**: This is the first message that a node sends out to the network, advertising its version and block count. The remote node then replies with the same information and the connection is then established.
- **Verack**: This is the response of the version message accepting the connection request.
- **Inv**: This is used by nodes to advertise their knowledge of blocks and transactions.
- **Getdata**: This is a response to `inv`, requesting a single block or transaction identified by its hash.
- **Getblocks**: This returns an `inv` packet containing the list of all blocks starting after the last known hash or 500 blocks.
- **Getheaders**: This is used to request block headers in a specified range.
- **Tx**: This is used to send a transaction as a response to the `getdata` protocol message.
- **Block**: This sends a block in response to the `getdata` protocol message.
- **Headers**: This packet returns up to 2,000 block headers as a reply to the `getheaders` request.
- **Getaddr**: This is sent as a request to get information about known peers.
- **Addr**: This provides information about nodes on the network. It contains the number of addresses and an address list in the form of an IP address and port number.
- **Ping**: This message is used to confirm if the TCP/IP network connection is active.
- **Pong**: This message is the response to a `ping` message confirming that the network connection is live.

When a Bitcoin Core node starts up, first, it initiates the discovery of all peers. This is achieved by querying DNS seeds that are hardcoded into the Bitcoin Core client and are maintained by Bitcoin community members. This lookup returns several DNS A records. The Bitcoin protocol works on TCP port `8333` by default for the main network and TCP `18333` for the testnet.



DNS seeds are declared (hardcoded) in the `chainparams.cpp` file in the Bitcoin source code, which can be viewed on GitHub at the following link: `https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/chainparams.cpp#L116`.

First, the client sends a protocol message, `version`, which contains various fields, such as the version, services, timestamp, network address, nonce, and some other fields. The remote node responds with its own `version` message, followed by a `verack` message exchange between both nodes, indicating that the connection has been established.

After this, `getaddr` and `addr` messages are exchanged to find the peers that the client does not know. Meanwhile, either of the nodes can send a `ping` message to see whether the connection is still active. `getaddr` and `addr` are message types defined in the Bitcoin protocol.

This process is shown in the following diagram of the protocol:

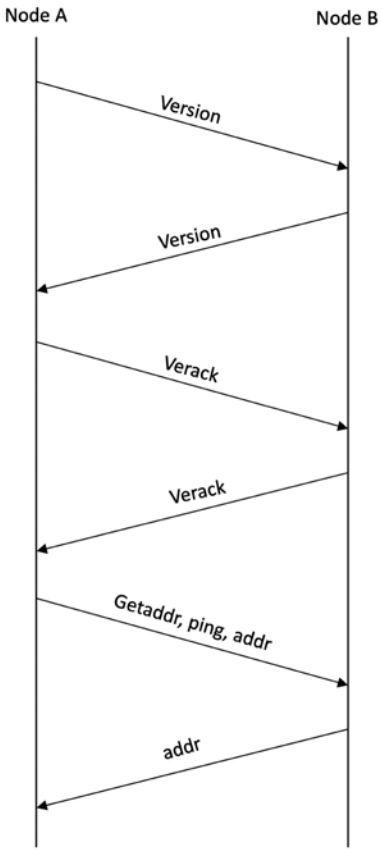


Figure 6.8: Visualization of node discovery protocol

This network protocol sequence diagram shows communication between two Bitcoin nodes during initial connectivity. **Node A** is shown on the left-hand side and **Node B** on the right.

First, **Node A** starts the connection by sending a version message that contains the version number and current time to the remote peer, **Node B**. **Node B** then responds with its own version message containing the version number and current time. **Node A** and **Node B** then exchange a `verack` message, indicating that the connection has been successfully established. Once this connection is successful, the peers can exchange `getaddr` and `addr` messages to discover other peers on the network.

Now, the block download can begin. In version 0.10.0, the initial block download method named *headers-first* was introduced. This resulted in major performance improvement, and blockchain synchronization that used to take days to complete started taking only a few hours. The core idea is that the new node first asks peers for block headers and then validates them. Once this has been completed, blocks are requested in parallel from all available peers. This happens because the blueprint of the complete chain is already downloaded in the form of the block header chain.

In this method, when the client starts up, it checks whether the blockchain is fully synchronized if the header chain is already synchronized; if not, which is the case the first time the client starts up, it requests headers from other peers using the `getheaders` message. If the blockchain is fully synchronized, it listens for new blocks via `inv` messages, and if it already has a fully synchronized header chain, then it requests blocks using `getdata` protocol messages. The node also checks whether the header chain has more headers than blocks, and then it requests blocks by issuing the `getdata` protocol message:

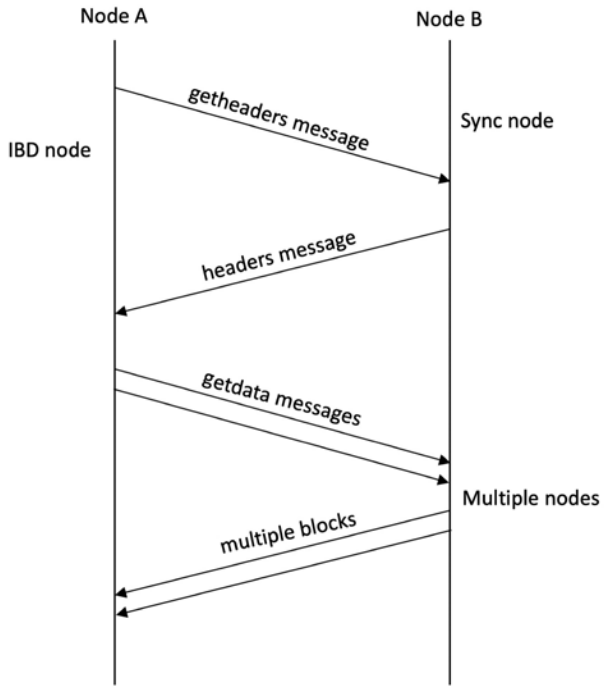


Figure 6.9: Bitcoin Core client >= 0.10.0 header and block synchronization

The preceding diagram shows the Bitcoin block synchronization process between two nodes on the Bitcoin network. **Node A**, shown on the left-hand side, is called an **Initial Block Download** (**IBD**) node, and **Node B**, shown on the right, is called a **sync node**.

IBD node means that this is the node that is requesting the blocks, while **sync node** means the node where the blocks are being requested from. The process starts with Node A first sending the `getheaders` message, which is met with a `getheaders` message response from the sync node. The payload of the `getheaders` message is one or more header hashes. If it's a new node, then there is only the first genesis block's header hash. Sync Node B replies by sending up to 2,000 block headers to IBD Node A. After this, the IBD node, Node A, starts to download more headers from Node B and blocks from multiple nodes in parallel; that is, it acts as the IBD and receives multiple blocks from multiple nodes, including Node B. If the sync node does not have more headers than 2,000 when the IBD node makes a `getheaders` request, the IBD node sends a `getheaders` message to other nodes. This process continues in parallel until the blockchain synchronization is complete.

The `getblockchaininfo` and `getpeerinfo` **Remote Procedure Calls** (**RPCs**) were updated with new functionality to cater to this change. An RPC known as `getchaintips` is used to list all known branches of the blockchain. This also includes headers-only blocks, `getblockchaininfo` is used to provide information about the current state of the blockchain. `getpeerinfo` is used to list both the number of blocks and the headers that are common between peers.

Wireshark can also be used to visualize message exchange between peers and can serve as an invaluable tool to learn about the Bitcoin protocol. A sample of this is shown here. This is a basic example showing the `version`, `verack`, `getaddr`, `ping`, `addr`, and `inv` messages. In the details, valuable information such as the packet type, command name, and results of the protocol messages can be seen:

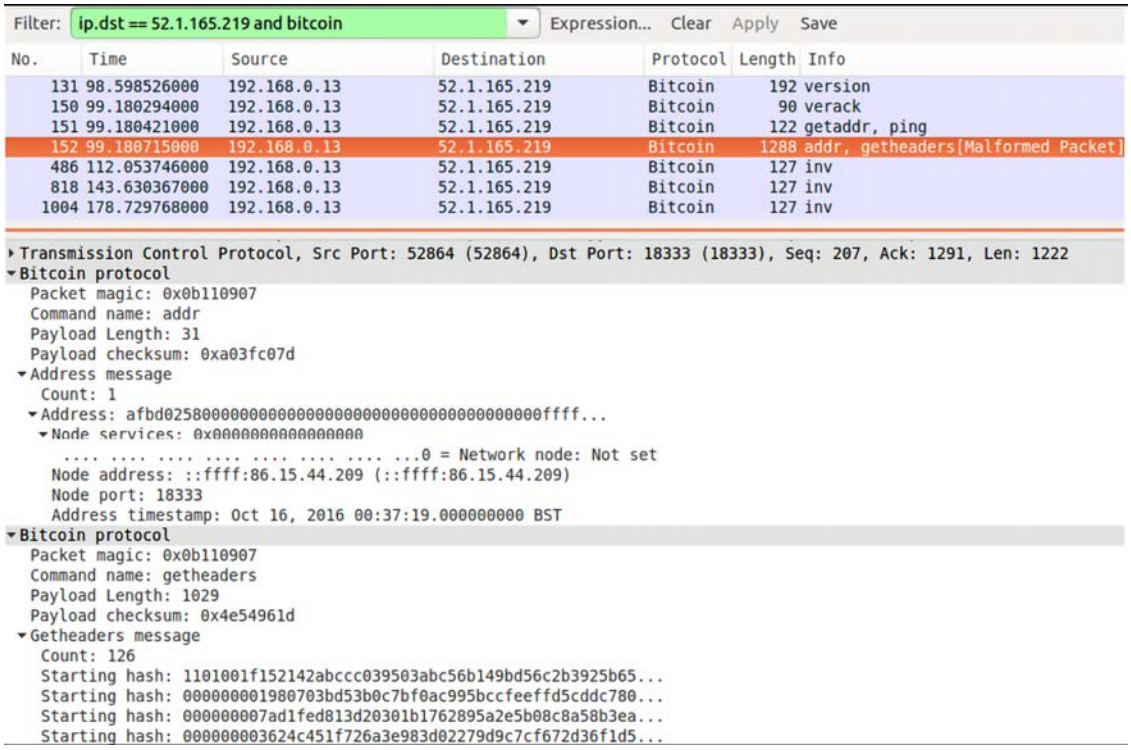


Figure 6.10: A sample block message in Wireshark

A protocol graph showing the flow of data between the two peers can be seen in the preceding screenshot. This can help you understand when a node starts up and what type of messages are used. This view can be accessed by selecting the appropriate network card interface and then applying the filter for Bitcoin.

In the following example, the Bitcoin dissector is used to analyze the traffic and identify the Bitcoin protocol commands. The exchange of messages such as `version`, `getaddr`, and `getdata` can be seen, along with the appropriate comment describing the message name.

This can be a very useful exercise in order to learn about the Bitcoin protocol and it is recommended that experiments be carried out on the Bitcoin testnet (`https://en.bitcoin.it/wiki/Testnet`), where various messages and transactions can be sent over the network and then analyzed by Wireshark.



Wireshark is a network analysis tool and is available at `https://www.wireshark.org`.

The analysis being performed here by Wireshark shows messages being exchanged between two nodes. If you look closely, you'll notice that the top three messages show the node discovery protocol that we introduced earlier:

Time	192.168.0.13 (57868) → 136.243.139.96 (18333)	Comment
97.734135000	version	Bitcoin: version
98.025045000	verack	Bitcoin: verack
98.025177000	getaddr, pin…	Bitcoin: getaddr, ping, addr
98.025468000	getheaders, …	Bitcoin: getheaders, [unknown command], [unknown command], [unknown command], headers
98.160419000	[TCP Retran…	Bitcoin: [TCP Retransmission] , getheaders, [unknown command], [unknown command], [unknown command]
98.598399000	getdata	Bitcoin: getdata
144.343544000	inv	Bitcoin: inv
176.152240000	getdata	Bitcoin: getdata
179.493755000	getdata	Bitcoin: getdata
218.101646000	ping	Bitcoin: ping
218.192004000	[unknown co…	Bitcoin: [unknown command]
218.444431000	[TCP Retran…	Bitcoin: [TCP Retransmission] , [unknown command]
336.234936000	getdata	Bitcoin: getdata
337.843423000	[unknown co…	Bitcoin: [unknown command]
338.143885000	ping	Bitcoin: ping
448.764093000	getdata	Bitcoin: getdata
457.894823000	[unknown co…	Bitcoin: [unknown command]
458.195265000	ping	Bitcoin: ping
578.011774000	[unknown co…	Bitcoin: [unknown command]
578.212044000	ping	Bitcoin: ping
585.587671000	inv	Bitcoin: inv
647.169633000	inv	Bitcoin: inv
671.962545000	getdata	Bitcoin: getdata
698.037067000	[unknown co…	Bitcoin: [unknown command]
698.237350000	ping	Bitcoin: ping
701.563581000	inv	Bitcoin: inv
701.986269000	inv	Bitcoin: inv
705.022173000	inv	Bitcoin: inv
812.115878000	inv	Bitcoin: inv
818.198570000	[unknown co…	Bitcoin: [unknown command]
818.298733000	ping	Bitcoin: ping

Figure 6.11: Bitcoin node discovery protocol in Wireshark

Nodes run different Bitcoin client software. We'll discuss those next.

Client software

There are different types of nodes on the network. The two main types of nodes are full nodes and **simple payment verification** (**SPV**) nodes.

Full nodes, as the name implies, are implementations of Bitcoin Core clients performing the wallet, miner, full blockchain storage, and network routing functions. These nodes download the entire blockchain; they provide the most secure method of validating the blockchain as a client and play a vital role in block propagation. However, it is not necessary for all nodes in a Bitcoin network to perform all these functions. Some nodes perform network routing functions only but do not perform mining or store private keys (the wallet function). Another type of node is solo miner nodes, which can perform mining, store full blockchains, and act as Bitcoin network routing nodes. Some nodes perform only mining functions and are called mining nodes.



Versioning information is coded in the Bitcoin client in the `version.h` file, which is available here: `https://github.com/bitcoin/bitcoin/blob/0cda5573405d75d695aba417e8f22f1301ded001/src/version.h#L9`.

It is possible to run SPV software that runs a wallet and network routing function without a blockchain while syncing with the network. SPV nodes only keep a copy of block headers of the current longest valid blockchain. When required, they can request transactions from full nodes. Verification is performed by looking at the Merkle branch, which links the transactions to the original block the transaction was accepted in. This is not very practical and requires a more pragmatic approach, which was implemented with BIP37, where bloom filters were used to filter for relevant transactions only. We will review bloom filters in the next section.

Finally, there are a few nonstandard but heavily used nodes. These are called pool protocol servers. These nodes make use of alternative protocols such as the stratum protocol, a line-based protocol that makes use of plain TCP sockets and human-readable JSON-RPC to operate and communicate between nodes. Stratum is commonly used to connect to mining pools. Nodes that only compute hashes use the Stratum protocol to submit their solutions to the mining pool.



Most protocols on the internet are line-based, which means that each line is delimited by a carriage return and newline `\r \n` character. More details on this protocol are available at this link: `https://en.bitcoin.it/wiki/Stratum_mining_protocol`.

Bloom filters

A bloom filter is a data structure (a bit vector with indexes) that is used to test the membership of an element in a probabilistic manner. It provides a probabilistic lookup with false positives but no false negatives.

This means that this filter can produce an output where an element that is not a member of the set being tested is wrongly considered to be in the set. Still, it can never produce an output where an element does exist in the set, but it asserts that it does not.

Elements are added to the bloom filter after hashing them several times and then setting the corresponding bits in the bit vector to 1 via the corresponding index. To check the presence of an element in the bloom filter, the same hash functions are applied and then compared with the bits in the bit vector to see whether the same bits are set to 1.

Note that not every hash function (such as SHA-1) is suitable for bloom filters as they need to be fast, independent, and uniformly distributed. Noncryptographic hash functions are used for bloom filters such as `fnv`, `murmur`, and `Jenkins`.

These filters are mainly used by SPV clients to request transactions and the Merkle blocks that they are interested in. A Merkle block is a lightweight version of the block, which includes a block header, some hashes, a list of 1-bit flags, and a transaction count. This information can then be used to build a Merkle tree. This is achieved by creating a filter that matches only those transactions and blocks from the full chain that have been requested by the SPV client. Once version messages have been exchanged and the connection is established between the peers, the nodes can set filters according to their requirements.

These probabilistic filters offer a varying degree of privacy or precision, depending on how accurately or loosely they have been set. A strict bloom filter will only filter transactions that have been requested by the node, but at the expense of the possibility of revealing the user addresses to adversaries who can correlate transactions with their IP addresses, thus compromising privacy.

On the other hand, a loosely set filter can result in retrieving more unrelated transactions but will offer more privacy. Also, for SPV clients, bloom filters allow them to use low bandwidth as opposed to downloading all transactions for verification.

BIP37 proposed the Bitcoin implementation of bloom filters and introduced three new messages to the Bitcoin protocol:

- `filterload`: This is used to set the bloom filter on the connection.
- `filteradd`: This adds a new data element to the current filter.
- `filterclear`: This deletes the currently loaded filter.



More details can be found in the BIP37 specification. This is available at `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`.

So far, we've discussed that, on a Bitcoin network, there are full clients (nodes), which perform the function of storing a complete blockchain. If you cannot run a full node, then SPV clients can be used to verify that particular transactions are present in a block by only downloading the block headers instead of the entire blockchain.

At times, even running an SPV node is not feasible (especially on low-resource devices such as mobile phones) and the requirement is only to be able to send and receive Bitcoin somehow. For this purpose, wallets (wallet software) are used that do not require downloading even the block headers.

Wallets

The wallet software is used to generate and store cryptographic keys. It performs various useful functions, such as receiving and sending Bitcoin, backing up keys, and keeping track of the balance available. Bitcoin client software usually offers both functionalities: a Bitcoin client and wallet. On disk, the Bitcoin Core client wallets are stored as a `Berkeley DB` file:

```
$ file wallet.dat
wallet.dat: Berkeley DB (Btree, version 9, native byte-order)
```

Private keys are generated by randomly choosing a 256-bit number provided by the wallet software. The rules of generation are predefined and were discussed in *Chapter 4*, *Asymmetric Cryptography*. Private keys are used by wallets to sign outgoing transactions. Wallets do not store any coins. In fact, in the Bitcoin network, coins do not exist; instead, only transaction information is stored on the blockchain (more precisely, UTXO, unspent outputs), which are then used to calculate the number of bitcoins.

Fundamentally, a cryptocurrency wallet depends on a **keystore** to store private keys. A keystore can be defined as a repository for storing keys. It can be as simple as a file or as complex as a **hardware security module** (**HSM**) or even a portable device such as a hardware wallet. The private key(s) must be protected against theft. Usually, encryption of the keystore and specialized secure hardware such as an HSM are used to protect the keys. These private keys are used in digital signatures to sign the transactions as proof of the ownership of the coins.

In Bitcoin and generally in cryptocurrencies, there are different types of wallets that can be used to store private keys. As software, they also provide some functions to the users to manage and carry out transactions on the Bitcoin network. Let's look at the common types of wallets:

- **Non-deterministic wallets**: These wallets contain randomly generated private keys and are also called **Just a Bunch of Key** wallets. The Bitcoin Core client generates some keys when first started and also generates keys as and when required. Managing many keys is very difficult and an error-prone process that can lead to the theft and, consequently, the loss of coins. Moreover, there is a need to create regular backups of the keys and protect them appropriately, for example, by encrypting them to prevent theft or loss.
- **Deterministic wallets**: In this type of wallet, keys are derived from a seed value via hash functions. This seed number is generated randomly and is commonly represented by human-readable **mnemonic code** words. Mnemonic code words are defined in BIP39, a Bitcoin improvement proposal for mnemonic code for generating deterministic keys. This BIP is available at `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki`. These phrases can be used to recover all keys and make private key management comparatively easier.
- **Hierarchical deterministic (HD) wallets**: Defined in BIP32 and BIP44, HD wallets store keys in a tree structure derived from a seed.

  The seed generates the parent key (master key), which is used to generate child keys and, subsequently, grandchild keys. Key generation in HD wallets does not generate keys directly; instead, it produces some information (private key generation information) that can be used to generate a sequence of private keys.

The complete hierarchy of private keys in an HD wallet is easily recoverable if the master private key is known.It is because of this property that HD wallets are very easy to maintain and are highly portable. There are many free and commercial HD wallets available, for example, **Trezor** (`https://trezor.io`), **Jaxx** (`https://jaxx.io`), and **Electrum** (`https://electrum.org/`).

- **Brain wallets**: The master private key can also be derived from a single human-readable and easy-to-remember (hence the phrase brain wallets) phrase. The idea here is that this easy-to-memorize passphrase is used to derive the private key and if used in HD wallets, this can result in a full HD wallet that is derived from a single memorized password. This is known as a brain wallet. While this is an easy-to-use solution for users, this method is prone to password guessing, dictionary attacks, and brute-force attacks. Some techniques such as **key stretching** can be used to slow down the progress made by the attacker.
- **Paper wallets**: As the name implies, this is a paper-based wallet with the required key material printed on it. It requires physical security to be stored. Paper wallets can be generated online from various service providers, such as `https://bitcoinpaperwallet.com/` or `https://www.bitaddress.org/`.
- **Hardware wallets**: Another method is to use a tamper-resistant device to store keys. This tamper-resistant device can be custom-built. With the advent of **Near-Field Communication** (**NFC**) enabled phones, this can also be a **secure element** (**SE**) in NFC phones. Trezor (`https://trezor.io`) and Ledger wallets (`https://www.ledger.com`) are the commonly used cryptocurrency hardware wallets.



Secure elements tamper-resistant chip that can securely host applications, confidential data, and cryptographic keys. Mobile smartphones, cryptocurrency hardware wallets, and tablets use secure elements to securely store sensitive data such as passwords, PIN codes, fingerprints, and in fact anything requiring confidentiality and integrity.

NFC chips short-range wireless communication between two devices in close proximity and is commonly used for contactless payments and other secure applications through smartphones.

- **Online wallets**: Online wallets, as the name implies, are stored entirely online and are provided as a service, usually via the cloud. They provide a web interface for users to manage their wallets and perform various functions, such as making and receiving payments. They are easy to use but require that the user trusts the online wallet service provider. An example of an online wallet is **GreenAddress**, which is available at `https://greenaddress.it/en/`.
- **Mobile wallets**: Mobile wallets, as the name suggests, are installed on mobile devices. They can provide us with various methods to make payments, most notably the ability to use smartphone cameras to scan QR codes quickly and make payments. There are many companies offering these wallets. It is, however, unwise to suggest which type of wallet should be used as it varies depending on personal preferences and the features available in the wallet. Therefore, we will not recommend any here.



An SPV client, which we introduced earlier, is also a type of Bitcoin wallet. Other types of Bitcoin wallets, especially mobile wallets, are mostly API wallets that rely on a mechanism where private and public keys are stored locally on the device where the wallet software is installed. Here, trusted backend servers are used for providing blockchain data via APIs.

The choice of Bitcoin wallet depends on several factors, such as security, ease of use, and available features. Out of all these attributes, security, of course, comes first, and when deciding which wallet to use, security should be of paramount importance.

Hardware wallets tend to be more secure compared to web wallets because of their tamper-resistant design. Web wallets, by their very nature, are hosted on websites, which may not be as secure as a tamper-resistant hardware device. Generally, mobile wallets for smartphone devices are quite popular due to a balanced combination of features, user experience, and security.

Summary

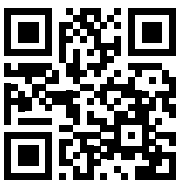
This chapter started with an introduction to Bitcoin and explained how a transaction works from the user's point of view. Then, an introduction to transactions from a technical point of view was presented. After this, the public and private keys used in Bitcoin were discussed. In the next section, we introduced addresses and their different types, followed by a discussion on transactions and their types and usage.

Following this, we looked at blockchain, with a detailed explanation of how blockchain works and the various components included in the Bitcoin blockchain. Next, mining processes and relevant concepts such as hardware systems, their limitations, and bitcoin rewards were introduced, along with an introduction to the Bitcoin network, followed by a discussion on Bitcoin node discovery and block synchronization protocols. Finally, we examined different types of Bitcoin wallets and discussed the various attributes and features of each type.

In the next chapter, we will examine some practical concepts related to Bitcoin payments, clients, and programming.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

7

Bitcoin in Practice

In this chapter, we will examine how the Bitcoin protocol works, as well as some advanced and modern Bitcoin protocols that have been developed to address limitations in the original Bitcoin protocol.

In this chapter, we will cover:

- Bitcoin in the real world
- Bitcoin payments
- Innovation in Bitcoin
- Bitcoin client installation
- Experimenting further with `bitcoin-cli`
- Bitcoin programming

Before we talk about the specifics of Bitcoin, let's briefly discuss the philosophy behind it, go over the official definition of Bitcoin, and consider Bitcoin from a user's perspective.

Bitcoin in the real world

For people with a libertarian ideology, Bitcoin is a platform that can be used instead of banks. However, some think that due to regulations, Bitcoin may become another institution that cannot be trusted. The original idea behind Bitcoin was to develop an e-cash system that requires no trusted third party and where users can be anonymous. If regulations require checks like **Know Your Customer** (**KYC**) and detailed information about business transactions to facilitate the regulatory process, then it might be too much information to share. As a result, Bitcoin may not remain attractive to some entities.

The regulation of Bitcoin is a controversial subject. As much as it is a libertarian's dream, law enforcement agencies, governments, and banks are proposing various regulations to control it.



Interested readers can read more about the regulation of Bitcoin and other relevant news and activities at `https://cointelegraph.com/tags/Bitcoin-regulation`.

At this point, the question arises that if Bitcoin is under so much pressure from regulatory bodies, then how has it managed to grow so significantly? The simple answer is due to its decentralized and trustless nature. In this context, the term *trustless* refers to the distribution of trust between users, rather than a central entity. No single entity can control this network, and even if some entities try to enforce some regulations, they can only go so far because the network is owned collectively by its users instead of a single entity. It is also protected by its PoW mechanism, which thwarts any adversarial attacks on the network. Moreover, the anonymity of the founder of Bitcoin has also played a role in its success.

The growth of Bitcoin is also due to the so-called **network effect**. Also called **demand-side economies of scale**, this is a concept that means that the more users use the network, the more valuable it becomes. Over time, an exponential increase has been seen in Bitcoin network growth. This increase in the number of users is mostly driven by financial incentives. Also, the scarcity of Bitcoin and its built-in inflation control mechanism gives it value, as there are only 21 million Bitcoins that can ever be mined. Also, the miner reward halves every four years, which increases scarcity, and consequently, the demand increases even more.

Despite its overall success, there are some concerns regarding Bitcoin as well. Bitcoin's **ESG** (**Environmental, Social, and Governance**) impact is also a concern. The main arguments include mining centralization where some powerful miners have most of the network (hash power), high energy consumption, and centralized development and governance. Moreover, Bitcoin volatility is another concern that is seen as a barrier to its day-to-day adoption.

However, despite the regulatory pressure, some limitations, and it even being made illegal in some countries, Bitcoin has gained significant acceptance in some parts of the world. For example, Bitcoin is legal tender in El Salvador and the Central African Republic.

In the next section, we will see how the Bitcoin network looks from a user's point of view—how a transaction is made, how it propagates from the user to the network, and how transactions are verified and finally accumulated in blocks.

Bitcoin payments

Having explored the various components of Bitcoin's architecture and design in the previous chapter, the following example will help you to understand how a payment transaction via the Bitcoin network looks from the end user's perspective. There are several steps involved in this process. In this example, we are using the **Blockchain wallet** for mobile devices.

The steps are described as follows:

1. First, either the payment is requested from a user, or the sender initiates a transfer to send money to another user. In both cases, the Bitcoin address of the beneficiary is required to be sent via an appropriate communication mechanism.
2. The sender either enters the receiver's address or scans the generated QR code that has the Bitcoin address, amount, and an optional description encoded in it. The wallet application recognizes this QR code and decodes it into something like:

```
"Please send <amount> BTC to address <receiver's Bitcoin address>".
```

With actual values, this will look like the following:

```
"Please send 0.00033324 BTC to address
1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3".
```

3. In the wallet application of the sender, this transaction is constructed, digitally signed using the private key of the sender, then broadcast to the Bitcoin network.
4. Bitcoin transactions are serialized for transmission over the network and encoded in hex format. As an example, this transaction is also shown in raw serialized hex format as follows:

```
01000000017d3876b14a7ac16d8d550abc78345b6571134ff173918a096ef
90ff0430e12408b0000006b483045022100de6fd8120d9f142a82d5da9389
e271caa3a757b01757c8e4fa7afbf92e74257c02202a78d4fbd52ae9f3a00
83760d76f84643cf8ab80f5ef971e3f98ccba2c71758d012102c16942555f
5e633645895c9affcb994ea7910097b7734a6c2d25468622f25e12ffffff
ff022c8200000000000001976a914c568ffeb46c6a9362e44a5a49deaa6ea
b05a619a88acc06c0100000000001976a9149386c8c880488e80a6ce8f18
6f788f3585f74aee88ac00000000
```

5. Once the QR code is decoded, the transaction will appear in the wallet. There are a number of parameters required for a transaction to work, such as `From`, `To`, `BTC`, and `Fee`. Bitcoin network fees ensure that your transaction will be included by miners in the block.
6. This transaction will be picked up by miners to be verified for legitimacy and included in the block. A confirmation will appear as soon as the transaction is verified, included in the candidate or proposed block, and mined.
7. Usually, at this point, users wait for up to six confirmations to be received before a transaction is considered final; however, a transaction can be considered final at the previous step. Confirmations serve as an additional mechanism to ensure that there is probabilistically a very low chance of a transaction being reverted, but otherwise, once a mined block is finalized and announced, the transactions within that block are final at that point.
8. The appropriate fee will be deducted from the original value to be transferred and will be paid to the miner who has included it in the block for mining.

In the transaction flow just described, a total payment of 0.00033324 BTC left the sender's address, 0.001267 BTC of which was paid to the receiver's address. A fee of 0.00010622 was deducted from the transaction as a mining fee.



Transactions are not encrypted and are publicly visible on the blockchain. We can see these details using a blockchain explorer, such as blockchain.info: `https://www.blockchain.com/btc/address/1JzouJCVmMQBmTcd8K4Y5BP36gEFNn1ZJ3`

Moreover, a view of various attributes of the transaction is available here: `https://www.blockchain.com/btc/tx/d28ca5a59b2239864eac1c96d3fd1c23b747f0ded8f5af0161bae8a616b56a1d`

Bitcoin can be accepted as payment using various techniques. It is increasingly being accepted as a payment method by many online merchants and e-commerce websites. There are a number of ways in which buyers can pay a business that accepts Bitcoin. For example, in an online shop, Bitcoin merchant solutions can be used, whereas in traditional, physical shops, **Point of Sale** (**POS**) terminals and other specialized hardware can be used. Customers can simply scan the QR barcode with the seller's payment URI in it and pay using their mobile devices. Bitcoin URIs allow users to make payments by simply clicking on links or scanning QR codes. A **Uniform Resource Identifier** (**URI**) is a string that represents the transaction information. The QR code can be displayed near the point of sale terminal, which can be decoded by wallets.

Various payment solutions, such as the 34 Bytes Bitcoin POS terminal, are available commercially. Generally, these solutions work by following these steps:

1. The salesperson enters the amount of money to be charged in fiat currency, for example, US dollars.
2. Once the value is entered into the system, the terminal prints a receipt with a QR code on it and other relevant information, such as the amount to be paid.
3. The customer can then scan this QR code using their mobile Bitcoin wallet to send the payment to the Bitcoin address of the seller embedded within the QR code.
4. Once the payment is received at the designated Bitcoin address, a receipt is printed out as physical evidence of the sale.

Bitcoin payment processors are offered by many online service providers. This allows integration with e-commerce websites to facilitate Bitcoin payments. These payment processors can be used to accept Bitcoin as payment. Some service providers also allow the secure storage of bitcoins, for example, **BitPay** (`https://bitpay.com`). Another example is the Bitcoin merchant solutions available at `https://www.bitcoin.com/merchant-solutions`.

Bitcoin, and blockchain technology in general, is continuously evolving, and we'll discuss some of the most relevant ideas next.

Innovation in Bitcoin

Bitcoin has undergone many changes and is still evolving into a more and more robust and better system by addressing various weaknesses in the system. Performance has been a topic of hot debate among Bitcoin experts and enthusiasts for many years. As such, various proposals have been made in the last few years to improve Bitcoin performance, resulting in greater transaction speed, increased security, payment standardization, and overall performance improvement at the protocol level.

These improvement proposals are usually made in the form of **Bitcoin Improvement Proposals** (**BIPs**) or fundamentally new versions of Bitcoin protocols, resulting in new networks altogether. Some of the changes proposed can be implemented via a soft fork, but a few need a hard fork and, as a result, give birth to a new currency.

Bitcoin improvement proposals

These documents, also referred to as **BIP**s, are used to propose improvements or inform the Bitcoin community about the improvements suggested, the design issues, or some aspects of the Bitcoin ecosystem. There are three types of BIPs:

- **Standard BIP**: Used to describe the major changes that have a major impact on the Bitcoin system; for example, block size changes, network protocol changes, or transaction verification changes.
- **Process BIP**: A major difference between standard and process BIPs is that standard BIPs cover protocol changes, whereas process BIPs usually deal with proposing a change in a process that is outside the core Bitcoin protocol. These are implemented only after a consensus among Bitcoin users.
- **Informational BIP**: These are usually used to just advise or record some information about the Bitcoin ecosystem, such as design issues.

Various BIPs have been proposed and finalized to introduce and standardize Bitcoin payments. Most notably, BIP70 (secure payment protocol) describes the protocol for secure communication between a merchant and customers. This protocol uses X.509 certificates for authentication and runs over HTTP and HTTPS. There are three messages in this protocol: **PaymentRequest**, **Payment**, and **PaymentACK**. The key features of this proposal are defense against man-in-the-middle attacks and secure proof of payment.

Man-in-the-middle attacks can result in a scenario where the attacker is sitting between the merchant and the buyer, and it would seem to the buyer that they are talking to the merchant, but in fact, the man in the middle is interacting with the buyer instead of the merchant. This can result in the manipulation of the merchant's Bitcoin address to defraud the buyer.

Several other BIPs, such as BIP71 (Payment Protocol MIME types) and BIP72 (URI extensions for Payment Protocol), have also been implemented to standardize payment schemes to support BIP70 (Payment Protocol).

Another innovative development is the Lightning Network. It is a solution for scalable off-chain instant payments. It was introduced in early 2016 and allows off-blockchain payments. This network makes use of payment channels that run off the blockchain, which allows for the greater speed and scalability of Bitcoin.



A paper is available at `https://lightning.network/` and those of you who are interested are encouraged to read the paper to understand the theory and rationale behind this invention.

Now that we've provided this brief discussion on Bitcoin's improvement and evolution, let's look at some of the excellent ideas that have emerged from research and innovation efforts related to Bitcoin.

Advanced protocols

In this section, we'll introduce various advanced protocols that have been suggested or implemented for improving the Bitcoin protocol. For example, transaction throughput is one of the critical issues that need a solution. The Bitcoin network can only process approximately three to seven transactions per second, which is a tiny number compared to other financial networks. For example, the Visa network can process approximately, on average, 24,000 transactions per second. PayPal can process approximately 200 transactions per second, whereas Ethereum can process up to, on average, 20. As the Bitcoin network has grown exponentially over the last few years, these issues have started to grow even further. The difference in processing speed is also shown in the following graph, which shows the scale of difference between Bitcoin and other networks' transaction speeds. The graph uses a logarithmic scale, which demonstrates the vast difference between the networks' transaction speeds:

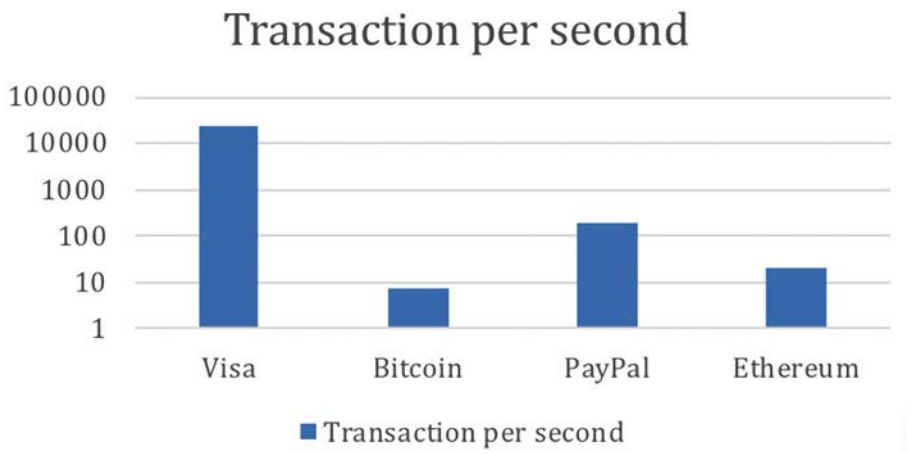


Figure 7.1: Bitcoin transaction speed compared to other networks (on a logarithmic scale)

Also, security issues such as transaction malleability are of real concern and can result in denial of service. Various proposals have been made to improve the Bitcoin proposal to address various weaknesses. A selection of these proposals will be presented here.

Segregated Witness

The **SegWit** or **Segregated Witness** is a soft fork-based upgrade of the Bitcoin protocol that addresses weaknesses such as throughput and security in the Bitcoin protocol. SegWit offers several improvements, as listed here:

- A fix for transaction malleability due to the separation of signature data from transactional data. In this case, it is no longer possible to modify the transaction ID because it is no longer calculated based on the signature data present within the transaction.
- By segregating the signature data and transaction data, lightweight clients do not need to download the transactions with all signatures unnecessarily. The transactions can be verified without useless signatures, which allows for increased bandwidth efficiency.

- Reduction in transaction signing and verification times, which results in faster transactions. A new transaction hashing algorithm for signature verification has been introduced and is detailed in BIP0143 (`https://en.bitcoin.it/wiki/BIP_0143`). Due to this change, the verification time grows linearly with the number of inputs instead of in a quadratic manner, resulting in a quicker verification time.
- A script versioning capability, which allows for easier script language upgrades. The version number is prefixed to the locking scripts to depict the version. This change allows upgrades and improvements to be made to the scripting language, without requiring a hard fork, by just increasing the version number of the script.
- Increased block size by introducing a weight limit instead of a size limit on the block and the removal of signature data. This concept will be explained in more detail shortly.
- An improved address format, also called a "bc1 address," which is encoded using the **Bech32** mechanism instead of base58. This improvement allows for better error detection and correction. Also, all characters are lowercase, which helps with readability. Moreover, this helps with distinguishing between legacy transactions and SegWit transactions. More information is available at this link: `https://en.bitcoin.it/wiki/Bech32`.

SegWit was proposed in BIP141, BIP143, BIP144, and BIP145. It was activated on Bitcoin's main network on August 24 2017 at block number 481824. The key idea behind SegWit is the separation of signature data from transaction data (that is, a transaction Merkle tree), which results in the size of the transaction being reduced. This change allows the block size to increase up to 4 MB in size. However, the practical limit is between 1.6 MB and 2 MB. Instead of a hard size limit of 1 MB blocks, SegWit introduced a new concept of a block weight limit.

The block weight is a new restriction mechanism where each transaction has a weight associated with it. This **weight** is calculated on a per-transaction basis. The formula used to calculate it is:

Weight = (*Transaction size without witness data*) x 3 + (*Transaction size*)

Blocks can have a maximum of four million weight units. As a comparison, a byte in a legacy 1 MB block is equivalent to 4 weight units, but a byte in a SegWit block weighs only 1 weight unit. This modification immediately results in increased block capacity.

To spend an **Unspent Transaction Output** (**UTXO**) in Bitcoin, a valid signature needs to be provided. In the pre-SegWit scenario, this signature is provided within the locking script, whereas in SegWit this signature is not part of the transaction and is provided separately.

There are four types of transactions introduced by SegWit. These types are:

1. **Pay to Witness Public Key Hash (P2WPKH)**: This type of script is similar to the usual P2PKH, but the crucial difference is that the transaction signature used as a proof of ownership in ScriptSig is moved to a separate structure known as the "witness" of the input. The signature is the same as P2PKH but is no longer part of ScriptSig; it is simply empty. The PubKey is also moved to the witness field.

This script is identified by a 20-byte hash. The ScriptPubKey is modified to a simpler format, as shown here:

- P2PKH ScriptPubKey:

```
OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

- P2WPKH ScriptPubKey:

```
OP_0 <pubKeyHash>
```

2. **Pay to Script Hash – Pay to Witness PubKey Hash (P2SH-P2WPKH)**: This is a mechanism introduced to make SegWit transactions backward-compatible. This is made possible by nesting the P2WPKH inside the usual P2SH.
3. **Pay to Witness Script Hash (P2WSH)**: This script is similar to legacy P2SH but the signature and redeem script are moved to the separate witness field. This means that ScriptSig is simply empty. This script is identified by a 32-byte SHA-256 hash. P2WSH is a simpler script compared to P2SH and has just two fields. The ScriptPubKey is modified as follows:

   P2SH ScriptPubKey:

   ```
   OP_HASH160 <pubKeyHash> OP_EQUAL
   ```

   P2WSH ScriptPubKey:

   ```
   OP_0 <pubKeyHash>
   ```

4. **Pay to Script Hash – Pay to Witness Script Hash (P2SH-P2WSH)**: Similar to P2SH-P2WPKH, this is a mechanism that allows backward-compatibility with legacy Bitcoin nodes.

SegWit adoption is still in progress as not all users of the network agree or have started to use SegWit.

Next, we'll introduce some other innovative ideas in the Bitcoin space. Not only has the original Bitcoin evolved quite significantly since its introduction, but there are also new blockchains that are either forks of Bitcoin or novel implementations of the Bitcoin protocol with advanced features.

Bitcoin Cash

Bitcoin Cash (**BCH**) increases the block limit to 8 MB. This change immediately increases the number of transactions that can be processed in one block to a much larger number compared to the 1 MB limit in the original Bitcoin protocol. It uses **Proof of Work** (**PoW**) as a consensus algorithm, and mining hardware is still ASIC-based. The block interval is changed from 10 minutes to 10 seconds and up to 2 hours. It also provides replay protection and wipe-out protection, which means that because BCH uses a different hashing algorithm, it prevents it from being replayed on the Bitcoin blockchain. It also has a different type of signature compared to Bitcoin to differentiate between two blockchains.



The BCH wallet and relevant information are available on their website: `https://www.bitcoincash.org`.

Bitcoin Unlimited

Bitcoin Unlimited increases the size of the block without setting a hard limit. Instead, miners come to a consensus on the block size cap over a period of time. Other concepts such as extremely thin blocks and parallel validation have also been proposed in Bitcoin Unlimited.



Its client is available for download at `https://www.bitcoinunlimited.info`.

Extremely thin blocks allow for faster block propagation between Bitcoin nodes. In this scheme, the node requesting blocks sends a `getdata` request, along with a bloom filter, to another node. The purpose of this bloom filter is to filter out the transactions that already exist in the **memory pool** (**mempool**) of the requesting node. The node then sends back a **thin block** only containing the missing transactions. This fixes an inefficiency in Bitcoin whereby transactions are regularly received twice—once at the time of broadcast by the sender and then again when a mined block is broadcast with the confirmed transaction.

Parallel validation allows nodes to validate more than one block, along with new incoming transactions, in parallel. This mechanism contrasts with Bitcoin, where a node, during its validation period after receiving a new block, cannot relay new transactions or validate any blocks until it has accepted or rejected the block.

Bitcoin Gold

This proposal has been implemented as a hard fork since block 491407 of the original Bitcoin blockchain. Being a hard fork, it resulted in a new blockchain, named Bitcoin Gold. The core idea behind this concept is to address the issue of mining centralization, which has hurt the original Bitcoin idea of decentralized digital cash, whereby more hash power has resulted in a power shift toward miners with more hashing power. Bitcoin Gold uses the Equihash algorithm as its mining algorithm instead of PoW; hence, it is inherently ASIC resistant and uses GPUs for mining.



Bitcoin Gold and relevant information are available at `https://bitcoingold.org`.

Taproot

Taproot is a significant upgrade to the Bitcoin protocol. It was activated at block 709,632. It improves transaction speed, privacy, and scalability. It also allows smart contracts on the Bitcoin network. Taproot includes several components, which we explain next:

- **Schnorr signatures**, which provide signature aggregation resulting in privacy and efficiency. Schnorr signatures are more secure than ECDSA.

- **Merkelized Alternative Script Tree** (**MAST**), which compresses complex bitcoin transactions into a single hash that reduces the transaction fee and memory usage. MAST allows enumerating distinct spending conditions separately. This allows funds to be spent by satisfying any one of the scripts.

  In MAST each script lives in a leaf on the Merkle tree. When funds are received, they are locked to the Merkle tree root. To spend the funds, a single leaf's script is revealed, satisfying the spending conditions mandated in the leaf. Here, a Merkle proof proves its inclusion in the tree. This way, other spending conditions that are not relevant are kept private. It also means that multiple different spending conditions can be encoded in many leaves.

- **Pay2Taproot** (**P2TR**) is a new type of script. This script combines the Schnorr signature and MAST in a single transaction. The Tapscript scripting language is used to enable various types of new transactions. It is like the Script language (the original Bitcoin script), but with some changes. The key change is the introduction of `OP_CHECKSIGADD` opcode, which enables the aggregation of signatures by utilizing Schnorr signatures.

  Tapscript also enables easier future soft fork upgrades by using the new `OP_SUCCESS` opcode. In practice, P2TR addresses combine signatures and scripts by placing the MAST of scripts in a public key. This means that the same funds can be spent either with a usual plain signature that corresponds to that public key (the old way) or through one of the scripts encoded in the MAST.

The Taproot upgrade is composed of three BIPs: BIP340 (BIP – Schnorr), BIP341 (BIP – Taproot), and BIP342 (BIP – Tapscript).

Extended protocols on top of Bitcoin

Several protocols, as discussed in the following sections, have been proposed and implemented on top of Bitcoin to enhance and extend the Bitcoin protocol, as well as to be used for various other purposes instead of just as a virtual currency.

Colored coins

Colored coins are a set of methods that have been developed to represent digital assets on the Bitcoin blockchain. Coloring a bitcoin refers colloquially to updating it with some metadata representing a digital asset (smart property). The coin still works and operates as a Bitcoin, but additionally carries some metadata that represents some assets. This can be some information related to the asset, some calculations related to transactions, or any arbitrary data. This mechanism allows issuing and tracking specific bitcoins. Metadata can be recorded using Bitcoin's `OP_RETURN` opcode or optionally in multi-signature addresses. The metadata can also be encrypted if required to address any privacy concerns. Some implementations also support the storage of metadata on publicly available torrent networks, which means that virtually unlimited amounts of metadata can be stored. Usually, these are JSON objects representing various attributes of the colored coin. Moreover, smart contracts are also supported.

Colored coins can be used to represent a multitude of assets, including, but not limited to, commodities, certificates, shares, bonds, and voting. It should also be noted that to work with colored coins, a wallet that interprets colored coins is necessary, and normal Bitcoin wallets will not work. Normal Bitcoin wallets will not work because they cannot differentiate between **colored coins** and **not colored coins**.

The idea of colored coins is very appealing as it does not require any modifications to be made to the existing Bitcoin protocol, and they can also make use of the already existing secure Bitcoin network. In addition to the traditional representation of digital assets, there is also the possibility of creating smart assets that behave according to the parameters and conditions defined for them. These parameters include time validation, restrictions on transferability, and fees. This opens the possibility of creating smart contracts, which we will discuss in *Chapter 8*, *Smart Contracts*.

A significant use case is the issuance of financial instruments on the blockchain. This will ensure low transaction fees, valid and mathematically secure proof of ownership, fast transferability without requiring some intermediary, and instant dividend payouts to investors.



There were a few services available online that used to provide colored coins services, but they are no longer active. However, check this link for some more information on this: *Colu* by *Coinprism* (`https://en.bitcoin.it/wiki/Coinprism`).

Counterparty

This is another service that can be used to create custom tokens that act as a cryptocurrency and can be used for various purposes, such as issuing digital assets on top of the Bitcoin blockchain. This is quite a robust platform and runs on Bitcoin blockchains at its core, but has developed its client and other components so that they support issuing digital assets. The architecture consists of the following components:

1. **Counterparty server**: This is the reference client and implements the core counterparty protocol. It is a combination of `counterparty-lib` and `counterparty-cli`.
2. **Counter block**: This component provides services in addition to the Counterparty server.
3. **Counter wallet**: This is a web wallet for Bitcoin and **Counterparty Coins** (**XCPs**).
4. **armory_utxsvr**: This is a service used for offline armory transactions.

Counterparty works based on the same idea as colored coins by embedding data into regular Bitcoin transactions but provides a much more productive library and a set of powerful tools to support the handling of digital assets. This embedding is also called **embedded consensus** because the counterparty transactions are embedded within Bitcoin transactions. The method of embedding the data is by using the `OP_RETURN` opcode in Bitcoin.

The currency produced and used by Counterparty is known as XCP and is used by smart contracts as the fee for running the contract. At the time of writing, its price is 1.03 USD. XCPs were created by using the PoB method discussed previously.

Counterparty allows the development of smart contracts on Ethereum using the Solidity language and allows interaction with the Bitcoin blockchain. To achieve this, BTC Relay is used to provide interoperability between Ethereum and Bitcoin. This is a clever concept where Ethereum contracts can talk to the Bitcoin blockchain and transactions through BTC Relay. The relayers (the nodes that are running BTC Relay) fetch the Bitcoin block headers and relay them to a smart contract on the Ethereum network that verifies the PoW. This process verifies that a transaction has occurred on the Bitcoin network.



BTC Relay is available at `http://btcrelay.org/`.

Technically, this is an Ethereum contract that can store and verify Bitcoin block headers, just like Bitcoin simple payment verification lightweight clients do, by using bloom filters. SPV clients were discussed in detail in the previous chapter. This idea can be visualized with the following diagram:

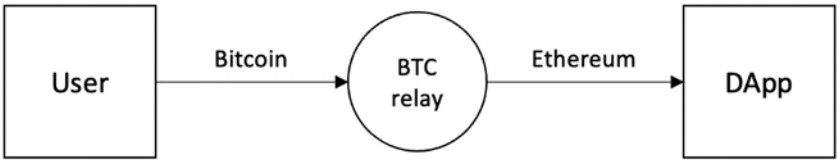


Figure 7.2: BTC relay concept



Counterparty is available at `http://counterparty.io/`.

Now, we will move on to a different topic, which explains how altcoins are developed, how they work, and how difficult it is to create a new coin.

Altcoins from Bitcoin

By definition, an altcoin is generated in the case of a hard fork. Altcoins must be able to attract new users, trades, and miners; otherwise, the currency will have no value. Currency gains its value, especially in the virtual currency space, due to the network effect and its acceptability by the community. If a coin fails to attract enough users, then soon, it will be forgotten. Users can be attracted by providing an initial amount of coins, which can be achieved by using various methods. There is, however, a risk that if the new coin does not perform well, then its initial investment may be lost.

Methods of providing an initial number of altcoins are as follows:

1. **Create a new blockchain**: Altcoins can create a new blockchain and allocate coins to initial miners, but this approach is now unpopular due to many scam schemes, or *pump-and-dump* schemes, where initial miners made a profit with the launch of a new currency and then disappeared.
2. **Proof of Burn** (**PoB**): Another approach to allocating initial funds to a new altcoin is PoB, also called a one-way peg or price ceiling. In this method, users permanently destroy a certain quantity of bitcoins in proportion to the number of altcoins to be claimed. For example, if 10 bitcoin are destroyed, then altcoins can have a value no greater than the bitcoins that were destroyed. This means that bitcoins are converted into altcoin by burning them.
3. **Proof of ownership**: Instead of permanently destroying bitcoins, an alternative method is to prove that users own a certain number of bitcoins. This proof of ownership can be used to claim altcoins by tethering altcoin blocks to Bitcoin blocks. For example, this can be achieved through merged mining in which, effectively, Bitcoin miners can mine altcoin blocks while mining for bitcoins without any extra work. Merged mining will be explained later in this chapter.
4. **Pegged sidechains**: Sidechains, as the name suggests, are blockchains separate from the Bitcoin network, but Bitcoin can be transferred to them. Altcoins can also be transferred back to the Bitcoin network. This concept is called a **two-way peg**.

We have now covered some new blockchains and implementations of the Bitcoin protocol. The next section introduces Bitcoin client installation and a basic overview of various APIs and tools that are available for developing Bitcoin applications and interacting with the Bitcoin blockchain.

Bitcoin client installation

The Bitcoin Core client can be installed from `https://bitcoin.org/en/download`. This is available for different architectures and platforms, ranging from x86 Windows to ARM Linux.

We will discuss a few topics relating to Bitcoin's installation and setup. We'll begin by discussing the different Bitcoin clients available and their associated tools, which enable you to run and manage the Bitcoin client and interact with the Bitcoin blockchain.

Types of clients and tools

There are different types of Bitcoin Core clients and relevant tools. A Bitcoin client is a piece of software that is responsible for generating private/public key pairs and facilitates Bitcoin payments using the Bitcoin blockchain. In addition, a client can implement a full synchronization function with a blockchain or choose to only implement basic wallet functionality or simple payment verification. A client can also provide other useful functions such as network monitoring, secure storage of keys, and user-friendly interfaces for interaction with the Bitcoin blockchain. Some of the core elements of the Bitcoin Core client and associated tools are as follows:

- `bitcoind`: This is the core client software that runs as a daemon (as a service), and it provides the JSON-RPC interface.

- `bitcoin-cli`: This is the command-line feature-rich tool for interacting with the Bitcoin daemon; the daemon then interacts with the blockchain and performs various functions. `bitcoin-cli` only calls JSON-RPC functions and does not perform any actions on its own on the blockchain.
- `bitcoin-qt`: This is the Bitcoin Core client GUI. When the wallet software starts up, first, it verifies the blocks on the disk and then starts the block synchronization process. The verification process is not specific to the `bitcoin-qt` client; it is performed by the `bitcoind` client as well.

There are also other clients available, such as **btcd**, which is a full-node Bitcoin client written in Golang. It is available at `https://github.com/btcsuite/btcd`.

Setting up a Bitcoin node

In this section, we will explore how we can set up a Bitcoin node in order to interact with the Bitcoin network and interact with a Bitcoin node using the command-line interface.

The Bitcoin Core software is available at `https://bitcoin.org/en/download`. You can download the software and install it according to the instructions provided, which are quite straightforward.

Alternatively, you can download the Bitcoin source code and compile it manually to produce binaries, which we'll cover next.

Setting up the source code

The Bitcoin source code can be downloaded and compiled if users wish to use the Bitcoin code, for learning purposes, or just want to produce binaries manually. The `git` command can be used to download the Bitcoin source code:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
```

Change the directory to `bitcoin`:

```
$ cd bitcoin
```

After the preceding steps are completed, the code can be compiled:

```
$ ./autogen.sh
$ ./configure.sh
$ make
$ sudo make install
```



Note that the `make` command shown here may take around 30 minutes to complete, depending on the speed of your computer.

Setting up bitcoin.conf

The `bitcoin.conf` file is a configuration file that is used by the Bitcoin Core client to save configuration settings. All command-line options for the `bitcoind` client, except for the `-conf` switch, can be set up in the configuration file, and when `bitcoin-qt` or `bitcoind` starts up, it will take the configuration information from that file.

In Linux systems, this is usually found in `$HOME/.bitcoin/`, but it can also be specified in the command line using the `-conf=<file>` switch in the `bitcoind` core client software. The configuration file can be generated using the tool available here: `https://github.com/bitcoin/bitcoin/tree/master/contrib/devtools#gen-bitcoin-confsh`.

Now that we have set up the Bitcoin client, let's see how to start up the Bitcoin client for use.

Starting up a node in the testnet

The Bitcoin node can be started in a test blockchain network (testnet) if you want to test the Bitcoin network and run some experiments. This is a completely alternative test network used for experimentation. It is a faster network compared to the main network and has relaxed rules for mining and transactions.

The key differences between the mainnet and the testnet are shown here:

Component	Mainnet	Testnet
Listening port	TCP 8333	TCP 18333
RPC connection port	TCP 8332	TCP 18332
DNS seeds are different for bootstrapping	Mainnet-specific	Testnet-specific
A different `ADDRESSVERSION` field in addresses to ensure testnet addresses do not work on Bitcoin's mainnet	0x00	0x6F
Genesis block	Mainnet-specific	Testnet-specific
`IsStandard()` check to ensure a transaction is standard	Enabled	Disabled

Various faucet services are also available for the Bitcoin test network. These services are used to get some test Bitcoin for testnet accounts. A list of faucets is available here on the Bitcoin wiki: `https://en.bitcoin.it/wiki/Testnet#Faucets`. The availability of test coins is very useful for experimentation on the testnet.

The command line to start up the Bitcoin testnet is as follows. To run the Bitcoin daemon for the testnet:

```
$ bitcoind --testnet -daemon
```

To run the Bitcoin command-line interface:

```
$ bitcoin-cli --testnet <command>
```

To allow the Bitcoin GUI to run in the testnet:

```
$ bitcoin-qt –testnet
```

A sample run is shown here:

1. Start up the Bitcoin node in daemon (as a background process) mode on the testnet:

```
$ bitcoind --testnet -daemon
Bitcoin server starting
```

2. Check the number of blocks and the difficulty. Note that there is a long list of various commands that the Bitcoin client supports. This is just an example to show how the Bitcoin command-line interface works:

```
$ bitcoin-cli --testnet getmininginfo
{
  "blocks": 566251,
  "difficulty": 400.6820950060902,
  "networkhashps": 572058533067.9225,
  "pooledtx": 0,
  "chain": "test",
  "warnings": ""
}
```

3. A complete list of commands can be obtained by running the following command:

```
$ bitcoin-cli --testnet help
```



The preceding command output shows various command-line options available in `bitcoin-cli`, the Bitcoin command-line interface. These commands can be used to query the blockchain, send transactions, and control the local node.

4. We now can stop the Bitcoin daemon by using the command shown here:

```
$ bitcoin-cli --testnet stop
Bitcoin server stopping
```

With this, we have completed a basic introduction to the Bitcoin testnet. We will do some more experimentation with this shortly, but first, we will look at another mode in which the Bitcoin node can run and that is especially useful for testing purposes.

Starting up a node in regtest

Regtest mode (regression testing mode) can be used to create a local private blockchain for testing purposes. In this mode, the user can control block generation for experimentation and testing, and a number of blocks with no value can be generated. It effectively creates a locally isolated new bitcoin blockchain for testing purposes.

The following commands can be used to start up a node in regtest mode:

1. Start up the Bitcoin daemon in regtest mode:

```
$ bitcoind -regtest -daemon
Bitcoin server starting
```

2. Check the balance:

```
$ bitcoin-cli -regtest getbalance
0.00000000
```

3. Generate blocks and addresses:

```
$ bitcoin-cli -regtest generatetoaddress 200 $(bitcoin-cli -regtest
getnewaddress)
[
  "366fce3c35031eaa3b085ae7d2631cb5b212bac7e3447bd8ffddb17ef97569c4
.
.
.
"33361a74d2586259d69a724921ff7b931cc6c95bd52f09fc05a4b8905695384f"
]
```



The reason why we generated 200 blocks in the preceding command is that on a regtest, a block must have 100 confirmations before the associated reward can be utilized. Therefore, we must generate more than 100 blocks to get access to this reward. In this command, we have generated 200 blocks, which will generate 5,000 bitcoins due to the hardcoded miner reward of 50 bitcoins.

4. Now, we can get the balance by running the following command:

```
$ bitcoin-cli -regtest getbalance
5000.00000000
```

5. Run a command, for example, `getmininginfo`:

```
$ bitcoin-cli -regtest getmininginfo
{
  "blocks": 200,
  "currentblockweight": 4000,
  "currentblocktx": 0,
  "difficulty": 4.656542373906925e-10,
  "networkhashps": 12,
  "pooledtx": 0,
  "chain": "regtest",
  "warnings": ""
}
```

6. We can also get information about the blockchain by using the following command:

```
$ bitcoin-cli -regtest getblockchaininfo
{
  "chain": "regtest",
  "blocks": 200,
  "headers": 200,
  "bestblockhash":
"1cafd1e540b6772f4fe4ab561def0de69945f84e701e5fffa8426ea572d3769b",
  "difficulty": 4.656542373906925e-10,
  "mediantime": 1577225980,
  .
  .
  .
```



Note that the complete output is not shown here due to its long length, but it is enough to explain the concept.

7. Stop the Bitcoin daemon:

```
$ bitcoin-cli -regtest stop
Bitcoin server stopping
```

If you want to delete the previous regtest node and start a new one, simply delete the directory named `regtest` under your computer's `$HOME` directory. On a Mac (macOS), it is located at `/$HOME/Library/Application Support/Bitcoin`.

After deleting the `regtest` directory, run the command shown in the first step again in this section to create a new `regtest` environment.

In this section, we covered how to start up a Bitcoin node in test and development (regtest) modes and how to interact with the Bitcoin blockchain using `bitcoin-cli`, the command-line tool for the Bitcoin client. Next, we will experiment further with some Bitcoin commands and interfaces.

Experimenting further with bitcoin-cli

As we've seen so far, `bitcoin-cli` is a powerful and feature-rich command-line interface available with the Bitcoin Core client and can be used to perform various functions using the RPC interface provided by the Bitcoin Core client.

We will now see how to send bitcoins to an address using the command line. For this, we will use the Bitcoin command-line interface on the Bitcoin regtest:

1. Generate a new address using the following command:

```
$ bitcoin-cli -regtest getnewaddress
2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd
```

2. Send 20 BTC to the newly generated address:

```
$ bitcoin-cli -regtest sendtoaddress \
2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd 20.00
```

The output of this command will show the transaction ID, which is:

```
a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5f2980
```

3. We can now generate a few more blocks to get some confirmation for this:

```
$ bitcoin-cli -regtest generatetoaddress 7 $(bitcoin-cli -regtest
getnewaddress)
```

4. We can also query the transaction information by using the following command:

```
$ bitcoin-cli -regtest gettransaction \
a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5f2980
```

This will show an output similar to the one shown here. Note that we use the same transaction ID hash output that was generated in *step 2* previously:


```
{
  "amount": 0.00000000,
  "fee": -0.00003320,
  "confirmations": 7,
  "blockhash": "7c50e79b54dcda17e32cc7b7b53fc095584befe4e952422bdf096de3b93fe539",
  "blockindex": 1,
  "blocktime": 1577228072,
  "txid": "a83ff460a32f29387d531f19e7092a5dcf6ce52d20931227447c0b9b7a5f2980",
  "walletconflicts": [
  ],
  "time": 1577227733,
  "timereceived": 1577227733,
  "bip125-replaceable": "no",
  "details": [
    {
      "address": "2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd",
      "category": "send",
      "amount": -20.00000000,
      "label": "",
      "vout": 0,
      "fee": -0.00003320,
      "abandoned": false
    },
    {
      "address": "2NC31WFFRwRkwd3S4TpyjN5GGDY7E63GSVd",
      "category": "receive",
      "amount": 20.00000000,
      "label": "",
      "vout": 0
    }
  ],
  "hex": "020000000001018712030 19a3456fc50b2044e79a4f078bc0ceb278734d44faf82ce4f24
35770000000017a914ce1afa4c71513d952194695adedfad119faf8f87870851d0b20000000017a914
c730f514cc654f222cecce09faf6a8e87d07c9a44a0220273478b3f452bec625d3d87002cb00831476
6ef37cf310609bce20e575c8000000"
}
```


Figure 7.3: gettransaction output

So far, we've used `bitcoin-cli` on regtest. However, we can use `bitcoin-cli` on any Bitcoin network, for example, the testnet or the mainnet. We simply use the `bitcoin-cli` command without specifying any network to query the mainnet blockchain. Next, we will show a quick example of querying the Bitcoin mainnet blockchain. The Bitcoin client provides three methods for interacting with the blockchain, as listed here:

- Bitcoin **Command-Line Interface** (**CLI**)
- JSON-RPC interface
- HTTP REST interface

First, we'll see an example of `bitcoin-cli` querying the blockchain using the `getblock` method. We will then see how the same `getblock` method can be invoked using the JSON-RPC interface and the HTTP REST interface.

We will query the 100th block of the Bitcoin blockchain with hash `000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a`.

Using the Bitcoin command-line tool

We can use `bitcoin-cli` for the purpose of using the Bitcoin command-line tool, as shown here:

```
$ bitcoin-cli getblock \
"000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a"
```

The output of the preceding command, which shows the details of block hash `000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a`, is shown below:

```
{
  "hash": "000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a",
  "confirmations": 232839,
  "strippedsize": 215,
  "size": 215,
  "weight": 860,
  "height": 100,
  "version": 1,
  "versionHex": "00000001",
  "merkleroot":
"2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc9020866",
  "tx": [
    "2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc9020866"
  ],
  "time": 1231660825,
  "mediantime": 1231656204,
  "nonce": 1573057331,
  "bits": "1d00ffff",
  "difficulty": 1,
```

```
  "chainwork": "0000000000000000000000000000000000000000000000000650065006
5",
  "nTx": 1,
  "previousblockhash":
"00000000cd9b12643e6854cb25939b39cd7a1ad0af31a9bd8b2efe67854b1995",
  "nextblockhash":
"00000000b69bd8e4dc60580117617a466d5c76ada85fb7b87e9baea01f9d9984"
}
```

The output shows several elements, including previous block hash, next block hash, transaction hashes included in the block, number of transactions, difficulty level, time, and some other data.

As an alternative to `bitcoin-cli`, we can query the blockchain using the JSON-RPC interface provided by the Bitcoin client as well. We'll show an example of that next.

Using the JSON-RPC interface

Now, we will run the same command but using the JSON-RPC. Note that we are using the Bitcoin mainnet for this example.

At a minimum, in order to use the JSON-RPC interface, we need to configure the RPC username and password in the `bitcoin.conf` file. We can easily do that. A sample configuration that will be used in this example is shown here:

```
$ cat bitcoin.conf
rpcuser=test1
rpcpassword=testpassword
```

We can use the `curl` command-line tool to interact with the JSON-RPC API, as shown here:

```
$ curl --user test1 --data-binary '{"jsonrpc": "1.0",
"id":"curltest", "method": "getblock", "params":
["000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a"] }' -H
'content-type: text/plain;' http://127.0.0.1:8332/
```

This will ask for the required password. Enter the password that has been set in the `bitcoin.conf` file:

```
Enter host password for user 'test1':
```

If the password is correct, after executing the command, the result shown here will be displayed in JSON format:

```
{"result":{"hash":"000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9
ebf9a715ed27fdb229a","confirmations":236138,"strippedsize":215,
"size":215,"weight":860,"height":100,"version":1,"versionHex":"00000001",
"merkleroot":"2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc9020866",
"tx":["2d05f0c9c3e1c226e63b5fac240137687544cf631cd616fd34fd188fc9020866"],
"time":1231660825,"mediantime":1231656204,"nonce":1573057331,"bits":
"1d00ffff","difficulty":1,"chainwork":"00000000000000000000000000000000
```

```
0000000000000000000000000006500650065","nTx":1,"previousblockhash":
"00000000cd9b12643e6854cb25939b39cd7a1ad0af31a9bd8b2efe67854b1995",
"nextblockhash":"00000000b69bd8e4dc60580117617a466d5c76ada85fb7b87e9bae
a01f9d9984"},"error":null,"id":"curltest"}
```



`curl` is an excellent command-line tool that is used to transfer data using URLs. It is commonly used to interact with REST APIs using HTTP. More information about `curl` is available at `https://curl.haxx.se`.

An example run will be shown next, which queries the same block that we queried in the previous command, but now using the HTTP REST interface.

Using the HTTP REST interface

Starting from Bitcoin Core client 0.10.0, the HTTP REST interface is also available. By default, this runs on the same TCP port (`8332`) as the JSON-RPC interface and requires no authentication. It is enabled either in the `bitcoin.conf` file by adding the `rest=1` option or on the `bitcoind` command line via the `-rest` flag.

We can use `curl` again for this purpose, as shown here:

```
$ curl http://localhost:8332/rest/
block/000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a.json
```

The output of the preceding command is shown here:

```
{"hash":"000000007bc154e0fa7ea32218a72fe2c1bb9f86cf8c9ebf9a715ed27fdb229a",
"confirmations":247383,"strippedsize":215,"size":215,"weight":860,"height":100,
"version":1,"versionHex":"00000001","merkleroot":"2d05f0c9c3e1c226e63b5fac240
137687544cf631cd616fd34fd188fc9020866","tx":[{"txid":"2d05f0c9c3e1c226e63b5
fac240137687544cf631cd616fd34fd188fc9020866","hash":"2d05f0c9c3e1c226e63b5
fac240137687544cf631cd616fd34fd188fc9020866","version":1,"size":134,"vsize":134,
"weight":536,"locktime":0,"vin":[{"coinbase":"04ffff001d014d",
"sequence":4294967295}],"vout":[{"value":50.00000000,"n":0,"scriptPubKey":
{"asm":"04e
70a02f5af48a1989bf630d92523c9d14c45c75f7d1b998e962bff6ff9995fc5bdb44f1793b37
495d80324acba7c8f537caaf8432b8d47987313060cc82d8a93 OP_CHECKSIG","hex":"4104e70
a02f5af48a1989bf630d92523c9d14c45c75f7d1b998e962bff6ff9995fc5bdb44f1793b37495d80
324acba7c8f537caaf8432b8d47987313060cc82d8a93ac","reqSigs":1,"type":"pubkey",
"addresses":["13A1W4jLPP75pzvn2qJ5KyyqG3qPSpb9jM"]}}],"hex":"01000000010000000
0000000000000000000000000000000000000000000000000000000
000000000000ffffffff0704ffff001d014dffffffff
0100f2052a01000000434104e70a02f5af48a1989bf630d92523c9d14c45c75f7d1b998e962bff6ff
9995fc5bdb44f1793b37495d80324acba7c8f537caaf8432b8d47987313060cc82d8a93ac00000
000"}],"time":1231660825,"mediantime":1231656204,"nonce":1573057331,"bits":
"1d00ffff","difficulty":1,"chainwork":"0000000000000000000000000000000000000000000000
```

```
000000000000006500650065","nTx":1,"previousblockhash":"00000000cd9b12643
e6854cb25939b39cd7a1ad0af31a9bd8b2efe67854b1995","nextblockhash":"000000
00b69bd8e4dc60580117617a466d5c76ada85fb7b87e9baea01f9d9984"}
```

With this, we have completed our basic introduction to the Bitcoin command-line tools and related interfaces. A complete introduction to all Bitcoin client RPC calls is not possible here. You are encouraged to check the comprehensive documentation of Bitcoin Core 0.21.0 RPC, which is available at `https://bitcoincore.org/en/doc/0.21.0/rpc/`.

Bitcoin programming

Bitcoin programming is a very rich field. The Bitcoin Core client exposes various JSON-RPC commands that can be used to construct raw transactions and perform other functions via custom scripts or programs. Also, the command-line tool `bitcoin-cli` is available, which makes use of the JSON-RPC interface and provides a rich toolset to work with Bitcoin.

These APIs are also available via many online service providers in the form of Bitcoin APIs, and they provide a simple HTTP REST interface. Bitcoin APIs, such as blockchain.info (`https://blockchain.info/api`), BitPay (`https://bitpay.com/api`), block.io (`https://www.block.io`), and many others, offer a myriad of options to develop Bitcoin-based solutions.

Various libraries are available for Bitcoin programming. A list is shown as follows. Those of you who are interested can explore the libraries further:

- **Libbitcoin**: Available at `https://libbitcoin.dyne.org/` and provides powerful command-line utilities and clients.
- **Pycoin**: Available at `https://github.com/richardkiss/pycoin`, this is a library for Python.
- **Bitcoinj**: This library is available at `https://bitcoinj.github.io/` and is implemented in Java.

There are many online Bitcoin APIs available; some commonly used APIs are listed as follows:

- `https://bitcore.io/`
- `https://bitcoinjs.org/`
- `https://blockchain.info/api`

As all APIs offer a similar type of functionality, it can get confusing to decide which one to use. It is also difficult to recommend which API is the best because all APIs are similarly feature-rich. One thing to keep in mind, however, is security. Therefore, whenever you evaluate an API for usage, in addition to assessing the offered features, also evaluate how secure the design of the API is.

Summary

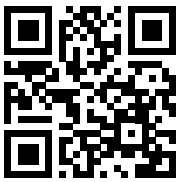
In this chapter, we looked at Bitcoin payments and payment processors, along with some important Bitcoin innovations, which included topics such as BIPs and advanced Bitcoin protocols.

The chapter continued with an introduction to Bitcoin installation, followed by a discussion on source code setup and how to set up Bitcoin clients for various networks. After this, we examined various command-line options available in Bitcoin clients. Lastly, we saw which APIs are available for Bitcoin programming and the main points to keep in mind while evaluating APIs for usage.

In the next chapter, we'll introduce smart contracts, which are an integral element of programmable blockchains and allow programmers to write programs that are stored on blockchain and implement business logic on the chain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

8

Smart Contracts

This chapter introduces smart contracts. This concept is not new; however, with the advent of blockchain technology, interest in this idea has been revived. Smart contracts are now an ongoing and intense area of research in the blockchain space. Many blockchains have emerged that support smart contracts.

Due to benefits such as the increased security, reliability, decentralization, cost-saving, and transparency that smart contracts can bring to many industries (especially the finance industry), smart contracts are seen as a technology that has the potential to change billions of lives.

In this chapter, we will cover the following topics:

- Introducing smart contracts
- Ricardian contracts
- Smart contract templates
- Oracles
- Deploying smart contracts
- The DAO
- Advances in smart contract technology

Introducing smart contracts

Human society is based on social contracts. Historically, we are used to agreements and contracts made between individuals or businesses. Historically, we had contracts written on stone, wood, and paper. Then in the computer age, digital contracts appeared. However, digital contracts are centralized and not reliable. A powerful entity could force its way out of the contract prematurely, influence the contract and its parties in one way or another, or not keep up with their side of the bargain.

Furthermore, they are only legally enforceable, and malicious parties can sometimes mislead the legal system. However, with the advent of blockchain, decentralized, more secure, and automatically enforceable contracts have come into existence where no single party can deviate or exert power to influence the performance of the contract. As you can imagine, such technology can alter, for good, the lives of billions of people.

Definitions

Nick Szabo first theorized smart contracts in the 1990s, in an article called *Formalizing and Securing Relationships on Public Networks*. This theory was presented almost 20 years before the real potential and benefits of smart contracts were appreciated, that is, before the invention of Bitcoin and the subsequent development of other more advanced blockchain platforms, such as Ethereum.

Smart contracts are described by Szabo as follows:

> *A smart contract is an electronic transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.*



The original article that was written by Szabo is available at `http://firstmonday.org/ojs/index.php/fm/article/view/548`.

Smart contract functionality was implemented in a limited fashion in Bitcoin in 2009. Bitcoin supports a restricted scripting language called **Script**, which allows the transfer of bitcoins between users. However, this is not a Turing-complete language and does not support arbitrary program development. It can be regarded as a limited-function calculator with only simple arithmetic operations, whereas smart contracts can be considered general-purpose computers that support writing any program.

The following is my attempt to provide a comprehensive generalized definition of a smart contract:

> *A smart contract is a secure and unstoppable computer program representing an agreement that is automatically executable and enforceable.*

Dissecting this definition reveals that a smart contract is, fundamentally, a computer program that is written in a language that a computer or target machine can understand. It also necessitates a few requirements for a smart contract to work effectively.

Properties

A smart contract has the following properties:

- **Automatically executable:** It is self-executable on a blockchain when certain conditions satisfy coded instructions without requiring any intervention.

- **Enforceable**: This means that all contractual terms perform as specified and expected, even in the presence of adversaries. Preferably, smart contracts should not rely on any traditional methods of enforcement. Instead, they should work on the principle that *code is the law*, which means that there is no need for an arbitrator or a third party to enforce, control, or influence the execution of a smart contract.



Smart contracts are self-enforcing as opposed to legally enforceable. This idea may sound like a libertarian's dream, but it is entirely possible and is in line with the true spirit of smart contracts.

- **Secure**: This means that smart contracts are tamper-proof (or tamper-resistant) and run with security guarantees. The underlying blockchain usually provides these security guarantees; however, the smart contract programming language and the smart contract code themselves must be correct, valid, and verified.



Blockchain platforms play a vital role in providing the necessary underlying network, with security guarantees required to run smart contracts.

- **Deterministic**: The deterministic feature ensures that smart contracts always produce the same output for a specific input. This property will allow a smart contract to be run by any node on a network and achieve the same result. If the result differs even slightly between nodes, then a consensus cannot be reached, and a whole paradigm of distributed consensuses on the blockchain can fail. Moreover, it is also desirable that the contract language itself is deterministic, thus ensuring the integrity and stability of smart contracts.



An example is some math functions in JavaScript, which can produce different results for the same input on different browsers and can, in turn, lead to various bugs. This scenario is unacceptable in smart contracts because if the results are inconsistent between the nodes, then a consensus will never be achieved.

- **Semantically sound**: This means that they are complete and meaningful to both people and computers.
- **Unstoppable**: This means that adversaries or unfavorable conditions cannot negatively affect the execution of a smart contract. When the smart contracts execute, they complete their performance deterministically in a finite amount of time.

It could be argued that the first four properties are required as a minimum, whereas the latter two may not be necessary or applicable in some scenarios and can be relaxed. For example, a financial derivatives contract does not, perhaps, need to be semantically sound and unstoppable but should at least be automatically executable, enforceable, deterministic, and secure.

On the other hand, a title deed needs to be semantically sound and complete; therefore, for it to be implemented as a smart contract, the language that it is written in must be understood by both computers and people.

Still, it will provide more significant benefits in the long run if security and unstoppable properties are included in the smart contract definition. This inclusion will allow researchers to focus on these aspects from the start and will help to build strong foundations on which further research can then be based.

There is also a suggestion by some researchers that smart contracts do not need to be **automatically executable**; instead, they can be what's called **automatable**, due to the manual human input required in some scenarios. For example, the manual verification of a medical record by a qualified medical professional might be needed. In such cases, fully automated approaches may not work best. While it is true that, in some instances, human input and control are helpful, they are not necessary. For a contract to be truly smart, in my opinion, it must be fully automated. Certain inputs that need to be provided by people can and should also be automated. Oracles can be used for this purpose. We will discuss oracles in more detail later in this chapter.

Real-world application

Smart contracts usually operate by managing their internal state using a state machine model provided by the underlying blockchain. This allows the development of a practical framework for programming smart contracts, where the state of a smart contract is advanced further based on some predefined criteria and conditions.

There is also an ongoing debate on whether computer code is acceptable as a conventional contract in a court of law. A smart contract is different in presentation from traditional legal prose, albeit it does represent and enforce all required contractual clauses. Still, a court of law does not understand computer code. This dilemma raises several questions about how a smart contract can be legally binding: can it be developed in such a way that it is readily acceptable and understandable in a court of law? Is it possible for dispute resolution to be implemented within code? Moreover, regulatory and compliance requirements are other topics that require addressing before smart contracts can become as efficient as traditional legal documents.

The legality of smart contracts is uncertain in many jurisdictions. Still, a recent exciting development in this space made crypto assets and smart contracts valid in English law by recognizing crypto assets as tradeable property and smart contracts as enforceable agreements. This announcement was made by the **UK Jurisdiction Taskforce** (**UKJT**) of the Lawtech Delivery Panel. Switzerland also has a favorable environment for cryptographic digital assets. Other "crypt-friendly" countries include Singapore, El Salvador, Puerto Rico, and Malta.



More information about this, including the full legal statement, is available here: `https://technation.io/news/uk-takes-significant-step-in-legal-certainty-for-smart-contracts-and-cryptocurrencies/`.

Even though smart contracts are called smart, they only do what they have been programmed to do. This very property of smart contracts ensures that they produce the same output every time they are executed. The deterministic nature of smart contracts is highly desirable in blockchain platforms due to consistency requirements.

Now, this gives rise to a problem whereby a large gap between the real world and the blockchain world emerges. In this situation, natural language is not understood by the smart contract, and, similarly, computer code is not acceptable within the natural world. So, a few questions arise: how can real-life contracts be deployed on a blockchain? How can this bridge between the real world and the smart contract world be built?

These questions open various possibilities, such as making smart contract code readable not only by machines but also by people. If humans and machines can both understand code written in a smart contract, it might become acceptable in legal situations, as opposed to just a piece of code that no one understands except for programmers. This desirable property is an area that is ripe for research, and a significant research effort has been expended in this area to answer questions about the semantics, meaning, and interpretation of smart contracts.

Some work has already been done to describe natural language contracts formally, by combining both smart contract code and natural language contracts by linking contract terms with machine-understandable elements. This is achieved using a markup language called the **Legal Knowledge Interchange Format** (**LKIF**), which is an XML schema for representing theories and proofs. It was developed under the Estrella project in 2008.



More information is available in the research paper at `https://doi.org/10.1007/978-3-642-15402-7_30`.

Further details can be found here: `http://www.estrellaproject.org/?page_id=5`.

Ian Grigg addressed this issue of interpretation with his invention of Ricardian contracts, which we will introduce in the next section.

Ricardian contracts

Ricardian contracts were initially used in a bond trading and payment system called **Ricardo**. The fundamental idea behind this contract is to write a document that is understood and accepted by both a court of law and computer software. Ricardian contracts address the challenge of the issuance of value over the internet by identifying the issuer and capturing all the terms and clauses of the contract in a document, making it acceptable as a legally binding contract.

A Ricardian contract has several of the following properties:

- It is a contract offered by an issuer to holders
- It is a valuable right held by holders and managed by the issuer
- It can be easily read by people (like a contract on paper)

- It can be read by programs (parsable, like a database)
- It is digitally signed
- It carries the keys and server information
- It is allied with a unique and secure identifier



The preceding information is based on the original definition by Ian Grigg at `http://iang.org/papers/ricardian_contract.html`.

In practice, the contracts are implemented by producing a single document that contains the terms of the contract in legal prose and the required machine-readable tags. This document is digitally signed by the issuer using their private key. This document is then hashed using a message digest function to produce a hash by which the document can be identified. This hash is then further used and signed by parties during the performance of the contract to link each transaction with the identifier hash, therefore serving as evidence of intent. This is depicted in the next diagram and is usually called a bowtie model:

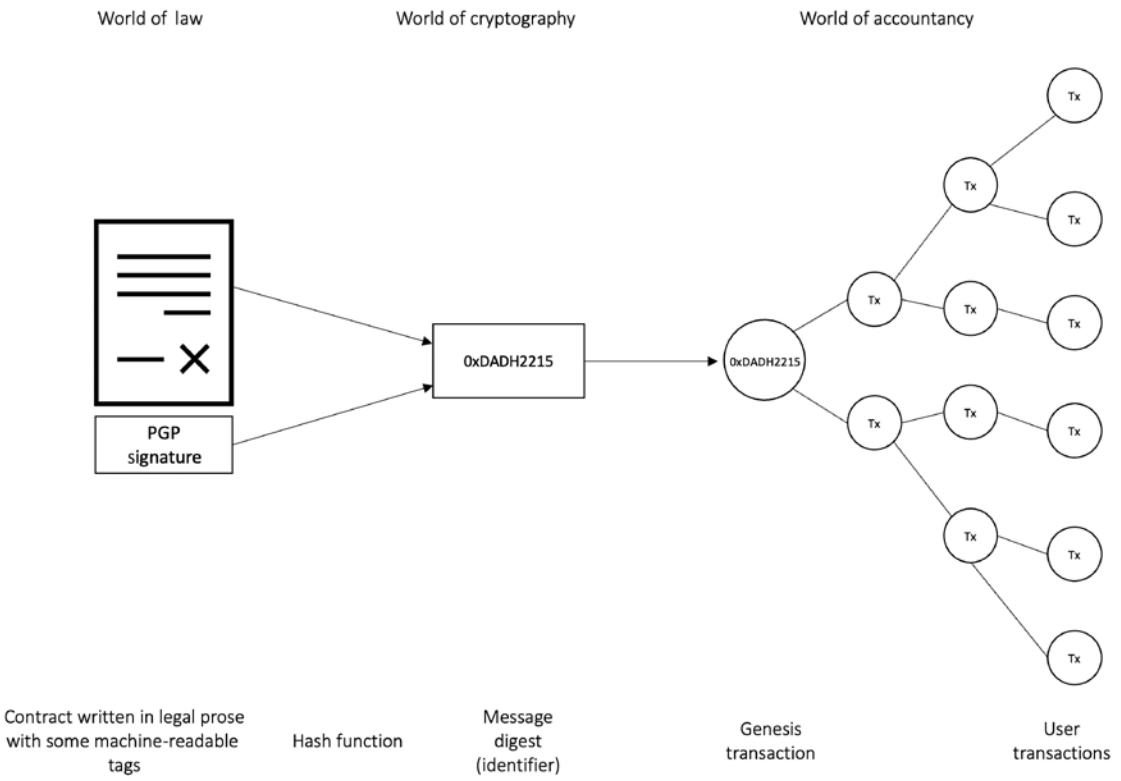


Figure 8.1: The Ricardian contract bowtie model

The diagram shows several elements:

1. The **World of law** is on the left-hand side from where the document originates. This document is a written contract in legal prose with some machine-readable tags.
2. This document is then hashed.
3. The resultant message digest is used as an identifier throughout the **World of accountancy**, as shown on the right-hand side of the diagram.

The **World of accountancy** element represents any accounting, trading, and information systems that are being used in the business to perform various business operations. The idea behind this flow is that the message digest generated by hashing the document is first used in a so-called **genesis transaction**, or first transaction, and then it is used in every transaction as an identifier throughout the operational execution of the contract. This way, a secure link is created between the original written contract and every transaction in the **World of accountancy**.

A Ricardian contract is different from a smart contract in the sense that a smart contract does not include any contractual document and is focused purely on the execution of the contract. A Ricardian contract, on the other hand, is more concerned with the semantic richness and production of a document that contains contractual legal prose. The semantics of a legal contract can be divided into two types:

- Operational semantics define the execution, correctness, and safety of the contract.
- Denotational semantics is concerned with the real "legal meaning" of the contract, i.e., the legal agreement.
- It makes sense to categorize smart contracts based on the difference between the semantics, but it is better to consider a smart contract as a standalone entity that is capable of encoding legal prose and code (business logic).



Some researchers have differentiated between smart contract code and smart legal contracts, where a smart contract is only concerned with the execution of the contract.

In Bitcoin, a straightforward implementation of basic smart contracts (conditional logic) can be observed, which is entirely oriented toward the execution and performance of the contract, whereas a Ricardian contract is more geared toward producing a document that is understood by humans with some parts that a computer program can understand.

This can be viewed as legal semantics versus operational performance (semantics versus performance), as shown in the following diagram:

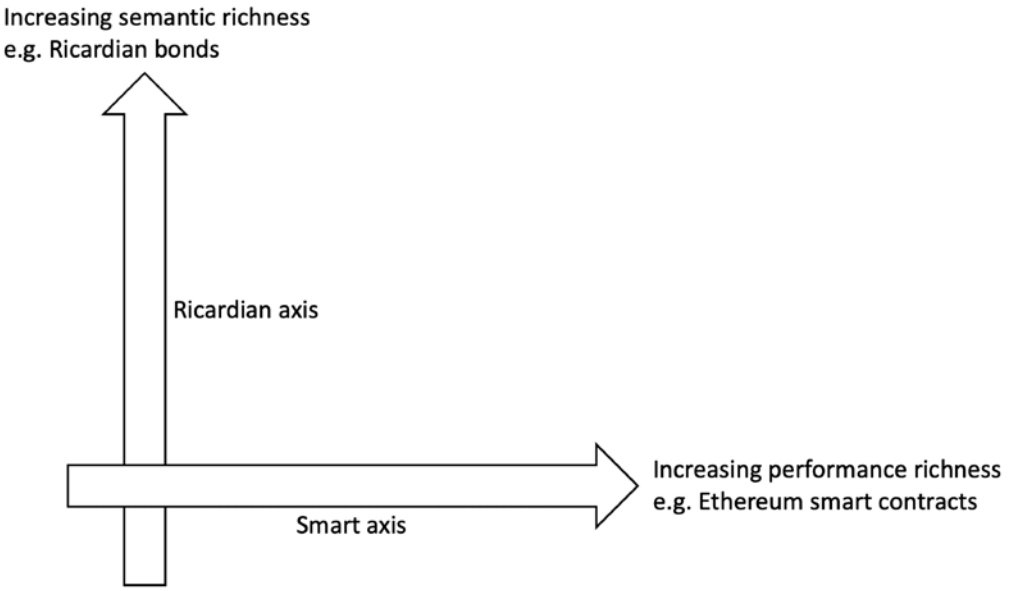


Figure 8.2: Diagram explaining that performance versus semantics is an orthogonal issue, as described by Ian Grigg; it is slightly modified to show examples of different types of contracts on both axes

The diagram shows that Ricardian contracts are more semantically rich, whereas smart contracts are more performance-rich.



This concept was initially proposed by *Ian Grigg* in his paper *On the intersection of Ricardian and Smart Contracts*. The paper is available at `https://iang.org/papers/intersection_ricardian_smart.html`.

A smart contract is made up of both of these elements (performance and semantics) embedded together, which completes the ideal model of a smart contract.

A Ricardian contract can be represented as a tuple of three objects, namely **prose**, **parameters**, and **code**. Prose represents the legal contract in natural language, code represents the program that is a computer-understandable representation of legal prose, and parameters join the appropriate parts of the legal contract to the equivalent code.



Ricardian contracts have been implemented in many systems, such as CommonAccord (`http://www.commonaccord.org`) and OpenBazaar (`https://github.com/OpenBazaar`).

Now that we understand what Ricardian contracts are, let's take a look at the concept of **smart contract templates**. These have been built on the idea of Ricardian contracts, and they aim to support the management of the entire life cycle of legal smart contracts.

Smart contract templates

Smart contracts can be implemented in any industry where they are required, but the most popular use cases relate to the financial sector. This is because blockchain first found many use cases in the finance industry and, therefore, sparked enormous research interest in the financial industry long before other areas. Recent work in the smart contract space specific to the financial sector has proposed the idea of smart contract templates. The idea is to build standard templates that provide a framework to support legal agreements for financial instruments.



Christopher D. Clack et al. proposed this idea in their paper published in 2016, named *Smart Contract Templates: foundations, design landscape and research directions*.

The paper is available at `https://arxiv.org/pdf/1608.00771.pdf`.

The paper also suggested that **Domain-Specific Languages** (**DSLs**) should be built to support the design and implementation of smart contract templates. A language named **Common Language for Augmented Contract Knowledge** (**CLACK**) has been proposed, and research has started to develop this language. This language is intended to be very rich and is expected to provide a large variety of functions, ranging from supporting legal prose to the ability to be executed on multiple platforms and cryptographic functions.

Clack et al. also carried out work to develop smart contract templates that support legally enforceable smart contracts. This proposal has been discussed in their research paper, *Smart Contract Templates: essential requirements and design options*.



This paper is available at `https://arxiv.org/pdf/1612.04496.pdf`.

The main aim of this paper is to investigate how legal prose could be linked with code using a mark-up language. It also covers how smart legal agreements can be created, formatted, executed, and serialized for storage and transmission. This work is ongoing and remains an open area for further research and development.

Contracts in the finance industry are not a new concept, and various DSLs are already in use in the financial services industry to provide a specific language for a particular domain. For example, there are DSLs available that support the development of insurance products, represent energy derivatives, or are being used to build trading strategies.



A comprehensive list of financial DSLs can be found at `http://www.dslfin.org/resources.html`.

It is also essential to understand the concept of DSLs, as this type of programming language can be developed to program smart contracts. DSLs are different from **General-Purpose Programming Languages** (**GPLs**). DSLs have limited expressiveness within a particular application or area of interest. These languages possess a small set of features that are sufficient and optimized for a specific domain only. Unlike GPLs, they are not suitable for building large general-purpose application programs.

Based on the design philosophy of DSLs, it can be envisaged that such languages will be developed specifically to write smart contracts. Some work has already been done, and **Solidity** is one such language that has been introduced with the Ethereum blockchain to write smart contracts. Vyper is another language that has been recently introduced for Ethereum smart contract development.

This idea of DSLs for smart contract programming can be further extended to a **GPL**. A smart contract modeling platform can be developed where a domain expert (not a programmer but a front-desk dealer, for example) can use a graphical user interface and a canvas (drawing area) to define and illustrate the definition and execution of a financial contract. Once the flow has been drawn and completed, it can be emulated first to test it and then be deployed from the same system to the target platform, which can be a smart contract on a blockchain or even a completely **Decentralized Application** (**DApp**). This concept is also not new, and a similar approach is already used in a non-blockchain domain, in the TIBCO StreamBase product, which is a Java-based system used for building event-driven, high-frequency trading systems.

It has been proposed that research should also be conducted in the area of developing high-level DSLs that can be used to program a smart contract in a user-friendly graphical user interface, thus allowing a non-programmer domain expert (for example, a lawyer) to design smart contracts. Another related concept is programmable money where certain conditions can be programmed into token spending. This could be a token that can only be spent if certain conditions are met, can be spent only on buying specific items, can be paid automatically to an entity if an event (for example, winning a game or delivering a service) occurs, or can only be released if several parties agree. It is possible for a graphical user interface to be developed that can facilitate the controlling flow of funds by programming specific conditions for the token. Here the domain expert could be an end user, a financial advisor, a trader, or an investor.

Apart from DSLs, there is also a growing interest in using general-purpose, already-established programming languages like Java, Go, and C++ to be used for smart contract programming. This idea is appealing, especially from a usability point of view, where a programmer who is already familiar with, for example, Java, can use their skills to write Java code instead of learning a new language. The high-level language code can then be compiled into a low-level bytecode for execution on the target platform. There are already some examples of such systems, such as in EOSIO blockchains, where C++ can be used to write smart contracts, which are compiled down to the web assembly for execution.

An inherent limitation of smart contracts is that they are unable to directly access any external data. The concept of oracles was introduced to address this issue.

Oracles

Oracles are an essential component of the smart contract and blockchain ecosystem. The limitation of smart contracts is that they cannot access external data because blockchains are closed systems, without any direct access to the real world. This external data might be required to control the execution of some business logic in the smart contract, for example, the stock price of a security product that is required by the contract to release dividend payments. In such situations, oracles can be used to provide external data to smart contracts. An oracle can be defined as an interface that delivers data from an external source to smart contracts. Oracles are trusted entities that use a secure channel to transfer off-chain data to a smart contract.

The following diagram shows a generic model of an oracle and smart contract ecosystem:

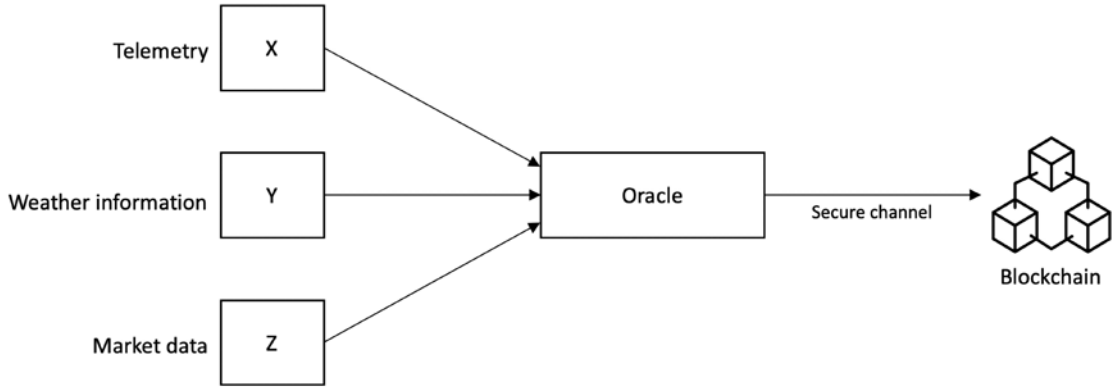


Figure 8.3: A generic model of an oracle and smart contract ecosystem with various data sources

Depending on the industry and use case requirements, oracles can deliver different types of data ranging from weather reports, real-world news, and corporate actions to data coming from an **Internet of Things** (**IoT**) device.

A list of some of the common use cases of oracles is shown here:

Type of data	Examples	Use case
Market data	Live price feeds of financial instruments: exchange rates, performance, pricing, and historic data of commodities, indices, equities, bonds, and currencies	DApps related to financial services, for example, decentralized exchanges and **decentralized finance** (**DeFi**)
Political events	Election results	Prediction markets
Travel information	Flight schedules and delays	Insurance DApps

Weather information	Flooding, temperature, and rain data	Insurance DApps
Sports	Results of football, cricket, and rugby matches	Prediction markets
Telemetry	Hardware IoT devices, sensor data, vehicle location, and vehicle tracker data	Insurance DApps Vehicle fleet management DApps

There are different methods used by oracles to write data into a blockchain, depending on the type of blockchain used. For example, in a Bitcoin blockchain, an oracle can write data to a specific transaction, and a smart contract can monitor that transaction in the blockchain and read the data. Other methods include storing the fetched data in a smart contract's storage, which can then be accessed by other smart contracts on the blockchain via requests between smart contracts depending on the platform. For example, in Ethereum, this can be achieved by using message calls.

The standard mechanics of how oracles work is presented here:

1. A smart contract sends a request for data to an oracle.
2. The request is executed, and the required data is requested from the source. There are various methods of requesting data from the source. These methods usually involve invoking APIs provided by the data provider, calling a web service, reading from a database (for example, in enterprise integration use cases where the required data may exist on a local enterprise legacy system), or requesting data from another blockchain. Sources can be any external off-chain data provider on the internet or in an internal enterprise network.
3. The data is sent to a notary to generate cryptographic proof (usually a digital signature) of the requested data to prove its validity (authenticity). Usually, TLSNotary is used for this purpose (`https://tlsnotary.org`). Other techniques include **Android proofs**, **Ledger proofs**, and **trusted hardware-assisted proofs**, which we will explain shortly.
4. The data with the proof of validity is sent to the oracle.
5. The requested data with its proof of authenticity can be optionally saved on a decentralized storage system, such as Swarm or IPFS, and can be used by the smart contract/blockchain for verification. This is especially useful when the proofs of authenticity are large and sending them to the requesting smart contracts (storing them on the chain) is not feasible.
6. Finally, the data, with the proof of validity, is sent to the smart contract.

This process can be visualized in the following diagram:

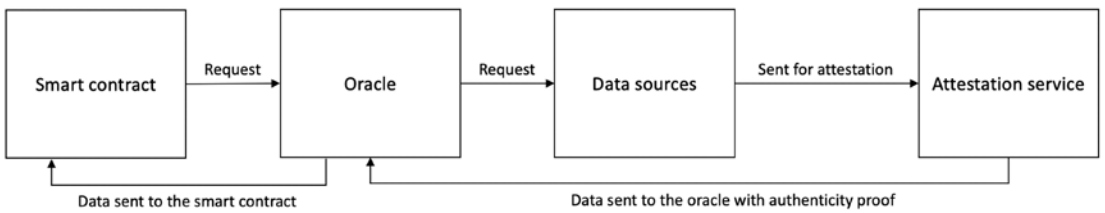


Figure 8.4: A generic oracle data flow

The preceding diagram shows the generic data flow of a data request from a smart contract to the oracle. The oracle then requests the data from the data source, which is then sent to the attestation service for notarization. The data is sent to the oracle with proof of authenticity. Finally, the data is sent to the smart contract with cryptographic proof (authenticity proof) that the data is valid.

Due to security requirements, oracles should also be capable of digitally signing or digitally attesting the data to prove that the data is authentic. This proof is called **proof of validity** or **proof of authenticity**.

Smart contracts subscribe to oracles. Smart contracts can either pull data from oracles, or oracles can push data to smart contracts. It is also necessary that oracles should not be able to manipulate the data they provide and must be able to provide factual data. Even though oracles are trusted (due to the associated proof of authenticity of data), it may still be possible that, in some cases, the data is incorrect due to manipulation or a fault in the system. Therefore, oracles must not be able to modify the data. This validation can be provided by using various cryptographic proofing schemes. We will now introduce different mechanisms to produce cryptographic proof of data authenticity.

Software-and network-assisted proofs

As the name suggests, these types of proofs make use of software, network protocols, or a combination of both to provide validity proofs. One of the prime examples of such proofs is TLSNotary.

TLSNotary

TLSNotary is a technology developed to be primarily used in the PageSigner project (`https://old.tlsnotary.org/pagesigner.html`) to provide web page notarization. This mechanism can also be used to provide the required security services to oracles. This protocol provides a piece of irrefutable evidence to an auditor that specific web traffic has moved between a client and a server. It is based on **Transport Layer Security** (**TLS**), which is a standard security mechanism that enables secure, bi-directional communication between hosts. It is extensively used on the internet to secure websites and allow HTTPS traffic.

A discussion of the internals of this protocol is beyond the scope of this book. Interested readers can read the standards document at `http://www.ietf.org/rfc/rfc2246.txt`. The link provided is only for TLS version 1.0 as TLSNotary only supports TLS version 1.0 or 1.1.

The key idea behind using TLSNotary is to utilize the TLS handshake protocol feature, which allows the splitting of the TLS master key into three parts. Each part is allocated to the server, the auditee, and the auditor. The oracle service provider (`https://provable.xyz`) becomes the auditee, whereas an **Amazon Web Services** (**AWS**) instance, which is secure and locked down, serves as the auditor.

TLS-N-based mechanism

This mechanism is one of the latest developments in this space. TLS-N is a TLS extension that provides secure non-repudiation guarantees. This protocol allows you to create privacy-preserving and non-interactive proofs of the content of a TLS session. TLS-N-based oracles do not need to trust any third-party hardware such as Intel SGX or a TLSNotary-type service to provide authenticity proofs of data web content (data) to the blockchain. In contrast to TLSNotary, this scheme works on the latest TLS 1.3 standard, which allows for improved security. More information regarding this protocol is available at `https://eprint.iacr.org/2017/578.pdf`.

Hardware device-assisted proofs

As the name suggests, these proofs rely on some hardware elements to provide proof of authenticity. In other words, they require specific hardware to work. Different mechanisms come under this category, and we will briefly introduce those next.

Android proof

This proof relies on Android's SafetyNet software attestation and hardware attestation to create a provably secure and auditable device. SafetyNet validates that a genuine Android application is being executed on a secure, safe, and untampered hardware device. Hardware attestation validates that the device has the latest version of the OS, which helps to prevent any exploits that existed due to vulnerabilities in the previous versions of the OS. This secure device is then used to fetch data from third-party sources, ensuring tamper-proof HTTPS connections. The very use of a provably secure device provides the guarantee and confidence (that is, proof of authenticity) that the data is authentic.



More information regarding SafetyNet and hardware attestation is available here:

`https://developer.android.com/training/safetynet`

`https://developer.android.com/training/articles/security-key-attestation.html`

Ledger proof

Ledger proof relies on the hardware cryptocurrency wallets built by the Ledger company (`https://www.ledger.com`). Two hardware wallets, **Ledger Nano S** and **Ledger Blue**, can be used for these proofs. The primary purpose of these devices is to be secure hardware cryptocurrency wallets. However, due to the security and flexibility provided by these devices, they also allow developers to build other applications for this hardware. These devices run a particular OS called **Blockchain Open Ledger Operating System** (**BOLOS**), which, via several kernel-level APIs, allows device and code attestation to provide a provably secure environment.

The secure environment provided by the device can also prove that the applications that may have been developed by oracle service providers and are running on the device are valid, authentic, and are indeed executing on the **Trusted Execution Environment** (**TEE**) of the ledger device. This environment, supported by both code and device attestation, provides an environment that allows you to run a third-party application in a secure and verifiable ledger environment to provide proof of data authenticity. Currently, this service is used by Provable, an oracle service, to provide untampered random numbers to smart contracts.



Ledger proofs, Android proofs, and TLSNotary proofs are used in provable oracles. The official documentation for these methods can be found here: `http://docs.provable.xyz`.

Currently, as these devices do not connect to the internet directly, the ledger devices cannot be used to fetch data from the internet.

Trusted hardware-assisted proofs

This type of proof makes use of trusted hardware, such as TEEs. A prime example of such a hardware device is Intel SGX. A general approach that is used in this scenario is to rely on the security guarantees of a secure and trusted execution provided by the secure element or enclave of the TEE device.

A prime example of a trusted hardware-assisted proof is Town Crier (`https://www.town-crier.org`), which provides an authenticated data feed for smart contracts. It uses Intel SGX to provide a security guarantee that the requested data has come from an existing trustworthy resource.



Intel SGX technology is developed by Intel, which provides a hardware TEE. More information about this is available here: `https://software.intel.com/en-us/sgx`. It is used by many different oracle service providers such as Town Crier and iExec (`https://docs.iex.ec/use-cases/iexec-doracle#the-iexec-solution-the-decentralized-oracle-doracle`).

Town Crier also provides a confidentiality service, which allows you to run confidential queries. The query for the data request is processed inside SGX Enclave, which provides a trusted execution guarantee, and the requested data is transmitted using a TLS-secured network connection, which provides additional data integrity guarantees.



It should be noted that in all the proof techniques mentioned earlier, there are many types of resources used to provide security guarantees, including hardware, network, and software. The categorization provided here is based on the principal element, either hardware or software, which plays a critical role in the overall security mechanism to provide security.

An issue can already be seen here, and that is the issue of trust. With oracles, we are effectively trusting a third party to provide us with the correct data. What if these data sources turn malicious, or simply due to a fault start providing incorrect data to the oracles? What if the oracle itself fails or the data source stops sending data? This issue can then damage the whole blockchain trust model. This phenomenon is called the **Blockchain oracle problem**. How do you trust a third party about the quality and authenticity of the data it provides? This question is especially real in the financial world, where market data must be accurate and reliable.

There are several proposed ways to overcome this issue. These solutions range from merely trusting a reputable third party to decentralized oracles. We have discussed some of the attestation techniques earlier; however, a third party due to a genuine fault or malicious intent may still provide data that is incorrect. Even if it is attested later on, the actual data itself is not guaranteed to be accurate. It might be acceptable for a smart contract designer in a certain use case to accept data for an oracle that is provided by a large, reputable, and trusted third party. For example, the source of the data can be a reputable weather reporting agency or airport information system directly relaying the flight delays, which can give some level of confidence. However, the issue of centralization remains.



The blockchain oracle problem can be defined formally as the conflict of trust between presumably trusted oracles (trusted third-party data sources) and completely trustless blockchain.

Based on the evolution of blockchain over the last few years, several types of blockchain oracles have emerged. We informally provided some background on these earlier, but now we will define these formally.

Types of blockchain oracles

There are various types of blockchain oracles, ranging from simple software oracles to complex hardware-assisted and decentralized oracles. Broadly speaking, we can categorize oracles into two categories: inbound oracles and outbound oracles.

Inbound oracles

This class represents oracles that receive incoming data from external services and feed it into the smart contract. We will shortly discuss software, hardware, and several other types of inbound oracle.

Software oracles

These oracles are responsible for acquiring information from online services on the internet. This type of oracle is usually used to source data such as weather information, financial data (stock prices, for example), travel information, and other types of data from third-party providers. The data source can also be an internal enterprise system, which may provide some enterprise-specific data. These types of oracles can also be called standard or simple oracles. A problem with these oracles is that as they are based simply on an online source of data, such as a website; if the source is not live anymore or has failed for some reason, then the feed to the blockchain smart contract will fail. However, redundancy can be used to make a single data source fault-tolerant.

Hardware oracles

This type of oracle is used to source data from hardware sources such as IoT devices or sensors. This is useful in use cases such as insurance-related smart contracts where telemetry sensors provide certain information, for example, vehicle speed and location. This information can be fed into the smart contract dealing with insurance claims and payouts to decide whether to accept a claim or not. Based on the information received from the source hardware sensors, the smart contract can decide whether to accept or reject the claim.

These oracles are useful in any situation where real-world data from physical devices is required. However, this approach requires a mechanism in which hardware devices are tamper-proof or tamper-resistant. This level of security can be achieved by providing cryptographic evidence (non-repudiation and integrity) of an IoT device's data and an anti-tampering mechanism on the IoT device, which renders it useless if there are any tampering attempts.

Computation oracles

These oracles allow computing-intensive calculations to be performed off-chain. As blockchain is not suitable for performing compute-intensive operations, a blockchain (that is, a smart contract on a blockchain) can request computations to be performed on off-chain, high-performance computing infrastructure and get the verified results back via an oracle. The use of an oracle, in this case, provides data integrity and authenticity guarantees.

An example of such an oracle is Truebit (`https://truebit.io`). It allows a smart contract to submit computation tasks to oracles, which are eventually completed by miners in return for an incentive. The fundamental idea is to offload compute-intensive operations to an off-chain system, and when the computation is completed, the results are posted back to the blockchain via the oracle.

Aggregation-based oracles

In this scenario, a single value is sourced from many different feeds. As an example, this single value can be the price of a financial instrument, and it can be risky to rely upon only one feed. To mitigate this problem, multiple data providers can be used where all of these feeds are inspected, and finally, the price value that is reported by most of the feeds can be picked up. The assumption here is that if the majority of the sources report the same price value, then it is likely to be correct. The collation mechanism depends on the use case: sometimes it's merely an **average** of multiple values, sometimes a **median** is taken of all the values, sometimes it's a **mean** value, and sometimes it's the **maximum value**. Regardless of the aggregation mechanism, the essential requirement here is to get a value that is valid and authentic, which eventually feeds into the system. Care must be taken here because a single malicious node can skew the results of a mean value by reporting a very large or small number. Usually, median values are used in practice because if most nodes are honest, the median value can be reasonably trusted. Of course, if most nodes are malicious, then even a median value cannot be trusted.



Two excellent examples of aggregation-based oracles that aggregate price feeds (also called price feed oracles) are MakerDAO (`https://makerdao.com/en/`) and price feed oracles (`https://developer.makerdao.com/feeds/`), which collate price data from multiple external price feed sources and provide a median ETHUSD price to MakerDAO.

Crowd wisdom-driven oracles

This is another way that the blockchain oracle problem can be addressed where a single source is not trusted. Instead, multiple public sources are used to eventually deduce the most appropriate data. In other words, it solves the problem where a single source of data may not be as trustworthy or accurate as expected. If there is only one source of data, it can be unreliable and risky to rely on entirely. It may turn malicious or become genuinely faulty.

In this case, to ensure the credibility of data provided by third-party sources for oracles, data is sourced from multiple sources. These sources can be users of the system or even members of the general public who have access to and have knowledge of some data, for example, a political event or a sporting event where members of the public know the results and can provide the required data. Similarly, this data can be sourced from multiple different news websites.

This data can then be aggregated, and if a sufficiently high number of the same information is received from multiple sources, then there is an increased likelihood that the data is correct and can be trusted.

Decentralized oracles

Another type of oracle, which primarily emerged due to decentralization requirements, is a **decentralized** oracle. Remember that in all the types of oracles discussed so far, there are some trust requirements to be placed in a trusted third party. As blockchain platforms such as Bitcoin and Ethereum are fully decentralized, it is expected that oracle services should also be decentralized. This way, we can address the **blockchain oracle problem**.

This type of oracle can be built based on a distributed mechanism. It can also be envisaged that the oracles can find themselves source data from another blockchain, which is driven by a distributed consensus, thus ensuring the authenticity of data. For example, one institution running its private blockchain can publish its data feed via an oracle that can then be consumed by other blockchains.

A decentralized oracle essentially allows off-chain information to be transferred to a blockchain without relying on a trusted third party.



Augur (visit `https://www.augur.net/whitepaper.pdf` for Jack Peterson et al.'s essay *A Decentralized Oracle and Prediction Market Platform*) is a prime example of such an oracle type. The Augur white paper is also available here: `https://arxiv.org/abs/1501.01042`.

The core idea behind Augur's oracle is that of crowd wisdom-supported oracles, in which information about an event is acquired from multiple sources and aggregated into the most likely outcome. The sources in the case of Augur are financially motivated reporters who are rewarded for correct reporting and penalized for incorrect reporting.



Decentralized, crowd wisdom-based, and aggregation-supported oracles can be categorized into a broader category of oracles called "consensus-driven oracles." Augur is based on the crowd wisdom-based oracle.

Smart oracles

An idea of a smart oracle has also been proposed by **Ripple Labs** (**Codius**). Its original whitepaper is available at `https://github.com/codius/codius-wiki/wiki/White-Paper#from-oracles-to-smart-oracles`. Smart oracles are entities just like oracles, but with the added capability of executing contract code. Smart oracles proposed by Codius run using Google Native Client, which is a sandboxed environment for running untrusted x86 native code.

Outbound oracles

This type, also called a **reverse oracle**, is used to send data out from the blockchain smart contracts to the outside world. There are two possible scenarios here; one is where the source blockchain is a producer of some data such as blockchain metrics, which are needed for some other blockchain.

The actual data somehow needs to be sent out to another blockchain smart contract. The other scenario is that an external hardware device needs to perform some physical activity in response to a transaction on-chain. However, note that this type of scenario does not necessarily need an oracle, because the external hardware device can be sent a signal as a result of the smart contract event.

On the other hand, it can be argued that if the hardware device is running on an external blockchain, then to get data from the source chain to the target chain, undoubtedly, it will need some security guarantees that oracle infrastructure can provide. Another situation is where we need to integrate legacy enterprise systems with the blockchain. In that case, the outbound oracle will be able to provide blockchain data to the existing legacy systems. An example scenario is the settlement of a trade done on a blockchain that needs to be reported to legacy settlement and backend reporting systems.

Cryptoeconomic oracles

These oracles are fundamentally decentralized exchanges, which are smart contracts that allow users to trade pairs of digital assets on-chain. As these decentralized exchanges need to maintain accurate and current prices for all digital assets, they can also be used to serve as an oracle to provide the correct and latest price of an asset.

Cryptoeconomic oracles provide economic assurance of correctness. In practice, this means that any adversarial entity that tries to game the system to its advantage, perhaps by introducing a fake price into the system, will lose its funds as a penalty due to the rules encoded in the system.

Now that we have discussed different types of oracles, we now introduce different service providers that provide these services. Several service providers provide oracle services for blockchain, some of which we will introduce now.

Blockchain oracle services

Various online services are now available that provide oracle services. These can also be called **oracle-as-a-service platforms**. All these services aim to enable a smart contract to securely acquire the off-chain data it needs to execute and make decisions:

- Town Crier: `https://www.town-crier.org`
- Provable: `https://provable.xyz`
- Witnet: `https://witnet.io`
- Chainlink: `https://chain.link`
- The Realitio project: `https://realit.io`
- Truebit: `https://truebit.io`
- iExec: `https://iex.ec`

Another service at `https://smartcontract.com/` is also available, where Ethereum, Bitcoin, and Town Crier oracles can be created. It allows smart contracts to connect to applications and allows you to feed data into the smart contracts from off-chain sources.

There are many oracle services available now, and it is challenging to cover all of them here. A random selection is presented in the preceding list.

Now that we've covered oracles in detail, let's look at smart contracts again and see how they can be deployed at a fundamental level.

Deploying smart contracts

Smart contracts may or may not be deployed on a blockchain, but it makes sense to do so on a blockchain due to the security and decentralized consensus mechanism provided by the blockchain. Ethereum is an example of a blockchain platform that natively supports the development and deployment of smart contracts. We will cover Ethereum in more detail in *Chapter 9*, *Ethereum Architecture*. Smart contracts on an Ethereum blockchain are typically part of a broader DApp.

In comparison, in a Bitcoin blockchain, the transaction timelocks, such as the `nLocktime` field, the `CHECKLOCKTIMEVERIFY` (**CLTV**), and the `CHECKSEQUENCEVERIFY` script operator in the Bitcoin transaction, can be seen as enablers of a simple version of a smart contract. These timelocks enable a transaction to be locked until a specified time or up to a number of blocks, thus enforcing a basic contract that a certain transaction can only be unlocked if certain conditions (elapsed time or number of blocks) are met. For example, you can implement conditions such as *Pay party* X N *number of bitcoins after 3 months*. However, this is very limited and should only be viewed as an example of a basic smart contract.

In addition to the example mentioned earlier, Bitcoin scripting language, though limited, can be used to construct basic smart contracts. One example of a basic smart contract is to fund a Bitcoin address that can be spent by anyone who demonstrates a **hash collision attack**. This was a contest that was announced on the Bitcointalk forum where bitcoins were set as a reward for whoever managed to find hash collisions (we discussed this concept in *Chapter 3*, *Symmetric Cryptography*) for hash functions. This conditional unlocking of Bitcoin solely for demonstrating of a successful attack is a basic type of smart contract.



This idea was presented on the Bitcointalk forum, and more information can be found at `https://bitcointalk.org/index.php?topic=293382.0`. This can be considered a basic form of a smart contract.

Various other blockchain platforms support smart contracts such as Monax, Lisk, Counterparty, Stellar, Hyperledger Fabric, Axoni Core, Neo, EOSIO, Solana, Polkadot, Avalanche, and Tezos. Smart contracts can be developed in various languages, either DSLs or GPLs. The critical requirement, however, is determinism, which is very important because it is vital that regardless of where the smart contract code is executed, it produces the same result every time and everywhere.

In blockchains where a GPL, such as, Rust, is used as a programming language for smart contracts, any non-deterministic features are removed before that language is implemented as a smart contract language in the blockchain protocol. For example, Solana uses the Rust language as a smart contract language, but with all non-deterministic features removed.



More details on this can be found here: `https://docs.solana.com/developing/on-chain-programs/developing-rust#restrictions`.

This requirement of the deterministic nature of smart contracts also implies that smart contract code is absolutely bug-free. Validation and verification of smart contracts is an active area of research, and a detailed discussion of this topic will be presented in *Chapter 17, Scalability and Chapter 19, Blockchain Security*.

Various languages have been developed to build smart contracts such as Solidity, which runs on the **Ethereum Virtual Machine** (**EVM**). It's worth noting that there are platforms that already support mainstream languages for smart contract development, such as Solana, which supports Rust. Another prominent example is Hyperledger Fabric, which supports Golang, Java, and JavaScript for smart contract development. Another example is EOSIO, which supports writing smart contracts in C++.

Security is of paramount importance for smart contracts. However, there are many vulnerabilities discovered in prevalent blockchain platforms and relevant smart contract development languages. These vulnerabilities result in some high-profile incidents, such as the DAO attack.

The DAO

The **Decentralized Autonomous Organization** (**DAO**), started in April 2016, was a smart contract written to provide a platform for investment. Due to a bug in the code, called the **reentrancy bug**, it was hacked in June 2016. An equivalent of approximately 3.6 million ether was siphoned out of the DAO into another account.

Even though the term "hacked" is used here, it was not really hacked. The smart contract did what it was asked to do but due to its vulnerabilities, the attacker was able to exploit it. It can be seen as an unintentional behavior (a bug) that the programmers of the DAO did not foresee. This incident resulted in a hard fork on the Ethereum blockchain, which was introduced to recover from the attack.

The DAO attack exploited a vulnerability (reentrancy bug) in the DAO code where it was possible to withdraw tokens from the DAO smart contract repeatedly before giving the DAO contract a chance to update its internal state, indicating how many DAO tokens have been withdrawn. The attacker was able to withdraw DAOs. However, before the smart contract could update its state, the attacker withdrew the tokens again. This process was repeated many times, but eventually, only a single withdrawal was logged by the smart contract, and the contract also lost a record of any repeated withdrawals.

The notion of *code is the law* or *unstoppable smart contracts* should be viewed with some skepticism, as the implementation of these concepts is still not mature enough to deserve complete and unquestionable trust. This is evident from the events after the DAO incident, where the Ethereum foundation was able to stop and change the execution of the DAO by introducing a hard fork on the Ethereum blockchain. Though this hard fork was introduced for genuine reasons, it goes against the true spirit of decentralization, immutability, and the notion that *code is the law*. Subsequently, resistance against this hard fork resulted in the creation of Ethereum Classic, where many users decided to keep mining on the old chain.

This chain is the original, non-forked Ethereum blockchain that still contains the DAO. It can be said that on this chain, *code is still the law*.



There are some interesting message threads and announcements related to this event, which readers may find informative and entertaining:

An open letter from *The Attacker* of the DAO: `https://pastebin.com/CcGUBgDG`

An announcement from Ethereum core dev: `https://twitter.com/avsa/status/745313647514226688`

Hard fork specification: `https://blog.slock.it/hard-fork-specification-24b889e70703`

The DAO attack highlights the dangers of not formally and thoroughly testing smart contracts. It also highlights the absolute need to develop a formal language for the development and verification of smart contracts. The attack also highlighted the importance of thorough testing to avoid the issues that the DAO experienced. There have been various vulnerabilities discovered in Ethereum over the last few years regarding the smart contract development language. Therefore, it is of utmost importance that a standard framework is developed to address all these issues.

Even though many different initiatives are aiming to explore and address the security of smart contracts, this field still requires further research to address limitations in smart contract programming languages. We will discuss these topics further in *Chapter 17, Scalability and Chapter 19, Blockchain Security*.

Advances in smart contract technology

Since the first availability of smart contracts on the Ethereum blockchain, significant development to create newer contract programming languages and improve existing languages has been made.

Solana Sealevel

Parallel execution of smart contracts is a significant area of research to improve the scalability of blockchain. Solana blockchain introduced **Sealevel**, a technology for the parallel execution of smart contracts, which helps increase the scalability and performance of blockchain. Sealevel is a runtime that allows the parallel execution of smart contracts on GPU cores available to a validator.



More information on Solana Sealevel is available here: `https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192`

We'll discuss the Solana blockchain in more detail in *Chapter 23*, *Alternative Blockchains*.

Digital Asset Modeling Language

DAML is a new Haskell-based functional programming language for blockchain that supports novel features like sub-transaction privacy, parallel transaction processing, scalability, ledger pruning, and high availability. The focus of DAML is on distributed business flows and lets developers focus more on programming them, rather than worrying about the underlying cryptography and other intricate details of blockchain architecture. This way, the underlying mechanics have been abstracted away, and developers can focus more on the business flows and actual business problems at hand. DAML is designed to build composable applications on an abstract DAML ledger model. DAML smart contracts are stored on a ledger. To execute, create, or read from the DAML ledger, the DAML Ledger API is used.

The DAML ledger model simplifies programming, and a developer doesn't have to think about the underlying ledger. Instead, developers can develop the code against the ledger model, and later implementors can select the actual ledger for implementation, be it Corda, VMware, or Hyperledger Fabric. It can even be a standard centralized database like PostgresSQL. In addition, it enables multiparty workflows by providing parties with a virtual shared ledger that encodes the current state of their shared contracts, written in DAML.

Due to business complexities, complex processes, and overly expansive workflows, current models are inefficient. Moreover, reconciliation between these systems is quite challenging, mainly due to a lack of interoperability. Usually, the answer is to automate the business processes, but the core complexity still exists and can result in problems later. Also, infrastructure code is usually interweaved with business applications, which results in fragile solutions lacking interoperability and portability.

DAML addresses such inefficiencies in current complex workflows by automatically isolating the business logic from the system (infrastructure) code. Furthermore, DAML allows for building composable applications on an abstract DAML ledger model, thus making developing interoperable and composable applications easy. Usually, different networks cannot talk with each other, even if the same technologies are used in both networks, resulting in isolated networks. Sometimes, these networks end up being able to communicate with each other only via rudimentary means, like sharing CSV or XML files. However, with DAML, the applications can readily communicate with each other regardless of the underlying platform by using DAML Ledger. For example, a Hyperledger Fabric network can atomically transact with a Quorum blockchain network. Also, these networks can exchange information with DAML applications running on centralized database servers, such as PostgreSQL, within an enterprise.

The diagram below shows this concept:

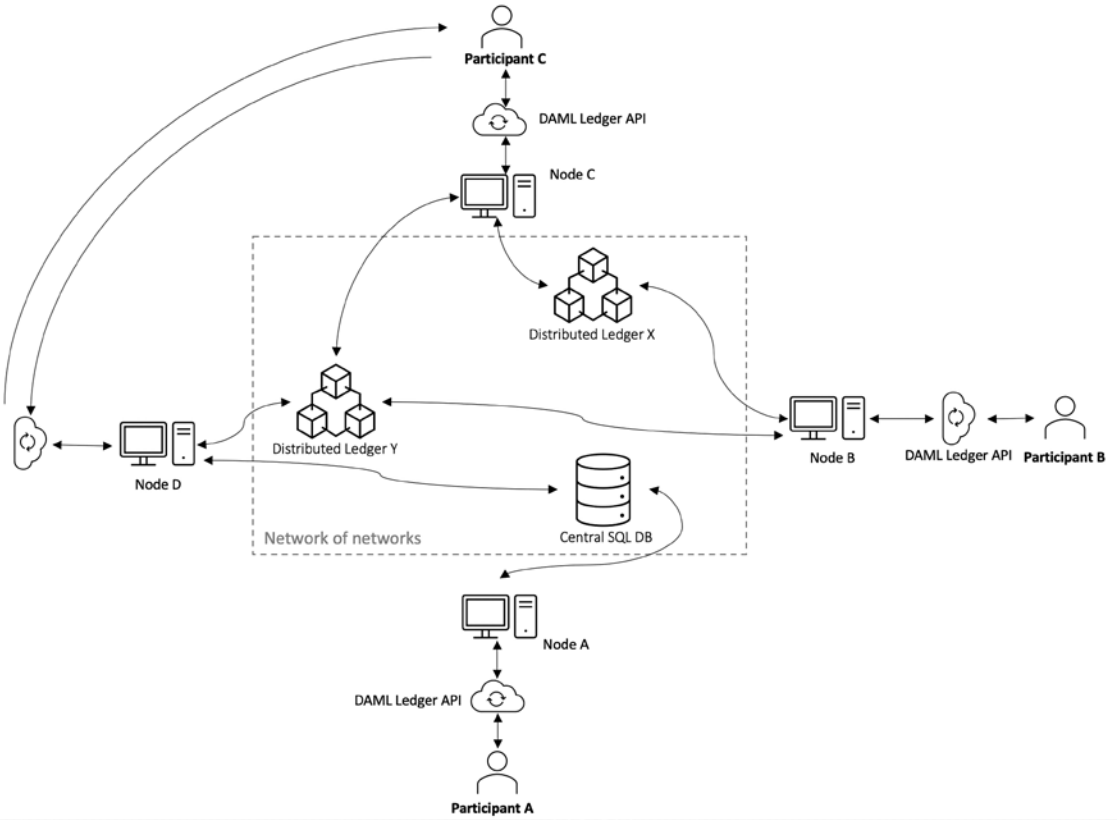


Figure 8.5: Interoperable ledger topology using DAML

In the preceding diagram, an example of a topology is shown where multiple ledgers are interoperating with multiple participants using the DAML ledger API as a *network of networks*.

DAML integrates with several distributed ledgers such as the VMware blockchain, Hyperledger Besu, and Hyperledger Fabric. It's a portable language and helps to achieve interoperability, as it can run on a DAML runtime engine that can be integrated with any blockchain or even a centralized database. So, in essence, the same code would work for a centralized database and a blockchain.

DAML is built with privacy in mind, which enables tracking and authorization at each workflow step; essentially, each smart contract has its own defined privacy.



More information on DAML is available here: `https://www.digitalasset.com/developers`

Other advances in smart contract languages include the development of tools and methods to analyze the security of smart contracts. These tools help to find vulnerable code patterns in smart contracts. Also, efforts have been made to use formal methods to verify smart contracts. Also, languages that are amenable to formal verification have been developed, such as the *Move* programming language. We will cover this in more detail in *Chapter 17, Scalability and Chapter 19, Blockchain Security*.

Summary

This chapter began by introducing a history of smart contracts followed by a detailed discussion of the definition of a smart contract. As there is no agreement on the standard definition of a smart contract, we attempted to introduce a definition that encompasses the crux of smart contracts.

An introduction to Ricardian contracts was also provided, and the difference between Ricardian contracts and smart contracts was explained, highlighting the fact that Ricardian contracts are concerned with the definition of a contract, whereas smart contracts are geared toward the actual execution of a contract.

The concept of smart contract templates was also discussed, in which high-quality active research is currently being conducted in academia and industry. Some ideas about the possibility of creating high-level DSLs were also discussed for creatin smart contracts or smart contract templates. In the later sections of the chapter, oracles were introduced, followed by a brief discussion on the DAO along with its security issues and smart contracts. Finally, we discussed advances in smart contracts.

In the next chapter, we will introduce Ethereum, which is one of the most popular blockchain platforms that inherently supports smart contracts.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

9

Ethereum Architecture

This chapter is an introduction to the Ethereum blockchain, its architecture, and design. We will introduce the fundamentals and various theoretical concepts behind Ethereum. A discussion on the different components, protocols, and algorithms relevant to the Ethereum blockchain is also presented.

We cover different elements of Ethereum such as transactions, accounts, the world state, the **Ethereum Virtual Machine** (**EVM**), the blockchain, the blockchain network, wallets, software clients, and supporting ecosystem protocols, with the goal of understanding the technical foundations on which Ethereum is built. The main topics we will explore in this chapter are as follows:

- Introducing Ethereum
- Cryptocurrency
- Keys and addresses
- Accounts
- Transactions and messages
- The EVM
- Blocks and blockchain
- Nodes and miners
- Networks
- Precompiled smart contracts
- Wallets and client software
- Supporting protocols

Let's begin with a brief overview of the foundation, architecture, and use of the Ethereum blockchain.

Introducing Ethereum

Vitalik Buterin conceptualized Ethereum in November 2013. The core idea proposed was the development of a Turing-complete language that allows the development of arbitrary programs (smart contracts) for blockchain and **Decentralized Applications** (**DApps**).

This concept is in contrast to Bitcoin, where the scripting language is limited and only allows necessary operations. It has gone through some key developments since its creation:

- The first version of Ethereum, called Olympic, was released in May 2015.
- Two months later, a version of Ethereum called Frontier was released in July.
- Another version named Homestead with various improvements was released in March 2016.
- The latest Ethereum release is called Arrow Glacier, which delays the difficulty bomb, a difficulty adjustment mechanism that eventually forces all miners to stop mining on Ethereum 1 and move to Ethereum 2.



A timeline of all major milestones and events is available here: `https://ethereum.org/en/history/`

A list of all releases as announced is maintained at `https://github.com/ethereum/go-ethereum/releases`

The formal specification of Ethereum has been described in the *yellow paper*, which can be used to develop Ethereum client implementations.

The Ethereum yellow paper (`https://ethereum.github.io/yellowpaper/paper.pdf`) was written by *Dr. Gavin Wood*, the founder of Ethereum and Parity (`http://gavwood.com`), and serves as a formal specification for the Ethereum protocol. Anyone can implement an Ethereum client by following the protocol specifications defined in the paper.

While this paper can be somewhat challenging to read, especially for those who do not have a background in algebra or mathematics and are not familiar with mathematical notation, it contains a complete formal specification for Ethereum. This specification can be used to implement a fully compliant Ethereum client. Therefore, it is necessary to understand this paper at a high level.

The Ethereum blockchain stack consists of various components. At the core, there is the Ethereum blockchain running on the peer-to-peer Ethereum network. Secondly, there's an Ethereum client (usually Geth) that runs on the nodes and connects to the peer-to-peer Ethereum network from where the blockchain is downloaded and stored locally. It provides various functions, such as mining and account management. The local copy of the blockchain is synchronized regularly with the network. Another component is the `web3.js` library, which allows interaction with the Geth client via the **Remote Procedure Call** (**RPC**) interface.

The overall Ethereum ecosystem architecture is visualized in the following diagram:

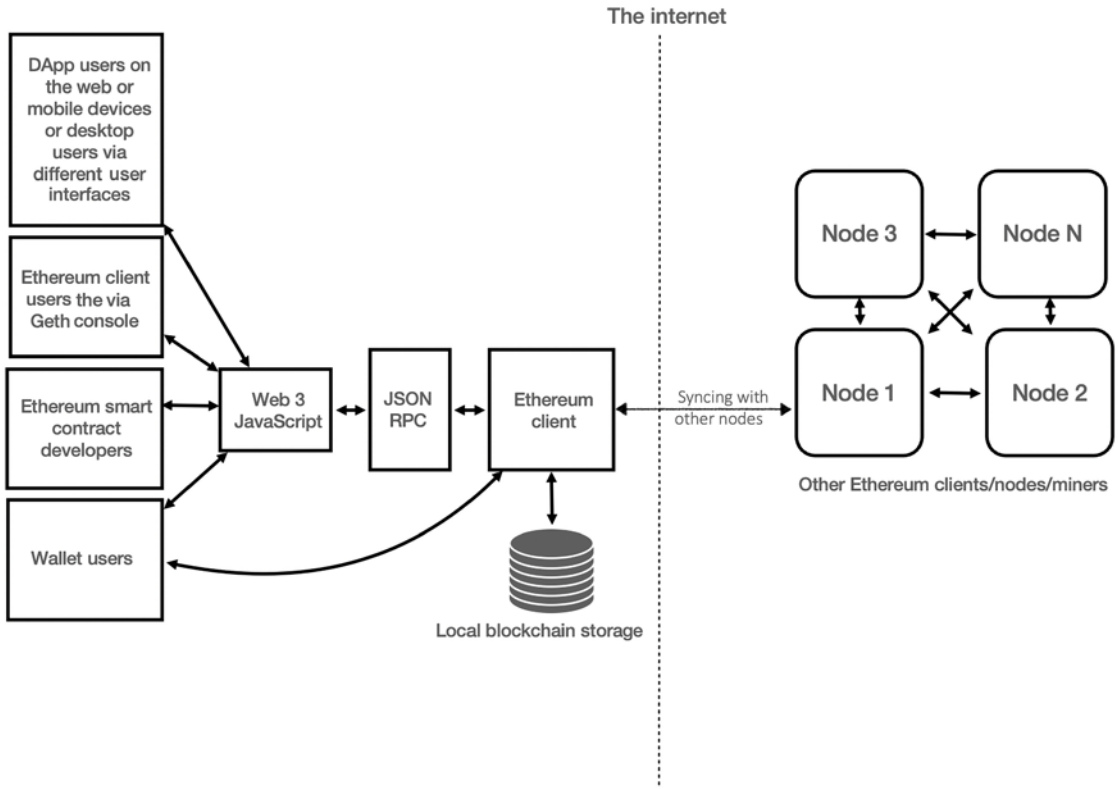


Figure 9.1: Ethereum's high-level ecosystem

A list of elements present in the Ethereum blockchain is presented here:

- Ether cryptocurrency/tokens
- Keys and addresses
- Accounts
- Transactions and messages
- The EVM
- The Ethereum blockchain
- Nodes and miners
- The Ethereum network
- Smart contracts and native contracts
- Wallets and client software

In the following sections, we will discuss each of these one by one.

Cryptocurrency

As an incentive to the miners, Ethereum rewards its own native currency, called **ether** (abbreviated as **ETH**). After the **Decentralized Autonomous Organization** (**DAO**) hack described in *Chapter 8*, *Smart Contracts*, a hard fork was proposed to mitigate the issue; therefore, there are now two Ethereum blockchains: one is called Ethereum Classic, and its currency is represented by ETC, whereas the hard-forked version is ETH, which continues to grow and on which active development is being carried out. ETC, however, has its own following with a dedicated community that is further developing ETC.

This chapter is focused on ETH, which is currently the most active and official Ethereum blockchain.

Ether is minted by miners as a currency reward for the computational effort they spend to secure the network by verifying transactions and blocks. Ether is used within the Ethereum blockchain to pay for the execution of contracts on the EVM. Ether is used to purchase gas as *crypto fuel*, which is required to perform computation on the Ethereum blockchain.



The denomination table is available at: `https://ethdocs.org/en/latest/ether.html#denominations`

Next, let's have a look at keys and addresses.

Keys and addresses

Keys and addresses are used in the Ethereum blockchain to represent ownership and transfer ether. The keys used are made up of pairs of private and public parts. The private key is generated randomly and is kept secret, whereas the public key is derived from the private key. Addresses are derived from public keys and are 20-byte codes used to identify accounts.

The process of key generation and address derivation is as follows:

1. First, a private key is randomly chosen (a 256-bit positive integer) under the rules defined by the elliptic curve `secp256k1` specification (in the range [`1`, `secp256k1n − 1`]).
2. The public key is then derived from this private key using the **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) recovery function. We will discuss this in the *Transactions and messages* section, in the context of digital signatures.
3. An address is derived from the public key, specifically, from the rightmost 160 bits of the Keccak hash of the public key.

An description of how keys and addresses look in Ethereum is as follows:

- A private key is basically a valid nonzero 256-bit integer randomly picked up from the range of the elliptic curve's field size (order). Ethereum client software makes use of the operating system's random number generator to generate this number.

Once this integer is picked up, it is represented in hex format (64 hexadecimal characters) as shown here: `b51928c22782e97cca95c490eb958b06fab7a70b9512c38c36974f47b954ffc4`.

- A public key is an *x*, *y* coordinate, a point on an elliptic curve that satisfies the elliptic curve equation. The public key is derived from the private key. Once we have the randomly chosen private key, it is multiplied by a preset point on the elliptic curve called the generator point, which produces another point somewhere on the curve. This other point is the public key and is represented in Ethereum as a hexadecimal string of 130 characters: `3aa5b8eefd12bdc2d26f1ae348e5f383480877bda6f9e1a47f6a4afb35cf998ab847 f1e3948b1173622dafc6b4ac198c97b18fe1d79f90c9093ab2ff9ad99260`.
- An address is a unique identifier on the Ethereum blockchain, which is derived from the public key by hashing it through the Keccak-256 hash function and keeping the last 20 bytes of the hash produced. An example is shown here: `0x77b4b5699827c5c49f73bd16fd5ce3d828c36f32`.



Note that the preceding private key is shown here only as an example and should not be reused.

The public and private key pair generated by the new account creation process is stored in key files located locally on the disk. These key files are stored in the `keystore` directory present in the relevant path according to the OS in use.

On the Linux OS, its location is as follows: `~/.ethereum/keystore`

The content of a `keystore` JSON file is shown here as an example. Can you correlate some of the names with what we have already learned in *Chapter 3*, *Symmetric Cryptography*, and *Chapter 4*, *Asymmetric Cryptography*? In the following example, it has been formatted for better visibility:

```
{
"address":"ba94fb1f306e4d53587fcdcd7eab8109a2e183c4",
"crypto":
{
    "cipher":"aes-128-ctr",
    "ciphertext":
        "b2ab4f94f5f44ce98e61d99641cd28eb00fd794129be25beb8a5fae89ef93241",
    "cipherparams":
        {"iv":"a0fdfe0a6d314a62ba6a370f438faa57"},
    "kdf":"scrypt",
    "kdfparams":
        {"dklen":32,"n":262144,"p":1,"r":8,
"salt":"be3e99203c24ffcb71a6be2823fc7a211c8cc10d66bc6b448fef420fa0669068"},
"mac":"1b0a42d4bf7a8e96d308179e9714718e902727ead7041b97a646ef1c9d6f9ad7"},
"id":"7e0772e0-965e-4a05-ad93-fc5d11245ba3",
"version":3
}
```

The private key is stored in an encrypted format in the `keystore` file. It is generated when a new account is created using the password and private key. The `keystore` file is also referred to as a UTC file as the naming format of it starts with UTC with the date timestamp therein. A sample name of a `keystore` file is shown here:

```
UTC--2022-01-18T17-46-46.604174215Z--ba94fb1f306e4d53587fcdcd7eab8109a2e183c4
```

On Unix-type operating systems, the `keystore` file is usually located under the user's home directory. For example, on a macOS, the `keystore` file is stored at the following location: `~/Library/Ethereum/keystore/`. For other operating systems, refer to the installation instructions available at `https://geth.ethereum.org/docs/getting-started/installing-geth`.



It is imperative to keep the associated passwords created at the time of creating the accounts and when the key files are produced safe.

As the `keystore` file is present on the disk, it is very important to keep it safe. It is recommended that it is backed up as well. If the `keystore` files are lost, overwritten, or somehow corrupted, there is no way to recover them. This means that any ether associated with the private key will be irrecoverable, too.

Now we have described the elements of the password `keystore` file and what they represent. The key purpose of this file is to store the configuration, which, when provided with the account password, generates a decrypted account private key. This private key is then used to sign transactions:

- `Address`: This is the public address of the account that is used to identify the sender or receiver.
- `Crypto`: This field contains the cryptography parameters.
- `Cipher`: This is the cipher used to encrypt the private key. In the following diagram, **AES-128-CTR** indicates the Advanced Encryption Standard, 128-bit, in counter mode. Remember that we covered this in *Chapter 3*, *Symmetric Cryptography*.
- `Ciphertext`: This is the encrypted private key.
- `Cipherparams`: This represents the parameters required for the encryption algorithm, **AES-128-CTR**.
- `IV`: This is the 128-bit initialization vector for the encryption algorithm.
- `KDF`: This is the key derivation function. It is `scrypt` in this case.
- `KDFParams`: These are the parameters for the **Key Derivation Function** (**KDF**).
- `Dklen`: This is the derived key length. It is 32 in our example.
- `N`: This is the iteration count.
- `P`: This is the parallelization factor; the default value is 1.
- `R`: This is the block size of the underlying hash function. It is set to 8, which is the default setting.
- `Salt`: This is the random value of salt for the **KDF**.
- `Mac`: This is the Keccak-256 hash output obtained following the concatenation of the second leftmost 16 bytes of the derived key together with the ciphertext.

- `ID`: This is a random identification number.
- `Version`: The version number of the file format, currently version 3.

Now we will explore how all these elements work together. This whole process is visualized in the following diagram:

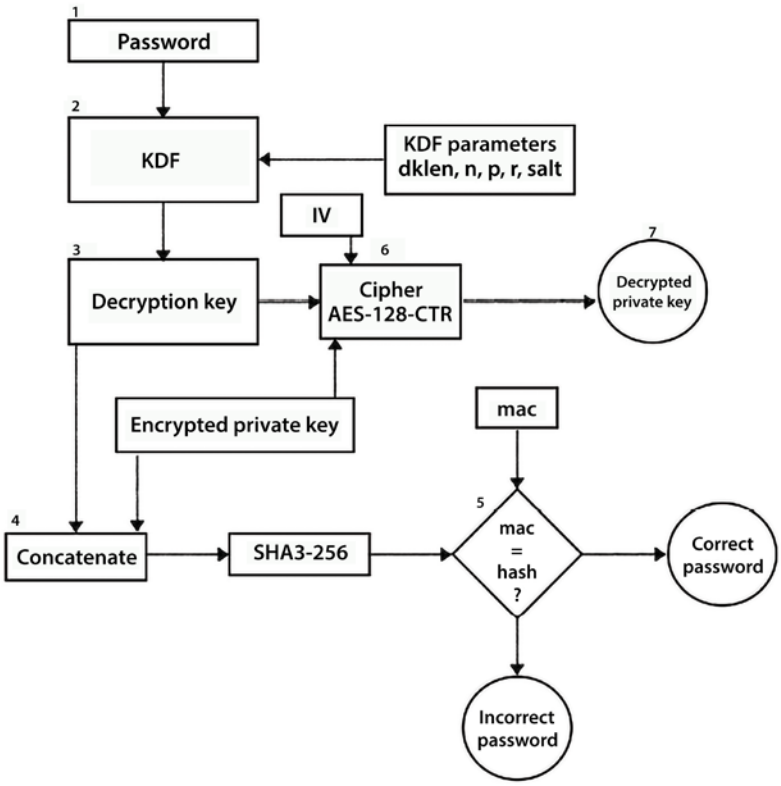


Figure 9.2: Private key decryption process

As shown in the preceding diagram, we can divide the process into different steps:

1. First, the password is fed into the **KDF**.
2. The KDF takes several parameters, namely, **dklen**, **n**, **p**, **r**, and **salt**, and produces a decryption key.
3. The decryption key is concatenated with an **encrypted private key** (ciphertext). The decryption key is also fed into the cipher algorithm, **AES-128-CTR**.
4. The concatenated **decryption key** and ciphertext are hashed using the **SHA3-256** hash function.
5. The **mac** is fed into the checking function where the hash produced from *step 4* is compared with the **mac**. If both match, then the password is valid, otherwise the password is incorrect.
6. The cipher function, which is fed with the **Initialization Vector** (**IV**), encrypted private key (ciphertext), and decryption key, decrypts the encrypted private key.
7. The **decrypted private key** is produced.

This decrypted private key is then used to sign transactions on the Ethereum network.

Another key element in Ethereum is an account, which is required to interact with the blockchain. It either represents a user or a smart contract.

Accounts

An account is one of the main building blocks of the Ethereum blockchain. It is defined by pairs of private and public keys. Accounts are used by users to interact with the blockchain via transactions. A transaction is digitally signed by an account before submitting it to the network via a node. With Ethereum being a *transaction-driven state machine*, the state is created or updated as a result of the interaction between accounts and transaction executions. All accounts have a state that, when combined together, represents the state of the Ethereum network. With every new block, the state of the Ethereum network is updated. Operations performed between and on the accounts represent state transitions. The state transition is achieved using what's called the Ethereum state transition function, which works as follows:

1. Confirm the transaction validity by checking the syntax, signature validity, and nonce.
2. The transaction fee is calculated, and the sending address is resolved using the signature. Furthermore, the sender's account balance is checked and subtracted accordingly, and the nonce is incremented. An error is returned if the account balance is insufficient.
3. Provide enough ETH (the gas price) to cover the cost of the transaction. We will cover gas and relevant concepts shortly in this chapter. This is charged per byte and is incrementally proportional to the size of the transaction. In this step, the actual transfer of value occurs. The flow is from the sender's account to the receiver's account. The account is created automatically if the destination account specified in the transaction does not exist yet. Moreover, if the destination account is a contract, then the contract code is executed. This also depends on the amount of gas available. If enough gas is available, then the contract code will be executed fully; otherwise, it will run up to the point where it runs **Out of Gas** (**OOG**) .
4. In cases of transaction failure due to insufficient account balance or gas, all state changes are rolled back except for the fee payment, which is paid to the miners.
5. Finally, the remainder (if any) of the fee is sent back to the sender as change and the fee is paid to the miners accordingly. At this point, the function returns the resulting state, which is also stored on the blockchain.

Now, as we understand accounts in Ethereum generally, let's examine the two kinds of accounts that exist in Ethereum.

Externally owned account (**EOA**): EOAs are similar to accounts in Bitcoin that are controlled by a private key. The same as in Bitcoin, where an address is controlled by a private key, Ethereum addresses are also controlled by their corresponding private keys. They are defined by three elements: *address*, *balance*, and *nonce*. They have the following properties:

- They have a state.
- They are associated with a human user, hence are also called user accounts.

- EOAs have an ether balance.
- They can send transactions, i.e., can transfer value and can initiate contract code.
- They have no associated code.
- They are controlled by private keys.
- An EOA's public address is derived from its private key.
- EOAs cannot initiate a call message.
- Accounts contain a key-value store.
- EOAs can initiate transaction messages.

Contract account (**CA**): CAs are accounts that have code associated with them in addition to being controlled with the private key. They are defined by five elements: *address*, *balance*, *nonce*, *StorageRoot*, and *codeHash*. They have the following properties:

- They have a state.
- They are not intrinsically associated with any user or actor on the blockchain.
- CAs have an ether balance.
- They have associated code that is kept in memory/storage on the blockchain. They can get triggered and execute code in response to a transaction or a message from other contracts. It is worth noting that due to the Turing-completeness property of the Ethereum blockchain, the code within CAs can be of any level of complexity. The code is executed by the EVM by each mining node on the Ethereum network. The EVM is discussed later in the chapter, in the *The EVM* section.
- A CA's public address is the combination of the public address of the EOA that created it and a transaction counter (how many transactions the EOA has sent) nonce.
- They have access to storage and can manipulate their storage.
- Also, CAs can maintain their permanent states and can call other contracts. It is envisaged that in future Ethereum releases, the distinction between EOAs and CAs may be eliminated.
- A CA can only be created from an EOA or from an already existing CA (contract).
- CAs cannot start transaction messages.
- CAs can initiate a call message, i.e., can call other contracts and contracts' functions.
- CAs can transfer ether.
- CAs contain a key-value store.
- CAs' addresses are generated when they are deployed. This address of the contract is used to identify its location on the blockchain.

Accounts allow interaction with the blockchain via transactions.

Transactions and messages

A transaction in Ethereum is a digitally signed (using a private key) data packet that contains the instructions that, when completed, either result in a message call or contract creation. Transactions are constructed by an actor external to the Ethereum blockchain or some external software tool. Smart contracts cannot send a transaction.

Initially, in Ethereum, there were simple transactions, message call transactions, and contract creation transactions. Later, as the system evolved, a need was felt to introduce more types and also to make it easier to introduce new features and future transaction types and be able to distinguish different types of transactions. New features such as access lists and EIP-1559 have been introduced while remaining compatible with legacy transactions. Further innovation resulted in the introduction of typed transactions, which were introduced in EIP-2718. They define a new transaction type, which is an envelope for future transaction types.

We will shortly explain what an Ethereum transaction is and consider its different types. But, before that, let's look at an important concept, the **Merkle Patricia Tree** (**MPT**), which is the foundation of integrity in the blockchain.

MPTs

It's essential to understand what an MPT is and how it guarantees blockchain data integrity. An MPT is used in all data structures relevant to the state and transactions, such as the world state, receipts, storage, and the account state. Remember, we discussed the Merkle tree in *Chapter 3*, *Symmetric Cryptography*.

Leaf nodes of a Merkle tree are paired and transactions are concatenated from a block for which the Merkle tree is to be constructed. Once these leaf nodes are hashed together, another set of nodes is created. It is again paired and concatenated and hashed until a single final hash value is obtained, called the Merkle root. In each block's header, there is a field in which this Merkle root is stored and serves as proof for all the transactions included in the block. Due to the collision-resistant property of hash functions, it is proven that changing even a single bit in any transaction will result in changing the Merkle hash value and, consequently, the block header's hash. Because of this, the Merkle root, the block header's hash, is a unique fingerprint for the block and all its transactions. Each new block also points to the preceding block by including its header's hash, and this pattern repeats up to the genesis block, making a hash-linked chain of blocks or simply a blockchain. This hash linking results in transaction linking due to the transaction's Merkle root being present in each block; we can verify this whole data structure for integrity.

The way integrity is checked is quite simple:

1. We need a reference to the latest block (block header) and the genuine genesis block.
2. Each hash field in the hash pointer present in the block header is compared to the previous block header's hash, and this process runs until the genesis block is verified. If the verification fails at any point, this means that there is some modification in the transactions or block header that resulted in a changed Merkle root. This process verifies the integrity of the chain.
3. Moreover, during the block validation checks, it is checked if the Merkle root in the block header matches the hash of all the transactions in the block.

To verify if a given transaction is part of the blockchain or not, the Merkle root needs to be recomputed and verified. This process is called *Merkle path authentication* or simply *Merkle proof*. Here's how it works:

If a transaction *t* is included in block *b*, then as long as the hash function is collision-resistant and the Merkle root value is authentic, *t* can be verified as follows:

1. First, calculate the hash of the element to be checked if it exists in the block. Let's say it is transaction *t*.
2. Next, get the Merkle root hash stored in the block header.
3. Calculate a candidate root hash value by using the transaction *t* for which the proof of inclusion is required and taking the off-path (i.e., off the authentication path) values. Note that the authentication path includes the siblings of nodes from transaction *t* to the root node (the Merkle root).
4. If the candidate root hash value matches the Merkle root hash stored in the block *b* header, it confirms that the transaction *t* is part of block *b*.

We now understand how Merkle proof of inclusion is performed and blockchain integrity is verified using Merkle trees; we can apply the same logic to MPTS. We can use the MPT to store, update, insert, delete, and retrieve key-value pairs. Each node in the tree can hold a part of a key, a value, and pointers to other nodes. An MPT is obtained by replacing the pointers with hash pointers, which means that the nodes are referenced by the hash of their contents instead of simple address pointers. The root of the MPT now serves the same purpose as the root of the Merkle tree we just saw, meaning that it represents the complete contents of the entire tree. This implies that the hash of the MPT's root node can be used to authenticate the entire key-value dataset of which the MPT is composed.

Ethereum goes a step further and customizes this MPT. The data of each node in the tree is stored in a database, and a hash value is used to look up this content. In Ethereum, there are three Merkle roots in the block header:

- **TransactionsRoot:** This is the root hash of the MPT of the transactions tree included in the block. Each block has its own separate transaction tree.
- **StateRoot:** This is the root hash of the MPT of the world state, i.e., persistent state.
- **ReceiptsRoot:** This is the root hash of the MPT of transaction receipts in the block. Each block has its own receipts tree.

Let's first now focus on transactions, after which we will discuss world state, account state, and transaction receipts, where the concept of *StateRoot* and *ReceiptsRoot* will become clear.

Transaction components

Transactions in Ethereum are composed of some standard fields, which are described as follows:

- Type: EIP-2718 transaction type 0 for legacy, 1 for EIP-2930, and 2 for EIP-1559. EIP-2718 defines a new generalized transaction envelope that allows the creation of different types of transactions. It is a simple format where the transaction type field is concatenated with a transaction payload, which can be of different types, depending on the transaction type. There can be 128 possible transaction types, as the transaction type is a number defined as ranging from 0 to 0x7f. Defining generalized envelopes like this allows for backward compatibility and less complexity while being able to create different types of new transactions. The transaction payload is a byte array defined by the transaction type.

- There are also three standard legacy transaction types based on the output they produce:
    - **Simple transactions**: This is the standard transaction used for payments. It transfers ether (funds) between EOAs.
    - **Message call transactions**: This transaction is used to execute smart contracts, i.e., used to invoke methods in the deployed smart contracts.
    - **Contract creation transactions**: These transactions result in the creation of a new contract account, i.e., a smart contract—an account with the associated code.

  We'll elaborate more on these core transaction types later, under the *Transaction types* section.

- **Nonce**: The nonce is a number that is incremented by one every time a transaction is sent by the sender. It must be equal to the number of transactions sent and is used as a unique identifier for the transaction. A nonce value can only be used once. This is used for replay protection on the network.
- **Gas price**: The **gas price** field represents the amount of Wei required to execute the transaction. In other words, this is the amount of Wei you are willing to pay for this transaction. This is charged per unit of gas for all computation costs incurred as a result of the execution of this transaction.



Wei is the smallest denomination of ether; therefore, it is used to count ether.

- **Gas limit**: The **gas limit** field contains the value that represents the maximum amount of gas that can be consumed to execute the transaction. The concept of gas and gas limits will be covered later in the chapter in more detail. For now, it is sufficient to say that this is the fee amount, in ether, that a user (for example, the sender of the transaction) is willing to pay for computation.
- **To**: As the name suggests, the **To** field is a value that represents the address of the recipient of the transaction. This is a 20-byte value.
- **Value**: Value represents the total number of Wei to be transferred to the recipient; in the case of a CA, this represents the balance that the contract will hold.
- **Signature**: Transactions are signed using recoverable ECDSA signatures. The signature is composed of three fields, namely **V**, **R**, and **S**. These values represent the digital signature (*R*, *S*) and some information that can be used to recover the public key (*V*). Also, the sender of the transaction can be determined from these values. The signature is based on the ECDSA scheme and makes use of the `secp256k1` curve. The theory of **Elliptic Curve Cryptography** (**ECC**) was discussed in *Chapter 4*, *Asymmetric Cryptography*. In this section, ECDSA will be presented in the context of its usage in Ethereum.

V is a single-byte value that depicts the size and sign of the elliptic curve point and can be either 27 or 28. *V* is used in the ECDSA recovery contract as a recovery value. This value is used to recover (derive) the public key from the private key. In `secp256k1`, the recovery value is expected to be either 0 or 1. In Ethereum, this is offset by 27.

R is derived from a calculated point on the curve. First, a random number is picked, which is multiplied by the generator of the curve to calculate a point on the curve. The *x*-coordinate part of this point is *R*. *R* is encoded as a 32-byte sequence. *R* must be greater than 0 and less than the `secp256k1n` limit (`115792089237316195423570985008687907852837564279074904382605163141518161494337`).

S is calculated by multiplying *R* with the private key and adding it to the hash of the message to be signed, and then finally dividing it by the random number chosen to calculate *R*. *S* is also a 32-byte sequence. *R* and *S* together represent the signature.

To sign a transaction, the `ECDSASIGN` function is used, which takes the message to be signed and the private key as an input and produces *V*, a single-byte value; *R*, a 32-byte value; and *S*, another 32-byte value. The equation is as follows:

ECDSASIGN (Message, Private Key) = (V, R, S)

- **Init**: The Init field is used only in transactions that are intended to create contracts, that is, contract creation transactions. This represents a byte array of unlimited length that specifies the EVM code to be used in the account initialization process. The code contained in this field is executed only once, when the account is created for the first time. It (init) gets destroyed immediately after that. Init also returns another code section, called the *body*, which persists and runs in response to message calls that the CA may receive. These message calls may be sent via a transaction or internal code execution.
- **Data**: If the transaction is a message call (i.e., calling methods in a deployed smart contract), then the **Data** field is used instead of init. It contains the input data of the message call. In other words, it contains data such as the name of the function in the smart contract to be called and relevant parameters. It is unlimited in size and is organized as a byte array.

The **Type 1** transaction (EIP-2930) introduced the following fields:

- **ChainID:** The transaction is only valid on the blockchain network with this specific ChainID.
- **AccessList**: This field contains a list of access entries. Each access list is a tuple of account addresses and a list of storage keys that the transaction intends to access.
- **yParity:** This is signature Y parity, instead of V in a legacy transaction. The parity of the y-value of a `secp256k1` signature is 0 for even and 1 for odd.

Legacy transactions do not have an AccessList field. In addition, ChainID and yParity are combined into a single field, W, described below, in legacy transactions.

- **W:** This is a scalar value that encodes Y parity and ChainID. This is based on EIP-155.

EIP-1559 introduces a new transaction type. We call them **Type 2** transactions. Most of the fields remain the same as the legacy structure, however, some new fields are added, which are described below:

- **ChainID**: This field is now part of the transaction payload. It used to be encoded in the signature V value due to EIP-155.
- **maxPriorityFeePerGas** and **maxFeePerGas**: Two new fields introduced as a replacement for the legacy gasPrice field.
- **Gas Limit**, **destination**, **amount**, and **data** remain the same as legacy transactions.
- **AccessList:** This is the same as EIP-2930, where the access list can be defined.
- **signatureYparity:** The signature V value from legacy transactions is replaced with signatureYparity, which can be either a 0 or 1, depending on which *y*-coordinate on the elliptic curve is used. In legacy transactions, it used to be 27, 28 or 35, 36 in EIP-155 transactions for a signature recovery identifier.
- **signatureR** and **signatureS** remain the same as legacy and Type 1 transactions. Recall from *Chapter 4*, *Asymmetric Cryptography*, that these are simply two points on the elliptic curve.

When a transaction is successfully executed, a record in the transaction tree is created containing all fields related to the transaction. This structure is visualized in the following diagram, where a transaction is a tuple of the fields mentioned earlier, which is then included in a **transaction trie** (a modified MPT) composed of the transactions to be included. Finally, the root node of the transaction trie is hashed using a Keccak 256-bit algorithm and is included in the block header along with a list of transactions in the block:

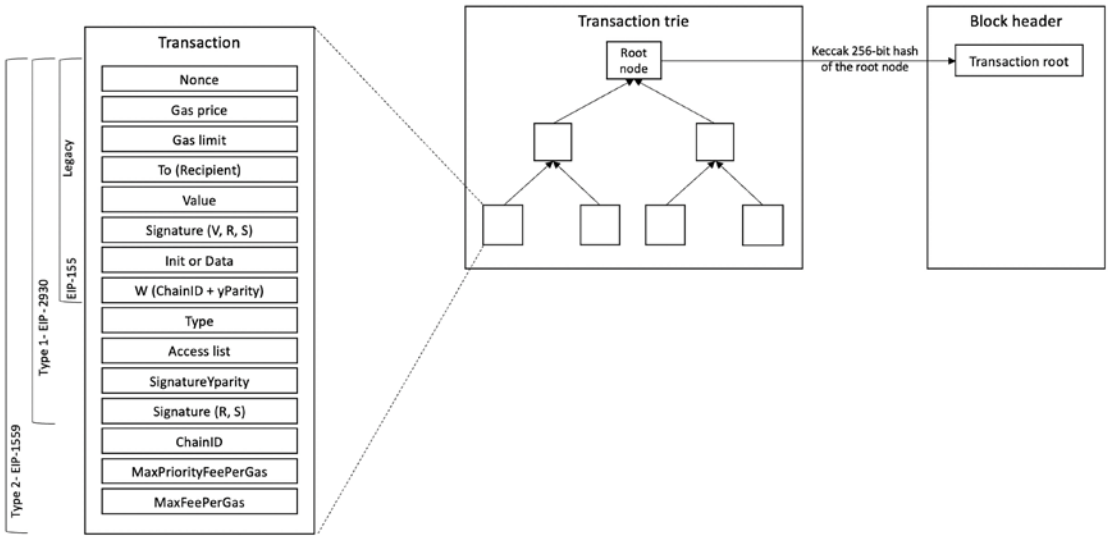


Figure 9.3: The relationship between the transaction, transaction trie, and block header

The question that arises here is how all these accounts, transactions, and related messages flow through the Ethereum network and how are they stored. So, before we move on to the different types of transactions and messages in Ethereum, let's explore how Ethereum data is encoded for storage and transmission. For this purpose, a new encoding scheme called **Recursive Length Prefix** (**RLP**) was developed, which we will cover here in detail.

Recursive Length Prefix

To define RLP, we first need to understand the concept of serialization. Serialization is simply a mechanism commonly used in computing to encode data structures into a format (a sequence of bytes) that is suitable for storage and/or transmission over the communication links in a network. Once a receiver receives the serialized data, it de-serializes it to obtain the original data structure. Serialization and deserialization are also referred to as marshaling and un-marshaling, respectively. Some commonly used serialization formats include XML, JSON, YAML, protocol buffers, and XDR. There are two types of serialization formats, namely *text* and *binary*. In a blockchain, there is a need to serialize and deserialize different types of data such as blocks, accounts, and transactions to support transmission over the network and storage on clients.



Why do we need a new encoding scheme when there are so many different serialization formats already available? The answer to this question is that RLP is a deterministic scheme, whereas other schemes may produce different results for the same input, which is absolutely unacceptable on a blockchain. Even a small change will lead to a totally different hash and will result in data integrity problems that will render the entire blockchain useless.

RLP is an encoding scheme developed by Ethereum developers. It is a specially developed encoding scheme that is used in Ethereum to serialize binary data for storage or transmission over the network and also to save the state in an MPT on storage media. It is a deterministic and consistent binary encoding scheme used to serialize objects on the Ethereum blockchain such as account state, transactions, messages, and blocks. It operates on strings and lists to produce raw bytes that are suitable for storage and transmission. RLP is a minimalistic and simple-to-implement serialization format that does not define any data types and simply stores structures as nested arrays. In other words, RLP does not encode specific data types; instead, its primary purpose is to encode structures.



More information on RLP is available on the Ethereum wiki, at `https://eth.wiki/en/fundamentals/rlp`.

Now, having defined RLP, we can delve deeper into transactions and other relevant elements of the Ethereum blockchain. Each operation on the Ethereum network costs some amount of ETH and is charged using a fee mechanism. This fee is also called gas. We will now introduce this mechanism in detail.

Gas

Gas is required to be paid for every operation performed on the Ethereum blockchain. This is a mechanism that ensures that infinite loops cannot cause the whole blockchain to stall due to the Turing-complete nature of the EVM. A transaction fee is charged as an amount of Ether and is taken from the account balance of the transaction originator.

Gas is charged in three scenarios as a prerequisite to the execution of an operation:

- The computation of an operation
- For contract creation or message calls
- An increase in the use of memory

A fee is paid for transactions to be included by miners for mining. If this fee is too low, the transaction may never be picked up; the higher the fee, the higher the chances that the transactions will be picked up by the miners for inclusion in the block. Conversely, if a transaction that has an appropriate fee paid is included in the block by miners but has too many complex operations to perform, it can result in an OOG exception if the gas cost is not enough. In this case, the transaction will fail but will still be made part of the block, and the transaction originator will not get a refund.

Transaction costs can be estimated using the following formula:

*Total cost = gasUsed * gasPrice*

Here, *gasUsed* is the total gas that is supposed to be used by the transaction during the execution, and *gasPrice* is specified by the transaction originator as an incentive to the miners to include the transaction in the next block. This is specified in ETH. Each EVM opcode has a fee assigned to it. It is an estimate because the gas used can be more or less than the value specified by the transaction originator originally. For example, if computation takes too long or the behavior of the smart contract changes in response to some other factors, then the transaction execution may perform more or fewer operations than intended initially and can result in consuming more or less gas. If the execution runs OOG, everything is immediately rolled back; otherwise, if the execution is successful and some gas remains, then it is returned to the transaction originator.



A website that keeps track of the latest gas price and provides other valuable statistics and calculators is available at `https://ethgasstation.info`

Each operation costs some gas; a high-level fee schedule of a few operations is shown as an example here:

Operation name	Gas cost
Stop	0
SHA3	30

SLOAD	800
Transaction	21000
Contract creation	32000

Based on the preceding fee schedule and the formula discussed earlier, an example calculation of the SHA-3 operation can be calculated as follows:

- SHA-3 costs 30 gas.
- Assume that the current gas price is 25 GWei, and convert it into ETH, which is 0.000000025 ETH. After multiplying both, 0.000000025 * 30, we get 0.00000075 ETH.
- In total, 0.00000075 ETH is the total gas that will be charged.

Transaction fees have been very high and at times the fee exceeded the value of the asset. On Ethereum, the high transaction fee is due to scalability constraints and how the economics of Ethereum works, instead of a design limitation. There are two aspects that one can think of when thinking about gas fees: opcodes and gas prices.

One idea that comes to mind is to reduce the EVM opcode gas cost, thereby lowering the fees. However, this would not work—the opcode gas cost estimates the time required to process an opcode, roughly calculated in milliseconds + some other constraints. In contrast, gas price is a function of Ethereum's fee market and supply and demand. This market exists because of the limited block size (maximum block size—12.5M gas—limit on the total gas that can be consumed by transaction in a block) where everyone is trying to get their transaction processed first. Therefore, it is a first-price auction market mechanism. All bidders bid simultaneously and the highest bidder wins. If the opcode cost is reduced, then due to supply and demand, the network will always have this high price fee due to this fee market mechanism.

Users are also willing to pay more to get their transaction processed first and miners can also ignore the low fee transaction and can order them as they like. So, the network gas price will remain high because, fundamentally, the Ethereum transaction fee market is a highest-bidder-wins market. Unless that is changed, only reducing the opcode cost will not result in any improvement. Opcodes' gas cost is there for a different reason. In comparison, gas price is a result of supply and demand and limited block size.

Other issues are DoS attacks and slower computers. Slower computers may not be able to process opcodes quick enough. For example, imagine a standard transfer instruction is 21,999 gas units. This is because it involved some ECC cooperation, which is estimated to take some time. Imagine the estimated time is hypothetically 21 milliseconds, because the gas cost is an estimate of the time it takes to process an opcode instruction. Now, if we reduce the gas cost, a faster computer will be able to do it quicker and within, let's say, 10, 000 gas untis. However, a slower computer will not be able to complete the whole operation in time.

Therefore, opcodes' cost cannot be just halved or reduced below a certain level. We can think of it as faster block times and bigger block sizes resulting in lower transaction fees. Some new layer 1 chains have introduced this, such as Solana.

Transaction types

We will continue by exploring different types of transactions on the Ethereum blockchain. We discussed this briefly before, under the section transaction components, however, now we'll dig deeper and learn about different elements of three core types of transactions in Ethereum.

Simple transactions

This is the standard value transfer transaction that Ethereum supports. It is used to transfer funds (ether) between accounts.

Contract creation transactions

A contract creation transaction is used to create smart contracts on the blockchain. There are a few essential parameters required for a contract creation transaction. These parameters are listed as follows:

- The sender
- The transaction originator
- Available gas
- Gas price
- Endowment, which is the amount of ether allocated
- A byte array of an arbitrary length
- Initialization EVM code
- The current depth of the message call/contract-creation stack (current depth means the number of items that are already present in the stack)

The initialization EVM code from the init field in the transaction contains the EVM bytecode compiled by tools such as the Solidity compiler. Once the contract is deployed, it is added as a new item in the world state trie. Its bytecode is stored in the codeHash field in the account state. Moreover, a new storage trie is created and its root is saved in the `StorageRoot` field in the account state.

Addresses generated as a result of a contract creation transaction are 160 bits in length. Precisely as defined in the yellow paper, they are the rightmost 160 bits of the Keccak hash of the RLP encoding of the structure containing only the sender and the nonce. Initially, the nonce in the account is set to zero. The balance of the account is set to the value passed to the contract. The storage is also set to empty. The codeHash is a Keccak 256-bit hash of the empty string.

The new account is initialized when the EVM code (the initialization EVM code, mentioned earlier) is executed. In the case of any exception during code execution, such as not having enough gas (running **OOG**), the state does not change. If the execution is successful, then the account is created after the payment of the appropriate gas costs.

Since **Ethereum Homestead**, the result of a contract creation transaction is either a new contract with its balance or no new contract is created with no transfer of value. This contrasts with versions prior to Homestead, where the contract would be created regardless of the contract code deployment being successful or not due to an OOG exception.

Message call transactions

The state is altered by transactions. These transactions are created by external factors (users) and are signed and then broadcast to the Ethereum network. After a smart contract is deployed successfully, its functions are called by transactions called message calls. We can simply think of these as *contract execution* transactions. A message call transaction requires several parameters for execution, which are listed as follows:

- The sender
- The transaction originator
- The recipient (contract address)
- The account whose code is to be executed (usually the same as the recipient)
- Available gas
- The value
- The gas price
- An arbitrary-length byte array
- The input data of the call
- The current depth of the message call/contract creation stack

Message calls result in a state transition. Message calls also produce output data, which is not used if transactions are executed. In cases where message calls are triggered by EVM code, the output produced by the transaction execution is used. As defined in the yellow paper, a message call is the act of passing a message from one account to another. If the destination account has an associated EVM code, then the EVM will start upon the receipt of the message to perform the required operations. If the message sender is an autonomous object (external actor), then the call passes back any data returned from the EVM operation.

Messages are passed between accounts using message calls. A description of messages and message calls is presented next.

Messages

Messages, as defined in the yellow paper, are the data and values that are passed between two accounts. A **message** is a data packet passed between two accounts. This data packet contains data and a value (the amount of ether). It can either be sent via a smart contract (autonomous object) or from an external actor (an **Externally Owned Account**, or **EOA**) in the form of a transaction that has been digitally signed by the sender.

Contracts can send messages to other contracts. Messages only exist in the execution environment and are never stored. Messages are similar to transactions; however, the main difference is that they are produced by the contracts, whereas transactions are produced by entities external to the **Ethereum Environment** (**EOAs**).

A message consists of the following components:

- The sender of the message

- The recipient of the message
- The amount of Wei to transfer and the message to be sent to the contract address
- An optional data field (input data for the contract)
- The maximum amount of gas (`startgas`) that can be consumed

Messages are generated when the `CALL` or `DELEGATECALL` opcodes are executed by the contract running in the EVM.

A call does not broadcast anything to the blockchain; instead, it is a local call and executes locally on the Ethereum node. It is almost like a local function call. It does not consume any gas as it is a read-only operation. It is akin to a dry run or a simulated run. Calls also do not allow ether transfer to CAs. Calls are executed locally on a node EVM and do not result in any state change because they are never mined. Calls are processed synchronously, and they usually return the result immediately.



Do not confuse a *call* with a *message call transaction*, which in fact results in a state change. A call basically runs message call transactions locally on the client and never costs gas nor results in a state change. It is available in the `web3.js` JavaScript API and can be seen as a simulated mode of the message call transaction. On the other hand, a *message call transaction* is a `write` operation and is used for invoking functions in a CA (i.e., smart contract), which does cost gas and results in a state change.

Transaction validation and execution

Transactions are executed after their validity has been verified. The initial checks are listed as follows:

- A transaction must be well formed and RLP-encoded without any additional trailing bytes.
- The digital signature used to sign the transaction must be valid.
- The transaction nonce must be equal to the sender's account's current nonce.
- The gas limit must not be less than the gas used by the transaction.
- The sender's account must have a sufficient balance to cover the execution cost.

A transaction substate is created during the execution of the transaction and is processed immediately after the execution completes. This transaction substate is a tuple that is composed of four items. These items are as follows:

- **Suicide set or self-destruct set**: This element contains the list of accounts (if any) that are disposed of after the transaction executes.
- **Log series**: This is an indexed series of checkpoints that allows the monitoring and notification of contract calls to the entities external to the Ethereum environment, such as application frontends. It works like a trigger mechanism that is executed every time a specific function is invoked, or a specific event occurs. Logs are created in response to events occurring in the smart contract. It can also be used as a cheaper form of storage. Events will be covered with practical examples in *Chapter 12*, *Web3 Development Using Ethereum*.

- **Refund balance**: This is the total price of gas for the transaction that initiated the execution. Refunds are not immediately executed; instead, they are used to offset the total execution cost partially.
- **Touched accounts**: Touched accounts can be defined as those accounts that are involved in any potential state-changing operation. Empty accounts from this set are deleted at the end of the transaction. The final state is reached after deleting accounts in the self-destruct set and emptying accounts from the touched accounts set.

State and storage in the Ethereum blockchain

Ethereum, just like any other blockchain, can be visualized as a transaction-based state machine. This definition is mentioned in the Ethereum yellow paper written by Dr. Gavin Wood.

The core idea is that in the Ethereum blockchain, a genesis state is transformed into a final state by executing transactions incrementally. The final transformation is then accepted as the absolute undisputed version of the state. In the following diagram, the Ethereum state transition function is shown, where a transaction execution has resulted in a state transition:

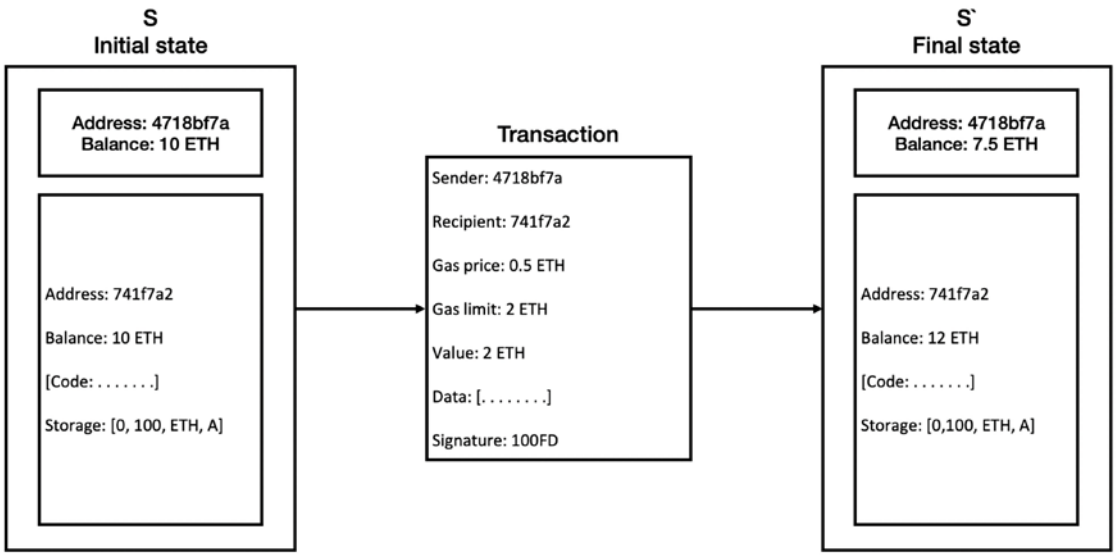


Figure 9.4: Ethereum state transition function

In the preceding example, a transfer of two ether from address **4718bf7a** to address **741f7a2** is initiated. The initial state represents the state before the transaction execution, and the final state is what the morphed state looks like. Mining plays a central role in state transition, and we will elaborate on the mining process in detail in later sections. The state is stored on the Ethereum network as the *world state*. This is the global state of the Ethereum blockchain.

The state needs to be stored permanently in the blockchain. For this purpose, the world state, the account state, transactions, and transaction receipts are stored.

The world state

This is a **mapping** between Ethereum addresses and account states. The addresses are 20 bytes (160 bits) long. This mapping is a data structure that is serialized using RLP. State root is the hash of the root of the MPT of the world state, i.e., persistent state. Here an Ethereum address (from EOA—[address, balance, nonce]) is used as a key to find the leaf node (lookup for the value), the account state which is composed of four elements [the nonce, balance, StorageRoot, codeHash]. StorageRoot is the root of another trie called the storage trie, where the contract code is stored. We describe the account state next.

The account state

The account state consists of four fields: nonce, balance, StorageRoot, and codeHash, and is described in detail here:

- **Nonce**: This is a scalar value that is incremented every time a transaction is sent from the address. In the case of contract accounts, it represents the number of contracts created by the account.
- **Balance**: This value represents the number of Weis held by the given address.
- **Storage root**: This field represents the root node of the MPT that encodes the storage contents of the account. It is a mapping encoded in the trie, from the Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. (i.e., the hash of the integer keys to the RLP-encoded integer values).
- **CodeHash**: This is an immutable field that contains the hash of the smart contract code that is associated with the account. In the case of normal accounts, this field contains the Keccak 256-bit hash of an empty string. This code is invoked via a message call.

The world state and its relationship with the accounts trie, accounts, and block header are visualized in the following diagram. It shows the **account state**, or data structure, which contains a **storage root** hash derived from the **root node** of the **account storage trie** shown on the left. The account data structure is then used in the **world state trie**, which is a mapping between addresses and account states.

The accounts trie is an MPT used to encode the storage contents of an account. The contents are stored as a mapping between Keccak 256-bit hashes of 256-bit integer keys to the RLP-encoded 256-bit integer values.

Finally, the **root node** of the world state trie is hashed using the Keccak 256-bit algorithm and made part of the **block header** data structure, which is shown on the right-hand side of the diagram as the **state root** hash:

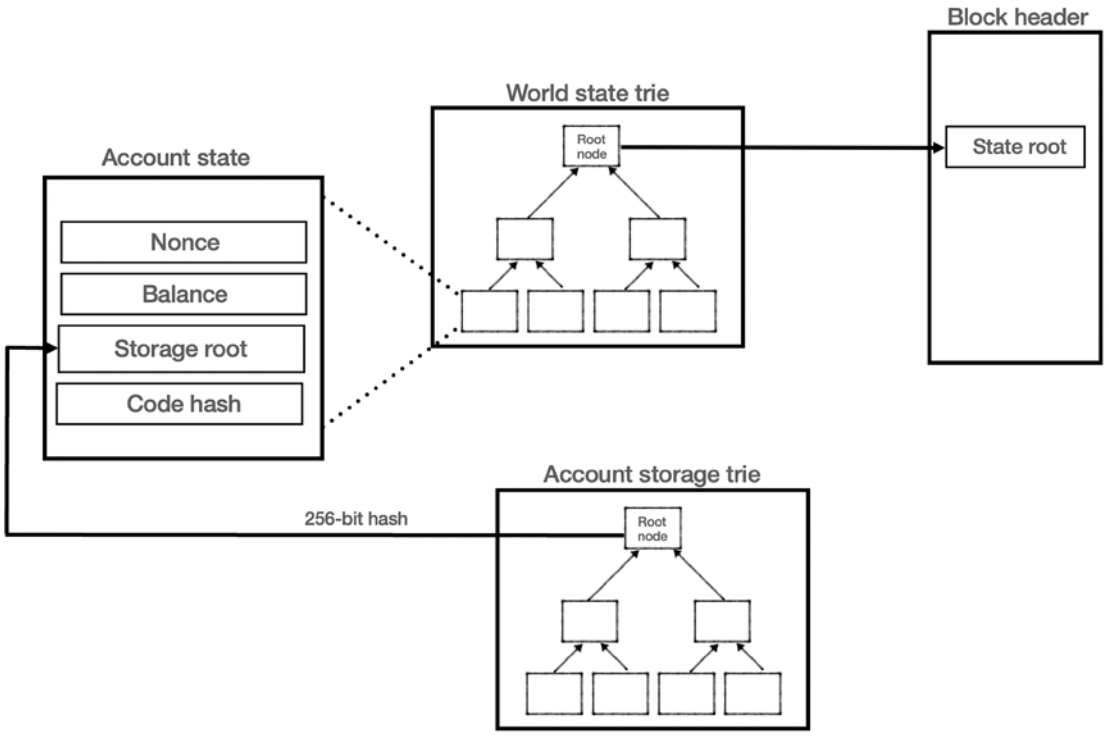


Figure 9.5: The accounts trie (the storage contents of account), account tuple, world state trie, and state root hash and their relationships

Transaction receipts

Transaction receipts are used as a mechanism to store the state after a transaction has been executed. In other words, these structures are used to record the outcome of the transaction execution. They are produced after the execution of each transaction. All receipts are stored in an index-keyed trie. The hash (a 256-bit Keccak hash) of the root of this trie is placed in the block header as the receipt's root. It is composed of five elements, as follows:

- **Type of the transaction**: This item is the type of transaction based on EIP-2718. `0` represents a legacy transaction, `1` represents EIP-2930, and `2` represents EIP-1559.
- **Status code**: Since the Byzantium release, an additional field returning the success (`1`) or failure (`0`) of the transaction is also available.
- **Cumulative gas used**: This item represents the total amount of gas used in the block that contains the transaction receipt. The value is taken immediately after the transaction execution is completed. The total gas used is a non-negative integer.
- **Series of log entries**: This field has a set of log entries created as a result of the transaction execution. Log entries contain the logger's address, a series of log topics, and the log data.

- **Bloom filter**: A bloom filter is created from the information contained in the set of logs discussed earlier. A log entry is reduced to a hash of 256 bytes, which is then embedded in the header of the block as the logs bloom. A log entry is composed of the logger's address, log topics, and log data. Log topics are encoded as a series of 32-byte data structures. The log data is made up of a few bytes of data.



More information about this change is available at `https://github.com/ethereum/EIPs/pull/658`.

This transaction receipt and its relationship with the block header is visualized in the following diagram:

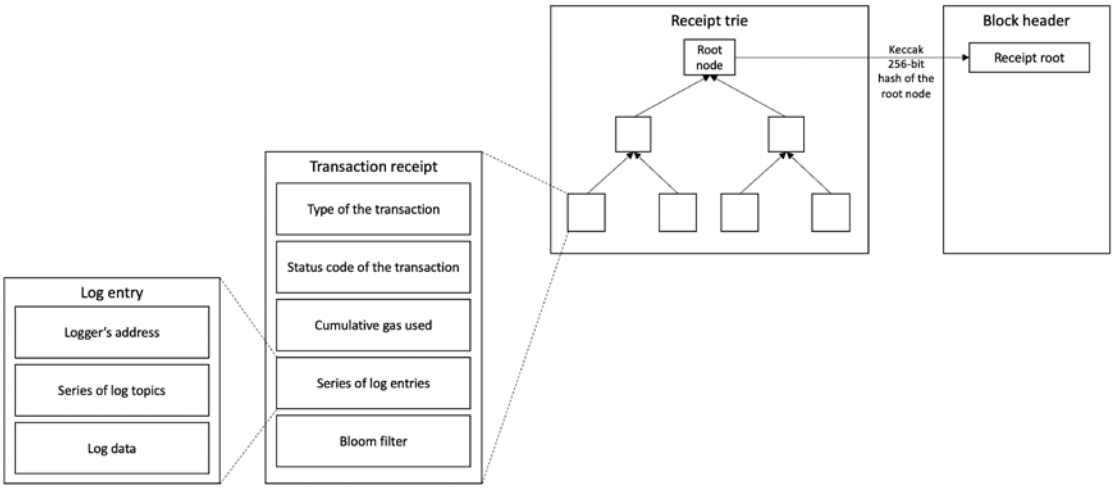


Figure 9.6: Transaction receipts and logs bloom

As a result of the transaction execution process, the state morphs from an initial state to a target state. This state needs to be stored and made available globally in the blockchain.

State is updated as a result of EVM execution, which is the core execution machine for the Ethereum blockchain and we'll discuss it next.

Ethereum virtual machine

The EVM is a simple stack-based execution machine that runs bytecode instructions to transform the system state from one state to another. The word size of the EVM is set to 256 bits. The stack size is limited to 1,024 elements and is based on the **Last In, First Out** (**LIFO**) queue. The EVM is a Turing-complete machine but is limited by the amount of gas that is required to run any instruction. This means that infinite loops that can result in denial-of-service attacks are not possible due to gas requirements.

The EVM also supports exception handling should exceptions occur, such as not having enough gas or providing invalid instructions, in which case the machine would immediately halt and return the error to the executing agent.

The EVM is an entirely isolated and sandboxed runtime environment. The code that runs on the EVM does not have access to any external resources such as a network or filesystem. This results in increased security, deterministic execution, and allows untrusted code (code that can be run by anyone) to be executed on the Ethereum blockchain.

As discussed earlier, the EVM is a stack-based architecture. The EVM is big-endian by design, and it uses 256-bit-wide words. This word size allows for Keccak 256-bit hash and ECC computations.

There are six types of locations available to smart contracts and the EVM for reading and writing information:

- **Memory**: The first location is called memory or volatile memory, which is a word-addressed byte array. When a contract finishes its code execution, the memory is cleared. It is akin to the concept of RAM. `write` operations to the memory can be of 8 or 256 bits, whereas `read` operations are limited to 256-bit words. Memory is unlimited but is constrained by gas fee requirements. It is available only for the duration of the transaction.
- **Storage**: The other location is called storage, which is a key-value store and is permanently persisted on the blockchain. Keys and values are each 256 bits wide. It is allocated to all accounts on the blockchain. As a security measure, storage is only accessible by its own respective CAs. It can be thought of as hard disk storage.
- **Stack**: EVM is a stack-based machine, and thus performs all computations in a data area called the stack. All in-memory values are also stored in the stack. It has a maximum depth of 1,024 elements and supports a word size of 256 bits. EVM opcodes `POP` data from and `PUSH` data to the stack. It is a temporary storage location, and it is empty once the transaction execution completes.
- **Call data**: This is the data field of a transaction. When the transaction is being executed, this acts as read-only memory that contains parameters to the message call.
- **Code**: This location contains the code currently being executed and is also used for static data storage.
- **Logs**: This is the write-only log and event output space.

The storage associated with the EVM is a word-addressable word array that is non-volatile and is maintained as part of the system state. Keys and values are 32 bytes in size and storage. The program code is stored in **virtual read-only memory** (**virtual ROM**), which is accessible using the `CODECOPY` instruction. The `CODECOPY` instruction copies the program code into the main memory. Initially, all storage and memory are set to zero in the EVM. There is another instruction called `EXTCODECOPY`, which can be used to copy program code from another contract to the memory.

The following diagram shows the design of the EVM where the virtual ROM stores the program code that is copied into the main memory using the `CODECOPY` instruction. The main memory is then read by the EVM by referring to the program counter and executes instructions step by step.

The program counter and EVM stack are updated accordingly with each instruction execution:

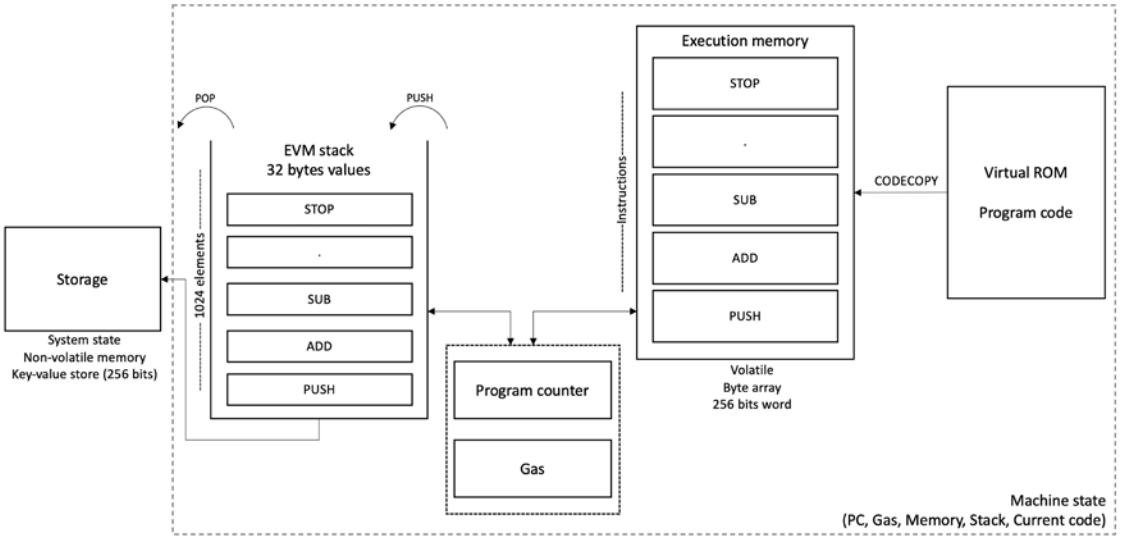


Figure 9.7: EVM operation

The preceding diagram shows an EVM stack on the left side showing that elements are pushed and popped from the stack. It also shows that a program counter is maintained and incremented with instructions being read from the main memory, with indexing the next EVM bytecode instruction to execute. There is also a gas register that keeps the count of available gas. The main memory gets the program code from the virtual ROM/storage via the `CODECOPY` instruction.



The success and wide adoption of Ethereum sparked an interest in developing more EVM chains. There are various EVM chains available in the blockchain ecosystem now. A list of EVM networks with necessary details is available here: `https://chainlist.org`

EVM optimization is an active area of research, and recent research has suggested that the EVM can be optimized and tuned to a very fine degree to achieve high performance. Research and development on **Ethereum WebAssembly** (**ewasm**)—an Ethereum-flavored iteration of WebAssembly—is already underway. **WebAssembly** (**Wasm**) was developed by Google, Mozilla, and Microsoft, and is now being designed as an open standard by the W3C community group. Wasm aims to be able to run machine code in the browser that will result in execution at native speed. More information and the GitHub repository for ewasm is available at `https://github.com/ewasm`.

Another intermediate language called YUL, which can compile to various backends such as the EVM and ewasm, is under development. More information on this language can be found at `https://solidity.readthedocs.io/en/latest/yul.html`.

Execution environment

There are some key elements that are required by the execution environment to execute the code. The key parameters are provided by the execution agent, for example, a transaction. These are listed as follows:

- The system state.
- The remaining gas for execution.
- Accrued substate
- The address of the account that owns the executing code.
- The address of the sender of the transaction. This is the originating address of this execution (it can be different from the sender).
- The gas price of the transaction that initiated the execution.
- Input data or transaction data depending on the type of executing agent. This is a byte array; in the case of a message call, if the execution agent is a transaction, then the transaction data is included as input data.
- The address of the account that initiated the code execution or transaction sender. This is the address of the sender if the code execution is initiated by a transaction; otherwise, it is the address of the account.
- The value of the transaction. This is the amount in Wei. If the execution agent is a transaction, then it is the transaction value.
- The code to be executed, presented as a byte array that the iterator function picks up in each execution cycle.
- The block header of the current block.
- Depth—the number of message calls or contract creation transactions (`CALL`, `CREATE` or `CREATE2`) currently in execution.
- Permission to make modifications to the state.

The execution results in producing the resulting state, the gas remaining after the execution, the self-destruct or suicide set, log series, and any gas refunds.

The machine state

The machine state is maintained internally and updated after each execution cycle of the EVM. An iterator function runs in the EVM, which outputs the results of a single cycle of the state machine. The iterator function performs various vital functions that are used to set the next state of the machine and eventually the world state. These functions include the following:

- It fetches the next instruction from a byte array where the machine code is stored in the execution environment.
- It adds/removes (`PUSH/POP`) items from the stack accordingly.
- Gas is reduced according to the gas cost of the instructions/opcodes. It increments the **Program Counter** (**PC**).

The EVM is also able to halt in normal conditions if `STOP`, `SUICIDE`, or `RETURN` opcodes are encountered during the execution cycle.

Machine state can be viewed as a tuple (g, pc, m, i, s) that consists of the following elements:

- g: Available gas
- pc: The PC, which is a positive integer of up to 256
- m: The contents of the memory (a series of zeroes of size 2^{256})
- i: The active number of words in memory (counting continuously from position 0)
- s: The contents of the stack

The EVM is designed to handle exceptions and will halt (stop execution) if any of the following exceptions occur:

- Not enough gas remaining for execution
- Invalid instructions
- Insufficient stack items
- Invalid destination of jump opcodes
- Invalid stack size (greater than 1,024)

With this, we complete our discussion on the EVM.

Blockchain is composed of blocks and blocks contain transactions. Next, we'll discuss what a block is, its structure, and how blocks are validated in the Ethereum blockchain.

Blocks and blockchain

Blocks are the main building structure of a blockchain. Ethereum blocks consist of various elements, which are described as follows:

- The list of headers of ommers or uncles
- The block header
- The transactions list

An uncle block is a block that is the child of a parent but does not have a child block. Ommers or uncles are valid but stale blocks that are not part of the main chain but contribute to the security of the chain. They also earn a reward for their participation but do not become part of the canonical truth.

Block header: Block headers are the most critical and detailed components of an Ethereum block. The header contains various elements, which are described in detail here:

- **Parent hash**: This is the Keccak 256-bit hash of the parent (previous) block's header.
- **Ommers hash**: This is the Keccak 256-bit hash of the list of ommers (or uncle) blocks included in the block.
- **The beneficiary**: The beneficiary field contains the 160-bit address of the recipient that will receive the mining reward once the block has been successfully mined.
- **State root**: The state root field contains the Keccak 256-bit hash of the root node of the state trie. It is calculated once all transactions have been processed and finalized.
- **Transaction root**: The transaction root is the Keccak 256-bit hash of the root node of the transaction trie. The transaction trie represents the list of transactions included in the block.
- **Receipts root**: The receipts root is the Keccak 256-bit hash of the root node of the transaction receipt trie. This trie is composed of receipts of all transactions included in the block. Transaction receipts are generated after each transaction is processed and contain useful post-transaction information. More details on transaction receipts are provided in the next section.
- **Logs bloom**: The logs bloom is a bloom filter that is composed of the logger address and log topics from the log entry of each transaction receipt of the included transaction list in the block. Logging is explained in detail in the next section.
- **Difficulty**: The difficulty level of the current block.
- **Number**: The total number of all previous blocks; the genesis block is block zero. i.e., the block number.
- **Gas limit**: This field contains the value that represents the limit set on the gas consumption per block.
- **Gas used**: This field contains the total gas consumed by the transactions included in the block.
- **Timestamp**: The timestamp is the epoch Unix time of the time of block initialization.
- **Extra data**: The extra data field can be used to store arbitrary data related to the block. Only up to 32 bytes are allowed in this field.
- **Mixhash**: The mixhash field contains a 256-bit hash that, once combined with the nonce, is used to prove that adequate computational effort (**Proof of Work**, or **PoW**) has been spent in order to create this block.
- **Nonce**: The nonce is a 64-bit hash (a number) that is used to prove, in combination with the *mixhash* field, that adequate computational effort (PoW) has been spent in order to create this block.
- **BaseFeePerGas**: A new field introduced after the London upgrade—EIP-1559—to record the protocol calculated fee required for a transaction to be included in the block.

The following diagram shows the detailed structure of the block and block header:

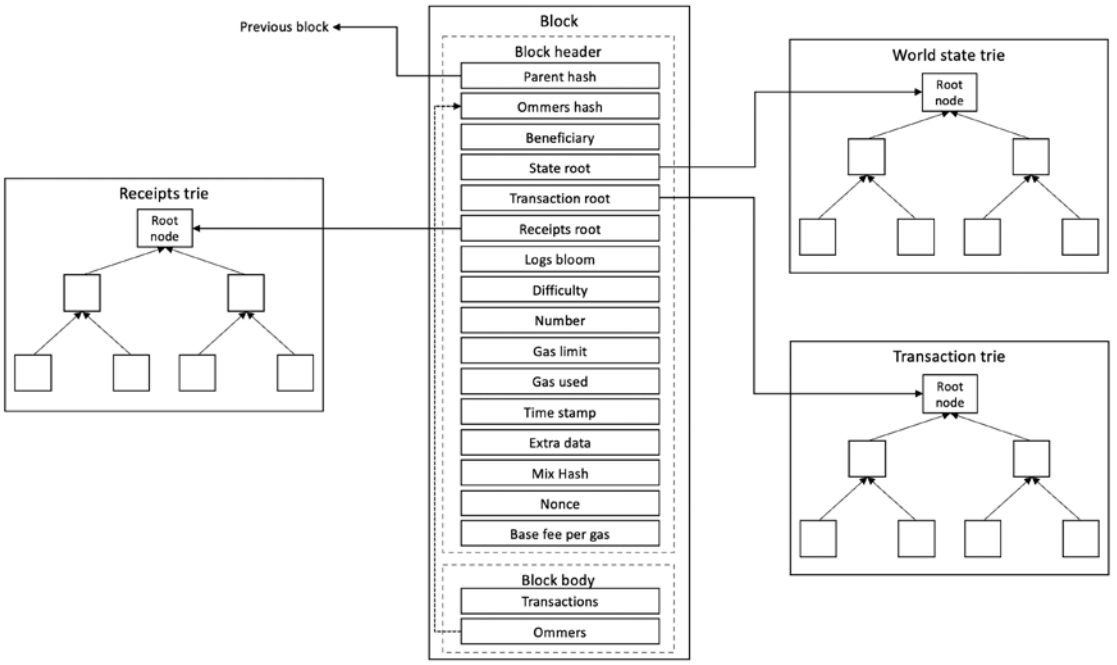


Figure 9.8: A detailed diagram of the block structure with a block header and relationship with tries

The transaction list is simply a list of all transactions included in the block. Also, the list of headers of uncles is included in the block.

The genesis block

The genesis block is the first block in a blockchain network. It varies slightly from normal blocks due to the data it contains and the way it has been created. The key difference is that it is a hardcoded block in the Ethereum software client, whereas the rest of the blocks after it are mined. It contains 15 items that can be viewed on the Etherscan block explorer: `https://etherscan.io/block/0`.

Block validation, finalization, and processing

An Ethereum block is considered valid if it passes the following checks:

- If it is consistent with uncles and transactions. This means that all ommers satisfy the property that they are indeed uncles. Also if the PoW for uncles is valid.
- If the previous block (parent) exists and is valid.
- If the timestamp of the block is valid. This means that the current block's timestamp must be higher than the parent block's timestamp. Also, it should be less than 15 minutes into the future. All block times are calculated in epoch time (Unix time).

- If any of these checks fails, the block will be rejected. A list of errors for which the block can be rejected is presented here:
    - The timestamp is older than the parent
    - There are too many uncles – more than two
    - There is a duplicate uncle
    - The uncle is an ancestor
    - The uncle's parent is not an ancestor
    - There is non-positive difficulty
    - There is an invalid mix digest
    - There is an invalid PoW

Block finalization is a process that is run by miners to validate the contents of the block and apply rewards. It results in four steps being executed:

1. **Ommers validation**: In the case of mining, determine ommers. The validation process of the headers of stale blocks checks whether the header is valid and whether the relationship between the uncle and the current block satisfies the maximum depth of six blocks. A block can contain a maximum of two uncles.
2. **Transaction validation**: In the case of mining, determine transactions. This process involves checking whether the total gas used in the block is equal to the final gas consumption after the final transaction, in other words, the cumulative gas used by the transactions included in the block.
3. **Reward application**: Apply rewards, which means updating the beneficiary's account with a reward balance. In Ethereum, a reward is also given to miners for stale blocks, which is 1/32 of the block reward. Uncles that are included in the blocks also receive 7/8 of the total block reward. The current block reward is 2 ethers. It was reduced first from 5 ether to 3 with the Byzantium release of Ethereum. Later, in the Constantinople release (`https://blog.ethereum.org/2019/02/22/ethereum-constantinople-st-petersburg-upgrade-announcement/`), it was reduced further to 2 ether. A block can have a maximum of two uncles.
4. **State and nonce validation**: Verify the state and block nonce. In the case of mining, compute a valid state and block nonce.

After the high-level view of the block validation mechanism, we'll now look into how a block is received and processed by a node. We will also see how it is updated in the local blockchain:

1. When an Ethereum full node receives a newly mined block, the header and the body of the block are detached from each other. Now, remember we introduced that there are three MPTs in an Ethereum blockchain whose roots are present in each block header as a state trie root node, a transaction trie root node, and a receipt trie root node. The account contains another root for the storage trie where the contract data is stored, called StorageRoot. We will now learn how these tries are used to validate the blocks.
2. A new MPT is constructed that comprises all transactions from the block.

3. All transactions from this new MPT are executed one by one in a sequence. This execution occurs locally on the node within the **EVM**. As a result of this execution, new transaction receipts are generated that are organized in a new receipts MPT. Also, the global state is modified accordingly, which updates the state MPT.
4. The root nodes of each respective trie, in other words, the state root, transaction root, and receipts root are compared with the header of the block that was split in the first step. If both the roots of the newly constructed tries and the trie roots that already exist in the header are equal, then the block is verified and valid.
5. Once the block is validated, new transaction, receipt, and state tries are written into the local blockchain database.

Block difficulty mechanism

Block difficulty is increased if the time between two blocks decreases, whereas it decreases if the block time between two blocks increases. This is required to maintain a roughly consistent block generation time. The difficulty adjustment algorithm in Ethereum's **Homestead** release is as follows:

$$block_diff = parent_diff + parent_diff // 2048 *$$

$$max(1 - (block_timestamp - parent_timestamp) // 10, -99) + int(2^{**}((block.number // 100000) - 2))$$



Note that // is the integer division operator.

The preceding algorithm means that, if the time difference between the generation of the parent block and the current block is less than 10 seconds, the difficulty goes up by *parent_diff // 2048 * 1*. If the time difference is between 10 and 19 seconds, the difficulty level remains the same. Finally, if the time difference is 20 seconds or more, the difficulty level decreases. This decrease is proportional to the time difference from *parent_diff // 2048 * -1* to a maximum decrease of *parent_diff // 2048 * -99*.

In the Byzantium release, the difficulty adjustment formula has been changed to take uncles into account for the difficulty calculation. This new formula is shown here:

$$adj_factor = max((2 \text{ if } len(parent.uncles) \text{ else } 1) - ((timestamp - parent.timestamp) // 9), -99)$$



Soon after the **Istanbul** upgrade, the difficulty bomb was delayed once again, under the **Muir Glacier** network upgrade with a hard fork, `https://eips.ethereum.org/EIPS/eip-2384`, for roughly another 611 days. The change was activated at block number 9,200,000 on January 2, 2020. It has been delayed yet again under Arrow Glacier: `https://eips.ethereum.org/EIPS/eip-4345`. We will consider the difficulty time bomb in more detail in *Chapter 10*, *Ethereum in Practice*.

Nodes and miners

The Ethereum network contains different nodes. Some nodes act only as wallets, some are light clients, and a few are full clients running the full blockchain. One of the most important types of nodes is mining nodes. We will see what constitutes mining in this section.

Transactions can be found in either transaction pools or blocks. In transaction pools, they wait for verification by a node, and in blocks, they are added after successful verification. When a mining node starts its operation of verifying blocks, it starts with the highest-paying transactions in the transaction pool and executes them one by one. When the gas limit is reached, or no more transactions are left to be processed in the transaction pool, the mining starts. As a result of the mining operation, currency (ether) is awarded to the nodes that perform mining operations as an incentive for them to validate and verify blocks made up of transactions. The mining process helps secure the network by verifying computations.

In this process, the block header is repeatedly hashed until a valid nonce is found, such that once hashed with the block header, it results in a value less than the difficulty target. At a theoretical level, a miner node performs the following functions:

- It listens for the transactions broadcast on the Ethereum network and determines the transactions to be processed.
- It determines stale ommer blocks and includes them in the blockchain.
- It updates the account balance with the reward earned from successfully mining the block.
- Finally, a valid state is computed, and the block is finalized, which defines the result of all state transitions.

Once a block is successfully mined, it is broadcast immediately to the network, claiming success, and will be verified and accepted by the network.

Miners play a vital role in reaching a consensus regarding the canonical state of the blockchain. The consensus mechanism that they contribute to is explained in the next section.

The consensus mechanism

The original method of mining in Ethereum was based on PoW, which is similar to that of Bitcoin. When a block is deemed valid, it must satisfy not only the general consistency requirements, but it must also contain the PoW for a given difficulty. Ethereum's PoW algorithm is called Ethash. The consensus mechanism in Ethereum is based on the **Greedy Heaviest Observed Subtree** (**GHOST**) protocol proposed initially by Zohar and Sompolinsky in December 2013.



Readers interested in this topic can find more information in the paper at `http://eprint.iacr.org/2013/881.pdf`.

Ethereum uses a simpler version of this protocol, where the chain that has the most computational effort spent on it to build it is identified as the definite version. Another way of looking at it is to find the longest chain, as the longest chain must have been built by consuming adequate mining efforts. The GHOST protocol was first introduced as a mechanism to alleviate the issues arising out of fast block generation times that led to the creation of stale or orphaned blocks. In GHOST, stale blocks, orphan blocks, uncles, or ommers, are added in calculations to figure out the chain of blocks to be selected as the main chain, hence the term heaviest chain. This also means that this is the chain that has the most computational work done behind it. The following diagram shows the difference between the longest and heaviest chains:

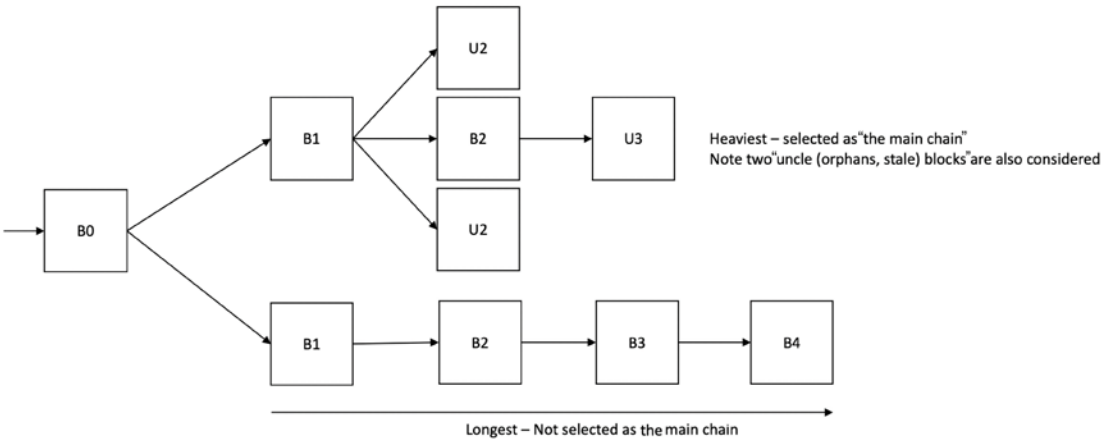


Figure 9.9: The longest versus the heaviest chain

The preceding diagram shows two rules for figuring out which blockchain is the canonical version of the truth. In the case of Bitcoin, shown on the left-hand side in the diagram, the longest chain rule is applied, which means that the active chain (true chain) is the one that has the most amount of PoW done. In the case of Ethereum, the concept is similar from the point of view of the longest chain, but it also includes ommers, the *orphaned* blocks, which means that it also rewards those blocks that were competing with other blocks during mining to be selected and performed significant PoW or were mined exactly at the same time as others but did not make it to the main chain. This makes the chain the *heaviest* instead of the *longest* because it also contains the *orphaned* blocks. This is shown on the right-hand side of the diagram.

Ethash was originally intended to be ASIC-resistant, where finding a nonce requires large amounts of memory. However, now some ASICs are available for *Ethash* too.

An algorithm named **Casper** has been developed that will replace the existing PoW algorithm in Ethereum. This is a security deposit based on the economic protocol where nodes are required to place a security deposit before they can produce blocks. Nodes have been named *bonded validators* in Casper, whereas the act of placing the security deposit is named *bonding*.



More information about Casper can be found here: `https://github.com/ethereum/research/tree/master/casper4`.

After "the merge", the consensus protocol is replaced with proof-of-stake mechanics, which we will discuss in detail in *Chapter 13*, *The Merge and Beyond*.

As the blockchain progresses (more blocks are added to the blockchain) governed by the consensus mechanism, on occasion, the blockchain can split into two. This phenomenon is called **forking**.

Forks in the blockchain

A fork occurs when a blockchain splits into two. This can be intentional or unintentional. Usually, as a result of a major protocol upgrade, a hard fork is created, while an unintentional fork can be created due to bugs in the software.

It can also be temporary, as discussed previously. In other words, the longest and heaviest chain. This temporary fork occurs when a block is created almost at the same time and the chain splits into two, until it finds the longest or heaviest chain to achieve eventual consistency.

The release of **Homestead** involved major protocol upgrades, which resulted in a hard fork. The protocol was upgraded at block number 1,150,000, resulting in the migration from the first version of Ethereum, known as **Frontier**, to the second version. Another version is called **Byzantium**. This was released as a hard fork at block number 4,370,000. The latest upgrade is the **London** hard fork under EIP-1559, activated at block number 12,965,000.

An unintentional fork, which occurred on November 24, 2016, at 14:12:07 UTC, was due to a bug in Ethereum's Geth client journaling mechanism. As a result, a network fork occurred at block number 2,686,351. This bug resulted in Geth failing to prevent empty account deletions in the case of the empty OOG exception. This was not an issue in **Parity** (another popular Ethereum client). This means that from block number 2,686,351, the Ethereum blockchain is split into two, one running with the Parity clients and the other with Geth. This issue was resolved with the release of Geth version 1.5.3. As a result of the DAO hack, the Ethereum blockchain was forked to recover from the attack.

Fundamentally, the Ethereum blockchain is a database that exists on every node in the Ethereum network. Let's now explore what the Ethereum network is.

The Ethereum network

The Ethereum network is a peer-to-peer network where nodes participate in order to maintain the blockchain and contribute to the consensus mechanism. Networks can be divided into three types, based on the requirements and usage: the main net, test nets, and private nets.

Main net

The **main net** is the current live network of Ethereum. Its network ID is 1 and its chainID is also 1. The network and chainIDs are used to identify the network. A block explorer that shows detailed information about blocks and other relevant metrics is available at `https://etherscan.io`. This can be used to explore the Ethereum blockchain.

Test nets

There are two main networks available for Ethereum testing. The aim of these test blockchains is to provide a testing environment for smart contracts and DApps before being deployed to the production live blockchain. Moreover, being test networks, they also allow experimentation and research. The main test net is called *Sepolia*, which is a PoW test net and contains all the features of the Ethereum main net. *Gorli* is another test net that is based on proof of authority.

Private nets

As the name suggests, these are private networks that can be created by generating a new genesis block. This is usually the case in private blockchain networks, where a private group of entities start their blockchain network and use it as a permissioned or consortium blockchain.



Network IDs and chainIDs are used by Ethereum clients to identify the network. ChainIDs were introduced in EIP-155" as part of the replay protection mechanism. EIP155 is detailed at `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md`. In EIP-155, ChainID was encoded with the signature V value. Currently, with EIP-1559, ChainID is a separate field in the transaction payload, identifying the blockchain network. A comprehensive list of Ethereum networks is maintained and available at `https://chainlist.org`

A private net allows the creation of an entirely new blockchain, usually on a local network. This is different from the testnet or main net in the sense that it uses its own genesis block and network ID.

On the main net, the Geth Ethereum client can discover **boot nodes** by default as they are hardcoded in the Geth client, and connects automatically. But on a private network, Geth needs to be configured by specifying appropriate flags and configurations in order for it to discover, or be discovered by, other peers. We will see how this is practically achieved in *Chapter 10, Ethereum in Practice*.

Let's get some theoretical aspects covered that are related to discovery and understand what actually happens when we disable node discovery. For this, first, we'll see how a node normally discovers other nodes on an Ethereum network and what protocols are involved.

The Ethereum network consists of four elements or layers: **Discovery**, **RLPx**, **DEVP2P**, and **application-level sub-protocols**:

- The **Discovery** protocol is responsible for discovering other nodes on the network by using a node discovery mechanism based on the Kademlia protocol's routing algorithm. This protocol works by using UDP. UDP is a connectionless and faster communication protocol as compared to TCP, which makes it suitable for time-sensitive applications and DNS lookups.

In Ethereum, there are two discovery protocols, named DiscV4 and DiscV5. DiscV4 is currently production-ready and implemented in Ethereum, whereas DiscV5 is in development.



The specification for DiscV4 can be found here: `https://github.com/ethereum/devp2p/blob/master/discv4.md`

The specification for DiscV5 can be found here: `https://github.com/ethereum/devp2p/blob/master/discv5/discv5.md`

For discovery, a new Ethereum node joining the network makes use of hardcoded bootstrap nodes, which provide an initial entry point into the network, from which further discovery processes then start. This list of bootstrap nodes can be found in the `bootnodes.go` file:

`https://github.com/ethereum/go-ethereum/blob/2b0d0ce8b0a02634b90b02bc038523eacd2b220a/params/bootnodes.go#L23`

- **RLPx** is a TCP-based transport protocol responsible for enabling secure communication between Ethereum nodes. It achieves this by using an asymmetric encryption mechanism called **Elliptic Curve Integrated Encryption Scheme** (**ECIES**) for handshaking and key exchange.



Note that RLPx is named after *RLP*, the serialization protocol introduced earlier. However, it is not related to the serialization protocol RLP, and the name is not an acronym.

Handshaking is a term used in computing to refer to a mechanism of exchanging initial information (signals, data, or messages) between different devices on a network to establish a connection for communication as per the protocol in use.

More information on ECIES and the RLPx specification can be found here: `https://github.com/ethereum/devp2p/blob/master/rlpx.md`.

- **DEVP2P** (also called the wire protocol) is responsible for negotiating an application session between two nodes that have been discovered and have established a secure channel using RLPx. This is where the `HELLO` message is sent between nodes to provide each other with details of the version of the DEVP2P protocol: the client name, supported application sub-protocols, and the port numbers the nodes are listening on. `PING` and `PONG` messages are used to check the availability of the nodes and `DISCONNECT` is sent if a response from a node is not received.
- Finally, the fourth element of the Ethereum network stack is where different **Ethereum sub-protocols** exist. After discovering and establishing a secure transport channel and negotiating an application session, the nodes exchange messages using so-called "capability protocols" or application sub-protocols. This includes **Eth** (versions 62, 63, and 64), **Light Ethereum Sub-protocol** (**LES**), **Whisper**, and **Swarm**. These capability protocols are responsible for different application-level communications: for example, Eth is responsible for block synchronization. It makes use of several protocol messages such as `Status`, `Transactions`, `GetBlockHeaders`, and `GetBlockBodies` messages to exchange blockchain information between nodes.

Note that Eth is also referred to as the "Ethereum wire protocol."



More detail on the Eth capability protocol can be found here: `https://github.com/ethereum/devp2p/blob/master/caps/eth.md`.

All four of these elements are visualized in the following diagram:

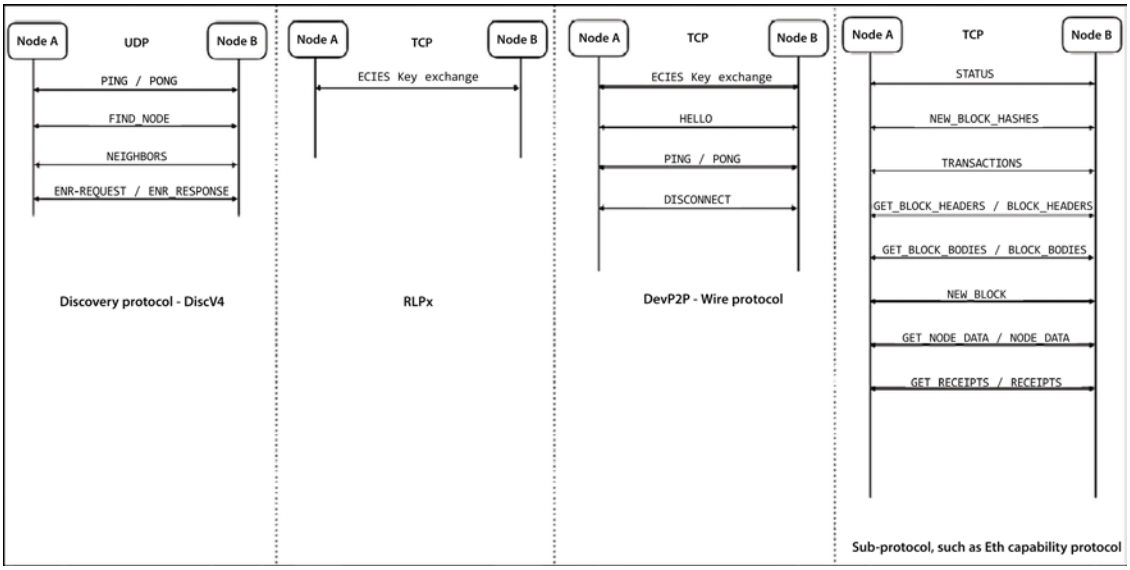


Figure 9.10: Ethereum node discovery, RLPx, DEVP2P, and capability protocols

When setting up private networks, it is advisable to disable the peer discovery mechanism, so that we can manually configure nodes that we need to connect to in our private network. We can then add nodes manually to the network by maintaining a static list of peers.

Now, after this introduction to some underlying theory, all the parameters required for creating a private network will be discussed in detail. In order to create a private net, three components are needed:

- Network ID
- The genesis file
- Data directory to store blockchain data

Let's consider each component in detail.

The **network ID** can be any positive number except any number that is already in use by another Ethereum network. For example, `1` and `3` are in use by Ethereum main net and test net (Ropsten), respectively.



A list of Ethereum networks is maintained at `https://chainid.network` and network IDs chosen already should not be used to avoid conflicts.

The **genesis file** contains the necessary fields required for a custom genesis block. This is the first block in the network and does not point to any previous block. The Ethereum protocol performs checking in order to ensure that no other node on the internet can participate in the consensus mechanism unless they have the same genesis block. The ChainID is usually used for identification of the network.

The genesis file has a number of parameters. Let's see what each of these parameters means:

- `nonce`: This is a 64-bit hash used to prove that PoW has been sufficiently completed. This works in combination with the `mixhash` parameter.
- `timestamp`: This is the Unix timestamp of the block. This is used to verify the sequence of the blocks and for difficulty adjustment. For example, if blocks are being generated too quickly, that difficulty increases.
- `parentHash`: Being the genesis (first) block, this is always zero as there is no parent block.
- `extraData`: This parameter allows a 32-bit arbitrary value to be saved with the block.
- `gasLimit`: This is the limit on the expenditure of gas per block.
- `difficulty`: This parameter is used to determine the mining target. It represents the difficulty level of the hash required to prove the PoW.
- `mixhash`: This is a 256-bit hash that works in combination with `nonce` to prove that a sufficient amount of computational resources has been spent in order to complete the PoW requirements.
- `coinbase`: This is the 160-bit address where the mining reward is sent as a result of successful mining.
- `alloc`: This parameter contains the list of pre-allocated wallets. The long hex digit is the account to which the balance is allocated.
- `config`: This section contains various configuration information defining the chainID and blockchain hard fork block numbers. This parameter is not required to be used in private networks.

The third component, the **data directory**, does not strictly need to be mentioned. However, if there is more than one blockchain already active on the system, then the data directory should be specified so that a separate directory is used for the new blockchain.

This is the directory where the blockchain data for the private Ethereum network will be saved. A number of parameters are specified in order to run the Ethereum node, fine-tune the configuration, and launch the private network.

If connectivity to only specific nodes is required, which is usually the case in private networks, then a list of **static nodes** is provided as a JSON file. This configuration file is read by the Geth client at the time of starting up and the Geth client only connects to the peers that are present in the configuration file.

We will discuss how to connect to a test net and set up private nets in *Chapter 10*, *Ethereum in Practice*.

Precompiled smart contracts

We discussed smart contracts at length in *Chapter 8*, *Smart Contracts*. It is sufficient to say here that Ethereum supports the development of smart contracts that run on the EVM. There are also various contracts that are available in precompiled format in the Ethereum blockchain to support different functions. These contracts, known as **precompiled contracts** or **native contracts**, are described in the following subsection.

These are not strictly smart contracts in the sense of user-programmed Solidity smart contracts but are in fact functions that are available natively to support various computationally intensive tasks. They run on the local node and are coded within the Ethereum client; for example, Parity or Geth.

There are nine **precompiled contracts** or **native contracts** in the Ethereum Istanbul release. The following subsections outline these contracts and their details.

- The elliptic curve public key recovery function: `ECDSARECOVER` (the ECDSA recovery function) is available at address `0x1`. It is denoted as `ECREC` and requires 3,000 gas units in fees for execution. If the signature is invalid, then no output is returned by this function. Public key recovery is a standard mechanism by which the public key can be derived from the private key in ECC. The ECDSA recovery function is shown as follows:

  $$ECDSARECOVER\ (H, V, R, S) = Public\ Key$$

  It takes four inputs: *H*, which is a 32-byte hash of the message to be signed, and *V*, *R*, and *S*, which represent the ECDSA signature with the recovery ID and produce a 64-byte public key. *V*, *R*, and *S* have been discussed in detail previously in this chapter.

- The SHA-256-bit hash function: The SHA-256-bit hash function is a precompiled contract that is available at address `0x2` and produces a `SHA256` hash of the input. The gas requirement for SHA-256 (`SHA256`) depends on the input data size. The output is a 32-byte value.
- The RIPEMD-160-bit hash function: The RIPEMD-160-bit hash function is used to provide a RIPEMD-160-bit hash and is available at address `0x3`. The output of this function is a 20-byte value. The gas requirement is dependent on the amount of input data.
- The identity/datacopy function: The identity function is available at address `0x4` and is denoted by the `ID`. It simply defines output as input; in other words, whatever input is given to the `ID` function, it will output the same value. The gas requirement is calculated by a simple formula: *15 + 3 [Id/32]*, where *Id* is the input data. This means that at a high level, the gas requirement is dependent on the size of the input data, albeit with some calculation performed, as shown in the preceding equation.
- The big mod exponentiation function: This function implements a native big integer exponential modular operation. This functionality allows for RSA signature verification and other cryptographic operations. This is available at address `0x05`.

- The BN256 Elliptic curve point addition function: We discussed elliptic curve addition in detail at a theoretical level in *Chapter 4*, *Asymmetric Cryptography*. This is the implementation of the same elliptic curve point addition function. This contract is available at address `0x06`.
- BN256 Elliptic curve scalar multiplication: We discussed elliptic curve multiplication (point doubling) in detail at a theoretical level in *Chapter 4*, *Asymmetric Cryptography*. This is the implementation of the same elliptic curve point multiplication function. Both elliptic curve addition and doubling functions allow for zk-SNARKs and the implementation of other cryptographic constructs. This contract is available at `0x07`.
- BN256 Elliptic curve pairing: The elliptic curve pairing functionality allows for performing elliptic curve pairing (bilinear maps) operations, which enables zk-SNARK verification. This contract is available at address `0x08`.
- Blake2 compression function 'F': This precompiled contract allows the `BLAKE2b` hash function and other related variants to run on the EVM. This improves interoperability with Zcash and other Equihash-based PoW chains. This precompiled contract is implemented at address `0x09`. The EIP is available at `https://eips.ethereum.org/EIPS/eip-152`.



More information on the Blake hash function is available at `https://blake2.net`

All the precompiled contracts can potentially become native extensions and might be included in the EVM opcodes in the future. There are more precompiled contracts proposed in EIP-2537 for BLS12-381 curve operations, however, the EIP is not implemented yet.



All pre-compiled contracts are present in the `contracts.go` file in the Ethereum source code. It is available at the following link: `https://github.com/ethereum/go-ethereum/blob/master/core/vm/contracts.go`

Information on EIP-2537 is available here: `https://eips.ethereum.org/EIPS/eip-2537`

Programming languages

Code for smart contracts in Ethereum is written in high-level languages such as Serpent, **Low-level Lisp-like Language** (**LLL**), Solidity, or Vyper, and is converted into the bytecode that the EVM understands for it to be executed.

Solidity

Solidity is one of the high-level languages that has been developed for Ethereum. It uses JavaScript-like syntax to write code for smart contracts. Once the code is written, it is compiled into bytecode that's understandable by the EVM using the Solidity compiler called **solc**.



The official Solidity documentation is available at `http://solidity.readthedocs.io/en/latest/`

For example, a simple program in Solidity is shown as follows:

```
pragma solidity ^0.8.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

This program is converted into bytecode, as shown in the following subsection. Details on how to compile Solidity code with examples will be provided in *Chapter 11*, *Tools, Languages, and Frameworks for Ethereum Developers*.

Runtime bytecode

The runtime bytecode is what executes on the EVM. The smart contract code (`contract Test1`) from the previous section is translated into binary runtime and opcodes, as follows:

Binary of the runtime (raw hex codes):

```
608060405234801560f57600080fd5b506004361060285760003560e01c80634d3e46e414
602d575b600080fd5b60336049565b604051808281526020019150506040518091039035b
60006002600054019050906fea2646970667358221204b6ab44aed24787c35f5a1942f8d
3005e7a35314d03d62ced225c1ee1353653864736f6c634300060b0033
```

Opcodes

There are many different **opcodes** that have been introduced in the EVM. The preceding runtime bytecode is translated into the following opcodes:

```
PUSH1 0x80 PUSH1 0x40 MSTORE PUSH1 0x2 PUSH1 0x0 SSTORE CALLVALUE DUP1 ISZERO
PUSH1 0x14 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1 0x8C DUP1 PUSH2
0x23 PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN INVALID PUSH1 0x80 PUSH1 0x40 MSTORE
CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP PUSH1
0x4 CALLDATASIZE LT PUSH1 0x28 JUMPI PUSH1 0x0 CALLDATALOAD PUSH1 0xE0 SHR DUP1
PUSH4 0x4D3E46E4 EQ PUSH1 0x2D JUMPI JUMPDEST PUSH1 0x0 DUP1 REVERT JUMPDEST
PUSH1 0x33 PUSH1 0x49 JUMP JUMPDEST PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE
PUSH1 0x20 ADD SWAP2 POP POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 RETURN
```

```
UMPDEST PUSH1 0x0 PUSH1 0x2 PUSH1 0x0 SLOAD ADD SWAP1 POP SWAP1 JUMP INVALID
LOG2 PUSH5 0x6970667358 0x22 SLT KECCAK256 0x4B PUSH11 0xB44AED24787C35F5A1942F
DUP14 ADDRESS SDIV 0xE7 LOG3 MSTORE8 EQ 0xD0 RETURNDATASIZE PUSH3 0xCED225 0xC1
0xEE SGT MSTORE8 PUSH6 0x3864736F6C63 NUMBER STOP MOD SIGNEXTEND STOP CALLER
```

Opcodes are divided into multiple categories based on the operation they perform. A list of opcodes, their meanings, and usage is available here: `https://ethereum.org/en/developers/docs/evm/opcodes/`.

Now let's move onto another topic, clients and wallets, which are responsible for mining, payments, and management functions, such as account creation on an Ethereum network.

Wallets and client software

As Ethereum is undergoing heavy development and evolution, there are many components, clients, and tools that have been developed and introduced over the last few years.

Wallets

A wallet is a generic program that can store private keys and, based on the addresses stored within it, it can compute the existing balance of ether associated with the addresses by querying the blockchain. It can also be used to deploy smart contracts. In *Chapter 10, Ethereum in Practice*, we will introduce the **MetaMask** wallet, which has become the tool of choice for developers.

Having discussed the role of wallets within Ethereum, let's now discuss some common clients.

Geth

This is the official **Go** implementation of the Ethereum client.

The latest version is available at the following link: `https://geth.ethereum.org/downloads/`.

There are other implementations, including C++ implementation Eth, and many others. A list is available here: `https://ethereum.org/en/developers/docs/nodes-and-clients/#execution-clients`.

Light clients

Simple Payment Verification (**SPV**) clients download only a small subset of the blockchain. This allows low-resource devices, such as mobile phones, embedded devices, or tablets, to be able to verify transactions.

A complete Ethereum blockchain and node are not required in this case, and SPV clients can still validate the execution of transactions. SPV clients are also called light clients. This idea is similar to Bitcoin SPV clients.

The critical difference between clients and wallets is that clients are full implementations of the Ethereum protocol that support mining, account management, and wallet functions. In contrast, wallets only store the public and private keys, provide essential account management, and interact with the blockchain usually only for payment (transfer of funds) purposes.

In the next section, we will introduce several supporting protocols that are part of the complete decentralized ecosystem based on the Ethereum blockchain.

Supporting protocols

Various supporting protocols are available to assist with the complete decentralized ecosystem. In addition to the contracts layer, which is the core blockchain layer, there are additional layers that need to be decentralized to achieve a completely decentralized ecosystem. This includes decentralized messaging and decentralized storage. These are addressed by the Whisper and Swarm protocols, respectively.

Whisper

Whisper provides decentralized peer-to-peer messaging capabilities to the Ethereum network. In essence, Whisper is a communication protocol that DApps use to communicate with each other. The data and routing of messages are encrypted within Whisper communications. Whisper makes use of the **DEVP2P** wire protocol for exchanging messages between nodes on the network. Moreover, it is designed to be used for smaller data transfers and in scenarios where real-time communication is not required.

Whisper is also designed to provide a communication layer that cannot be traced and provides *dark communication* between parties. Blockchain can be used for communication, but that is expensive, and a consensus is not really required for messages exchanged between nodes. Therefore, Whisper can be used as a protocol that allows censor-resistant communication. Whisper messages are ephemeral and have an associated **Time to Live** (**TTL**). Whisper is already available with Geth and can be enabled using the `–shh` option while running the Geth Ethereum client.



The official Whisper documentation is available at `https://eth.wiki/concepts/whisper/whisper`.

Swarm

Swarm has been developed as a distributed file storage platform. It is a decentralized, distributed, and peer-to-peer storage network. Files in this network are addressed by the hash of their content. This contrasts with traditional centralized services, where storage is available at a central location only. It has been developed as a native base layer service for the Ethereum Web3 stack. Swarm is integrated with DEVP2P, which is the multiprotocol network layer of Ethereum. Swarm is envisaged to provide a **Distributed Denial-of-Service** (**DDOS**)-resistant and fault-tolerant distributed storage layer for Ethereum Web 3.0. Similar to `shh` in Whisper, Swarm has a protocol called `bzz`, which is used by each Swarm node to perform various Swarm protocol operations.



The official Swarm documentation is available at: `https//swarm-guide.readthedocs.io/en/latest/`

The following diagram gives a high-level overview of how Swarm and Whisper fit together and work with the Ethereum blockchain:

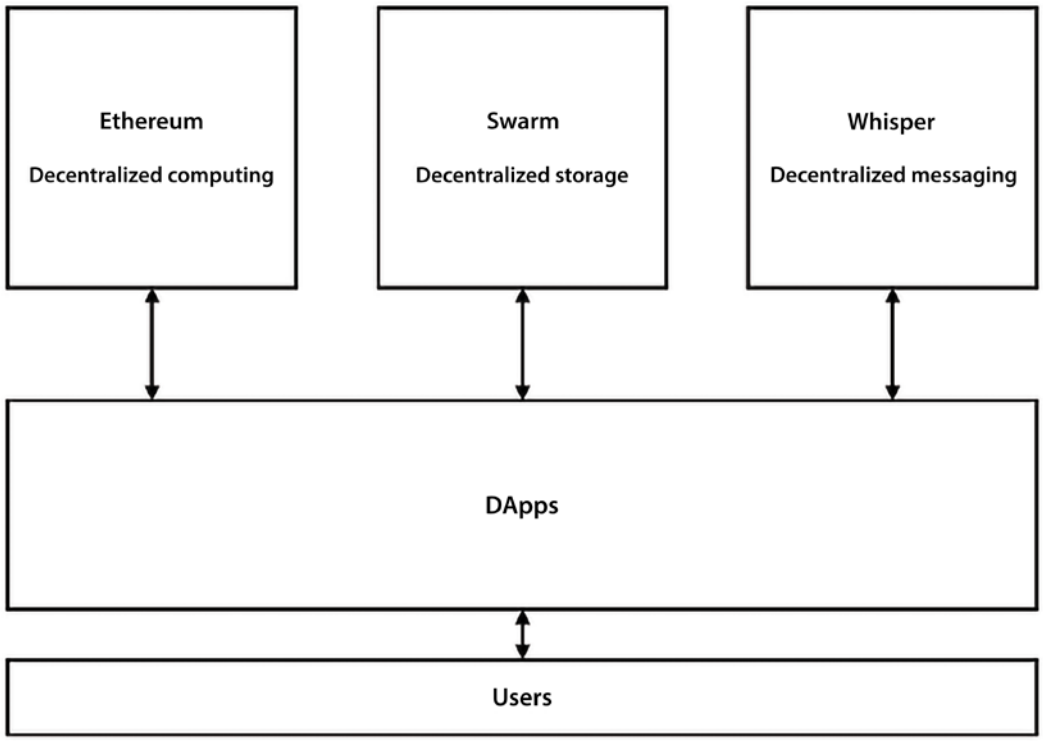


Figure 9.11: Blockchain, Whisper, and Swarm

With the development of Whisper and Swarm, a completely decentralized ecosystem emerges, where Ethereum serves as a decentralized computer, Whisper as a decentralized means of communication, and Swarm as a decentralized means of storage. In simpler words, Ethereum—or more precisely, the EVM—provides compute services, Whisper handles messaging, and Swarm provides storage.

If you recall, in *Chapter 2*, *Decentralization*, we mentioned that decentralization of the whole ecosystem is highly desirable as opposed to decentralization of just the core computation element. The development of Whisper for decentralized communication and Swarm for decentralized storage is a step toward the decentralization of the entire blockchain ecosystem.

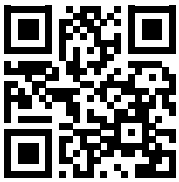
Summary

This chapter mainly covered the Ethereum architecture and ecosystem. We introduced the core concepts of the Ethereum blockchain, such as the state machine model, the world and machine states, accounts, and transactions. Moreover, a detailed introduction to the core components of the EVM was also presented. In addition, the Ethereum blockchain and network, wallets, software clients, and supporting protocols were also discussed.

In the next chapter, we will continue to explore Ethereum development. We will look at more concepts, such as programming languages and developing programs for Ethereum blockchain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

10

Ethereum in Practice

This chapter introduces the Ethereum development environment. Several examples will be presented in this chapter to complement the theoretical concepts provided in the previous chapter.

We will begin by considering the practical payment process in Ethereum, and some recent and upcoming innovations to the blockchain. We will then look at **test nets**, which can play a vital role in testing the smart contracts before deploying on the **main net**. Then, we will look at the private net option in Ethereum, which allows the creation of an independent private network. This private network can be used as a shared distributed ledger between participating entities and for the development and testing of smart contracts.

While there are other clients available for Ethereum, Geth is the leading client for Ethereum and the standard tool of choice, and as such, this chapter uses Geth for the examples. We will also use Remix IDE to write smart contracts and MetaMask to deploy contracts on a private network that we create.

Along the way, we consider the following topics:

- Ethereum payments
- Innovations in Ethereum
- Programming with Geth
- Setting up a development environment
- Introducing Remix IDE
- Interacting with the Ethereum blockchain with MetaMask

Ethereum payments

In this section, we will see how Ethereum works from a user's point of view. For this purpose, we will present the most common use case of transferring funds—in our use case, from one user (Bashir) to another (Irshad). We will use two Ethereum clients, one for sending funds and the other for receiving. There are several steps involved in this process, as follows:

1. First, either a user requests money by sending the request to the sender, or the sender decides to send money to the receiver. We can use any Ethereum wallet software. The request can be sent by sending the receiver's Ethereum address to the sender.

   For example, there are two users, Bashir and Irshad. If Irshad requests money from Bashir, then she can send a request to Bashir by using a QR code. Once Bashir receives this request, he will either scan the QR code or manually type in Irshad's Ethereum address and send the Ether to Irshad's address. This request is encoded as a QR code, shown in the following screenshot, that can be shared via email, text, or be made available for visual scanning or any other communication method:



Figure 10.1: QR code as shown in a blockchain wallet application

2. Once Bashir receives this request, he will either scan this QR code or copy or type the Ethereum address in the Ethereum wallet software and initiate a transaction. Imagine Bashir wants to send funds to Irshad. The sender enters both the amount and the destination address in a wallet software to send the Ether to the receiver. Just before sending the Ether, the final step is to confirm the transaction.



The transaction flow works in fundamentally the same way in all wallet software, so any wallet software can be used. There are many different types of wallet software available online for the iOS, Android, and desktop operating systems.

3. Once the request (transaction) for money to be sent is constructed in the wallet software, it is then broadcasted to the Ethereum network. The transaction is digitally signed by the sender as proof that they are the owner of the Ether.
4. This transaction is then picked up by nodes called miners on the Ethereum network for verification and inclusion in the block. At this stage, the transaction is still unconfirmed.

5. Once it is verified and included in the block, the **Proof of Work** (**PoW**) process starts.
6. Once a miner finds the answer to the PoW problem by repeatedly hashing the block with a new nonce, this block is immediately broadcast to the rest of the nodes, which then verify the block and PoW.
7. If all the checks pass, then this block is added to the blockchain, and miners are paid rewards accordingly.
8. Finally, Irshad gets the Ether, which will be displayed in the wallet software in use.



On the blockchain, this transaction is identified by the following transaction hash: `0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e`. This can be visualized on the block explorer at `https://etherscan.io/tx/0xc63dce6747e1640abd63ee63027c3352aed8cdb92b6a02ae25225666e171009e`

With this example, we complete our discussion on the most common usage of the Ethereum network: transferring Ether from one user to another.

Innovations in Ethereum

In this section we will cover some of the innovations in Ethereum. Ethereum is under constant development. Just like BIPs, we have **Ethereum improvement proposals** (**EIPS**) in Ethereum to suggest and implement improvements in Ethereum. You can track EIPs here: `https://eips.ethereum.org`. Now we will cover some important improvements.

Difficulty time bomb

In addition to timestamp difference-based difficulty adjustment, there is also another element that increases mining difficulty exponentially after every 100,000 blocks. This is the so-called **difficulty time bomb**, or **ice age**, introduced in the Ethereum network, which will make it very hard to mine on the Ethereum blockchain at some point in the future.

Once activated, over time, the difficulty bomb makes mining on Ethereum 1.x so prohibitively slow that it becomes infeasible, resulting in a so-called "ice age." In other words, this mechanism exponentially increases the PoW mining difficulty to a level where block generation becomes impossible, thus forcing the miners to migrate to Ethereum's **Proof of Stake** (**PoS**) system called **Casper**.



More information about Casper is available here: `https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf`.

The **London** upgrade (`https://ethereum.org/en/history/#london`) introduced EIP-1559. The most recent upgrade is **Paris** (The Merge: `https://ethereum.org/en/history/#paris`), which we will discuss in more detail in *Chapter 13*, *The Merge and Beyond*.

EIP-1559

A way to mitigate high transaction fees to some extent is EIP 1559. However, note that EIP-1559 does not aim to reduce the gas fee; it just helps to better predict the fee, which can mean that users won't end up overpaying, resulting in an overall reduction in fees paid over time.

EIP-1559 was implemented as part of the London upgrade. EIP-1559 enforces a base fee (per gas unit) that must be paid and aims to match supply and demand. This fee is burnt forever, which can decrease the Ether inflation rate. The base fee recalibrates with every block depending on the "target." Blocks can expand in size by changing `MaxGasLimit`, and if the block is larger than the target, then the fee is higher, and lower otherwise. Users can also pay the miners a tip to prioritize their transactions in addition to the base fee.

With the increased block size, it might be possible to launch a DoS attack where the network could be flooded with successive large-size blocks. EIP-1559 makes gas fee calculation more predictable and improves user experience, which can result in users not overpaying too much.

Note that EIP-1559 is not a mechanism to reduce the transaction fee because a high transaction fee is a problem due to an inherent scalability problem, i.e., a fixed block size, and consequently a supply and demand mechanism. When the network is busy, a fixed block size results in users waiting for their transaction to be included in the block for indefinite periods of time.



Simple solutions do come to mind in resolving this issue, but they are not as simple as they may seem. If we can somehow enforce a set fee and strict consensus-driven universal transaction ordering along with flexible block sizes, then this problem can be solved. However, if there is no incentive for the miners left due to set fee restrictions, then perhaps miners will leave the blockchain.

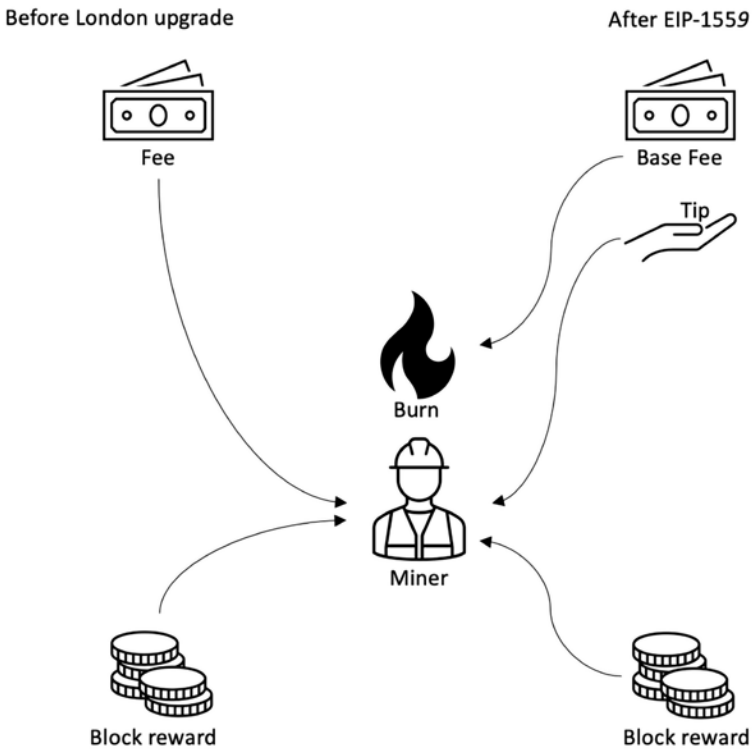


Figure 10.2: EIP 1559 mechanism, before and after the London upgrade

EIP-1559 works on the basis of three variables:

- **baseFeePerGas:** This is a new field introduced in the block structure. It represents the reserve price that must be paid for a transaction to be included in a block. With EIP-1559, blocks can increase or decrease in size as per the network demand until the block limit of 30 million gas.

  The base fee is calculated based on previous blocks, which makes predicting gas fees more predictable for users. The calculation is based on a formula that compares the size of the previous block with the target size. The base fee increases or decreases based on how far the current block size is from the target, which is 15 million gas for each block. If a block is 100% full the base fee will increase by 12.5%, if the block is only 50% full the base fee will remain the same, and if the block is 0% full the base fee decreases by 12.5%. This means that the network achieves equilibrium at 50% capacity by calibrating fees according to the level of network utilization.

This exponential growth makes it economically non-viable for block size to remain high indefinitely. Wallets can calculate the base deterministically based on the information available in the previous blocks. Also, the base fee is burnt to thwart an attack where miners could collude to increase the base fee permanently.

- **maxPriorityFeePerGas**: This is a variable controlled by the users. Users set this and add it to the transactions. It represents the part of the transaction fee that goes to the miner. This is the miner tip that allows transactions to be executed quicker.
- **maxFeePerGas**: This is another user-settable material. It represents the maximum amount that a user is willing to pay for their transaction. It is inclusive of `baseFeePerGas` and `maxPriorityFeePerGas`. The difference between `maxFreePerGas` and `baseFeePerGas` + `maxPriorityFeePerGas` is refunded to the user.

In summary, EIP-1559 makes transaction fees more predictable, reduces delay in transaction inclusion and confirmation, improves user experience through an automated fee/bid mechanism, and adjusts fees based on network activity, i.e., demand and supply.

One way to think about this is that EIP-1559 replaces the volatile fee prices with volatility in block size, and over time, the block size smoothes out because the users who cannot pay for a miner tip to prioritize their block will drop off, and eventually the equilibrium of 50% block sizes will be maintained again.

In summary, before the London upgrade, the gas calculation was:

Gas units (*limit*) * *Gas price per unit*

After the London upgrade, the gas calculation works out as:

Gas units (*limit*) * (*Base fee* + *Tip*)

The merge and upcoming upgrades

Upcoming upgrades are the so-called "merge" and shard chains. The merge merged the current main net Ethereum on September 15, 2022 with the Beacon chain. The Beacon chain is a PoS chain and is currently live, providing the core chain for shards and the merged Ethereum chain.

Note that there is no Ethereum 2.0 now, as was originally planned. Eth2 or Ethereum 2.0 was a term used for a new distinct version of Ethereum 2.0, but it has now been replaced with The Merge. The Merge upgrade has merged both Ethereum1 and what was supposed to be Ethereum 2.0, and a single platform has emerged, called Just Ethereum, not Ethereum 2.0. A mental model to clarify this is that Eth1 is now the so-called *execution layer* where transaction execution occurs, and Eth2 is the *consensus layer* that is responsible for PoS. We will cover more about this in a separate chapter on Ethereum Merge and future upgrades in *Chapter 13*, *The Merge and Beyond*.

Next, we discuss the installation procedure and usage of some of the Ethereum clients.

Programming with Geth

First, we describe Geth and explore various operations that can be performed using this client.

Installing and configuring the Geth client

The installation procedure detailed at `https://geth.ethereum.org/docs/getting-started/installing-geth` describes the installation of Ethereum clients on macOS and Linux. Instructions for other operating systems are also available.

Once installation is complete, Geth can be launched simply by issuing the `geth` command at the terminal. It comes preconfigured with all the required parameters to connect to the live Ethereum network (main net):

```
$ geth
```

When the Ethereum client starts up, it starts to synchronize with the rest of the network. There are three types of synchronization mechanisms available, namely, `snap`, `full`, and `light`:

- **Snap**: This is the default mode that keeps the most recent 128 blocks' states in memory, enabling immediate access to transactions in this range. The node in this mode also stores checkpoints between the initial sync block (a recent block) and the 128 most recent blocks that allow you to rebuild states. This mode starts by downloading the headers for a set of blocks. Once the headers are verified, the rest of the data, such as receipts and block bodies, is downloaded. It also downloads raw state data and builds state tries in parallel.
- **Full**: In this synchronization mode, the Geth client generates the latest state by executing and verifying all blocks since the genesis block. In this mode only the most recent 128 block states are stored. Older blocks are pruned regularly and represented with checkpoints that can be used to regenerate the state if required. There are also "full archive" nodes that keep all data since the genesis block, and old data is never deleted. Currently, the Ethereum blockchain size is too large to download and maintain without difficulty on entry-level hardware. Therefore, usually, SSDs are recommended for a full archive node so that disk latency cannot cause any processing delays.
- **Light**: This is the quickest mode and only downloads and stores the current state trie. In this mode, the client does not download any historic blocks and only processes newer blocks.

Synchronization mode is configurable on the Geth client via the flag:

```
--syncmode value
```

Here, the value can be either `snap`, `full`, or `light`, e.g., `--syncmode light` starts up the Geth client in light sync mode.

Creating a Geth new account

New accounts can be created via the command line using Geth or any other client's command-line interface. Execute the following command to add a new account:

```
$ geth account new
```

This command will produce output similar to the following:

```
INFO [11-23|21:51:53.440] Maximum peer count                       ETH=50 LES=0
total=50
Your new account is locked with a password. Please give a password. Do not
forget this password.
Password:
Repeat password:

Your new key was generated

Public address of the key:   0x26Ef606C24682D0C73933389d42D9023D1E29679
Path of the secret key file: /Users/imran/Library/Ethereum/keystore/UTC--2022-
11-23T21-51-55.579586000Z--26ef606c24682d0c73933389d42d9023d1e29679

- You can share your public address with anyone. Others need it to interact
with you.
- You must NEVER share the secret key with anyone! The key controls access to
your funds!
- You must BACKUP your key file! Without the key, it's impossible to access
account funds!
- You must REMEMBER your password! Without the password, it's impossible to
decrypt the key!
```

The list of accounts can be displayed using the Geth client by issuing the following command:

```
$ geth account list
```

This command will produce output similar to the following:

```
INFO [11-23|21:53:12.201] Maximum peer count                       ETH=50 LES=0
total=50
INFO [11-23|21:53:12.204] Set global gas cap
cap=50,000,000
Account #0: {bbe89c77e56c109d4ac91e79dbaf68e5b6955123} keystore:///Users/imran/
Library/Ethereum/keystore/UTC--2022-05-20T16-57-01.035303000Z--bbe89c77e56c109d
4ac91e79dbaf68e5b6955123
Account #1: {38f5b137676a960ad52bce7b35e597b23c3e3121} keystore:///Users/imran/
Library/Ethereum/keystore/UTC--2022-05-20T17-07-51.380957000Z--38f5b137676a960a
d52bce7b35e597b23c3e3121
Account #2: {26ef606c24682d0c73933389d42d9023d1e29679} keystore:///Users/
imran/Library/Ethereum/keystore/UTC--2022-11-23T21-51-55.579586000Z--26ef606c
24682d0c73933389d42d9023d1e29679INFO [01-18|17:59:25.595] Maximum peer count
ETH=50 LES=0
```



Note that you will see different addresses and directory paths when you run this on your computer.

Next, we'll see different methods to interact with the blockchain.

Querying the blockchain using Geth

There are different methods available to query with the blockchain. First, in order to connect to the running instance of the client, either a local IPC or RPC API can be used.

There are three methods of interacting with the blockchain using Geth:

- Geth console
- Geth attach
- Geth JSON RPC

Geth console and Geth attach are used to interact with the blockchain using an REPL JavaScript environment. JSON RPC is a remote procedure call mechanism that makes use of JSON data format to encode its calls. In simpler terms, it is an RPC encoded in JSON.

Geth console

Geth can be started in console mode by running the following command:

```
$ geth console
```

This will start the interactive JavaScript environment in which JavaScript commands can be run to interact with the Ethereum blockchain, e.g., getting balance, block numbers, and many other commands.

Geth attach

When a Geth client is already running, the interactive JavaScript console can be invoked by attaching to that instance. This is possible by running the `geth attach` command.

The Geth JavaScript console can be used to perform various functions. For example, an account can be created by attaching Geth.

Geth can be attached with the running daemon, as shown in the following screenshot:

```
$ geth attach
```

This command will produce output similar to the following:

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.10.14-stable/darwin-arm64/go1.17.5
coinbase: 0xbbe89c77e56c109d4ac91e79dbaf68e5b6955123
```

```
at block: 0 (Thu Jan 01 1970 01:00:00 GMT+0100 (BST))
 datadir: /Users/imran/Library/Ethereum
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
>
```

Once Geth is successfully attached to the running instance of the Ethereum client, it will display the command prompt, >, which provides an interactive command-line interface to interact with the Ethereum client using JavaScript notations.

For example, a new account can be added using the following command in the Geth console:

```
> personal.newAccount()
Password:
Repeat password:
"0x76bcb051e3cedfcc9c7ff0b25a57383dacbf833b"
```



Note that readers will see a different address.

The list of accounts can also be displayed similarly:

```
> eth.accounts
["0x07668e548be1e17f3dcfa2c4263a0f5f88aca402",
"0xba94fb1f306e4d53587fcdcd7eab8109a2e183c4",
"0x1df5ae40636d6a1a4f8db8a0c65addce5a992a14",
"0x76bcb051e3cedfcc9c7ff0b25a57383dacbf833b"]
```

Geth JSON RPC API

JSON stands for **JavaScript Object Notation**. It is a lightweight and easy-to-understand text format used for data transmission and storage. A remote procedure call is a distributed systems concept. It is a mechanism used to invoke a procedure in a different computer. It appears as if a local procedure call is being made because there is a requirement to write code to handle remote interactions.



Further details on RPC and JSON can be found here: `https://www.jsonrpc.org`.

For this chapter, it is sufficient to know that the JSON-RPC API is used in Ethereum extensively to allow users and dApps to interact with the blockchain.

There are several methods available to interact with the blockchain. One is to use the Geth console, which makes use of the Web3 API to query the blockchain. The Web3 API makes use of the JSON-RPC API. Another method is to make JSON-RPC calls directly, without using the Web3 API. In this method, direct RPC calls can be made to the Geth client over HTTP. By default, Geth RPC listens at TCP port `8545`.

Now, we will look at some examples involving the utilization of the JSON RPC API. We will use a common utility called curl (`https://curl.haxx.se`) for this purpose.

For these examples to work, first, the Geth client needs to be started up with appropriate flags. If there is an existing session of Geth running with other parameters, stop that instance and run the `geth` command, as shown here, which will start Geth with the RPC available. The user can also control which APIs are exposed:

```
$ geth  --http --http.api "eth,net,web3,personal"
```

In this command, Geth is started up with the `--rpc` and `--rpcapi` flags, along with a list of APIs that are exposed:

- The `rpc` flag enables the HTTP-RPC server.
- The `rpcapi` flag is used to define which APIs are made available over the HTTP-RPC interface. This includes a number of APIs such as `eth`, `net`, `web3`, and `personal`.

For each of the following examples, run the `curl` command in the terminal, as shown in each of the following examples.

The list of accounts can be obtained by issuing the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_accounts","params":[], "id":64}' http://
localhost:8545
```

This will display the following JSON output, which lists all the Ethereum accounts owned by the client:

```
{"jsonrpc":"2.0","id":64,"result":["0x07668e548be1e17f3dcfa2c4263a0f5
f88aca402","0xba94fb1f306e4d53587fcdcd7eab8109a2e183c4","0x1df5ae4063
6d6a1a4f8db8a0c65addce5a992a14","0x76bcb051e3cedfcc9c7ff0b25a57383dac bf833b"]}
```

We can query if the network is up by using the command shown here:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"net_listening","params":[], "id":64}' http://
localhost:8545
```

This will display the output as shown here with the result `true`, indicating that the network is up:

```
{"jsonrpc":"2.0","id":64,"result":true}
```

We can find the Geth client version using this command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"web3_clientVersion","params":[], "id":64}' http://
localhost:8545
```

This will display the Geth client version:

```
{"jsonrpc":"2.0","id":64,"result":"Geth/v1.9.9-stable-01744997/linux-amd64/
go1.13.4"}
```

To check the latest synchronization status, we can use the following command:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_syncing","params":[], "id":64}' http://
localhost:8545
```

This will display the data pertaining to the synchronization status or return `false`:

```
{"jsonrpc":"2.0","id":64,"result":{"currentBlock":"0x1d202","highest
lock":"0x8a61c8","knownStates":"0x23b9b","pulledStates":"0x2261a",
"startingBlock":"0xa5c5"}}
```

The coinbase address can be queried using:

```
$ curl -X POST --insecure -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"eth_coinbase","params":[], "id":64}' http://
localhost:8545
```

This will display the output shown here, indicating the client `coinbase` address:

```
{"jsonrpc":"2.0","id":64,"result":"0x07668e548be1e17f3dcfa2c4263a0f5f88aca402"}
```

These are just a few examples of extremely rich APIs that are available in Ethereum's Geth client.



More information and official documents regarding Geth RPC APIs are available at the following link: `https://eth.wiki/json-rpc/API`.

In this section, we have covered Geth JSON RPC APIs and seen some examples of how to query the blockchain via the RPC interface. With this, we have completed the introduction to Geth.

Setting up a development environment

A usual and sensible approach to develop and test Ethereum smart contracts is within a local private net or a simulated environment like Ganache. After all the relevant tests are successful on a public test net, the contracts can then be deployed to the public main net. There are, however, variations in this process.

Many developers opt to only develop and test contracts on a local simulated environment and then deploy them onto the public main net or their private/enterprise production blockchain networks. Developing first on a simulated environment and then deploying directly to a public network can lead to faster time to production, as setting up private networks may take longer compared to setting up a local development environment with a blockchain simulator.

There are new tools and frameworks available, like **Truffle** and **Ganache**, which make development and testing for Ethereum easier. We will look into these tools in more depth in *Chapter 11*, *Tools, Languages, and Frameworks for Ethereum Developers*, but first, we will use a manual approach whereby we develop a smart contract and deploy it manually via the command line to the private network. Frameworks and tools make development easier but hide most of the finer "under the hood" details that are useful for beginners to build a strong foundation of knowledge.

Let us start by connecting to a test network.

Connecting to test networks

The Ethereum Go client (`https://geth.ethereum.org`), Geth, can be connected to a test network using one of the following switches available with Geth. Geth can be issued with a command-line flag to connect to the desired network:

- `--goerli`: Görli network: pre-configured proof-of-authority test network
- `--rinkeby`: Rinkeby network: pre-configured proof-of-authority test network
- `--ropsten`: Ropsten network: pre-configured proof-of-work test network
- `--sepolia`: Sepolia network: pre-configured proof-of-work test network

This command will connect to the Sepolia network:

```
$ geth --sepolia
```

A blockchain explorer for the sepolia test net is located at `https://sepolia.etherscan.io` and can be used to trace transactions and blocks on the Ethereum test network.

Now let us experiment with building a private network and then we will see how a contract can be deployed on this network using command-line tools.

Creating a private network

In this section, we will start up a private network and prepare it for use. We will use the following components, as specified in the previous chapter:

- Network ID: Network ID `786` has been chosen for the example private network.
- Genesis file: A custom genesis file that will be used is shown here:

```
{
"nonce": "0x0000000000000042",
"timestamp": "0x00",
```

```
"parentHash ": "0x0000000000000000000000000000000000000000000000000000000000000000",
"extraData": "0x00",
"gasLimit": "0x8000000",
"difficulty": "0x0400",
"mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
"coinbase": "0x3333333333333333333333333333333333333333",
"alloc": {},
"config": {
"chainId": 786,
"homesteadBlock": 0,
"eip150Block": 0,
"eip155Block": 0,
"eip158Block": 0,
"byzantiumBlock": 0,
"constantinopleBlock": 0
    }
}
```

This file is saved as a text file with the JSON extension, for example, `privategenesis.json`. Optionally, Ether can be pre-allocated by specifying the beneficiary's addresses and the amount of Wei, but this is usually not necessary, as being on a private network, Ether can be mined very quickly.

In order to pre-allocate an account with Ether, a section can be added to the genesis file, as shown here:

```
"alloc": {
        "0xcf61d213faa9acadbf0d110e1397caf20445c58f ":
            {"balance": "100000"},
}
```

- **Data directory**: In the following example, it is `~/etherprivate/`.

In addition to the aforementioned three components, it is desirable that you disable node discovery so that other nodes on the internet cannot discover your private network and that it is secure. This can be achieved by running Geth with the flag `--nodiscover`, which disables the peer discovery mechanism. If other networks happen to have the same genesis file and network ID, they may connect to your private net, which can result in security issues. The chance of having the same network ID and genesis block is very low, but, nevertheless, disabling node discovery is a good practice, and is recommended.

Another network or node connecting to your private network can result in security breaches, information leakage, and other undesirable security incidents. However, note that private networks are usually run within enterprise environments and are protected by usual enterprise security practices such as firewalls.

Disabling peer discovery also allows us to define a list of static peers that we have on our network. This gives us the additional ability to control who can join our private network.

In order to configure this list, the node IDs of these nodes are added to a configuration file called `static-nodes.json`. This file is usually placed in the data directory of the Geth (Ethereum client) executable. This directory is also where the `chaindata` (database) and `keystore` files are saved. By default, the data directory is located at `<user's home directory>/Library/Ethereum` but can be configured by using the flag `--datadir`.

The filename should be `static-nodes.json` under the data directory. This is valuable in a private network because this way, the network is limited to only known nodes. An example of the `static-nodes.json` file is shown as follows:

```
[
"enode://
44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc5
7a25380481c8a36fb3be2787375e932fb4885885f6452f6efa77f@xxx.xxx.xxx.xxx:TCP_PORT"
]
```

Here, `xxx` is the IP address and `TCP_PORT` can be any valid and available TCP port on the system. The long hex string is the node ID.

As we now understand the various aspects and components required to set up a private network, including the genesis file and other relevant configuration files, let's now move on to setting up our own private network using Ethereum.



Before starting, it is useful to note that there are many configurations and settings options available with Geth, help can be retrieved by issuing the command:

```
$ geth help
```

Starting up the private network

The first step is to create a directory named `etherprivate` under the home directory of the user.

```
$ mkdir ~/etherprivate
```

This command will create the directory. Once the directory is created, place the `privategenesis.json` file shown earlier in it. At this point, stored under the home directory of the user, we have a directory named `~/etherprivate`, which contains the genesis file called `privategenesis.json`. We are ready to start our network. The initial command to start the private network is shown as follows:

```
$geth init ~/etherprivate/privategenesis.json --datadir ~/etherprivate
```

This will produce an output similar to what is shown in the following:

```
INFO [05-21|13:35:40.359] Maximum peer count
```

```
ETH=50 LES=0 total=50
INFO [05-21|13:35:40.361] Set global gas cap
cap=50,000,000
INFO [05-21|13:35:40.361] Allocated cache and file handles         database=/
Users/imran/etherprivate/geth/chaindatacache=16.00MiB handles=16
INFO [05-21|13:35:40.433] Writing custom genesis block
INFO [05-21|13:35:40.433] Persisted trie from memory database      nodes=0
size=0.00B time="205.416µs" gcnodes=0 gcsize=0.00B gctime=0s livenodes=1
livesize=0.00B
INFO [05-21|13:35:40.434] Successfully wrote genesis state
database=chaindatahash=6650a0..b5c158
INFO [05-21|13:35:40.434] Allocated cache and file handles         database=/
Users/imran/etherprivate/geth/lightchaindatacache=16.00MiB handles=16
INFO [05-21|13:35:40.491] Writing custom genesis block
INFO [05-21|13:35:40.491] Persisted trie from memory database      nodes=0
size=0.00B time="4.084µs"   gcnodes=0 gcsize=0.00B gctime=0s livenodes=1
livesize=0.00B
INFO [05-21|13:35:40.491] Successfully wrote genesis state
database=lightchaindatahash=6650a0..b5c158
```

If you see a `Successfully wrote genesis state` message, then all is well and you should see a directory structure as shown below:

```
├── geth
│   ├── LOCK
│   ├── chaindata
│   │   ├── 000001.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000000
│   ├── lightchaindata
│   │   ├── 000001.log
│   │   ├── CURRENT
│   │   ├── LOCK
│   │   ├── LOG
│   │   └── MANIFEST-000000
│   └── nodekey
├── keystore
└── privategenesis.json
```

This director structure contains chain data, logs, and a keystore. In order for Geth to start, the following command can be issued:

```
$ geth --datadir ~/etherprivate/ --networkid 786 --http --http.api
'web3,eth,net,debug,personal' --http.corsdomain '*'
```

This will produce many output messages as Geth starts up. An important part of the output to note is the following log lines:

```
INFO [05-21|17:12:54.789] IPC endpoint opened                      url=/Users/
imran/etherprivate/communication
INFO [05-21|17:12:54.790] HTTP server started
endpoint=127.0.0.1:8545 prefix= cors=* vhosts=localhost
```

These lines show the information about the **Inter-Process Communication** (**IPC**) endpoint, HTTP endpoint, and Etherbase (coinbase) account information. This information is useful for the examples provided later in this section.



IPC mechanisms are used to allow communication between different processes or threads running in an operating system locally on a computer. There are different IPC mechanisms including pipes and signals. A pipe is a bounded buffer between processes realized in practice by a file descriptor shared between processes. A signal is a method used to get a process's attention. Geth uses pipes to enable communication between Geth instances.

Now Geth can be attached via IPC to the running Geth client on a private network using the following command:

```
$geth attach ~/etherprivate/geth.ipc
```

This command will open the interactive JavaScript console to run the private net session:

```
Welcome to the Geth JavaScript console!

instance: Geth/v1.10.14-stable/darwin-arm64/go1.17.5
coinbase: 0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f
at block: 188 (Sat May 21 2022 17:47:53 GMT+0100 (BST))
datadir: /Users/imran/etherprivate
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0

To exit, press ctrl-d or type exit
```

You may notice that a warning message appears when Geth starts up:

```
WARNING: No etherbase set and no accounts found as default.
```

This message appears because there are no accounts currently available in the new test network and no account is set as `etherbase` to receive mining rewards. This issue can be addressed by creating a new account and setting that account as etherbase. This will also be required when mining is carried out on the test network.

This is shown in the following commands. Note that these commands are entered in the Geth JavaScript console. The following command creates a new account. In this context, the account will be created on the private network ID 786, as this is the network we created earlier:

```
>personal.newAccount("Password123")
"0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f"
```

Once the account is created, the next step is to set it as an etherbase/coinbase account so that the mining reward goes to this account. This can be achieved using the following command:

```
>miner.setEtherbase(personal.listAccounts[0])
true
```

Currently, the etherbase account has no balance, as can be seen by using the following command:

```
>eth.getBalance(eth.coinbase).toNumber();
0
```

In this section, we created a private network with a custom genesis file. Also, we created a new account in our private network. Now, before further exploration of different methods, let's understand some basics of using the JavaScript console.

Experimenting with the Geth JavaScript console

In the JavaScript console, we can perform several operations. A general tip is that if two spaces and two tabs on the keyboard are pressed in a sequence, a complete list of the available objects will be displayed.

Furthermore, when a command is typed, it can be autocompleted by pressing *Tab* twice. If *Tab* is pressed twice, then the list of available methods is also displayed. In addition to the aforementioned command, in order to get a list of available methods of an object, just press *Enter*. An example is shown below, which shows a list of all the methods available for `net`:

```
> net
{
  listening: true,
peerCount: 0,
  version: "786",
getListening: function(callback),
getPeerCount: function(callback),
getVersion: function(callback)
}
```

There is also the `eth` object available, which has several methods. While there are several methods under this object, the most common is `getBalance`, which we can use to query the current balance of Ether. This is shown in the following example:

```
> eth.getBalance(eth.coinbase)
150000000000000000000
```

After mining, a significant amount can be seen here. Mining is extremely fast as it is a private network with no competition for solving the PoW, and also in the genesis file, the network difficulty has been set to quite low.

The preceding balance is shown in Wei. If we want to see the output in Ether, we can use the `web3` object, as shown here:

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")
150
```

There are a few other commands that can be used to query the private network. Some examples are shown as follows:

Get the current gas price:

```
> eth.gasPrice
1000000000
```

Get the latest block number:

```
> eth.blockNumber
30
```

`debug` can come in handy when debugging issues. A sample command is shown here; however, there are many other methods available. A list of these methods can be viewed by typing `debug`.

The following method will return the RLP of block `0`:

```
> debug.getBlockRlp(0)
"0xf901f8f901f3a00000000000000000000000000000000000000000000000000000000000000000
000000a01dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347943
3333333333333333333333333333333333333333a056e81f171bcc55a6ff8345e692c0f86e5b
48e01b996cadc001622fb5e363b421a056e81f171bcc55a6ff8345e692c0f86e5b48e01b996
cadc001622fb5e363b421a056e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc00162
2fb5e363b42
1b901000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000
```

Mining and sending transactions

Now that we have started up our private network, mining can start by simply issuing the following command. This command takes one parameter: the number of threads. In the following example, two threads will be allocated to the mining process by specifying `2` as an argument to the `start` function:

```
> miner.start(2)
```

Here we can also provide an integer parameter. For example, if we provide `1`, it will only use one CPU core for mining, which helps with performance issues, if using all CPU resources is reducing system performance. An example command of using only one CPU is `miner.start(1)`. On systems where there is only one CPU, issuing the preceding command will inevitably use only one CPU. However, on a multicore system, providing the number of cores that can be used for mining helps to address any performance concerns, as multiple mining threads can run in parallel on multiple cores.

After the preceding command is issued as preparation for mining, the DAG generation process starts, which produces an output similar to the one shown here:

```
INFO [05-21|17:17:54.075] Commit new mining work                   number=1
sealhash=d18bb2..2acb58 uncles=0 txs=0 gas=0 fees=0 elapsed=2.161ms
INFO [05-21|17:17:54.798] Generating DAG in progress               epoch=0
percentage=0 elapsed=335.435ms
INFO [05-21|17:17:55.135] Generating DAG in progress               epoch=0
percentage=1 elapsed=673.142ms
INFO [05-21|17:17:55.524] Generating DAG in progress               epoch=0
percentage=2 elapsed=1.062s
INFO [05-21|17:17:55.926] Generating DAG in progress               epoch=0
percentage=3 elapsed=1.463s
INFO [05-21|17:17:56.343] Generating DAG in progress               epoch=0
percentage=4 elapsed=1.880s
```



DAG stands for **Directed Acyclic Graph**. In the context of Ethereum's Ethash PoW algorithm, DAG refers to **Dagger**, which is a memory-hard PoW algorithm based on moderately connected directed acyclic graphs. The aim of the Dagger algorithm is to provide an ASIC-resistant, memory-hard PoW algorithm. The Dagger algorithm works by generating a DAG of a few gigabytes every 30,000 blocks. This DAG serves as a resource for PoW where the PoW mechanism needs to choose subsets from the DAG, which is dependent on the nonce and the block header.

Once DAG generation is finished, the mining process starts and blocks are produced. Geth will produce an output similar to that shown in the following. It can be seen that blocks are being mined successfully with the `mined potential block` message:

```
INFO [05-21|17:19:21.154] Successfully sealed new block            number=8
sealhash=c4e772..dff715 hash=83f1a1..2409db elapsed=16.735s
```

```
INFO [05-21|17:19:21.154] 🔗 block reached canonical chain          number=1
hash=b0e469..54acc8
INFO [05-21|17:19:21.154] 🔨 mined potential block                  number=8
hash=83f1a1..2409db
INFO [05-21|17:19:21.155] Commit new mining work                   number=9
sealhash=07728b..be7c03 uncles=0 txs=0 gas=0 fees=0 elapsed=1.224ms
INFO [05-21|17:19:23.255] Successfully sealed new block            number=9
sealhash=07728b..be7c03 hash=6ecea5..43c4bd elapsed=2.101s
INFO [05-21|17:19:23.255] 🔗 block reached canonical chain          number=2
hash=c1b8eb..8176e1
```

Mining can be stopped using the following command:

```
> miner.stop()
null
```

Now let's see how to create a new account and send a transaction on our private network.

- Create a new account. Note that `Password123` is the password chosen as an example, but you can choose any:

  ```
  > personal.newAccount("Password123")
  "0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b"
  ```

- Unlock the account before sending transactions. We use `allow unsecure unlocking`; otherwise, the accounts cannot be unlocked with HTTP access. If that is the case you will see an error message as follows:

  ```
  > personal.unlockAccount("0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b")
  Unlock account 0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b
  Passphrase:
  GoError: Error: account unlock with HTTP access is forbidden at web3.
  js:6365:37(47)
       at github.com/ethereum/go-ethereum/internal/jsre.MakeCallback.func1
  (native)
       at <eval>:1:24(3)
  ```

To circumvent the error, restart Geth using the `–allow-insecure-unlock` flag:

```
$ geth --datadir ~/etherprivate/ --allow-insecure-unlock --networkid 786 --http
--http.api 'web3,eth,net,debug,personal' --http.corsdomain '*'
```

Now we unlock both accounts that we created earlier with the `personal.newAccount()` command at the start of the private network we're creating, and just before the `unlockAccount()` command on this page.

The first account we created at the start of our private network is `0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f` and another one, `0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b`, is just at the start of this page:

```
> personal.listAccounts
["0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
"0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b"]
> personal.unlockAccount("0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f")
Unlock account 0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f
Passphrase:
true

> personal.unlockAccount("0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b")
Unlock account 0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b
Passphrase:
true
```

Now once we have these accounts unlocked we can issue some further commands to query the balances they hold. First, let's check the balance of our account that we created earlier when we started our private network for the first time:

```
> web3.fromWei(eth.getBalance("0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f"),
"ether")

150
```

As this account is also our coinbase account, which is the default account that receives the mining reward; we can also query the balance slightly differently by specifying `eth.coinbase` in the command, as shown here:

```
> web3.fromWei(eth.getBalance(eth.coinbase), "ether")
150
```

Finally, we check the balance of the other account that we created. As this has not received any mining reward, the balance is `0` as expected:

```
> web3.fromWei(eth.getBalance("0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b"),
"ether")
0
```

Now let's try to send transactions from one account to another. In this example we'll send Ether from account `0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f` to `0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b` using the `sendTransaction` command, shown as follows.

First, we send a value of `100`:

```
>eth.sendTransaction({     from: "0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
to: "0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b",     value: 100 })
```

This command outputs the transaction hash, which is a unique identifier used to identify a specific transaction. The output is shown as follows:

```
"0xba427f7c5590530d75375e3670b856200c684141f9c787bf8e1bf622305880d9"
```

This command will transfer 100 Wei (the smallest Ether unit) to the target account. To transfer the value in Ether, 100 ETH in our example, we can use the command slightly differently and use `web3.toWei`, which will convert the value from Ether into Wei. To achieve this we issue the command as shown here:

```
>eth.sendTransaction({     from: "0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
to: "0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b",     value: web3.toWei(100,
"ether") })
```

This command outputs the transaction hash as shown here:

```
"0xd33390a61dc91cdeefdf51af302f13b731f94d042718f7749e169c0a7266ca80"
```

The preceding command makes use of `web3.toWei`, which takes two parameters, `value`, which is 100 in our example, and a string, `Ether`, which is the unit of value. This means that 100 ETH will be converted into its equivalent Wei and will be used in the `sendTransaction` command. The result of this command will be transferring 100 ETH from our source account to the target account. Also notice the log message in the logs in the other console/terminal window, from where we ran the Geth command to run our private network earlier at the start of the private network:

```
INFO [05-21|17:44:14.565] Submitted transaction
hash=0xd33390a61dc91cdeefdf51af302f13b731f94d042718f7749e169c0a7266ca80
from=0x6E94BDb15141491bC3B9De3A9CaB9d87ae2af82f nonce=1
recipient=0x0F044cb4A0F924b6CfCF07c6c57945A0AF75Ec5b
value=100,000,000,000,000,000,000
```

This message shows that the transaction has been submitted, but it needs to be mined first in order for the transaction to take effect. As we stopped mining earlier, for this transaction to go through we need to start mining again; otherwise, the transaction won't be processed:

```
> miner.start()
null
```

Now check in the logs that the mining has started to work:

```
INFO [05-21|17:45:27.215] 🔨 mined potential block                  number=70
hash=d4eb33..b0ce2a
INFO [05-21|17:45:27.215] Commit new mining work                   number=71
sealhash=910916..cfad54 uncles=0 txs=0 gas=0     fees=0
elapsed="68.166µs"
```

```
INFO [05-21|17:45:27.374] Successfully sealed new block            number=71
sealhash=910916..cfad54 hash=04c1c2..8200b5 elapsed=159.180ms
INFO [05-21|17:45:27.374] 🔗 block reached canonical chain          number=64
hash=5e852c..204f33
INFO [05-21|17:45:27.374] 🔨 mined potential block                  number=71
hash=04c1c2..8200b5
INFO [05-21|17:45:27.374] Commit new mining work                   number=72
sealhash=3f2d41..8b73bc uncles=0 txs=0 gas=0     fees=0
elapsed="94.459µs"
INFO [05-21|17:45:27.502] Successfully sealed new block            number=72
sealhash=3f2d41..8b73bc hash=744a22..c3f5e0 elapsed=127.906ms
```

Note that the amount of funds has been transferred:

```
> web3.fromWei(eth.getBalance("0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b"),
"ether")
100.0000000000000001
> web3.fromWei(eth.getBalance("0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f"),
"ether")
839.9999999999999999
```

Remember that as the mining progresses, you will see slightly different numbers and more Ether in the source account (coinbase) and the target account will now have 100 ETH, which we transferred.

It could be a bit cumbersome to type all these account IDs. Instead of typing these long account IDs, we can also use the `listAccounts[]` method, which takes an integer parameter to address the account, for example, `0`, which represents the first account that we created.

Now let's see an example of using the `listAccounts[]` method, as shown here:

```
> eth.sendTransaction({from: personal.listAccounts[0], to: personal.
listAccounts[1], value: 100})
"0x479c9d07361bf60c97c45ee286c188d59953b60499ee62eb2033f98a7ab905b9"
```

We can also now query information about the transaction that we executed earlier. Remember how it returned a transaction hash? We can use that to find out details about the transaction:

```
eth.getTransactionReceipt("0x479c9d07361bf60c97c45ee286c188d59953b60499ee62eb2
033f98a7ab905b9")
```

This will produce the output shown as follows:

```
{
blockHash: "0xe90f08d4836e58689b59788d3f8f554f859bbc8b122b574d9493ddae778c7c6f",
blockNumber: 189,
contractAddress: null,
cumulativeGasUsed: 42000,
effectiveGasPrice: 1000000000,
```

```
  from: "0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
gasUsed: 21000,
  logs: [],
logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000",
  root: "0xd71daf69bdee5d1aaaa0ea1bedaa3019967b965b865a747169fa13124de7706b",
  to: "0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b",
transactionHash:
"0x479c9d07361bf60c97c45ee286c188d59953b60499ee62eb2033f98a7ab905b9",
transactionIndex: 1,
  type: "0x0"
}
```



If you see no transaction receipt, make sure mining is running, issue `miner.start()` to start mining. The block containing this transaction must have been mined in order for receipts to be produced.

Notice the `root` in the preceding output, the transaction root, which will be available in the block header as the Merkle root of the transaction trie. Similarly, we can query more information about the transaction using the `getTransaction` method:

```
> eth.getTransaction("0x479c9d07361bf60c97c45ee286c188d59953b60499ee62eb20
33f98a7ab905b9")
```

This will produce the output shown as follows:

```
{
blockHash: "0xe90f08d4836e58689b59788d3f8f554f859bbc8b122b574d9493ddae778c7c6f",
blockNumber: 189,
  from: "0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
  gas: 21000,
  gasPrice: 1000000000,
  hash: "0x479c9d07361bf60c97c45ee286c188d59953b60499ee62eb2033f98a7ab905b9",
  input: "0x",
  nonce: 3,
  r: "0x4b451415fe581c9ad4b7413f0b2a08128133ae35265f9683b402ceb2486b09b8",
  s: "0x524cb5ee0fa1496c7caf593f8a47cbbcf499987a26208e9657d75ad7de8a7371",
```

```
  to: "0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b",
transactionIndex: 1,
  type: "0x0",
  v: "0x648",
  value: 100
}
```

In this section, we covered how to start mining on a private network and ran some transactions. We also saw how transactions can be created, balance can be queried, and we performed an Ether transfer transaction from one account to another. We also covered how the transaction results can be queried using a transaction receipt and other relevant methods available via RPC in the Geth client.

We've seen in the examples previously that there is a rich interface available with hundreds of methods to interact with the blockchain. While this console- and terminal-based mechanism is quite useful, it can become a bit difficult to manage and deploy smart contracts using the command-line console. For this, we need better alternatives, which we will discuss next.

Introducing Remix IDE

There are various **Integrated Development Environments** (**IDEs**) available for Solidity development. Most of the IDEs are available online and are presented via web interfaces. Remix (formerly Browser-Solidity) is the most commonly used IDE for building and debugging smart contracts. It is discussed here.

Remix is a web-based environment for the development and testing of contracts using Solidity. It is a feature-rich IDE that does not run on a live blockchain; in fact, it is a simulated environment in which contracts can be deployed, tested, and debugged. It is available at `https://remix.ethereum.org`.

An example interface is shown as follows:

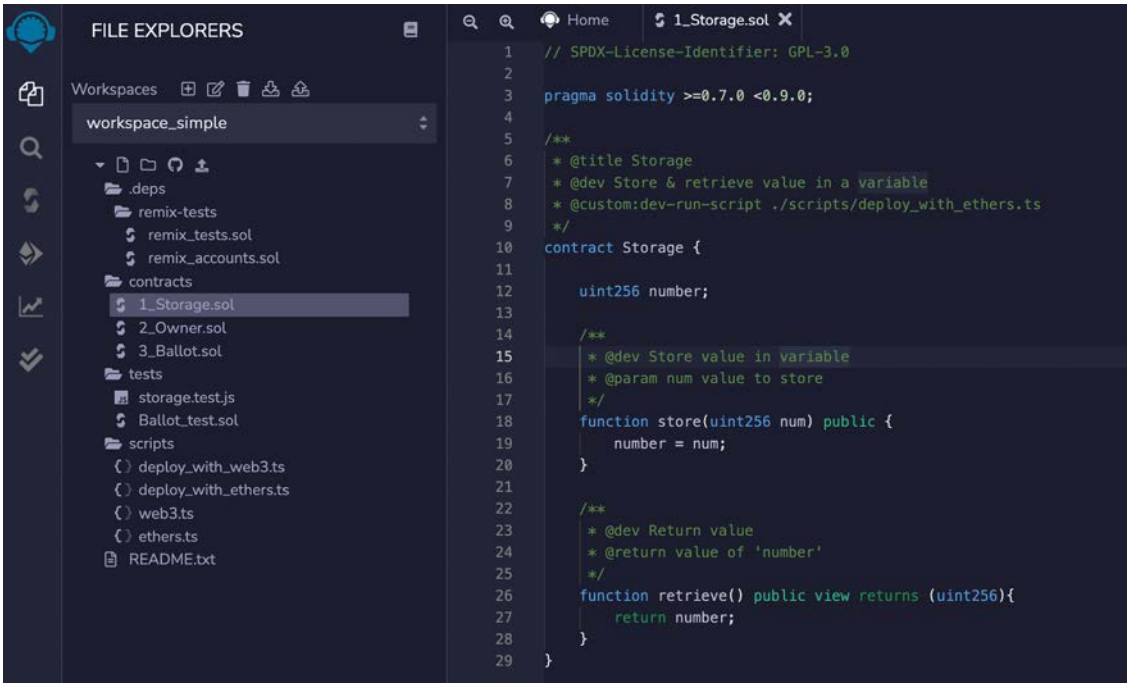


Figure 10.3: Remix IDE

On the left-hand side, there is a column with different icons. These icons represent various plugins of the Remix IDE. When you run Remix for the first time, it won't show any of the plugins. In order to add plugins to Remix IDE, you need to access the plugin manager to activate the plugins you need.

There are a number of plugins available, including but not limited to a Ganache provider, solidity static analysis, and a contract deployer, just to name a few; there are many others. They can be accessed by clicking on the plugin icon in the left bottom corner of the IDE.

Once activated, the plugins will appear in the leftmost column of the IDE, as shown in the following image. Again, note that this will only show the plugins that are activated, and other plugins or local plugins can be activated as required.

On the right-hand side, there is a code editor with syntax highlighting and code formatting, and on the left-hand side, there are a number of plugins available that can be used to deploy, debug, test, and interact with the contract.

Various features, such as transaction interaction, options to connect to the JavaScript VM, the configuration of an execution environment, a debugger, formal verification, and static analysis are available. They can be configured to connect to execution environments such as the JavaScript VM, injected Web3—where Mist, MetaMask, or a similar environment has provided the execution environment—or the Web3 provider, which allows connection to the locally running Ethereum client (for example, `geth`) via IPC or RPC over HTTP (a Web3 provider endpoint).

Remix also has a debugger plugin for EVM that is very powerful and can be used to perform detailed level tracing and analysis of the EVM bytecode. An example is shown here:

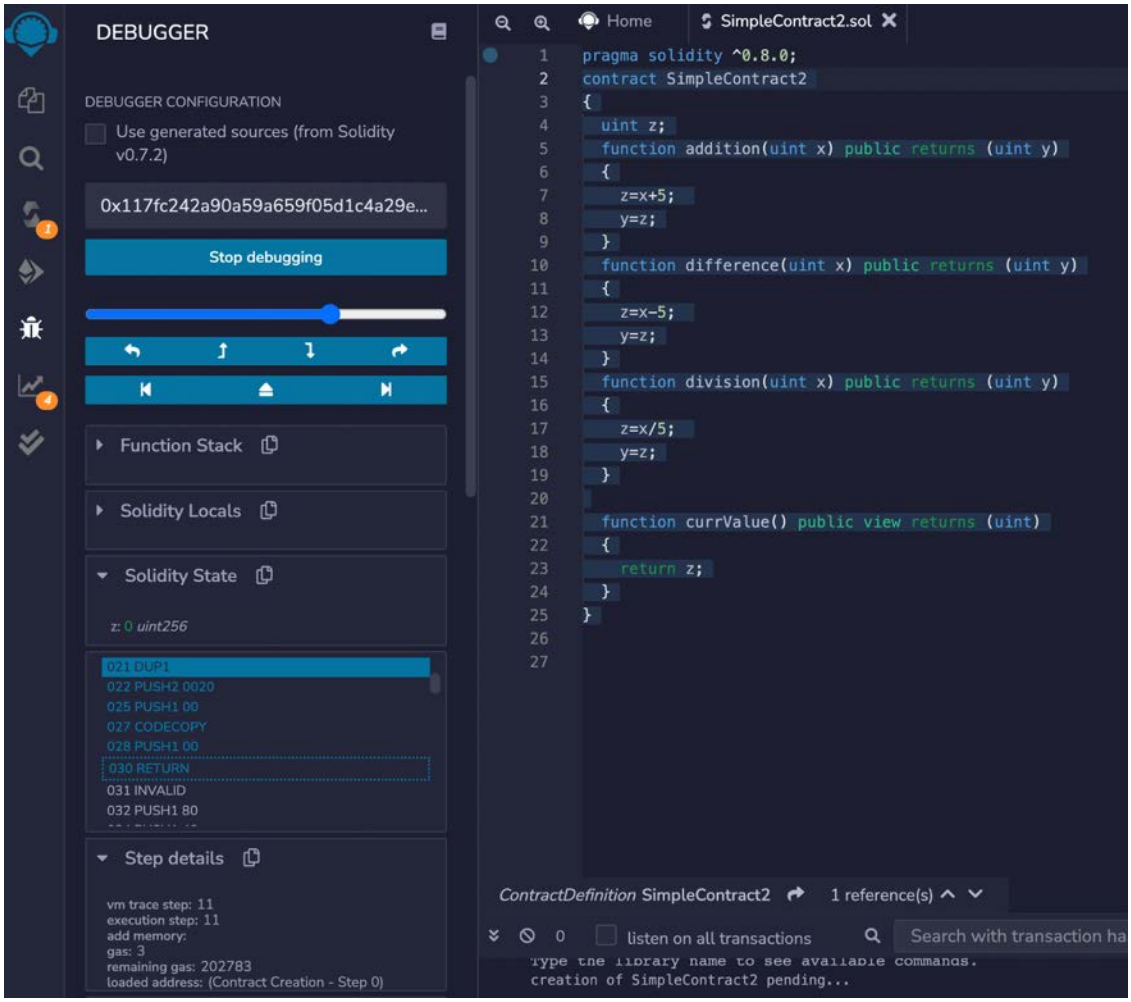


Figure 10.4: Remix IDE, debugging

The preceding screenshot shows different elements of the Remix IDE when **DEBUGGER** is running. The debugger has the source code decoded into EVM instructions. The user can step through the instructions one by one and examine what the source code does when executed.

On the right-hand side, the source code is shown. Below that is the output log, which shows informational messages and data related to compilation and the execution status, and transaction information of the transaction/contract.

In the debugger, the opcodes are displayed, along with memory, state, and stack information, which makes debugging easier and intuitive. This is a very useful feature and comes in handy, especially in complex code debugging.

In the next section, we'll be using MetaMask, a browser extension that serves as a cryptocurrency wallet and an interface to blockchains and dApps.

Interacting with the Ethereum Blockchain with MetaMask

MetaMask allows interaction with the Ethereum blockchain via the Firefox and Chrome browsers. It injects a `web3` object within the running website's JavaScript context, which allows immediate interface capability for dApps. This injection allows dApps to interact directly with the blockchain.



Further information is available at `https://github.com/MetaMask/metamask-plugin`.

MetaMask also allows account management and records all transactions for these accounts. This acts as a verification method before any transaction is executed on the blockchain. The user is shown a secure interface to review the transaction for approval or rejection before it can reach the target blockchain.

It allows connectivity with various Ethereum networks. An interesting feature to note is that MetaMask can connect to any Ethereum blockchain via the custom RPC connection. It can connect to not only remote blockchains but also to locally running blockchains. All it needs is an RPC connection exposed by a node running on the blockchain. If it's available, MetaMask can connect and will allow a web browser to connect to it via the `web3` object. MetaMask can also be used to connect to a locally running test (or simulated) blockchain like Ganache and TestRPC.

Installing MetaMask

MetaMask runs as a plugin or add-on in the web browser. It is available for Chrome, Firefox, Opera, and Brave browsers. The key idea behind the development of MetaMask is to provide an interface with the Ethereum blockchain. It allows efficient account management and connectivity to the Ethereum blockchain without running the Ethereum node software locally. MetaMask allows connectivity to the Ethereum blockchain through the infrastructure available at Infura (`https://infura.io`). This allows users to interact with the blockchain without having to host any node locally.

Here, we'll go over the installation process. We are using Chrome for this example; however, instructions are the same for Firefox and Brave:

1. Browse to `https://metamask.io/`, where links to the relevant source are available to download the extension for your browser.

2. Install the extension for the browser. It will install quickly, and if all goes well, you will see a message saying `Welcome to MetaMask`.
3. Click on **Get Started**, and it will show two options, either to import an already existing wallet using the seed, or to create a new wallet. We will select **Create a Wallet** here.
4. Next, create a **Password**.
5. Optionally, set up a **Secret Backup Phrase**.
6. Once everything has been completed, you will see a congratulatory message.
7. After clicking on the **All Done** button, the MetaMask main view will open.
8. After the setup is completed, you will see the main accounts view.

Once installed, the Ethereum provider API will be available in the browser context, which can be used to interact with the blockchain. MetaMask injects a global API into websites at `window.ethereum`. An example of the Ethereum API is shown here using the JavaScript console in Chrome:

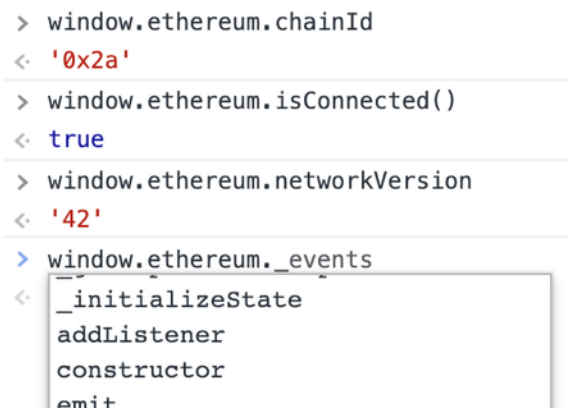


Figure 10.5: Ethereum object

MetaMask can connect to various networks. By default, it connects to the Ethereum main net. Other networks include, but are not limited to, the **Sepolia** test network, and the **Goerli** test network. In addition, it can connect to any local node via `Localhost 8545`, and to a custom RPC, which means that it can connect to any network as long as RPC connection information is available.

Now, let's see how account management works in MetaMask.

Creating and funding an account with MetaMask

Let's now create an account and fund it with some ETH, all using MetaMask. Note that we are connected to the Sepolia test network. Open MetaMask and ensure that it is connected to the Sepolia test network:

1. Open the MetaMask extension and click on the circle in the top-right corner. The menu will open; click **Create Account**.
2. Enter the new account name, `SepoliaTestAccount`, and click on **Create**, which will immediately create a new account.

   Once created, we can fund the account with the following steps.

3. Copy the account address to the clipboard.
4. Go to `https://sepolia-faucet.pk910.de` and enter the `SepoliaTestAccount` address. You will receive some ETH in your account.



An alternative faucet can be found at `https://sepolia-faucet.pk910.de/`. If you don't see any faucet working, then simply search for `sepolia testnet faucet` on your favorite internet search engine.

Now, notice in MetaMask that some ETH is available.

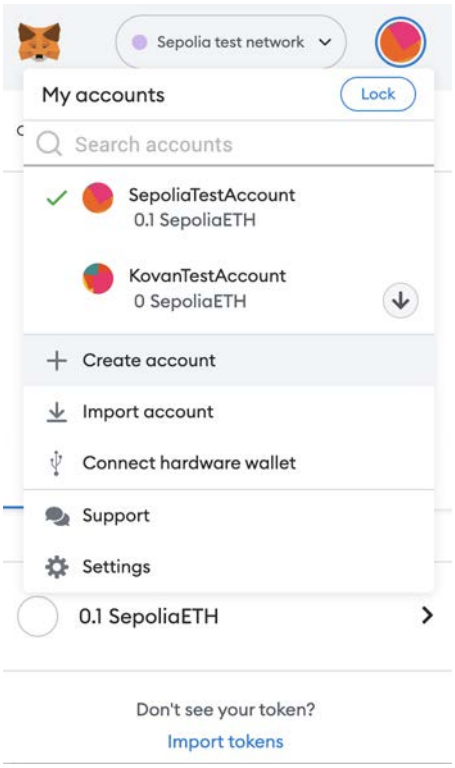


Figure 10.6: MetaMask My accounts menu

5. Now create another account and transfer some ETH from `SepoliaTestAccount` to the new account. You can do this by taking the following steps:
6. Click on the send icon.
7. Select the target account from the **Transfer between my accounts** option.
8. Press next.
9. Click the **EDIT** option item in the top right-hand corner to edit the priority for the transaction, either low, medium, or high.
10. Further click on **Advanced options** to change the gas limit (the gas limit is the maximum units of gas you are willing to use).



Units of gas are can be specified for "Max priority fee" and "Max fee". Max priority fee (aka the "miner tip") goes directly to miners and incentivizes them to prioritize your transaction. You'll most often pay your max setting. Max fee is the most you'll pay (base fee + priority fee). We usually don't need to change any of the settings, but to control the transaction fee and transaction speed, these options are adjusted accordingly. The higher the miner tip, the greater the transaction speed because miners will pick it up because of higher transaction fees.

11. Confirm the transaction.
12. Now, notice that the Ether has been sent to the other account.

Also note the detailed logs and information pertaining to the transaction by clicking on the transaction. Detailed information can also be viewed in the relevant block explorer by clicking the **View on block explorer** option, which takes to this link to the transaction summary on etherscan: `https://sepolia.etherscan.io/tx/0x96e3688cba95c68c2922453872a3fd1d98060b5884f74866f69bda1db8f610cd`

As we now understand what MetaMask is, we can now delve deeper and see how the Remix IDE can deploy smart contracts through MetaMask.

Using MetaMask and Remix IDE to deploy a smart contract

As MetaMask injects a `web3` object into the browser, we can use it with Remix IDE to deploy contracts to the blockchain. It is very easy to deploy new contracts using MetaMask and Remix. Remix IDE provides an interface where contracts can be written in solidity and then deployed onto the blockchain.

We will use Remix IDE and MetaMask to deploy a sample smart contract to the locally running private blockchain that we created earlier in the chapter. In the exercise, a simple contract that can perform various simple arithmetic calculations on the input parameter will be used. As we have not yet introduced **Solidity**, the aim here is to demonstrate the contract deployment and interaction process only.

More information on coding and Solidity will be provided later in *Chapter 11, Tools, Languages, and Frameworks for Ethereum Developers*, after which the following code will become easy to understand. Those of you who are already familiar with JavaScript or any other similar language such as the C language will find the code almost self-explanatory.

Now, let's look at some examples of how MetaMask can be used in practice with the private net that we created earlier in this chapter.

Adding a custom network to MetaMask and connecting it with Remix IDE

First, ensure that you have MetaMask available as we set it up earlier. In this section, we will add our local private network in MetaMask and then interact with it using Remix IDE.

1. Open the Google Chrome web browser, where MetaMask is installed. Select **Localhost 8545**, where the Geth instance of our private net is listening on:

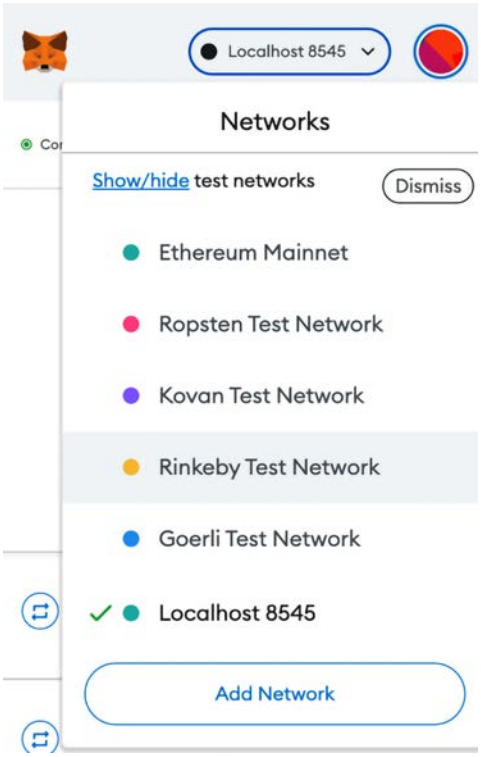


Figure 10.7: MetaMask network selection

2. Navigate to Remix IDE for Ethereum smart contract development on your browser at `https://remix.ethereum.org`.

3. Once on the website, notice the **DEPLOY & RUN TRANSACTIONS** option in the left-hand column. Choose **Injected Web3** as the **ENVIRONMENT**, and a MetaMask window will open:

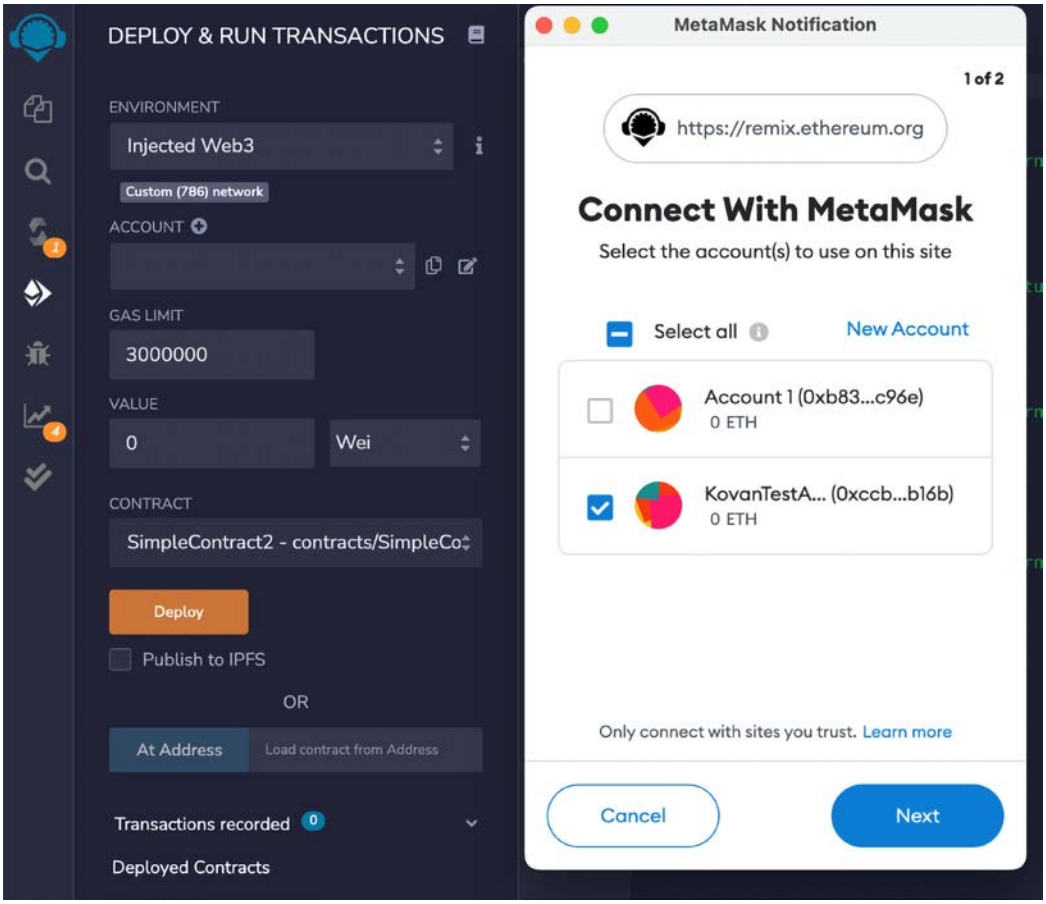


Figure 10.8: Remix IDE and interaction with MetaMask

Note that when Remix IDE is connected to MetaMask, it will show network information, such as **Custom (786) network**, and account information, as shown in the following screenshot.

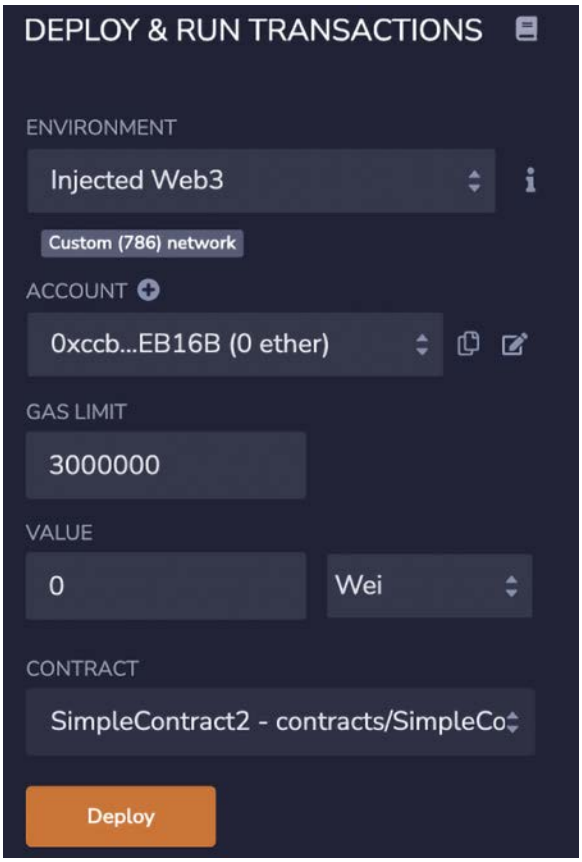


Figure 10.9: Network information and account information in Remix

In this section, we saw how Remix can connect to MetaMask using the injected Web3 environment. MetaMask can be connected to the main net or any other network (a local or test network) but from Remix's point of view, as long as the injected Web3 object is available, it will connect to it.

In our example, as we are using a private network, we have connected to our private network through MetaMask, and Remix is connecting to MetaMask using the injected Web3 environment. Now even if we have this connectivity available, MetaMask does not know about the accounts that we have created in our private network. In order for MetaMask to operate on existing accounts, we need to import them from the existing keystore, in our case, the private network 786 keystore. We will see how this is done in the next section.

Importing accounts into MetaMask using keystore files

We can import existing Ethereum accounts into MetaMask by importing the keystore files generated by Geth as a result of creating accounts. To demonstrate how this works we will now import the accounts from our private network blockchain, named `786`, into MetaMask.

In MetaMask, the **Import Account** option is available under the **My Accounts** menu, as shown in the following screenshot.

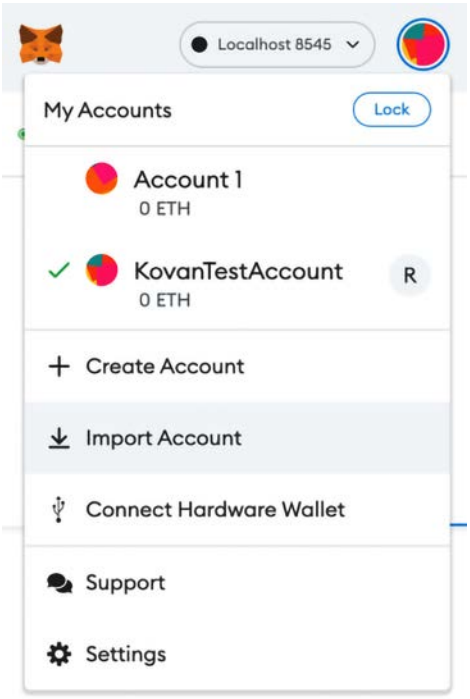


Figure 10.10: MetaMask Import Account option

Now let's import the accounts. Remember we created two accounts earlier in our private network `786`:

```
>eth.accounts
["0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f","0x0f044cb4a0f9
24b6cfcf07c6c57945a0af75ec5b"]
```

We can import these accounts into MetaMask using their associated keystore JSON files.

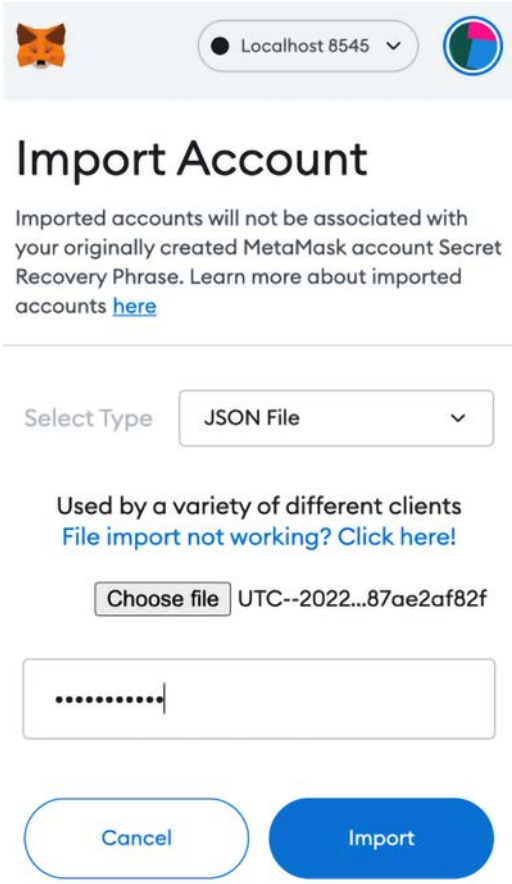


Figure 10.11: MetaMask JSON import file

The steps are as follows:

1. First, choose the JSON file from the private net `keystore` directory, `~/etherprivate/keystore`. The `keystore` files for our private net are listed here:

```
UTC--2022-05-21T16-16-47.193932000Z--6e94bdb15141491bc3b9de3a9cab9d87ae2
af82f
UTC--2022-05-21T16-28-12.270088000Z--0f044cb4a0f924b6cfcf07c6c57945a0af7
5ec5b
```

2. Simply browse to the key store and select the `keystore` file, then enter the password (earlier, when we created the accounts for the first time, we set the password as `Password123`), and click **Import**. It may take a few seconds to import. When imported, the account will be visible in the MetaMask window as shown here:

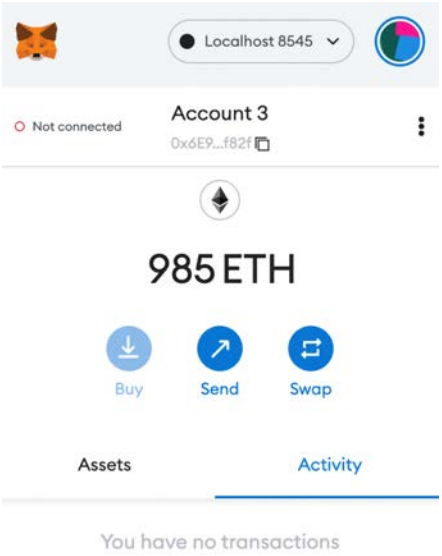


Figure 10.12: Account imported in MetaMask

3. Similarly, we can import the other account, by selecting the `keystore` file and importing it. Finally, we will have two accounts listed in MetaMask, as shown in the following image:

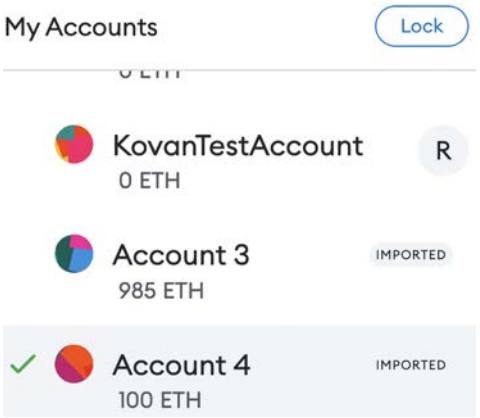


Figure 10.13: MetaMask—two accounts imported

Now that our accounts have been imported successfully, we can move on to using MetaMask to deploy a contract in our private network.

Deploying a contract with MetaMask

In this section, we will write a simple smart contract and deploy it on our private network using MetaMask. First, in Remix, we create a new file. Enter the code shown for `SimpleContract2` as follows:

```
pragma solidity 0.5.0;
contract SimpleContract2
{
uint z;
function addition(uint x)public returns(uint y)
{
    z=x+5;
    y=z;
  }
function difference(uint x)public returns(uint y)
{
    z=x-5;
    y=z;
  }
function division(uint x)public returns(uint y)
{
    z=x/5;
    y=z;
  }

function currValue()public view returns(uint)
{
return z;
  }
}
```

The code will look like this in Remix.

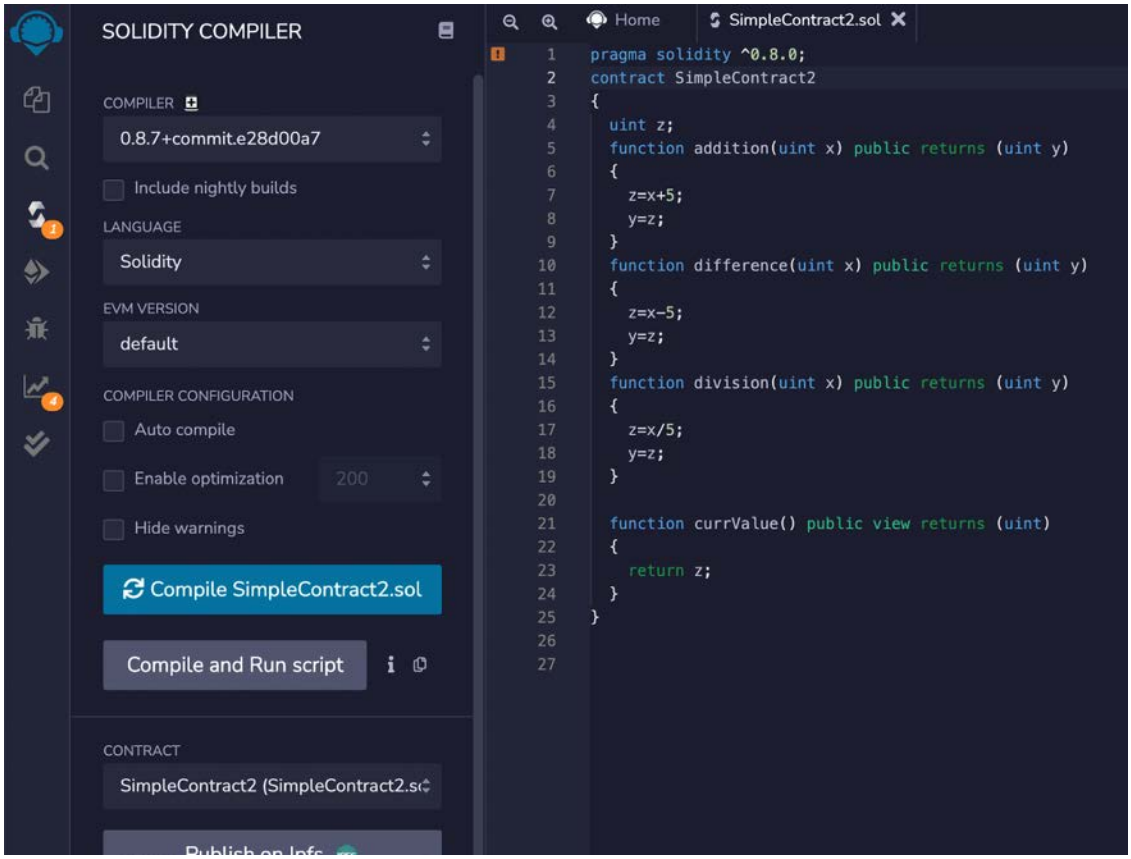


Figure 10.14: SimpleContract2 in Remix

Now we compile the smart contract `SmartContract2` by clicking on the **Compile SimpleContract2.sol** button, as shown above.

Once compiled successfully, we deploy the smart contract in the private net:

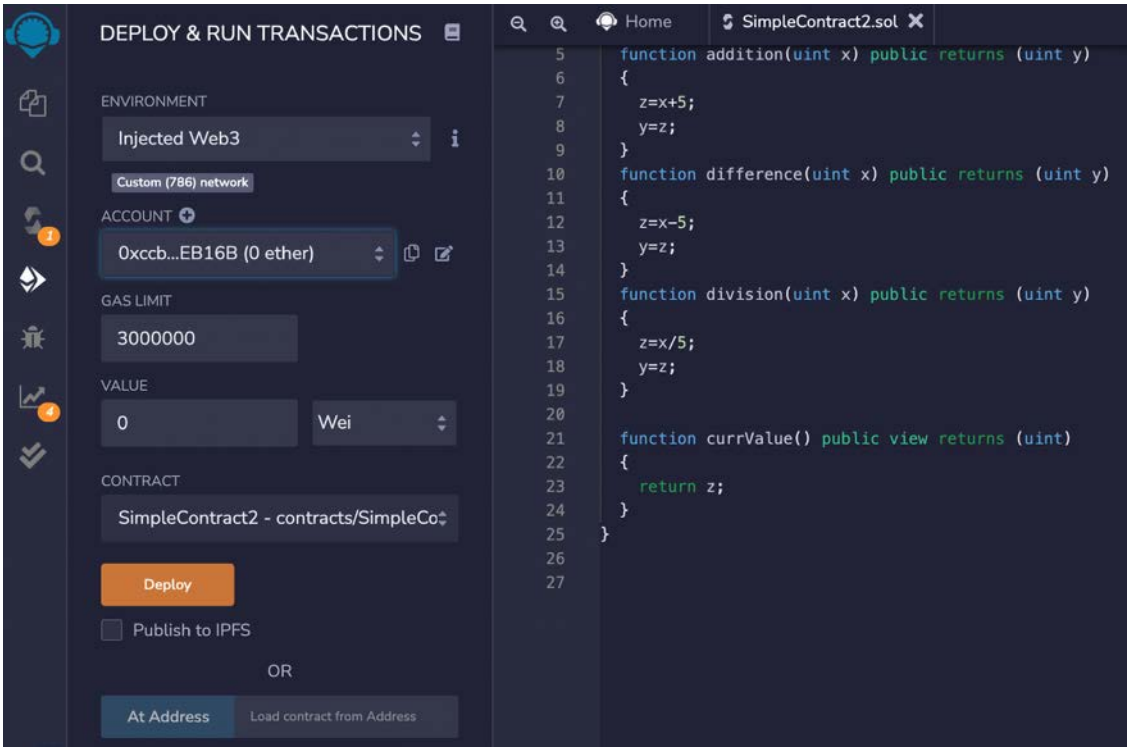


Figure 10.15: Smart contract deployment

Click **Deploy**, and review the contract details. In the **DATA** tab, you can see the contract code. Then click on the **Confirm** button.

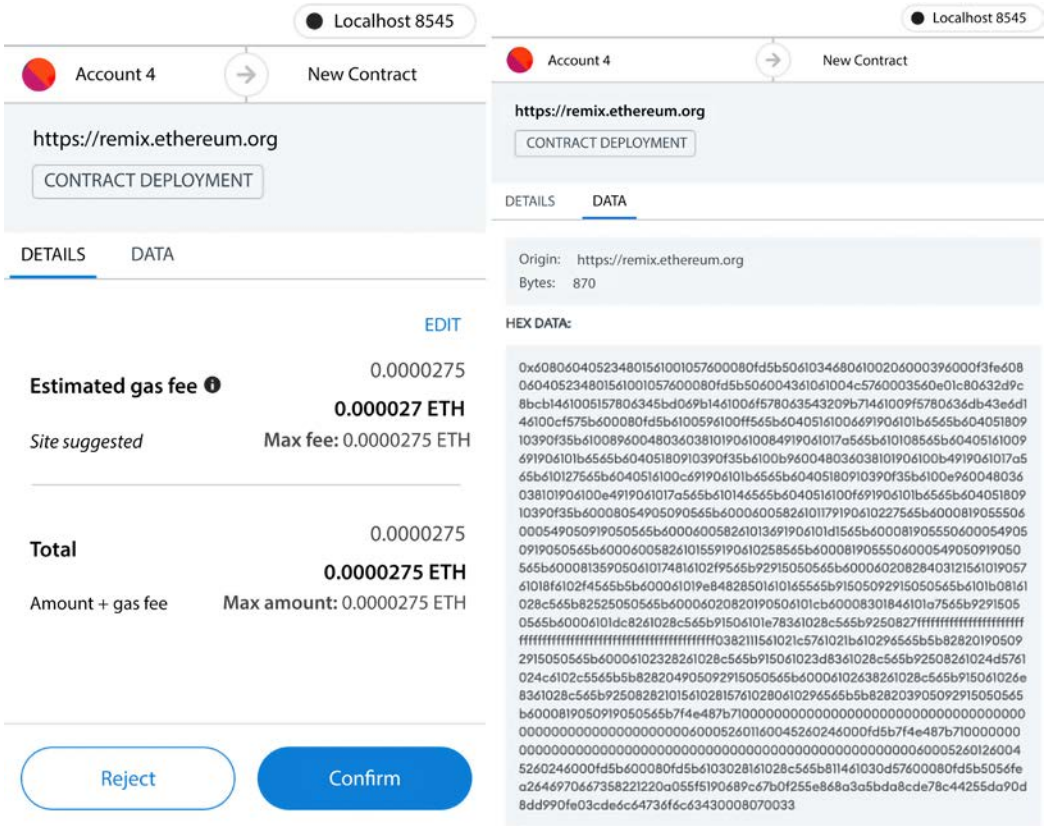


Figure 10.16: MetaMask contract deployment

Notice how in the Geth logs, we see `Submitted contract creation`, which means that a contract creation transaction has been submitted and acknowledged as a result of the deploy action from the Remix IDE and the **Confirm** action from MetaMask. Note that it also shows the full hash and contract address:

```
INFO [05-21|19:21:39.357] Submitted contract creation
hash=0xb2b13506270dcfbc5256b2d61944b3a6240cfe9b9067d64fda61f82a1f200e5b
from=0x0F044cb4A0F924b6CfCF07c6c57945A0AF75Ec5b nonce=1
contract=0x0810f73FF422Dc7D6c27BCee611dc3F325EA9030 value=0
```

Now we start mining again, by issuing the following command in the JavaScript console of Geth. If mining is already running, this step is not required:

```
> miner.start()
null
```

Once the contract is mined and deployed, you can see it in Remix, under the **Deployed Contracts** view:

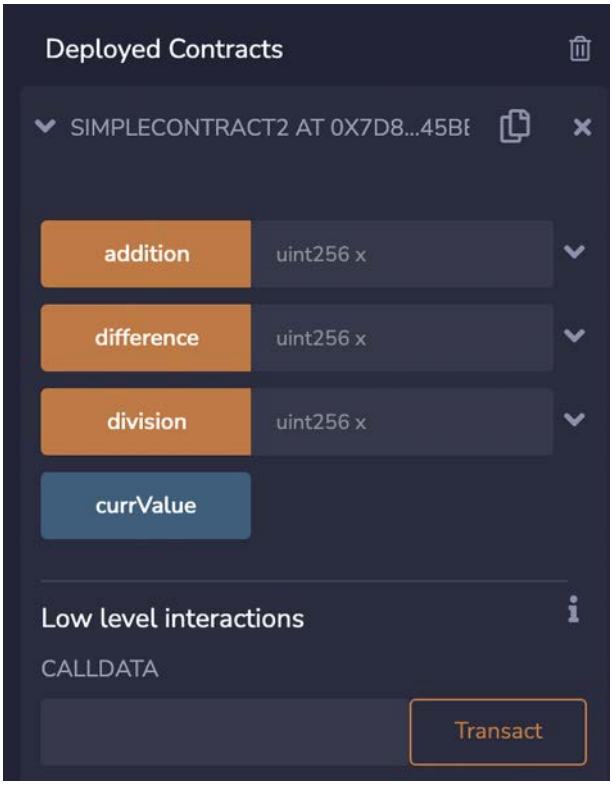


Figure 10.17: MetaMask deployed contract

With this, we have successfully deployed our example smart contract on the private network. Next, we'll see how we can interact with this contract using Remix IDE and MetaMask.

Interacting with a contract through MetaMask using Remix IDE

Now we can interact with the contract using MetaMask. We run the **addition** option, enter a value of `100` in the box, and click on the **addition** button, as shown in the following screenshot, on the left-hand side.

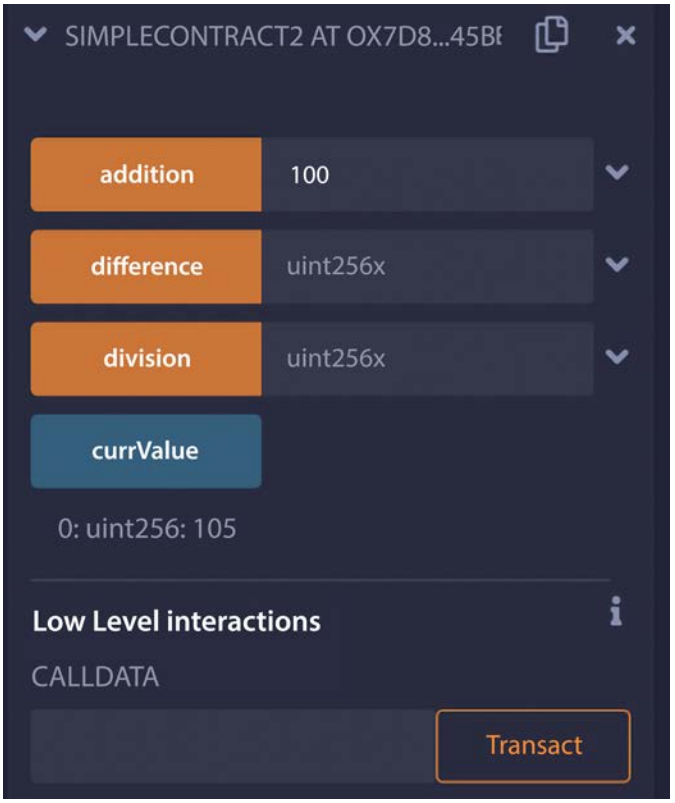


Figure 10.18: Smart contract functions

This will invoke the MetaMask window, as shown in the following screenshot, on the right-hand side.

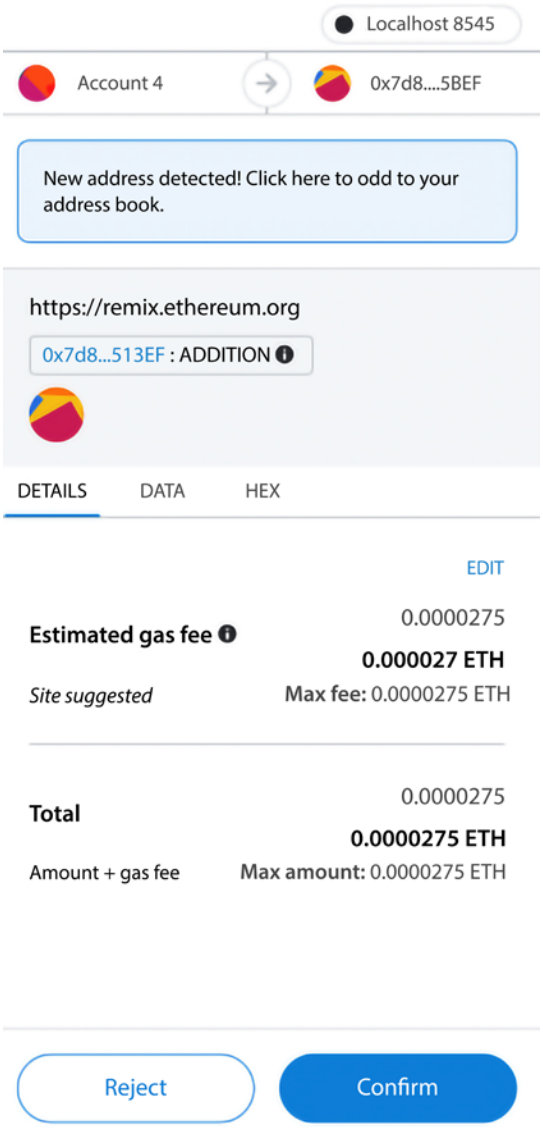


Figure 10.19: MetaMask contract interaction

Click **Confirm**, and the transaction will be mined as usual because we have a miner running the background in Geth.

Now click on the **currValue** button, which will read the contract to find the current value, which is, as expected, `105`. This value is the result of the addition operation being performed.

Recall that we provided a value of `100`, and in the smart contract we have already hardcoded `5` in the `addition` function, as shown here:

```
function addition(uint x)publicreturns(uint y)
{
    z=x+5;
    y=z;
  }
```

This means that `100` provided by us as input is added to `5`, which is hardcoded in the smart contract `addition` function, and the result of this calculation is `105`, as expected.

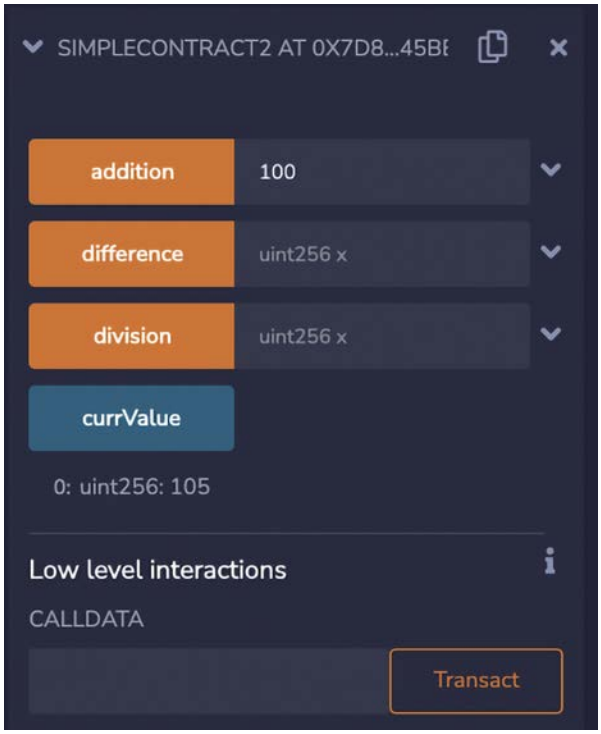


Figure 10.20: Retrieve the current value from the smart contract

This is a very simple program, but the aim of this exercise is to demonstrate the use of MetaMask and how it can connect with the local private chain, using the `localhost` RPC provider running in Geth on port `8545`.

Pay special attention to the Solidity compiler settings. Choose the appropriate compiler version and the EVM version according to your source code and your Geth client's EVM version. If this is not selected appropriately, you can run into issues. For example, if an incorrect version of EVM is chosen, then when interacting with the contract, you may see a **Gas estimation failed** error.

Similarly, if an incorrect EVM version is chosen, then you will also see an error message in the Remix window. It's best to configure the settings correctly, which can just default in most cases, unless there are specific features of an EVM version that you are testing. Compiling for the wrong version of the EVM can result in incorrect and failing behavior. For example, if you have chosen an outdated EVM version, then even if the compilation is successful, the bytecode at runtime may fail to run correctly and will report an error saying that the `opcode is invalid`. In some other cases, the runtime may even fail silently or with little indication in the Geth logs about what happened, leading to hours of wasted time in debugging. This is especially true in private networks; therefore, make sure that the correct EVM version that matches the Geth client release is used.

Also notice that the compiler version in the Remix IDE is set to the version of EVM specified in the Solidity source code in the line `pragma solidity 0.8.0`. Also, the EVM version should be left to default, which is the latest EVM release.

In the MetaMask view, you can also see the transaction history of the contract by clicking on the activity tab:

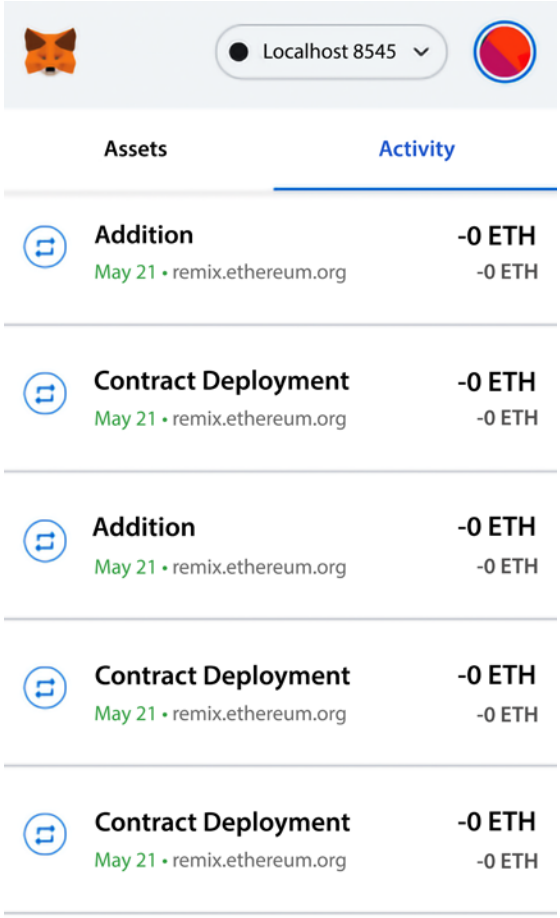


Figure 10.21: Transaction history in MetaMask

Another way to connect to the local private net (Geth node) is to directly connect using the Web3 provider, available under **DEPLOY & RUN TRANSACTIONS**. This will allow direct communication with the blockchain through Remix via RPC without requiring an injected Web3 environment:

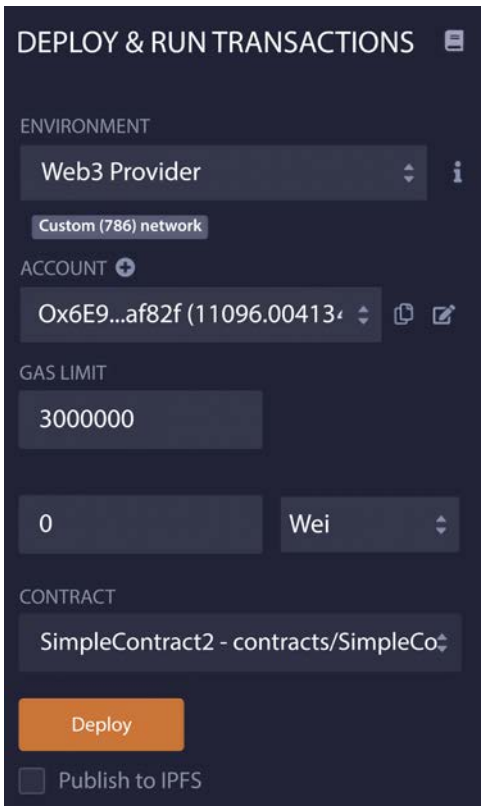


Figure 10.22: Deploy and run transactions in Remix IDE

Notice in the preceding screenshot that **ENVIRONMENT** is set to **Web3 Provider**, which simply connects to `http://localhost:8545`.

One advantage of this method is that it is a quick and easy method to connect Remix with the underlying blockchain (the Geth node). There is no need for any additional tools; all you need is Remix IDE (which is also browser-based) and your local running Geth node. The disadvantage, however, is that this method is not very secure. In order to connect Remix IDE to a local node, you can run the Geth client using the `--http.corsdomain` flag, as shown in the screenshot below:

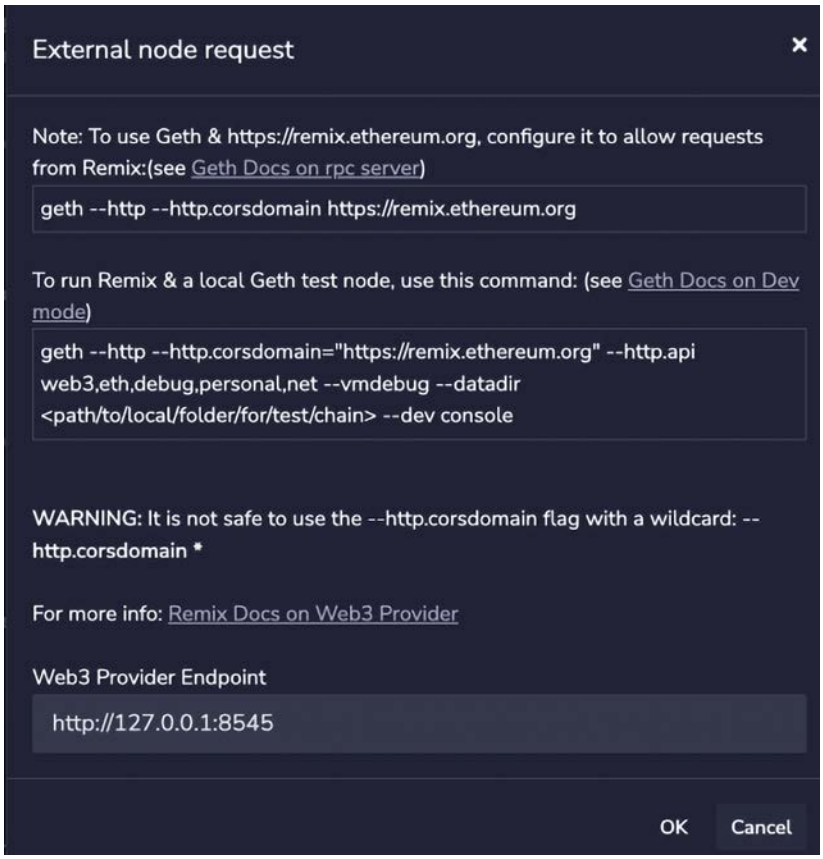


Figure 10.23: Remix Web3 provider

Notice that our Geth process is running on the private network with the required parameters already, which allows Web3 provider connection, as shown here. Notice the `--http.corsdomain` flag in *Figure 10.22* is used here. If the `c` process is not already running, issue the following command:

```
$ geth –datadir ~/etherprivate/ --networkid 786 --http --http.api
'web3,eth,net,debug,personal' --http.corsdomain '*'
```



As we are using a private network our command looks different from what is shown previously but makes use of the `--http.corsdomain` flag, which is essential to allow connectivity of the local Geth node with Remix.

Notice that in the preceding command, we have the `–-http.corsdomain` flag set. If we didn't have this flag set, then the only option would have been to connect via injected Web3, which in fact is a more secure way to connect to the chain using MetaMask. As such, we use that as a standard, but if you have a simple local test network, you can always use the **Web3 Provider** option to quickly interact with the blockchain using Remix IDE.

Using the flag `--http.corsdomain` with the wildcard (`--http.corsdomain '*'`) can be used with a local test chain with test accounts. However, it is not advisable on public networks, as the wildcard * will allow everyone to connect to the node. Therefore, it is recommended that the access is restricted by specifying only trusted URLs.

Now, we can stop the miner; recall that this is the miner we started up earlier when setting up the private network:

```
> miner.stop()
null
```

With this example, we have now covered how MetaMask can be used to connect to a local network and how Remix and MetaMask, or only Remix, can be used to connect to the local network and deploy smart contracts. We also saw how to interact with a deployed contract.

Remix is a very feature-rich IDE and we have not covered everything; however, this chapter has formally introduced you to Remix, describing most of the main features in detail, and we will keep using it in the chapters to come.

It is also useful to have a mechanism to view a consolidated list and details of all the transactions in the blockchain. For this purpose, block explorers are used. There are many services available for public blockchains on the internet. There are many open source projects available as well. As there are many choices we are not going to describe a specific one here; they all work in more or less the same way. Do a search on GitHub at `https://github.com/search?q=blockchain+explorer` to find an appropriate one and explore further.

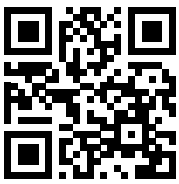
Summary

In this chapter, we have explored Ethereum test networks and how to set up private Ethereum networks. After the initial introduction to private network setup, we also saw how the Geth command-line tool can be used to perform various functions and how we can interact with the Ethereum blockchain. We also saw how MetaMask and Remix can be used to deploy smart contracts.

In the next chapter, we will see in greater detail what tools, programming languages, and frameworks are available for the development of smart contracts on Ethereum.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

11

Tools, Languages, and Frameworks for Ethereum Developers

This chapter is an introduction to the tools, languages, and frameworks used for Ethereum smart contract development. We will examine different methods of developing smart contracts for the Ethereum blockchain. We will discuss various constructs of the **Solidity** language in detail, which is currently the most popular development language for smart contract development on Ethereum.

In this chapter, we will cover the following topics:

- Languages
- Compilers
- Tools and libraries
- Frameworks
- Contract development and deployment
- The Solidity language

There are a number of tools available for Ethereum development, including clients, IDEs, and development frameworks.

The content of this chapter does not include all frameworks and tools that are out there for development on Ethereum. It shows the most commonly used tools and frameworks, including some that we will use in our examples in the following chapter.



There are a number of resources available related to development tools for Ethereum at the following address: `http://ethdocs.org/en/latest/contracts-and-transactions/developer-tools.html#developer-tools`

Since we have discussed some of the main tools available in Ethereum in previous chapters, such as Remix IDE and MetaMask, this chapter will focus mainly on Solidity, Ganache, `solc`, and Truffle. Some other elements, such as prerequisites (Node.js), will also be introduced briefly.

We'll start by exploring some of the programming languages that can be used in the Ethereum blockchain.

Languages

Smart contracts can be programmed in a variety of languages for the Ethereum blockchain. There are three main languages that can be used to write contracts, and three that are of historical interest:

- **Solidity**: This language has now become a standard for contract writing for Ethereum. This language is the focus of this chapter.
- **Vyper**: This language is a Python-like experimental language that is being developed to bring security, simplicity, and auditability to smart contract development.
- **Yul**: This is an intermediate language that has the ability to compile to different backends such as EVM and `ewasm`. The design goals of Yul mainly include readability, easy control flow, optimization, formal verification, and simplicity.
- **Mutan**: This is a Go-style language, which was deprecated in early 2015 and is no longer used.
- **LLL**: This is a **Low-level Lisp-like Language**, hence the name LLL. This is also no longer used.
- **Serpent**: This is a simple and clean Python-like language. It is not used for contract development anymore and is not supported by the community.

As Solidity code needs to be compiled into bytecode, we need a compiler to do so. In the next section, we will introduce the Solidity compiler.

The Solidity compiler

Compilers are used to convert high-level contract source code into the format that the Ethereum execution environment understands. The Solidity compiler, `solc`, is the most common one in use.

`solc` converts from a high-level Solidity language into **Ethereum Virtual Machine** (**EVM**) bytecode so that it can be executed on the blockchain by the EVM.

Installing solc

`solc` can be installed on a Linux Ubuntu operating system using the following command:

```
$ sudo apt-get install solc
```

If **Personal Package Archives** (**PPAs**) are not already installed, those can be installed by running the following commands:

```
$ sudo add-apt-repository ppa:ethereum/ethereum
$ sudo apt-get update
```

To install `solc` on macOS, execute the following commands:

```
$ brew tap ethereum/ethereum
```

This command will add the Ethereum repository to the list of `brew` formulas:

```
$ brew install solidity
```

This command will produce a long output and may take a few minutes to complete. If there are no errors produced, then eventually it will install Solidity.

In order to verify that the Solidity compiler is installed and to validate the version of the compiler, the following command can be used:

```
$ solc --version
```

This command will produce the output shown as follows, displaying the version of the Solidity compiler:

```
solc, the solidity compiler commandline interface
Version: 0.8.11+commit.d7f03943.Darwin.appleclang
```

This output confirms that the Solidity compiler is installed successfully.

Experimenting with solc

`solc` supports a variety of functions. A few examples are shown as follows. As an example, we'll use a simple contract, `Addition.sol`:

```
pragma solidity ^0.8.0;
contract Addition
{
  uint8 x;
  function addx(uint8 y, uint8 z ) public
  {
    x = y + z;
  }
  function retrievex() view public returns (uint8)
  {
    return x;
  }
}
```

In order to see the smart contract in compiled binary format, we can use the following command:

```
$ solc --bin Addition.sol
```

This command will produce an output similar to the following:

```
======= Addition.sol:Addition =======
Binary:
```

```
6080604052348015610010576000080fd5b506101008061002060000396000f3fe608060
4052348015600f57600080fd5b506004361060325760003560e01c806336718d801460
37578063ac04e0a0146072575b600080fd5b607060048036036040811015604b576000
80fd5b81019080803560ff16906020019092919080356 0ff1690602001909291905050
506094565b005b607860b4565b604051808260ff1660ff16815260200191505060405 1
80910390f35b8082016000806101000a81548160ff021916908360ff16021790555050
50565b6000806000905490610100a900460ff1690509056fea2646970667358221220
20bae3e7dea36338ad4ced23dee3370621e38b08377e773854fcc1b22260924d64736f
6c63430006010033
```

This output shows the binary translation of the `Addition.sol` contract code represented in hex.

As a gas fee is charged for every operation that the EVM executes, it's a good practice to estimate gas before deploying a contract on a live network. Estimating gas gives a good approximation of how much gas will be consumed by the operations specified in the contract code, which gives an indication of how much ether is required to be spent in order to run a contract. We can use the `--gas` flag for this purpose, as shown in the following example:

```
$ solc --gas Addition.sol
```

This will give the following output:

```
======= Addition.sol:Addition =======
Gas estimation:
construction:
   147 + 100800 = 100947
external:
   addx(uint8,uint8):   infinite
   retrievex(): 2479
```

This output shows how much gas usage is expected for these operations in the `Addition.sol` smart contract. The gas estimation is shown next to each function. For example, the `retrieve()` function is expected to use `2479` gas.

We can generate the **Application Binary Interface** (**ABI**) using `solc`, which is a standard way to interact with the contracts:

```
$ solc --abi Addition.sol
```

This command will produce a file named `Addition.abi` as output. The following are the contents of the output file `Addition.abi`:

```
======= Addition.sol:Addition =======
Contract JSON ABI
[{"inputs":[{"internalType":"uint8","name":"y","type":"uint8"},
{"internalType":"uint8","name":"z","type":"uint8"}],"name":"addx",
"outputs":[],"stateMutability":"nonpayable","type":"function"},
```

```
{"inputs":[],"name":"retrievex","outputs":[{"internalType":"uint8",
"name":"","type":"uint8"}],"stateMutability":"view","type":"function"}]
```

The preceding output displays the contents of the `Addition.abi` file, which are formatted in JSON style. It consists of inputs and outputs along with their types. We will generate and use ABIs later in this chapter to interact with the deployed smart contracts.

Another useful command to compile and produce a binary compiled file along with an ABI is shown here:

```
$ solc --bin --abi -o bin Addition.sol
```

The message displays if the compiler run is successful; otherwise, errors are reported.

```
Compiler run successful. Artifact(s) can be found in directory bin.
```

This command will produce a message and two files in the output directory `bin`:

- `Addition.abi`: This contains the ABI of the smart contract in JSON format.
- `Addition.bin`: This contains the hex representation of the binary of the smart contract code.



The ABI encodes information about smart contracts' functions and events. It acts as an interface between EVM-level bytecode and high-level smart contract program code. To interact with a smart contract deployed on the Ethereum blockchain, external programs require an ABI and the address of the smart contract.

`solc` is a very powerful command and further options can be explored using the `--help` flag, which will display detailed options. However, the preceding commands used for compilation, ABI generation, and gas estimation should be sufficient for most development and deployment requirements.

Tools, libraries, and frameworks

There are various tools and libraries available for Ethereum. The most common ones are discussed here. In this section, we will first install the prerequisites that are required for developing applications for Ethereum.

Node.js

Node.js is a popular development platform for executing JavaScript code primarily on the backend server side; however, it can also be used for frontends.

As Node.js is required for most of the tools and libraries, we recommend installing it first. Node.js can be installed for your operating system by following the instructions on the official website: `https://nodejs.org/en/`

Ganache

At times, it is not possible to test on the testnet, and the mainnet is obviously not a place to test contracts. A private network can be time-consuming to set up at times. Ganache is a simulated personal blockchain with a command-line interface or a user-friendly GUI to view transactions, blocks, and relevant details. This is a fully working personal blockchain that supports the simulation of Ethereum and its different hard forks, such as **Homestead**, **Byzantium**, **Istanbul**, **Petersburg, or London**. It's available as a CLI and also a GUI.

ganache-cli

Ganache comes in handy when quick testing is required and no testnet is available. It simulates the Ethereum `geth` client behavior and allows faster development and testing. The Ganache command line is available via `npm` as a Node.js package. As such, Node.js should already have been installed and the `npm` package manager should be available. `ganache` can be installed using this command:

```
$ npm install -g ganache
```

In order to start the `ganache` command-line interface, simply issue this command, keep it running in the background, and open another terminal to work on developing contracts:

```
$ ganache
```

When Ganache runs, it will automatically generate 10 accounts and private keys, along with an HD wallet. It will start to listen for incoming connections on TCP port `8545`.

`ganache-cli` has a number of flags to customize the simulated chain according to your requirements. For example, flag `-a` allows you to specify the number of accounts to generate at startup. Similarly `-b` allows the user to configure block time for mining.

Detailed help is available using the following command:

```
$ ganache --help
```

Ganache is a command-line tool. However, at times, it is desirable to have a fully featured tool with a rich **graphical user interface** (**GUI**).

Ganache UI

Ganache UI is based on a JavaScript implementation of the Ethereum blockchain, with a built-in block explorer and mining, making testing locally on the system very easy. You can view transactions, blocks, and addresses in detail on the frontend GUI. It can be downloaded from `https://www.trufflesuite.com/ganache`.

When you start Ganache for the first time, it will ask whether you want to create a quick blockchain or create a new workspace that can be saved, and it also has advanced setup options:

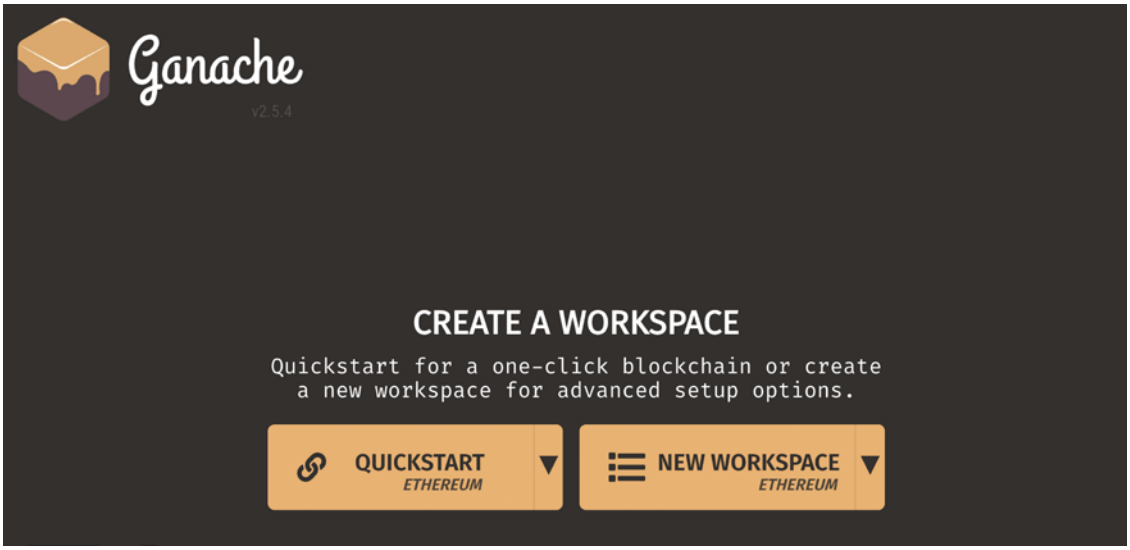


Figure 11.1: Creating a workspace

Select the **QUICKSTART** or **NEW WORKSPACE** option as required. For a quicker temporary setup with default options, which could be useful for simple testing, you can choose **QUICKSTART**. We will choose **NEW WORKSPACE** as we want to explore more advanced features.

If **NEW WORKSPACE** is selected, there are a number of options available to configure the blockchain. One of the configuration options is **WORKSPACE NAME**, where you can specify a name for your project. Additionally, Truffle projects can also be added here—we will cover Truffle in more detail later in the chapter.

Other options include **SERVER**, **ACCOUNTS & KEYS**, **CHAIN**, and **ADVANCED**. The **SERVER** tab is used to configure RPC connectivity by specifying the hostname, port number, and network ID:

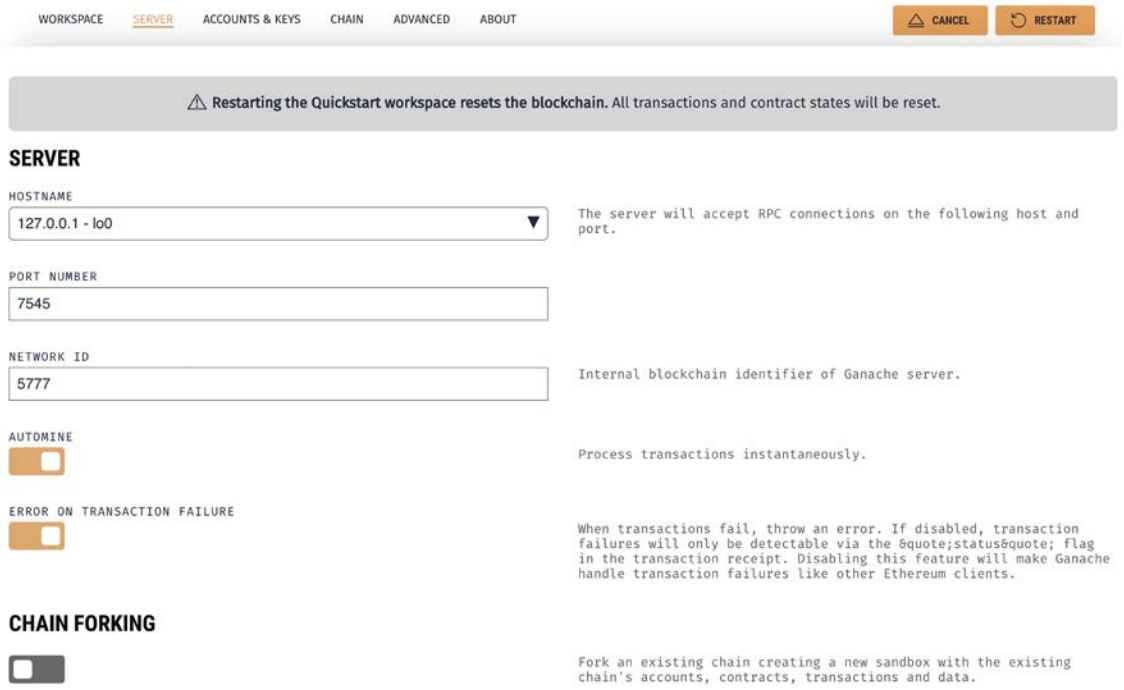


Figure 11.2: Server configuration

ACCOUNTS & KEYS provides options to configure the balance and the number of accounts to generate. The **CHAIN** option provides a configuration interface for specifying the gas limit, gas price, and hard fork, which is required to be simulated, such as Byzantine or Petersburg.

The **ADVANCED** option is available to configure logging and analytics-related settings. Once you have all the configuration options set, save the workspace by selecting **SAVE WORKSPACE**, and the main transaction view of the Ganache personal blockchain will show:

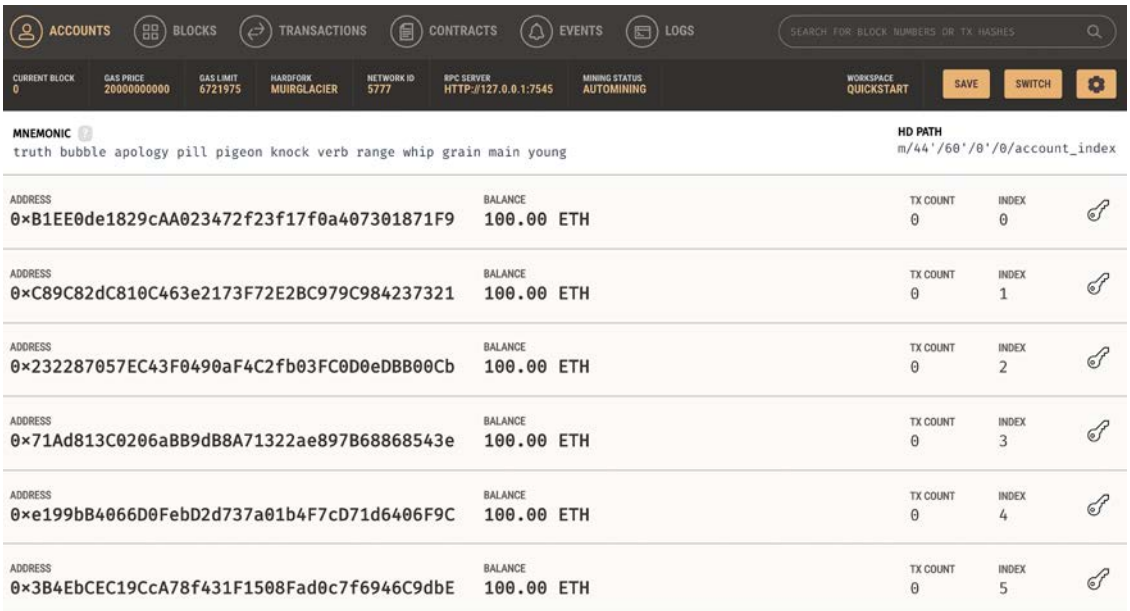


Figure 11.3: Ganache main view

With this, we conclude our introduction to Ganache, a mainstream tool used in blockchain development. Now we will move on to different development frameworks that are available for Ethereum.

There are a number of other notable frameworks available for Ethereum development. It is almost impossible to cover all of them, but an introduction to some of the mainstream frameworks and tools is given as follows.

Truffle

Truffle (available at `https://www.trufflesuite.com`) is a development environment that makes it easier and simpler to test and deploy Ethereum contracts. Truffle provides contract compilation and linking along with an automated testing framework using Mocha and Chai. It also makes it easier to deploy the contracts to any private, public, or testnet Ethereum blockchain. Also, an asset pipeline is provided, which makes it easier for all JavaScript files to be processed, making them ready for use by a browser.

Before installation, it is assumed that `node` is available, which can be queried as shown here:

```
$ node -v
v16.15.0
```

If `node` is not available already, then the installation of `node` is required first in order to install truffle. The installation of truffle is very simple and can be done using the following command via **Node Package Manager** (**npm**):

```
$ npm install truffle -g
```

This will take a few minutes; once it is installed, the `truffle` command can be used to display help information and verify that it is installed correctly.

Type `truffle` in the terminal to display usage help:

```
$ truffle
```

This will display all the options that Truffle supports. Alternatively, the repository is available at `https://github.com/trufflesuite/truffle`, which can be cloned locally to install `truffle`. **Git** can be used to clone the repository using the following command:

```
$ git clone https://github.com/trufflesuite/truffle.git
```

We will use Truffle later in *Chapter 12*, *Web3 Development Using Ethereum*, to test and deploy smart contracts on the Ethereum blockchain. For now, we'll continue to explore some of the frameworks used for development on the Ethereum blockchain.

Drizzle

Drizzle is a collection of frontend libraries that allows the easy development of web UIs for decentralized applications. It is based on the Redux store and allows seamless synchronization of contract and transaction data.

Drizzle is installed using the following command:

```
$ npm install --save @drizzle/store
```

Web **User Interface** (**UI**) development is an important part of dApp development. As such, many web techniques and tools, ranging from simple HTML and JavaScript to advanced frameworks such as Redux and React, are used to develop web UIs for dApps.

Other tools

There are many other tools and frameworks available. Information about these tools is available here: `https://ethereum.org/en/developers/local-environment/`. However, we will discuss a few here:

- **Embark** is a complete and powerful developer platform for building and deploying decentralization applications. It is used for smart contract development, configuration, testing, and deployment. It also integrates with **Swarm**, **IPFS**, and **Whisper**. There is also a web interface called **Cockpit** available with Embark, which provides an integrated development environment for easy development and debugging of decentralized applications.
- **Brownie** is a Python-based framework for Ethereum smart contract development and testing. It has the full support of Solidity and Vyper with relevant testing and debugging tools. More information is available at `https://eth-brownie.readthedocs.io/en/stable/`.

- **Waffle** is a framework for smart contract development and testing. It claims to be faster than Truffle. This framework allows dApp development, debugging, and testing in Solidity and Vyper. It is based on `Ethers.js`. More details are available on the official website at `https://getwaffle.io`.
- The `OpenZeppelin` toolkit has a rich set of tools that allow easy smart contract development. It supports compiling, deploying, upgrading, and interacting with smart contracts. Further information is available here: `https://openzeppelin.com/sdk/`.

In this section, we have covered some of the mainstream frameworks that are used in the Ethereum ecosystem for development. In the next section, we will explore which tools are available for writing and deploying smart contracts.

Contract development and deployment

There are various steps that need to be taken in order to develop and deploy contracts. Broadly, these can be divided into three steps: writing, testing, and deploying. After deployment, the next optional step is to create the UI and present it to the end users via a web server. We'll cover that in the following chapter. A web interface is sometimes not needed in contracts where no human input or monitoring is required, but usually there is a requirement to create a web interface so that end users can interact with the contract using familiar web-based interfaces.

Writing smart contracts

The writing step is concerned with writing the contract source code in Solidity. This can be done in any text editor. There are various plugins and add-ons available for Vim in Linux, Atom, and other editors that provide syntax highlighting and formatting for Solidity source code.

Visual Studio Code has become quite popular and is used commonly for Solidity development. There are Solidity plugins available that allow syntax highlighting, formatting, and **IntelliSense**. Also, **Truffle** is available as a plugin, which improves the developer experience considerably.

Both can be installed via the **Extensions** option in Visual Studio Code.



The Solidity plugin for Visual Studio is available in the Visual Studio Marketplace at `https://marketplace.visualstudio.com/items?itemName=JuanBlanco.solidity`

Truffle for VS Code is available here: `https://marketplace.visualstudio.com/items?itemName=trufflesuite-csi.truffle-vscode`

Testing smart contracts

Testing is usually performed by automated means. Earlier in the chapter, you were introduced to Truffle, which uses the Mocha framework to test contracts. However, manual functional testing can be performed as well by using Remix IDE, which was discussed in *Chapter 10*, *Ethereum in Practice*, and running functions manually and validating results. We will cover this in *Chapter 12*, *Web3 Development Using Ethereum*.

Deploying smart contracts

Once the contract is verified, working, and tested on a simulated environment (for example, Ganache) or on a private net, it can be deployed to a public testnet such as **Ropsten** and eventually to the live blockchain (mainnet). We will cover all these steps, including verification, development, and creating a web interface, in the next chapter.

Now that we have covered which tools can be used to write Solidity smart contracts, we will introduce the Solidity language. This will be a brief introduction to Solidity, which should provide the base knowledge required to write smart contracts. The syntax of the language is very similar to C and JavaScript, and it is quite easy to program. We'll start by exploring what a smart contract written in the Solidity language looks like.

The Solidity language

Solidity is the domain-specific language of choice for programming contracts in Ethereum. Its syntax is close to both JavaScript and C. Solidity has evolved into a mature language over the last few years and is quite easy to use, but it still has a long way to go before it can become advanced, standardized, and feature-rich, like other well-established languages such as Java, C, and C#. Nevertheless, it is the most widely used language currently available for programming contracts.

It is a statically typed language, which means that variable type checking in Solidity is carried out at compile time. Each variable, either state or local, must be specified with a type at compile time. This is beneficial in the sense that any validation and checking are completed at compile time and certain types of bugs, such as the interpretation of data types, can be caught earlier in the development cycle instead of at runtime, which could be costly, especially in the case of the blockchain/smart contract paradigm.

Other features of the language include inheritance, libraries, and the ability to define composite data types. Solidity is also called a contract-oriented language. In Solidity, contracts are equivalent to the concept of classes in other object-oriented programming languages.

In the following subsections, we will look at the components of a Solidity source code file, which is important to cover before we move on to writing smart contracts in the next section.

In order to address compatibility issues that may arise from future versions of `solc`, `pragma` can be used to specify the version of the compatible compiler, as in the following example:

```
pragma solidity ^0.8.0
```

This will ensure that the source file does not compile with versions lower than `0.8.0`.

`import` in Solidity allows the importing of symbols from existing Solidity files into the current global scope. This is similar to the `import` statements available in JavaScript, as in the following, for example:

```
import "module-name";
```

Comments can be added to the Solidity source code file in a manner similar to the C language. Multiple-line comments are enclosed in `/*` and `*/`, whereas single-line comments start with `//`.

An example `solidity` program is as follows, showing the use of `pragma`, `import`, and comments:

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;

contract valueChecker
{
    uint8 price=10;
    //price variable declared and initialized with a value of 10
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y= true;
        }
    }
}
```



Note that an SPDX license identifier needs to be added as well as the first line, otherwise the Solidity compiler will generate a warning. If all is in order, the program will compile successfully.

In this section, we examined what the Solidity code of a smart contract looks like. Now it's time to learn about the Solidity language.

Functions

Functions are pieces of code within a smart contract. For example, look at the following code block:

```
pragma solidity >8.0.0;
contract Test1
{
    uint x=2;
    function addition1() public view returns (uint y)
    {
        y=x+2;
    }
}
```

In the preceding code example, with `contract Test1`, we have defined a function called `addition1()`, which returns an unsigned integer after adding `2` to the value supplied via the variable `x`, initialized just before the function.

In this case, `2` is supplied via variable `x`, and the function will return `4` by adding `2` to the value of `x`. It is a simple function, but demonstrates how functions work and what their different elements are.

There are two function types – i*nternal* and *external* functions:

- **Internal functions** can be used only within the context of the current contract.
- **External functions** can be called via external function calls.

A **function** in Solidity can be marked as a constant. Constant functions cannot change anything in the contract; they only return values when they are invoked and do not cost any gas. This is the practical implementation of the concept of *call* as discussed in the previous chapter.

Functions in Solidity are modules of code that are associated with a contract. Functions are declared with a name, optional parameters, access modifiers, state mutability modifiers, and an optional return type.

The syntax of defining a function is shown as follows:

```
function <name of the function>(<parameters>) <visibility specifier> <state
mutability modifier> returns (<return data type> <name of the variable>)
{
    <function body>
}
```

It contains the following elements:

- **Function signature**: Functions in Solidity are identified by their **signature**, which is the first four bytes of the Keccak-256 hash of its full signature string. This is also visible in Remix IDE. For example, the signature for the `Matcher` function we saw earlier is the first four bytes of the 32-byte Keccak-256 hash of the function:

  ```
  {
      "f9d55e21": "Matcher(uint8)"
  }
  ```

  In this example function, `Matcher` has the signature hash of `f9d55e21`. This information is useful in order to build interfaces.

- **Input parameters of a function**: Input parameters of a function are declared in the form of `<data type> <parameter name>`. This example clarifies the concept where `uint x` and `uint y` are input parameters of the `checkValues` function:

  ```
  contract myContract
  {
      function checkValues(uint x, uint y)
  ```

```
    {
    }
}
```

- **Output parameters of a function:** Output parameters of a function are declared in the form of `<data type> <parameter name>`. This example shows a simple function returning a `uint` value:

```
contract myContract
{
    function getValue() returns (uint z)
    {
        z=x+y;
    }
}
```

A function can return multiple values, as well as taking multiple inputs. In the preceding example function, `getValue` only returns one value, but a function can return up to 14 values of different data types. The names of the unused return parameters can optionally be omitted. An example of such a function could be:

```
pragma solidity ^0.8.0;
contract Test1
{

 function addition1(uint x, uint y) public pure returns (uint z, uint a)
    {
        z= x+y ;
        a=x+y;
        return (z,a);

    }
}
```

Here, when the code runs, it will take two parameters as input, `x` and `y`, add both, then assign them to `z` and `a`, and finally return `z` and `a`. For example, if we provide `1` and `1` for `x` and `y`, respectively, then when the variables `z` and `a` are returned by the function, both will contain `2` as a result.

- **Internal function calls**: Functions within the context of the current contract can be called internally in a direct manner. These calls are made to call the functions that exist within the same contract. These calls result in simple `JUMP` calls at the EVM bytecode level.
- **External function calls**: External function calls are made via message calls from a contract to another contract. In this case, all function parameters are copied to the memory. If a call to an internal function is made using the `this` keyword, it is also considered an external call. The `this` variable is a pointer that refers to the current contract. It is explicitly convertible to an address and all members of a contract are inherited from the address.

- **Fallback functions:** This is an unnamed function in a contract with no arguments and return data. This function executes every time ether is received. It is required to be implemented within a contract if the contract is intended to receive ether; otherwise, an exception will be thrown and ether will be returned. This function also executes if no other function signatures match in the contract. If the contract is expected to receive ether, then the fallback function should be declared with the payable **modifier**.
- The payable is required; otherwise, this function will not be able to receive any ether. This function can be called using the `address.call()` method as, for example, in the following:

```
function ()
{
    throw;
}
```

  In this case, if the fallback function is called according to the conditions described earlier; it will call `throw`, which will roll back the state to what it was before making the call. It can also be some other construct than `throw`; for example, it can log an event that can be used as an alert to feed back the outcome of the call to the calling application.

- **Modifier functions**: These functions are used to change the behavior of a function and can be called before other functions. Usually, they are used to check some conditions or verification before executing the function. _ (underscore) is used in the modifier functions that will be replaced with the actual body of the function when the modifier is called. Basically, it symbolizes the function that needs to be *guarded*. This concept is similar to *guard* functions in other languages.
- **Constructor function**: This is an optional function that has the same name as the contract and is executed once a contract is created. Constructor functions cannot be called later on by users, and there is only one constructor allowed in a contract. This implies that no overloading functionality is available.
- **Function visibility specifiers** (access modifiers/access levels): Functions can be defined with four access specifiers as follows:
    - **External:** These functions are accessible from other contracts and transactions. They cannot be called internally unless the `this` keyword is used.
    - **Public**: By default, functions are public. They can be called either internally or by using messages.
    - **Internal**: Internal functions are visible to other derived contracts from the parent contract.
    - **Private:** Private functions are only visible to the same contract they are declared in.
- **Function modifiers**:
    - **pure:** This modifier prohibits access or modification to state.
    - **view**: This modifier disables any modification to state.
    - **payable**: This modifier allows payment of ether with a call.

- **virtual**: This allows the function's or modifier's behavior to be changed in the derived contracts.
- **override**: This states that this function, modifier, or public state variable changes the behavior of a function or modifier in a base contract.

We can see these elements in an example shown below:

```
function orderMatcher (uint x)
private view returns (bool return value)
```

In the preceding code example, `function` is the keyword used to declare the function. `orderMatcher` is the function name, `uint x` is an optional parameter, `private` is the **access modifier** or **specifier** that controls access to the function from external contracts, `view` is an optional keyword used to specify that this function does not change anything in the contract but is used only to retrieve values from the contract, and `returns (bool return value)` is the optional return type of the function.

Variables

Just like any programming language, variables in Solidity are the named memory locations that hold values in a program. There are three types of variables in Solidity: **local variables**, **global variables**, and **state variables**.

Local variables

These variables have a scope limited to only within the function they are declared in. In other words, their values are present only during the execution of the function in which they are declared.

Global variables

These variables are available globally as they exist in the global namespace. They are used to perform various functions such as ABI encoding, cryptographic functions, and querying blockchain and transaction information.

Solidity provides a number of **global variables** that are always available in the global namespace. These variables provide information about blocks and transactions. Additionally, cryptographic functions, ABI encoding/decoding, and address-related variables are available.

A subset of available global variables is shown as follows.

- This returns the current block number:

```
block.number
```

- This returns the gas price of the transaction:

```
tx.gasprice (uint)
```

- This variable returns the address of the miner of the current block:

```
block.coinbase (address payable)
```

- This returns the current block timestamp:

```
now (uint)
```

- This returns the current block difficulty:

```
block.difficulty (uint)
```

A subset of some functions is shown as follows:

- The function below is used to compute the Keccak-256 hash of the argument provided to the function:

```
keccak256(...) returns (bytes32)
```

- This function returns the associated address of the public key from the elliptic curve signature:

```
ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address)
```



There are several other global variables available. A comprehensive list and details can be found in Solidity's official documentation: `https://solidity.readthedocs.io/en/latest/units-and-global-variables.html`.

State variables

State variables have their values permanently stored in smart contract storage. State variables are declared outside the body of a function, and they remain available throughout the contract depending on the accessibility assigned to them and as long as the contract persists:

```
pragma solidity >=0.8.0;
contract Addition {
    uint x; // State variable
}
```

Here, `x` is a state variable whose value will be stored in contract storage.

There are three types of state variables, based on their visibility scope:

- **Private**: These variables are only accessible internally from within the contract that they are originally defined in. They are also not accessible from any derived contract from the original contract.
- **Public**: These variables are part of the contract interface. In simple words, anyone is able to get the value of these variables. They are accessible within the contract internally by using the `this` keyword. They can also be called from other contracts and transactions. A `getter` function is automatically created for all public variables.
- **Internal**: These variables are only accessible internally within the contract that they are defined in. In contrast to private state variables, they are also accessible from any derived contract from the original (parent) contract.

There are two modifiers for state variables:

- `constant`: This disallows assignment, except initialization.
- `immutable`: This allows exactly one assignment at construction time and is constant afterward.

In the next section, we will introduce the data types supported in Solidity.

Data types

Solidity has two categories of data types – v**alue types** and **reference types**:

- Value types are variables that are always passed by a value. This means that value types hold their value or data directly, allowing a variable's value held in memory to be directly accessible by accessing the variable.
- Reference types store the address of the memory location where the value is stored. This is in contrast with value types, which store the actual value of a variable directly with the variable itself.

Recall from *Chapter 9*, *Ethereum Architecture*, that EVM can read and write data in different locations. The specific location used for storing values of a variable depends on the data type of the variable and where the variable has been declared. For example, function parameter variables are stored in memory, whereas state variables are stored in storage.

Now we'll describe value types in detail.

Value types

Value types mainly include **Booleans**, **integers**, **addresses**, and **literals**, which are explained in detail here.

Boolean

This data type has two possible values, `true` or `false`, for example:

```
bool v = true;
bool v = false;
```

This statement assigns the value `true` or `false` to `v` depending on the assignment.

Integers

This data type represents integers. Various keywords are used to declare integer data types:

- `int`: Signed integer. `int8` to `int256`, which means that keywords are available from `int8` up to `int256` in increments of 8, for example, `int8`, `int16`, and `int24`.
- `uint`: Unsigned integer. `uint8`, `uint16`, up to `uint256`. Usage is dependent on how many bits are required to be stored in the variable.

For example, in this code, note that `uint` is an alias for `uint256`:

```
uint256 x;
```

```
uint y;
uint256 z;
```

These types can also be declared with the `constant` keyword, which means that no storage slot will be reserved by the compiler for these variables. In this case, each occurrence will be replaced with the actual value:

```
uint constant z=10+10;
```

Address

This data type holds a 160-bit long (20-byte) value. This type has several members that can be used to interact with and query the contracts. These members are described here:

- **Balance**: The `balance` member returns the balance of the address in Wei.
- **Send**: This member is used to send an amount of ether to an address (Ethereum's 160-bit address) and returns `true` or `false` depending on the result of the transaction, for example, the following:

  ```
  address to = 0x6414cc08d148dce9ebf5a2d0b7c220ed2d3203da;
  address from = this;
  if (to.balance < 10 && from.balance > 50) to.send(20);
  ```

- **Call functions**: The `call`, `callcode`, and `delegatecall` functions are provided in order to interact with functions that do not have an ABI. These functions should be used with caution as they are not safe to use due to the impact on the type safety and security of the contracts.
- **Array value types (fixed-size and dynamically sized byte arrays)**: Solidity has fixed-size and dynamically sized byte arrays. Fixed-size keywords range from `bytes1` to `bytes32`, whereas dynamically sized keywords include `bytes` and `string`. The `bytes` keyword is used for raw byte data, and `string` is used for strings encoded in UTF-8. As these arrays are returned by the value, calling them will incur a gas cost.

  An example of a static (fixed-size) array is as follows:

  ```
  bytes32[10] bankAccounts;
  ```

  An example of a dynamically sized array is as follows:

  ```
  bytes32[] trades;
  ```

  `length` is a member of array value types and returns the length of the byte array:

  ```
  trades.length;
  ```

Literals

These are used to represent a fixed value. There are different types of literals that are described as follows:

- **Integer literals:** These are a sequence of decimal numbers in the range of 0–9. An example is shown as follows:

```
uint8 x = 2;
```

- **String literals:** This type specifies a set of characters written with double or single quotes. An example is shown as follows:

```
'packt'
"packt"
```

- **Hexadecimal literals:** These are prefixed with the keyword `hex` and specified within double or single quotation marks. An example is shown as follows:

```
(hex'AABBCC');
```

- **Enums:** This allows the creation of user-defined types. An example is shown as follows:

```
enum Order {Filled, Placed, Expired };
Order private ord;
ord=Order.Filled;
```

Explicit conversion to and from all integer types is allowed with enums.

Reference types

As the name suggests, these types are passed by reference and are discussed in the following section. These are also known as **complex types**. Reference types include **arrays, structs**, and **mappings**.

When using reference types, it is essential to explicitly specify the storage area where the type is stored, for example, *memory*, *storage*, or *calldata*.

Arrays

Arrays represent a contiguous set of elements of the same size and type laid out at a memory location. The concept is the same as any other programming language. Arrays have two members, named `length` and `push`.

Structs

These constructs can be used to group a set of dissimilar data types under a logical group. These can be used to define new custom types, as shown in the following example:

```
pragma solidity ^0.8.0;
contract TestStruct {
  struct Trade
```

```
  {
    uint tradeid;
    uint quantity;
    uint price;
    string trader;
  }
  //This struct can be initialized and used as below
  Trade tStruct = Trade({tradeid:123, quantity:1, price:1, trader:"equinox"});
}
```

In the preceding code, we declared a `struct` named `Trade` that has four fields. `tradeid`, `quantity`, and `price` are of the `uint` type, whereas `trader` is of the `string` type. Once the `struct` is declared, we can initialize and use it. We initialize it by using `Trade tStruct` and assigning `123` to `tradeid`, `1` to `quantity`, and `"equinox"` to `trader`.

Sometimes it is desirable to choose the location of the variable data storage. This choice allows for better gas expense management. We can use the data location name to specify where a particular complex data type will be stored. Depending on the default or specified annotation, the location can be `storage`, `memory`, or `calldata`. This is applicable to arrays and structs and can be specified using the `storage` or `memory` keyword. `calldata` behaves almost like memory. It is an unmodifiable and temporary area that can be used to store function arguments.

For example, in the preceding structs example, if we want to use only memory (temporarily) we can do that by using the `memory` keyword when using the structure and assigning values to fields in the `struct`, as shown here:

```
Trade memory tStruct;
tStruct.tradeid = 123;
```

As copying between memory and storage can be quite expensive, specifying a location can be helpful to control the gas expenditure at times.

Parameters of external functions use **calldata** memory. By default, parameters of functions are stored in **memory**, whereas all other local variables make use of **storage**. State variables, on the other hand, are required to use storage.

Mappings

Mappings are used for key-to-value mapping. This is a way to associate a value with a key. All values in this map are already initialized with all zeroes, as in the following example:

```
mapping (address => uint) offers;
```

This example shows that `offers` is declared as a mapping. Another example makes this clearer:

```
mapping (string => uint) bids;
bids["packt"] = 10;
```

This is basically a dictionary or a hash table, where string values are mapped to integer values. The mapping named `bids` has the string `packt` mapped to value `10`.

Control structures

The control structures available in the Solidity language are `if...else`, `do`, `while`, `for`, `break`, `continue`, and `return`. They work exactly the same as other languages, such as the C language or JavaScript.

Some examples are shown here:

- `if`: If `x` is equal to `0`, then assign value `0` to `y`, else assign `1` to `z`:

```
if (x == 0)
    y = 0;
else
    z = 1;
```

- `do`: Increment `x` while `z` is greater than `1`:

```
do{
    x++;
} (while z>1);
```

- `while`: Increment `z` while `x` is greater than `0`:

```
while(x > 0){
    z++;
}
```

- `for`, `break`, and `continue`: Perform some work until `x` is less than or equal to `10`. This `for` loop will run `10` times; if `z` is `5`, then break the `for` loop:

```
for(uint8 x=0; x<=10; x++)
{
    //perform some work
    z++
    if(z == 5) break;
}
```

- `continue` can be used in situations where we want to start the next iteration of the loop immediately without executing the rest of the code. For example, take the code shown next:

```
        for (uint i = 0; i < 5; i++) {
             if (i == 2) {
                 // Skip to next
                 continue;
```

```
            }

            if (i == 5) {
                // Exit loop
                break;

            }
        }
```

It will continue the work in a similar vein, but when the condition is met, the loop will start again.

- `return`: `return` is used to stop the execution of a function and returns an optional value. For example:

```
return 0;
```

It will stop the execution and return a value of `0`.

Events

Events in Solidity can be used to log certain events in EVM logs. These are quite useful when external interfaces are required to be notified of any change or event in the contract. These logs are stored on the blockchain in transaction logs. Logs cannot be accessed from the contracts but are used as a mechanism to notify change of state or the occurrence of an event (meeting a condition) in the contract.

In a simple example here, the `valueEvent` event will return `true` if the `x` parameter passed to the function `Matcher` is equal to or greater than `10`:

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;
contract valueChecker
{
    uint8 price=10;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y = true;
        }
    }
}
```

Inheritance

Inheritance is supported in Solidity. The `is` keyword is used to derive a contract from another contract. In the following example, `valueChecker2` is derived from the `valueChecker` contract. The derived contract has access to all non-private members of the parent contract:

```
// SPDX-License-Identifier: MIT

pragma solidity >=0.8.0;
contract valueChecker
{
    uint8 price = 20;
    event valueEvent(bool returnValue);
    function Matcher(uint8 x) public returns (bool y)
    {
        if (x>=price)
        {
            emit valueEvent(true);
            y = true;
        }
    }
}
contract valueChecker2 is valueChecker
{
    function Matcher2() public view returns (uint)
    {
        return price+10;
    }
}
```

In the preceding example, if the `uint8 price = 20` is changed to `uint8 private price = 20`, then it will not be accessible by the `valueChecker2` contract. This is because now the member is declared as `private`, and thus it is not allowed to be accessed by any other contract. The error message that you will see when attempting to compile this contract is as follows:

```
browser/valuechecker.sol:20:8: DeclarationError: Undeclared identifier.
return price+10;
       ^---^
```

Libraries

Libraries are deployed only once at a specific address and their code is called via the `CALLCODE` or `DELEGATECALL` opcode of the EVM. The key idea behind libraries is code reusability. They are similar to contracts and act as base contracts to the calling contracts.

A library can be declared as shown in the following example:

```
library Addition
{
    function Add(uint x,uint y) returns (uint z)
    {
        return x + y;
    }
}
```

This library can then be called in the contract, as shown here. First, it needs to be imported and then it can be used anywhere in the code. A simple example is shown as follows:

```
import "Addition.sol"
function Addtwovalues() returns(uint)
{
    return Addition.Add(100,100);
}
```

There are a few limitations with libraries; for example, they cannot have state variables and cannot inherit or be inherited. Moreover, they cannot receive ether either; this is in contrast to contracts, which can receive ether.

Now let's consider how Solidity approaches handling errors.

Error handling

Solidity provides various functions for error handling. By default, in Solidity, whenever an error occurs, the state does not change and reverts back to the original state.

Some constructs and convenience functions that are available for error handling in Solidity are introduced as follows:

- `assert`: This is used to check for conditions and throw an exception if the condition is not met. Assert is intended to be used for internal errors and invariant checking. When called, this method results in an invalid opcode and any changes in the state are reverted.
- `require`: Similar to `assert`, this is used to check conditions and throws an exception if the condition is not met. The difference is that `require` is used to validate inputs, return values, or calls to external contracts. The method also results in reverting to the original state. It can also take an optional parameter to provide a custom error message.
- `revert`: This method aborts the execution and reverts the current call. It can also optionally take a parameter to return a custom error message to the caller.
- `try/catch`: This construct is used to handle a failure in an external call.
- `throw`: `throw` is used to stop execution. As a result, all state changes are reverted. In this case, no gas is returned to the transaction originator because all the remaining gas is consumed.

This completes a brief introduction to the Solidity language. The language is very rich and under constant improvement. Detailed documentation and coding guidelines are available online at `http://solidity.readthedocs.io/en/latest/`.

Summary

This chapter started with the introduction of development tools for Ethereum, such as Ganache CLI. The installation of Node.js was also introduced, as most of the tools are JavaScript- and Node.js-based. Then we discussed some frameworks such as Truffle, along with local blockchain solutions for development and testing, such as Ganache and Drizzle. We also introduced Solidity in this chapter and explored different concepts, such as value types, reference types, functions, and error handling concepts. We also learned how to write contracts using Solidity.

In the next chapter, we will explore the topic of Web3, a JavaScript API that is used to communicate with the Ethereum blockchain.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

12

Web3 Development Using Ethereum

Web3 is a JavaScript library that can be used to communicate with an Ethereum node via RPC communication. Web3 works by exposing methods that have been enabled over RPC. This allows the development of user interfaces (UIs) that make use of the Web3 library in order to interact with contracts deployed over the blockchain.

In this chapter, we'll explore the Web3 API, and introduce some detailed examples of how smart contracts are written, tested, and deployed to the Ethereum blockchain. We will use various tools such as the Remix IDE and Ganache to develop and test smart contracts and look at the methods used to deploy smart contracts to Ethereum test networks and private networks. The chapter will explore how HTML and JavaScript frontends can be developed to interact with smart contracts deployed on the blockchain, and introduce advanced libraries such as Drizzle. The topics we will cover are as follows:

- Exploring Web3 with Geth
- Contract deployment
- Interacting with contracts via frontends
- Development frameworks

We will start with Web3, and gradually build our knowledge with various tools and techniques for smart contract development. You will be able to test your knowledge in the bonus content pages for this book, where we will develop a project using all the techniques that this chapter will cover.

Interacting with contracts using Web3 and Geth

Web3 is a powerful API and can be explored by attaching a Geth instance. To expose the required APIs via Geth, the following command can be used:

```
$ geth --datadir ~/etherprivate --networkid 786 --http --http.api
"web3,net,eth,debug" --http.port 8001 --http.corsdomain http://localhost:7777
```

The `--http` flag in the preceding command allows the `web3`, `eth`, `net`, and `debug` methods. There are other APIs such as `personal`, `miner`, and `admin` that can be exposed by adding their names to this list.

Note that I have used Geth version v1.10.14-stable. All commands work correctly for this version but may not work with previous versions. Try to use the latest version of Geth when running examples in this chapter, or at least v1.10.14-stable.

The Geth instance can be attached using the following command:

```
$ geth attach ~/etherprivate/geth.ipc
```

Once the Geth JavaScript console is running, Web3 can be queried:

```
Welcome to the Geth JavaScript console!
instance: Geth/v1.10.14-stable/darwin-arm64/go1.17.5
coinbase: 0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f
at block: 2421 (Sat May 21 2022 19:41:23 GMT+0100 (BST))
 datadir: /Users/imran/etherprivate
 modules: admin:1.0 debug:1.0 eth:1.0 ethash:1.0 miner:1.0 net:1.0 personal:1.0
rpc:1.0 txpool:1.0 web3:1.0
To exit, press ctrl-d or type exit
> web3.version
{
  api: "0.20.1",
  ethereum: undefined,
  network: "786",
  node: "Geth/v1.10.14-stable/darwin-arm64/go1.17.5",
  whisper: undefined,
  getEthereum: function(callback),
  getNetwork: function(callback),
  getNode: function(callback),
  getWhisper: function(callback)
}
>
```

Now that we've introduced Web3, let's consider how the Remix IDE can be used to deploy a contract, and how the Geth console can interact with the deployed contract.

Deploying contracts

A simple contract can be deployed using Geth and interacted with using Web3 via the **command-line interface** (**CLI**) that Geth provides (`console` or `attach`). The following are the steps to achieve that.

Run the Geth client using one of the following commands. We've used all these commands in different contexts before, and you can use any of them to run Geth. The first command, shown next, allows connectivity with the Remix IDE as we have specified `--http.corsdomain https://remix.ethereum.org`:

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure-unlock --http
--http.api "web3,net,eth,debug,personal" --http.port 8001 --http.corsdomain
https://remix.ethereum.org
```

The second command, shown next, allows `localhost:7777` to access the RPC server exposed by Geth. This option is useful if you have an application running on this interface and you want to give it access to RPC:

```
$ geth --datadir ~/etherprivate --networkid 786 --allow-insecure-unlock --http
--http.api "web3,net,eth,debug,personal" --http.port 8001 --http.corsdomain
http://localhost:7777
```

It also exposes RPC on port `8001` via the flag `http.port 8001`, which is useful in case you have some other service or application listening already on port `8545`, which would mean that Geth won't be able to use that already-in-use port. This is because Geth listens on port `8545` for the HTTP-RPC server by default.

The last command, shown next, allows all incoming connections as `--http.corsdomain` is set to `*`:

```
$ geth --datadir ~/etherprivate --networkid 786 --http --http.api
"web3,net,eth,debug" --http.port 8001 --http.corsdomain "*"
```

If the Geth console is not already running, open another terminal and run the following command:

```
$ geth attach ~/etherprivate/geth.ipc
```

In order to deploy a smart contract, we use the Web3 deployment script. The main elements of the script, the Application Binary Interface (ABI) and the bytecode, can be generated from the Remix IDE. We will discuss the Remix IDE in more detail later in this chapter; for now, we are using this IDE only to get the required elements (ABI and bytecode) for the Web3 deployment script used for the deployment of the contract.



To learn how to download and use the Remix IDE, refer to *Chapter 10*, *Ethereum in Practice*.

First, paste the following source code into the Remix IDE:

```
pragma solidity ^0.8.0;
contract valueChecker
{
    uint price=10;
    event valueEvent(bool returnValue);
    function Matcher (uint8 x) public returns (bool)
    {
        if (x>=price)
```

```
        {
            emit valueEvent(true);
            return true;
        }
    }
}
```

Once the code is pasted in the Remix IDE, it will appear as follows in the Solidity compiler:

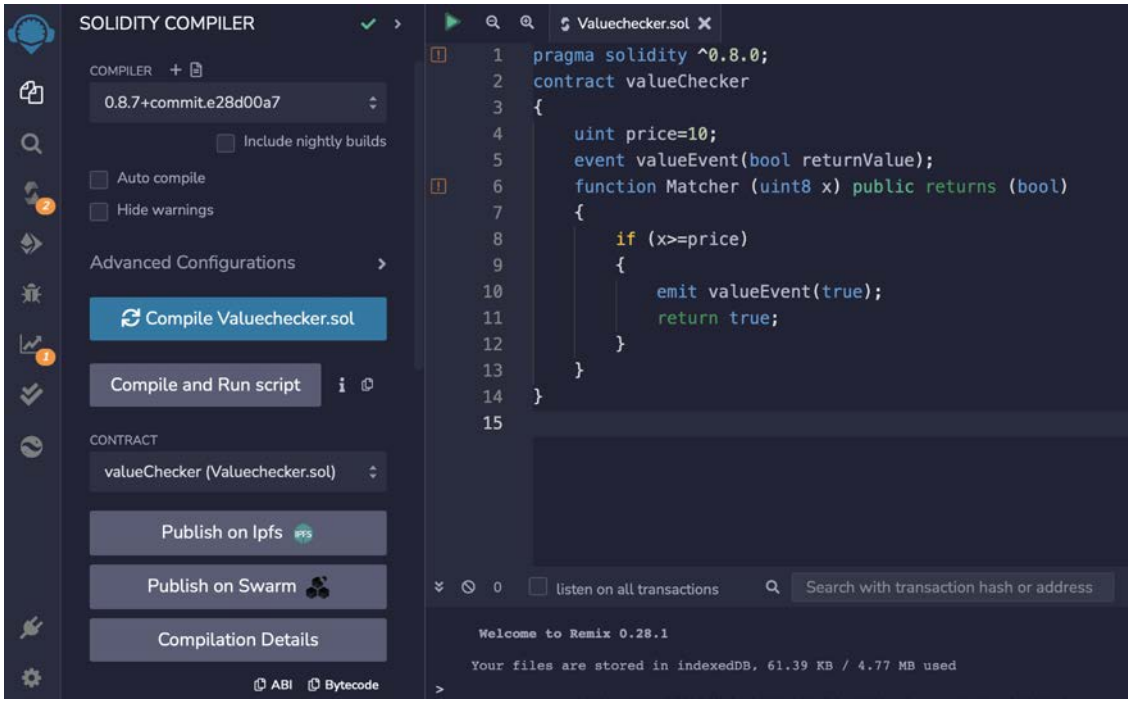


Figure 12.1: Code shown in Remix

Now create the Web3 deployment script as follows. First, generate the ABI and bytecode by using the **ABI** and **Bytecode** buttons in the Remix IDE (shown on the lower-left side of *Figure 12.1*) and paste them into the following script after the `web3.eth.contract(` and `data:` elements, respectively. The example shown below already has `ABI` and `Bytecode` pasted in:

```
var valuecheckerContract = web3.eth.contract([{"constant": false,"inputs":
[{"name": "x","type": "uint8"}],"name": "Matcher","outputs": [{"name":
"","type": "bool"}],"payable": false,"stateMutability": "nonpayable","type":
"function"},{"anonymous": false,"inputs": [{"indexed": false,"name":
"returnValue","type": "bool"}],"name": "valueEvent","type": "event"}]);
var valuechecker = valuecheckerContract.new({
     from: web3.eth.accounts[0],
```

```
     data: '0x6080604052600a600055348015610015760080fd5b5061010d806100256000
396000f3006080604052600436106603f576000357c0100000000000000000000000000000000-
0000000000000000000000000900463ffffffff168063f9d55e21146044575b600080fd
5b348015604f57600080fd5b50606f60048036038101908080356ff16906020019092
919050506089565b604051808215151515815260200191505060405180910390f35b
600080548260ff1610151560db577f3eb1a229ff7995457774a4bd31ef7b13b6f4491a
d1ebb8961af120b8b4b6239c60016040518082151515158152602001915050604051805
910390a16001905060dc565b5b9190505600a165627a7a72305820fe915bf3811b38f8
c7adcdca8d34daa2db1e11a309134504c285daff8d960e560029',
     gas: '4700000'
   }, function (e, contract){
    console.log(e, contract);
    if (typeof contract.address !== 'undefined') {
         console.log('Contract mined! address: ' + contract.address + '
transactionHash: ' + contract.transactionHash);
    }
})
```

Ensure that the account is unlocked. We created accounts earlier in *Chapter 10*, *Ethereum in Practice*; we can use that account or can create a new one if required.

First, list the accounts by using the following command, which outputs account `0` (the first account), as shown here:

```
> personal.listAccounts[0]
"0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811"
```

Now unlock the account using the following command. It will need the passphrase (password) that you used originally when creating this account. Enter the password to unlock the account:

```
> personal.unlockAccount(personal.listAccounts[0])
Unlock account 0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811
Password:
true
```

For more convenience, the account can be unlocked permanently for the length of the Geth console/attach session by using the command shown here:

```
> web3.personal.unlockAccount("0xc9bf76271b9e42e4bf7e1888e0f52351bdb65811",
"Password123", 0);
true
```

Now, we can open the Geth console that has been opened previously and deploy the contract. However, before deploying the contract, make sure that mining is running on the Geth node; this will allow contracts to be mined and, as a result, deployed on the blockchain. The following command can be used to start mining under the Geth console:

```
> miner.start()
```

Now paste the previously mentioned Web3 deployment script into the Geth console as shown below:

```
> var valuecheckerContract = web3.eth.contract([{"constant": false,
"inputs": [{"name": "x","type": "uint8"}],"name": "Matcher","outputs":
[{"name": "","type": "bool"}],"payable": false,"stateMutability":
"nonpayable","type": "function"},{"anonymous": false,"inputs":
[{"indexed": false,"name": "returnValue","type": "bool"}],"name":
"valueEvent","type": "event"}]);
```

After this, run the script as shown below, which contains the bytecode of the smart contract:

```
> var valuechecker = valuecheckerContract.new({
......      from: web3.eth.accounts[0],
......      data: '0x6080604052600a600055348015610015576000 80fd5b5061010
d80610025600039 6000f3006080604052600436106103f576000357c01000000000000000000
0000000000000000000000000000000000000000900463ffffffff168063f9d55e2114604
4575b600080fd5b348015604f57600080fd5b50606f60048036038101908080356 0ff1690
6020019092919050505060896565b604051808215151515815260200191505060405180910
390f35b600080548260ff1610151560db577f3eb1a229ff7995457774a4bd31ef7b13b6f4
491ad1ebb8961af120b8b4b6239c60016040518082151515158152602001915050604051 80
910390a16001905060dc565b5b9190505600a165627a7a72305820fe915bf3811b38f8c7ad
cdca8d34daa2db1e11a309134504c285daff8d960e560029',
......      gas: '4700000'
......    }, function (e, contract){
......     console.log(e, contract);
......     if (typeof contract.address !== 'undefined') {
.........          console.log('Contract mined! address: ' + contract.address +
' transactionHash: ' + contract.transactionHash);
.........     }
...... })
```

The previous command line shows how it looks when the Web3 deployment script is pasted into the Geth console for deployment.

Using solc to generate ABI and code

ABI and code can also be obtained by using the Solidity compiler, as shown in the following code snippets.

In order to generate the ABI, we can use the command shown as follows:

```
$ solc --abi valuechecker.sol
```

This command will produce the following output, with the contract ABI in `JSON` format:

```
======= valuechecker.sol:valueChecker =======
```

```
Contract JSON ABI
[{"anonymous":false,"inputs":[{"indexed":false,"internalType":"bool",
"name":"returnValue","type":"bool"}],"name":"valueEvent","type":"event"},
{"inputs":[{"internalType":"uint8","name":"x","type":"uint8"}],"name":"Matcher",
"outputs":[{"internalType":"bool","name":"","type":"bool"}],"stateMutability":
"nonpayable","type":"function"}]
```

The next step is to generate code, which we can use the following command to do:

```
$ solc --bin valuechecker.sol
```

This command will produce the binary (represented as hex) of the smart contract code:

```
======= valuechecker.sol:valueChecker =======
Binary:
6080604052600a600055348015610015576000 80fd5b5061018b806100256000396000f3fe608
060405234801561001057600080fd5b506004361061002b5760003560e01c8063f9d55e2114610
030575b600080fd5b61004a600480360381019061004591906100f2565b610060565b604051610
057919061013a565b60405180910390f35b600080548260ff16106100ae577f3eb1a229ff79954
57774a4bd31ef7b13b6f4491ad1ebb8961af120b8b4b6239c600160405161009d919061013a565
b60405180910390a1600190506100af565b5b919050565b600080fd5b600060ff8216905091905
0565b6100cf816100b9565b81146100da57600080fd5b50565b6000813590506100ec816100c65
65b92915050565b60006020828403121561010857610076100b4565b5b6000610116848285016
100dd565b91505092915050565b60008115159050919050565b6101348161011f565b825250505
65b60006020820190506101 4f600083018461012b565b9291505056fea26469706673582212200
e651ef5fb0ca476c266684e18228bc96ce0e786775fe640d915f250b6fc07d164736f6c6343000
80b0033
```

When generating binary for use with the deployment script we saw earlier, just add `0x` as a prefix to the generated binary string before using it in the script, to avoid any errors occurring due to hexadecimal not being recognized. Otherwise, you may see a message similar to the following:

```
Error: invalid argument 0: json: cannot unmarshal hex string without 0x prefix
into Go struct field TransactionArgs.data of type hexutil.Bytes undefined
Undefined
```

Querying contracts with Geth

We can also see the relevant message in the Geth logs to verify that the contract creation transaction has been submitted; you will see messages similar to the one shown as follows:

```
INFO [06-03|19:19:02.330] Submitted contract creation
hash=0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9
from=0x6E94BDb15141491bC3B9De3A9CaB9d87ae2af82f nonce=5
contract=0x3B52828C63Ffdcb27fA5105BC66B8F1CB1BC648C value=0
```

Also notice that in the Geth console, the following message appears as soon as the transaction is mined, indicating that the contract has been successfully mined:

```
Contract mined! address:
0x3b52828c63ffdcb27fa5105bc66b8f1cb1bc648c transactionHash:
0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9
```

Notice that in the preceding output, the transaction hash `0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9` is also shown.

After the contract is deployed successfully, you can query various attributes related to this contract, which we will also use later in this example, such as the contract address and ABI definition. Remember, all of these commands are issued via the Geth console, which we have already opened and used for contract deployment:

```
> valuechecker.
valuechecker.Matcher         valuechecker.abi             valuechecker.
allEvents       valuechecker.transactionHash
valuechecker._eth            valuechecker.address         valuechecker.
constructor     valuechecker.valueEvent
> valuechecker.abi
[{
    constant: false,
    inputs: [{
        name: "x",
        type: "uint8"
    }],
    name: "Matcher",
    outputs: [{
        name: "",
        type: "bool"
    }],
    payable: false,
    stateMutability: "nonpayable",
    type: "function"
}, {
    anonymous: false,
    inputs: [{
        indexed: false,
        name: "returnValue",
        type: "bool"
    }],
    name: "valueEvent",
```

```
    type: "event"
}]
>
```

There are a number of methods that are now exposed, and the contract can be further queried now, for example:

```
> eth.getBalance(valuechecker.address)
0
```

We can now call the actual methods in the contract. A list of the various methods that have been exposed now can be seen as follows:

```
> valuechecker.
valuechecker.Matcher         valuechecker.abi             valuechecker.
allEvents       valuechecker.transactionHash valuechecker._eth
valuechecker.address         valuechecker.constructor     valuechecker.
valueEvent
```

The contract can be further queried as follows.

First, we find the transaction hash, which identifies the transaction:

```
> valuechecker.transactionHash
```

The output of this command is the transaction hash of the contract creation transaction:

```
0xe55da8b738248c8cfa90519ed7d0985d48d4a826fa019377a118510eea1685a9
```

We can also query the ABI, using the following command:

```
> valuechecker.abi
```

The output will be as follows. Note that it shows all the inputs and outputs of our example contract:

```
> valuechecker.abi
[{
    constant: false,
    inputs: [{
        name: "x",
        type: "uint8"
    }],
    name: "Matcher",
    outputs: [{
        name: "",
        type: "bool"
    }],
    payable: false,
```

```
    stateMutability: "nonpayable",
    type: "function"
}, {
    anonymous: false,
    inputs: [{
        indexed: false,
        name: "returnValue",
        type: "bool"
    }],
    name: "valueEvent",
    type: "event"
}]
```

In the following example, the `Matcher` function is called with the arguments. Arguments, also called parameters, are the values passed to the functions. Remember that in the smart contract code, there is a condition that checks if the value is equal to or greater than 10, and if so, the function returns `true`; otherwise, it returns `false`. To test this, type the following commands into the Geth console that you have open.

Pass `12` as an argument, which will return `true` as it is greater than 10:

```
> valuechecker.Matcher.call(12)
true
```

Pass `10` as an argument, which will return `true` as it is equal to 10:

```
> valuechecker.Matcher.call(10)
true
```

Pass `9` as an argument, which will return `false` as it is less than 10:

```
> valuechecker.Matcher.call(9)
false
```

In this section, we learned how to use the Remix IDE to create and deploy contracts. We also learned how the Geth console can be used to interact with a smart contract and explored which methods are available to interact with smart contracts on the blockchain. We also introduced the Web3.js library, which allows us to interact with an Ethereum node. Now we'll see how we can interact with Geth using JSON RPC over HTTP.

Interacting with Geth using POST requests

It is possible to interact with Geth via JSON RPC over HTTP. For this purpose, the `curl` tool can be used.



`curl` is available at `https://curl.haxx.se/`.

An example is shown here to familiarize you with the POST request and show how to make POST requests using `curl`.

POST is a request method supported by HTTP which is used to send text to a web server for processing. It is widely used in webpages to send data from forms to the server and update databases.

Before using the JSON RPC interface over HTTP, the Geth client should be started up with appropriate switches, as shown here:

```
--http.api "web3 "
```

This switch will enable the `web3` interface over HTTP. The Linux command `curl` can be used for the purpose of communicating over HTTP, as shown in the following example.

For example, in order to retrieve the list of accounts using the `personal_listAccounts` method, the following command can be used:

```
$ curl -X POST -H "Content-Type: application/json" --data
'{"jsonrpc":"2.0","method":"personal_listAccounts","params":[],"id":67}'
http://localhost:8001
```

This will return the output, a JSON object with the list of accounts:

```
{"jsonrpc":"2.0","id":67,"result":["0x6e94bdb15141491bc3b9de3a9cab9d87ae2af82f",
"0x0f044cb4a0f924b6cfcf07c6c57945a0af75ec5b"]}
```

In the preceding `curl` command, `--request` is used to specify the request command, `POST` is the request, and `--data` is used to specify the parameters and values. Finally, `localhost:8545` is used where the HTTP endpoint from Geth is opened.

In this section, we covered how we can interact with the smart contract using the JSON RPC over HTTP. While this is a common way of interacting with the contracts, the examples we have seen so far are command line-based. In the next section, we'll see how we can interact with the contracts by creating a user-friendly web interface.

Interacting with contracts via frontends

So far, we have seen how we can interact with a contract using the Geth console via the command line, but in order for an application to be usable by end users, who will mostly be familiar with web interfaces, it becomes necessary to build web frontends so that users can communicate with the backend smart contracts using familiar webpage interfaces.

It is possible to interact with the contracts using the `web3.js` library from HTML-/JS-/CSS-based web-pages.

The HTML content can be served using any HTTP web server, whereas `web3.js` can connect via local RPC to the running Ethereum client (Geth) and provide an interface to the contracts on the blockchain. This architecture can be visualized in the following diagram:

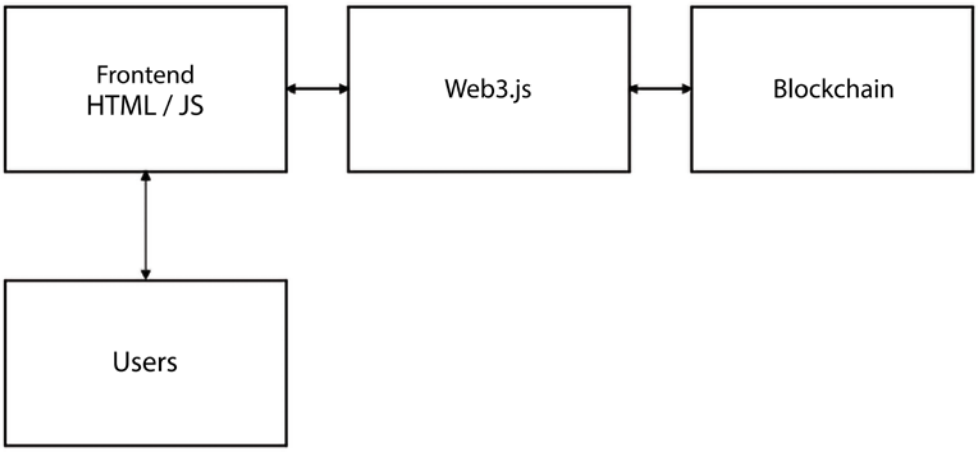


Figure 12.2: web3.js, frontend, and blockchain interaction architecture

Installing the web3.js JavaScript library

If you do not have `web3.js` installed, use these steps; otherwise, move on to the next section.

Web3, which we discussed earlier in this chapter, was looked at in the context of the Web3 API exposed by Geth. In this section, we will introduce the Web3 JavaScript library (`web3.js`), which is used to introduce different functionalities related to the Ethereum ecosystem in DApps. The `web3.js` library is a collection of several modules, which are listed as follows with the functionality that they provide.

- `web3-eth`: Ethereum blockchain and smart contracts
- `web3-shh`: Whisper protocol (P2P communication and broadcast)
- `web3-bzz`: Swarm protocol, which provides decentralized storage
- `web3-utils`: Provides helper functions for DApp development

The `web3.js` library can be installed via `npm` by simply issuing the following command:

```
$ npm install web3
```



`web3.js` can also be directly downloaded from `https://github.com/ethereum/web3.js`.

`web3.min.js`, downloaded via `npm`, can be referenced in the HTML files. This can be found under `node_modules`, for example, `cd`.



Note that `drequinox` is specific to the user under which these examples were developed; you will see the name of the user that you are running these commands under.

The file can optionally be copied into the directory where the main application is and can be used from there. Once the file is successfully referenced in HTML or JavaScript, Web3 needs to be initialized by providing an HTTP provider. This is usually the link to the `localhost` HTTP endpoint exposed by running the Geth client. This can be achieved using the following code:

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545'));
```

Once the provider is set, further interaction with the contracts and blockchain can be done using the `web3` object and its available methods.

Creating a web3 object

The first step when creating a `web3.js`-based application is to create the `web3` object. It is created by selecting the appropriate available Web3 provider, which serves as an "entry point" to the blockchain through the HTTP RPC server exposed on a locally running Geth node. The necessary code will look similar to the following:

```
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}
else
{
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost: 8545"));
}
```

This code checks whether there is already an available Web3 provider; if yes, then it will set the provider to the current provider. Otherwise, it sets the provider to `localhost: 8545`; this is where the local instance of Geth is exposing the HTTP-RPC server. In other words, the Geth instance is running an HTTP-RPC server, which is listening on port `8545`. In our case, as we started Geth with port `8001`, it will be `8001` that we use in our code. Note that there is no need to run this code, as we will create a `web3` object later in the exercise.

So far, we have explored how to install `web3.js`, the Ethereum JavaScript API library, and how to create a `web3` object that can be used to interact with the smart contracts using the HTTP provider running on the localhost as part of the Geth instance.

Creating an app.js JavaScript file

In the following section, an example will be presented that will make use of `web3.js` to allow interaction with the contracts, via a webpage served over a simple HTTP web server.

This can be achieved by following these steps. First, create a directory named `/simplecontract/app`, the home directory. This is the main directory under your user ID on Linux or macOS. This can be any directory, but in this example, the home directory is used.

Then, create a file named `app.js`, and write or copy the following code into it:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined') {
  web3 = new Web3(web3.currentProvider);
} else {
  // set the provider you want from Web3.providers
  web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8001"));
}
web3.eth.defaultAccount = web3.eth.accounts[0];
var SimpleContract = web3.eth.contract([
    {
        "constant": false,
        "inputs": [
            {
                "name": "x",
                "type": "uint8"
            }
        ],
        "name": "Matcher",
        "outputs": [
            {
                "name": "",
                "type": "bool"
            }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
```

```
                "indexed": false,
                "name": "returnValue",
                "type": "bool"
            }
        ],
        "name": "valueEvent",
        "type": "event"
    }
]);
var simplecontract = SimpleContract.
at("0x3b52828c63ffdcb27fa5105bc66b8f1cb1bc648c");
        console.log(simplecontract);
function callMatchertrue()
{
var txn = simplecontract.Matcher.call(12);
{
};
console.log("return value: " + txn);
}
function callMatcherfalse()
{
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}
function myFunction()
{
  var x = document.getElementById("txtValue").value;
    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
    document.getElementById("decision").innerHTML = txn;
}
```

This file contains various elements:

First, we created the `web3` object and provided a `localhost geth` instance listening on port `8545` as the Web3 provider.

After this, `web3.eth.accounts[0]` is selected as the account with which all the interactions will be performed with the smart contract.

Next, the ABI is provided, which serves as the interface between the user and the contract. It can be queried using Geth, generated using the Solidity compiler, or copied directly from the Remix IDE contract details.

After this, the `simplecontract` is created, which refers to the smart contract with the address `0x3b52828c63ffdcb27fa5105bc66b8f1cb1bc648c`.

Finally, we declared three functions: `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`. Once the `web3` object is correctly created and a `simplecontract` instance is created, calls to the contract functions can be made easily.

`callMatchertrue()` simply calls the smart contract function matcher using the `simplecontract` object we created in the `app.js` file we created earlier.

Similarly, `callMatcherfalse()` calls the smart contract's `Matcher` function by providing a value of `1`.

Finally, the `myFunction()` function is defined, which contains simple logic to read the value provided by the user on the webpage in a `txtValue` textbox and uses that value to call the smart contract's matcher function.

After that, the functions, the return value is also logged in the debug console, available in browsers by using `console.log`.

Calls can be made using `simplecontractinstance.Matcher.call` and then by passing the value for the argument. Recall the `Matcher` function in the Solidity code:

```
function Matcher (uint8 x) returns (bool)
```

It takes one argument `x` of type `uint8` and returns a Boolean value, either `true` or `false`. Accordingly, the call is made to the contract, as shown here:

```
var txn = simplecontractinstance.Matcher.call(12);
```

In the preceding example, `console.log` is used to print the value returned by the function call. Once the result of the call is available in the `txn` variable, it can be used anywhere throughout the program, for example, as a parameter for another JavaScript function.

Finally, check the availability of the contract. This line of code simply uses `console.log` to print the simple contract attributes, in order to verify the successful creation of the contract:

```
var simplecontract = SimpleContract.
at("0x82012b7601fd23669b50bb7dd79460970ce386e3");
console.log(simplecontract);
```

Once this call is executed, it will display various contract attributes indicating that the `web3` object has been created successfully and `HttpProvider` is available. Any other call can be used to verify the availability, but here, printing simple contract attributes has been used.

In this part of the example, we have created a `simplecontract` instance and then used `console.log` to display some attributes of the contract, which indicates the successful creation of the `simplecontract` instance.

Creating a frontend webpage

The next stage is to create the frontend webpage. For this, we create a file named `index.html` with the source code shown as follows:

```
<html>
<head>
<title>SimpleContract Interactor</title>

<script src="./web3.min.js"></script>
<script src="./app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the smart
contract.</p>
<button onclick="myFunction()">get decision</button>
<p id="decision"></p>
<p>Calling the contract manually with hardcoded values, result logged in
browser debug console:</p>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

This file will serve as the frontend of our decentralized application. In other words, it provides the UI for interacting with the smart contracts.

First, we referred to the JavaScript `web3.js` library and `app.js`, which we created earlier in this section. This will allow the HTML file to call the required functions from these files.

After that, standard HTML is used to create an input text field so that users can enter the values.

Then we used the `onclick` event to call the `myFunction()` function that we declared in our JavaScript `app.js` file.

Finally, two `onclick` events with the buttons `callTrue` and `callFalse` are used to call the `callMatchertrue()` and `callMatcherfalse()` functions, respectively.

We are keeping this very simple on purpose; there is no direct need to use jQuery, React, or Angular here, which would be a separate topic. Nevertheless, these frontend frameworks make development easier and a lot faster, and are commonly used for blockchain-related JavaScript frontend development.

In order to keep things simple, we are not going to use any frontend JavaScript frameworks in this section, as the main aim is to focus on blockchain technology and not the HTML, CSS, or JavaScript UI frameworks.

In this part of the example, we have created a web frontend and a JavaScript file backend, where we have defined all the functions required for our application.

The `app.js` file we created in the previous section is the main JavaScript file that contains the code to create a `web3` object. It also provides methods that are used to interact with the contract on the blockchain.

In the next stage of this example, we will explore how contract functions can be called.

Calling contract functions

Contract functions can be called as shown in the following code, which is part of the `app.js` file created earlier:

```
function callMatchertrue()
{
var txn = simplecontract.Matcher.call(12);
{
};
console.log("return value: " + txn);
}
function callMatcherfalse()
{
var txn = simplecontract.Matcher.call(1);{
};
console.log("return value: " + txn);
}
function myFunction()
{
  var x = document.getElementById("txtValue").value;
    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
    document.getElementById("decision").innerHTML = txn;
}
```

The preceding code shows three simple functions, `callMatchertrue()`, `callMatcherfalse()`, and `myFunction()`.

Creating a frontend webpage

Finally, the HTML file named `index.html` is created with the following code. This HTML file will serve as the frontend UI for the users, who can browse to this page served via an HTTP server to interact with the smart contract:

```
<html>
<head>
<title>SimpleContract Interactor</title>
<script src="./web3.js"></script>
<script src="./app.js"></script>
</head>

<body>

<p>Enter your value:</p>
<input type="text" id="txtValue" value="">

<p>Click the "get decision" button to get the decision from the smart
contract.</p>
<button onclick="myFunction()">get decision</button>
<p id="decision"></p>
<p>Calling the contract manually with hardcoded values, result logged in
browser debug console:</p>
<button onclick="callMatchertrue()">callTrue</button>
<button onclick="callMatcherfalse()">callFalse</button>

</body>
</html>
```

It is recommended that a web server is running in order to serve the HTML content (`index.html` as an example). Alternatively, the file can be browsed from the filesystem but that can cause some issues related to serving the content correctly with larger projects; as a good practice, always use a web server.

A web server in Python can be started using the following command. This server will serve the HTML content from the same directory that it has been run from:

```
$ python -m SimpleHTTPServer 7777
Serving HTTP on 0.0.0.0 port 7777
```

The web server does not have to be in Python; it can be an Apache server or any other web container.

If you are using Visual Studio Code, you can use the live server extension, available here: `https://marketplace.visualstudio.com/items?itemName=ritwickdey.LiveServer`. Note that, do not load the HTML file directly into the browser – it won't work. You need a web server, which can be run as shown above.

Now any browser can be used to view the webpage served over the chosen TCP port. This is shown in the following screenshot:

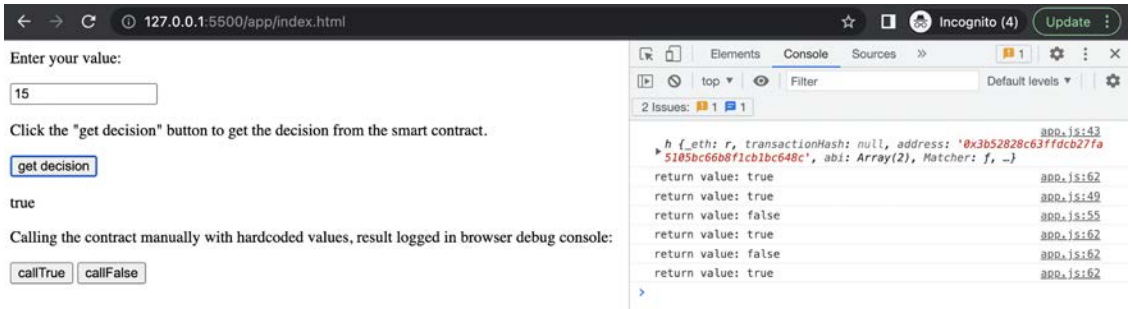


Figure 12.3: Interaction with the contract

It should be noted that the output shown here is in the browser's console window. The browser's console must be enabled in order to see the output.

As the values (12 and 1) are hardcoded in the code for simplicity, two buttons shown in the screenshot, `callTrue` and `callFalse`, have been created in `index.html`. Both of these buttons call functions with hardcoded values. This is just to demonstrate that parameters are being passed to the contract via Web3 and values are being returned accordingly.

There are three functions being called behind these buttons. We will describe them as follows:

1. The `get decision` button returns the decision from the contract:

```
<button onclick="myFunction()">get decision</button>
function myFunction()
{
  var x = document.getElementById("txtValue").value;
    var txn = simplecontract.Matcher.call(x);{
};
console.log("return value: " + txn);
    document.getElementById("decision").innerHTML = txn;
}
```

   The `get decision` button invokes the smart contract's `Matcher` function with the value entered on the webpage. The variable `x` contains the value passed to this function via the webpage, which is `12`. As the value is `12`, which is greater than 10, the `get decision` button will return `true`.

2. The `callTrue` button will call the `Matcher` function with a value that is always greater than 10, such as 12, always returning `true`. The `callMatchertrue()` method has a hardcoded value of `12`, which is sent to the contract using the following code:

```
simplecontractinstance.Matcher.call(12)
```

The return value is printed in the console using the following code, which first invokes the `Matcher` function and then assigns the value to the `txn` variable to be printed later in the console:

```
simplecontractinstance.Matcher.call(1) function callMatchertrue()
{
var txn = simplecontractinstance.Matcher.call(12);{
};
console.log("return value: " + txn);
}
```

3. The `callFalse` button invokes the `callMatcherfalse()` function. The `callMatcherfalse()` function works by passing a hardcoded value of `1` to the contract using this code:

```
simplecontractinstance.Matcher.call(1)
```

The return value is printed accordingly:

```
console.log("return value: " + txn);
function callMatcherfalse()
{
var txn = simplecontractinstance.Matcher.call(1);{
};
console.log("return value: " + txn);
}
```

Note that there is no real need for the `callTrue` and `callFalse` methods here; they are just presented for pedagogical reasons so that readers can correlate the functions with the hardcoded values and then to the called function within the smart contract, with `value` as a parameter.

This example demonstrates how the Web3 library can be used to interact with the contracts on the Ethereum blockchain. First, we created a web frontend using the JavaScript `app.js` file and the HTML file. We also included the Web3 library in our HTML so that we could create the `web3` object and use that to interact with the deployed smart contract.

In the next section, we will explore some development frameworks that aid Ethereum development, including a commonly used framework called Truffle.

Deploying and interacting with contracts using Truffle

There are various development frameworks now available for Ethereum. As seen in the examples discussed earlier, it can be quite time-consuming to deploy the contracts via manual means. This is where Truffle and similar frameworks such as Embark can be used to make the process simpler and quicker. We have chosen Truffle because it has a more active developer community and is currently the most widely used framework for Ethereum development. However, note that there is no best framework as all frameworks aim to provide methods to make development, testing, and deployment easier.

You can explore other frameworks and tools here: `https://ethereum.org/en/developers/local-environment/`

We discussed Truffle briefly in *Chapter 11*, *Tools, Languages, and Frameworks for Ethereum Developers*. In this section, we will see an example project that will demonstrate how Truffle can be used to develop a decentralized application. We will see all the steps involved in this process such as initialization, testing, migration, and deployment. First, we will see the installation process.

Installing and initializing Truffle

If Truffle is not already installed, it can be installed by running the following command:

```
$ npm install -g truffle
```

Next, Truffle can be initialized by running the following commands. First, create a directory for the project, for example:

```
$ mkdir testdapp
```

Then, change the directory to the newly created `testdapp` and run the following command:

```
$ truffle init

Starting init...
================

> Copying project files to /Users/imran/testdapp

Init successful, sweet!
```

Once the command is successful, it will create the directory structure, as shown here. This can be viewed using the `tree` command in Linux:

```
$ tree .
.
├── contracts
│   └── Migrations.sol
├── migrations
│   └── 1_initial_migration.js
├── test
└── truffle-config.js

3 directories, 3 files
```

This command creates three main directories:

- `contracts`: This directory contains Solidity contract source code files. This is where Truffle will look for Solidity contract files during migration.

- `migration`: This directory has all the deployment scripts.
- `test`: As the name suggests, this directory contains relevant test files for applications and contracts.

Finally, Truffle configuration is stored in the `truffle.js` file, which is created in the root folder of the project from where `truffle init` was run.

Now that we have initialized Truffle, let's see how it is used to compile, test, and migrate smart contracts.

Compiling, testing, and migrating using Truffle

In this section, we will demonstrate how to use various operations available in Truffle. We will learn how a sample project available with Truffle can be downloaded, and introduce how to use compilation, testing, and migration commands in Truffle to deploy and test Truffle boxes, which are essentially sample projects available with Truffle. We will use the MetaCoin Truffle box. Later, further examples will be shown on how to use Truffle for custom projects.

We will use Ganache as a local blockchain to provide the RPC interface. Make sure that Ganache is running in the background and mining.

We covered the Ganache setup in *Chapter 11*, *Tools, Languages, and Frameworks, for Ethereum Developers* . You can refer to that chapter for a refresher.

Ganache typically runs on port `7545` with 10 accounts, however, these choices can be changed in the **Settings** option in Ganache.

We will use the Ganache workspace that we saved in *Chapter 11*, *Tools, Languages, and Frameworks, for Ethereum Developers*. Alternatively, a new environment can also be set up.

After the successful setup of Ganache, the following steps need to be performed in order to unpack the webpack Truffle box and run the MetaCoin project:

1. First, create a new directory:

```
$ mkdir tproject
$ cd tproject
```

2. Now, unbox the metacoin sample from Truffle, by issuing the command below:

```
$ truffle unbox metacoin

Starting unbox...
=================

✓ Preparing to download box
✓ Downloading
✓ Cleaning up temporary files
✓ Setting up box
Unbox successful, sweet!
```

3. Edit the `truffle.js` file: if Ganache is running on a different port, then change the port from the default to where Ganache is listening.
4. Edit the file and uncomment the defaults. The file should look like the one shown here:

```
module.exports = {
  // Uncommenting the defaults below
  // provides for an easier quick-start with Ganache.
  // You can also follow this format for other networks;
  // see <http://truffleframework.com/docs/advanced/configuration>
  // for more details on how to specify configuration options!
  //
  networks: {
    development: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    },
    test: {
      host: "127.0.0.1",
      port: 7545,
      network_id: "*"
    }
  }
};
```

Now, after unboxing the webpack sample and making the necessary configuration changes, we are ready to compile all the contracts.

5. Now run the following command to compile all the contracts:

```
$ truffle compile
```

This will show the following output:

```
Compiling your contracts...
===========================
> Compiling ./contracts/ConvertLib.sol
> Compiling ./contracts/MetaCoin.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/imran/tproject/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
```

6. Now we can test the contracts using the `truffle test` command as shown here:

```
$ truffle test
```

This command will produce the output shown as follows, indicating the progress of the testing process:

```
Compiling your contracts...
===========================
> Compiling ./test/TestMetaCoin.sol
> Artifacts written to /var/folders/c5/dnw63gx93j9cl0d7r4nhc6k00000gn/T/
test--1429-MvQiDl7EIBLN
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


  TestMetaCoin
    ✓ testInitialBalanceUsingDeployedContract (151ms)
    ✓ testInitialBalanceWithNewMetaCoin (109ms)

  Contract: MetaCoin
    ✓ should put 10000 MetaCoin in the first account (53ms)
    ✓ should call a function that depends on a linked library (75ms)
    ✓ should send coin correctly (206ms)
 5 passing (5s)
```

We can also see that in Ganache, the transactions are being processed as a result of running the test:

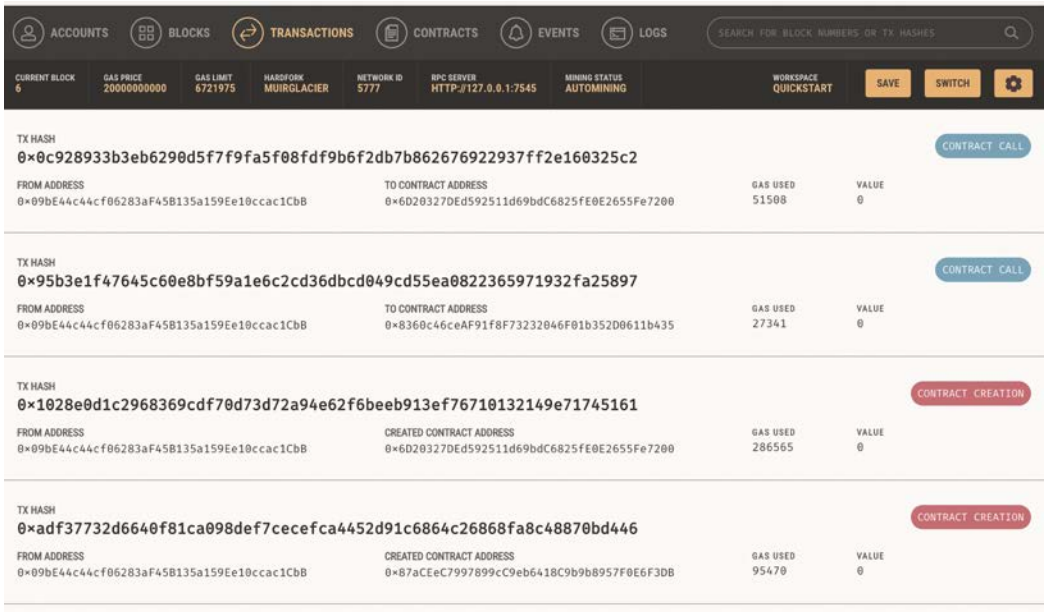


Figure 12.4: Truffle screen as a result of testing

7. Once testing is completed, we can migrate to the blockchain using the code below. Migration will use the settings available in `truffle.js` that we edited in the second step of this process to point to the port where Ganache is running. This is achieved by issuing the following command:

```
$ truffle migrate
```

The output is shown as follows. Note that not the entire output is shown, just a summary, indicating the success of the deployment:

```
Starting migrations...
Summary
=======
> Total deployments:   3
> Final cost:          0.0109242 ETH
```

Notice that when the migration runs, it will reflect on Ganache; for example, the balance of accounts will go down and you can also view transactions that have been executed. You can see **BALANCE** updating in Ganache, as the transactions run and **ETH** is consumed:

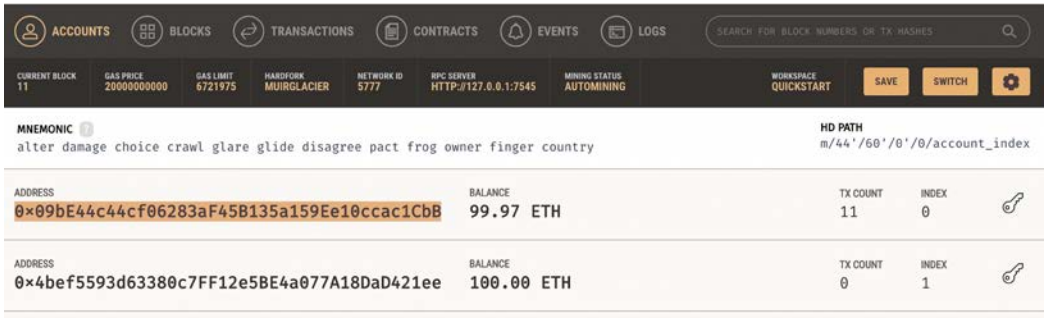


Figure 12.5: Ganache displaying accounts

In the Ganache workspace, we can also add the Truffle project to enable extra features. For this, click on the upper right-hand corner gear icon, and open the **Settings** screen. Add the Truffle project by selecting the **truffle-config.js** file in the **tproject** directory.

Save and restart to commit the changes. When Ganache is back up again, you will be able to see additional information about the contract, as shown in the following screenshot:

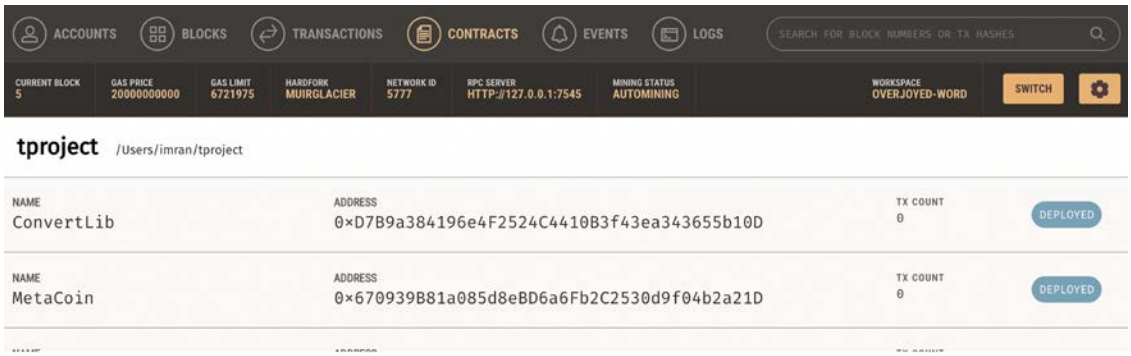


Figure 12.6: More contract details are visible after adding the Truffle project

In this section so far, we've explored how Truffle can be used to compile, test, and deploy smart contracts on the blockchain. We used Ganache, the Ethereum personal blockchain (a simulated version of the Ethereum blockchain), to perform all the exercises.



Note that you will see slightly different outputs and screens depending on your local environment and Ganache and Truffle versions.

Now, we can interact with the contract using the Truffle console. We will explore this in the following section.

Interacting with the contract

Truffle also provides a console (a CLI) that allows interaction with the contracts. All deployed contracts are already instantiated and ready to use in the console. This is a REPL-based interface, meaning Read, Evaluate, and Print Loop. Similarly, in the Geth client (via `attach` or `console`), REPL is used via exposing the **JavaScript runtime environment** (**JSRE**):

1. The console can be accessed by issuing the following command:

```
$ truffle console
```

2. This will open a CLI:

```
truffle(development)>
```

   Once the console is available, various methods can be run in order to query the contract. A list of methods can be displayed by typing the preceding command and tab-completing.

3. Other methods can also be called in order to interact with the contract; for example, in order to retrieve the address of the contract, the following method can be called using the Truffle console:

```
truffle(development)> MetaCoin.address
'0x670939B81a085d8eBD6a6Fb2C2530d9f04b2a21D'
```

4. This address is also shown in the contract creation transaction in Ganache:

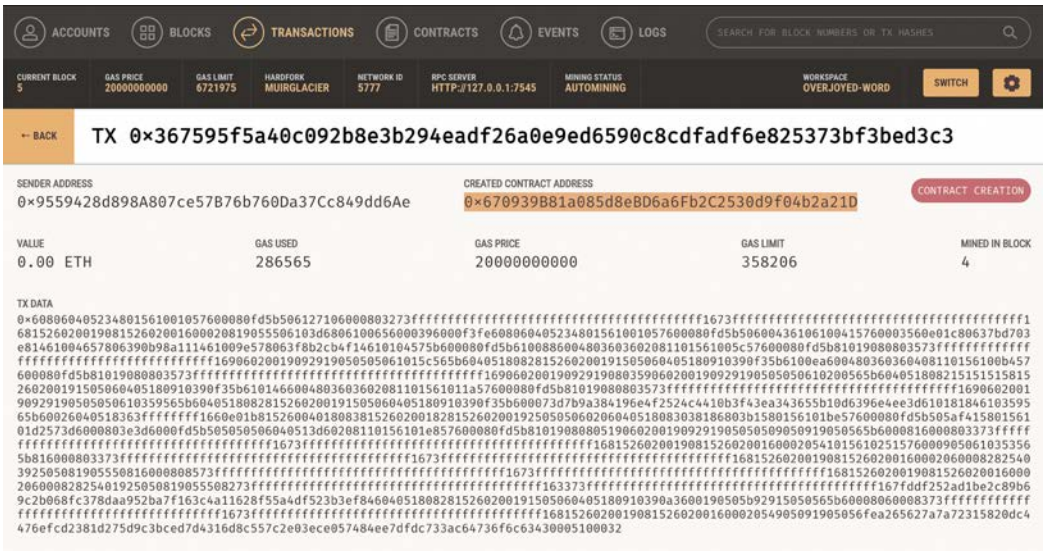


Figure 12.7: Contract creation transaction shown in Ganache

A few examples of other methods that we can call in the truffle console are shown here.

5. To query the accounts available, enter the following command:

```
truffle(development)> MetaCoin.web3.eth.getAccounts()
```

6. This will return the output shown here:

```
[
  '0x9559428d898A807ce57B76b760Da37Cc849dd6Ae',
  '0x63B9315C73AD1646e1D4b004F4A89f230f552897',
  '0xB45708b2b3d2bBD9F1Ab80a7B39568179d11eFac',
  '0x5d21E49Da087AD4A44c1ca581aB34631C071dD69',
  '0x95aCC244602Ac08db4A06198cd07C980dD17F7B3',
  '0x96A7910c8C7f4DF5F313cBa0720B744DD1A56e94',
  '0x6511B815E79BC26e9c839A8D88699c4BBA79E384',
  '0x5df2ac93A9e9D45c7832274ef5E1f116edfa90c0',
  '0x904025806f01483365587b2B339274D69A7091CD',
  '0xa96275Fb47d10Ecb1FE801f631cbAe94A3776441'
]
```

7. To query the balance of the contract, enter the following command:

```
truffle(development)> MetaCoin.web3.eth.
getBalance("0x9559428d898A807ce57B76b760Da37Cc849dd6Ae")
```

8. This will show the output shown here:

```
99987682160000000000
```

9. To end a session with the truffle console, the `.exit` command is used.

This completes our introduction to the sample webpack Truffle box and the MetaCoin application using Truffle. In this section, we discovered how Truffle can be used to interact with deployed smart contracts. MetaCoin, in this section, is an example of a decentralized application, however, we used this merely as an example to learn how Truffle works.

In the next section, we will see how a contract can be developed from scratch, tested, and deployed using Truffle, Ganache, and our privatenet.

Using Truffle to test and deploy smart contracts

This section will demonstrate how we can use Truffle for testing and deploying smart contracts. Let's look at an example of a simple contract in Solidity, which performs addition. We will see how migrations and tests can be created for this contract with the following steps:

1. Create a directory named `simple`:

```
$ mkdir simple
```

2. Change directory to `simple`:

```
$ cd simple
```

3. Initialize Truffle to create a skeleton structure for smart contract development:

```
$ truffle init
```

4. Once initialization completes, place the two files `Addition.sol` and `Migrations.sol` in the `contracts` directory. The code for both of these files is listed as follows:

`Addition.sol`:

```
pragma solidity ^0.5.0;

    contract Addition
    {
    uint8 x; //declare variable x

    // define function addx with two parameters y and z, and modifier
public
    function addx(uint8 y, uint8 z ) public
    {
    x = y + z; //performs addition
    }
    // define function retrievex to retrieve the value stored, variable x
    function retrievex() view public returns (uint8)
    {
    return x;
    }
    }
Migrations.sol:
pragma solidity >=0.4.21 <0.7.0;
contract Migrations {
  address public owner;
  uint public last_completed_migration;
  constructor() public {
    owner = msg.sender;
  }
  modifier restricted() {
    if (msg.sender == owner) _;
  }
  function setCompleted(uint completed) public restricted {
    last_completed_migration = completed;
  }
}
```

5. Now compile the contracts:

```
Compiling your contracts...
===========================
> Compiling ./contracts/Addition.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /Users/imran/simple/build/contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang
```

6. Under the `migration` folder, place two files `1_initial_migration.js` and `2_deploy_contracts.js` as shown here:

```
1_initial_migration.js:
var Migrations = artifacts.require("./Migrations.sol");
module.exports = function(deployer) {
deployer.deploy(Migrations);
};
2_deploy_contracts.js:
var SimpleStorage = artifacts.require("Addition");
module.exports = function(deployer) {
deployer.deploy(SimpleStorage);
};
```

7. Under the `test` folder, place the file `TestAddition.sol`. This will be used for unit testing:

`TestAddition.sol`:

```
pragma solidity ^0.4.2;
import "truffle/Assert.sol";
import "truffle/DeployedAddresses.sol";
import "../contracts/Addition.sol";
contract TestAddition {
  function testAddition() public {
    Addition adder = Addition(DeployedAddresses.Addition());
    adder.addx(100,100);
    uint returnedResult = adder.retrievex();
    uint expected = 200;
    Assert.equal(returnedResult, expected, "should result 200");
  }
}
```

8. The test is run using the command shown here:

```
Using network 'development'.
```

```
Compiling your contracts...
===========================
> Compiling ./test/TestAddition.sol
> Artifacts written to /var/folders/c5/dnw63gx93j9cl0d7r4nhc6k00000gn/T/
test--26262-L5233Jk29gwo
> Compiled successfully using:
   - solc: 0.8.5+commit.a4f2e591.Emscripten.clang


  TestAddition
    ✓ testAddition (417ms)


  1 passing (6s)
```

9. This means that our tests are successful. Once the tests are successful, we can deploy it to the network, in our case, Ganache:

```
$ truffle migrate
```

In Ganache we see this activity, which corresponds to our migration activity under the **Transactions** tab.

As the `Addition` contract is already instantiated and available in the truffle console, it is quite easy to interact with the contract. In order to interact with the contract, the following methods can be used:

10. Run the following command:

```
$ truffle console
```

This will open the `truffle console`, which will allow interaction with the contract. For example, in order to retrieve the address of the deployed contract, the following method can be called:

```
truffle(development)> Addition.address
0x742CD9222cccd146972d89fE491D6Bd0B1167426
```

11. To call the functions from within the contract, the deployed method is used with the contract functions. An example is shown here. First instantiate the contract:

```
truffle(development)> let additioncontract = await Addition.deployed()
undefined
```

12. Now call the `addx` function:

```
truffle(development)> additioncontract.addx(2,2)
```

This will produce the following output indicating the execution status of the transaction that was created as a result of calling the `addx` function:

```
{
  tx:
'0x7386a0cb0a4c3f57121df668efe80e2ea2eb4d8dc09d72cefbb81dddf2e39334',
  receipt: {
    transactionHash:
'0x7386a0cb0a4c3f57121df668efe80e2ea2eb4d8dc09d72cefbb81dddf2e39334',
    transactionIndex: 0,
    blockHash:
'0x951378cbaf6089b9eb71a2bd51e59571c7abc4739f48bdb92bd598f27ca752f1',
    blockNumber: 335,
    from: '0x9559428d898a807ce57b76b760da37cc849dd6ae',
    to: '0x742cd9222cccd146972d89fe491d6bd0b1167426',
    gasUsed: 42986,
    cumulativeGasUsed: 42986,
    contractAddress: null,
    logs: [],
    status: true,
    logsBloom: '0x00000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
000000000000000000',
    rawLogs: []
  },
  logs: []
}
```

We see the transaction in Ganache too, under the **Transactions** tab.

13. Finally, we can call the `retrievex` function to see the current value after `addition`:

```
truffle(development)> additioncontract.retrievex()

BN { negative: 0, words: [ 4, <1 empty item> ], length: 1, red: null }
```

Finally, we see the value 4 returned by the contract.

In this section, we created a simple smart contract that performs an addition function and learned how the Truffle framework can be used to test and deploy the smart contract. We also learned how to interact with the smart contract using the Truffle console.



Can you deploy this project using Truffle on the privatenet we created earlier? Hint: you have to change the `truffle-config.js` to point to the local private network.

In the next section, we will look at IPFS, which can serve as the decentralized storage layer for a decentralized ecosystem. We will now see how we can host one of our DApps on IPFS.

Deployment on decentralized storage using IPFS

As discussed in *Chapter 1*, *Blockchain 101*, in order to fully benefit from decentralized platforms, it is desirable that you decentralize the storage and communication layer too in addition to decentralized state/computation (blockchain). Traditionally, web content is served via centralized servers, but that part can also be decentralized using distributed filesystems. The HTML content shown in the earlier examples can be stored on a distributed and decentralized IPFS network in order to achieve enhanced decentralization.

IPFS is available at `https://ipfs.io/`.



Note that IPFS is under heavy development and is an alpha release. Therefore, there is the possibility of security bugs. Security notes are available here: `https://ipfs.io/ipfs/QmYwAPJzv5CZsnA625s3Xf2nemtYgPpHdWEz79ojWnPbdG/security-notes`

IPFS can be installed by following this process:

1. Download the IPFS package using the command below:

```
$ curl -O https://dist.ipfs.io/go-ipfs/v0.12.2/go-ipfs_v0.12.2_darwin-amd64.tar.gz
```

2. And decompress the `.gz` file:

```
$ tar -xvzf go-ipfs_v0.12.2_darwin-amd64.tar.gz
```

3. Change directory to where the files have been decompressed:

```
$ cd go-ipfs
```

4. Start the installation:

```
$ ./install.sh
```

This will produce the following output:

```
Moved ./ipfs to /usr/local/bin
```

5. Check the version to verify the installation:

```
$ ipfs --version
```

6. This will produce the following output:

```
ipfs version 0.12.2
```

7. Initialize the IPFS node:

```
$ ipfs init
```

This will produce the following output:

```
generating ED25519 keypair...done
peer identity: 12D3KooWNY2s8dKsocHQuk4sUR9TFjPxETZcf8rkTSs2NiwiiTyy
initializing IPFS node at /Users/imran/.ipfs
to get started, enter:

ipfs cat /ipfs/QmQPeNsJPyVWPFDVHb77w8G42Fvo15z4bG2X8D2GhfbSXc/readme
```

8. Enter the following command to ensure that IPFS has been successfully installed:

```
$ ipfs cat /ipfs/QmQPeNsJPyVWPFDVHb77w8G42Fvo15z4bG2X8D2GhfbSXc/readme
```

9. Now, start the IPFS daemon:

```
$ ipfs daemon
```

This will produce the following output (not the entire output is shown):

```
Initializing daemon...
 . . . .
Daemon is ready
```

10. Copy files onto IPFS using the following command:

```
$ ipfs add . --recursive --progress
```

This will produce the following output, indicating the progress of adding the pages to IPFS:

```
added QmcCGHgtAn4CpAw185Wd4VmSHsVFwbA3GJLTckMtAvqXqo simplecontract/app/
app.js
added Qmcuo7cC9ycKaSgNN12kdrpx6h6RoHbtZxyMX8d4myqLUM simplecontract/app/
index.html
. . . . .
```

Now it can be accessed in the browser by using the URL that is pointing to the IPFS filesystem: `http://localhost:8080/ipfs/Qmcuo7cC9ycKaSgNN12kdrpx6h6RoHbtZxyMX8d4myqLUM/app/`

11. Finally, in order to make the changes permanent, the following command can be used:

```
$ ipfs pin add Qmcuo7cC9ycKaSgNN12kdrpx6h6RoHbtZxyMX8d4myqLUM
```

This will show the following output:

```
pinned Qmcuo7cC9ycKaSgNN12kdrpx6h6RoHbtZxyMX8d4myqLUM recursively
```

The preceding example demonstrated how IPFS can be used to provide decentralized storage for the web part (UI) of smart contracts.

Remember that in *Chapter 1*, *Blockchain 101*, we described how a decentralized application consists of a frontend interface (usually a web interface), backend smart contracts on a blockchain, and the underlying blockchain. We have covered all these elements in this example and created a decentralized application.

Summary

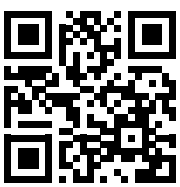
This chapter started with the introduction of Web3. We explored various methods to develop smart contracts. Also, we saw how the contract can be tested and verified using a local test blockchain before implementation on a public blockchain or private production blockchain.

We worked with various tools such as Ganache, the Geth client console, and the Remix IDE to develop, test, and deploy smart contracts. Moreover, the Truffle framework was also used to test and migrate smart contracts. We also explored how IPFS can be used to host the webpages that we created for our DApp, serving as the decentralized storage layer of the blockchain ecosystem.

In the next chapter, we will introduce how Ethereum is evolving into a more advanced form previously called Ethereum 2.0 but now called simply Ethereum - the combination of Execution layer (Eth1) and consensus layer (Eth2).

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

13

The Merge and Beyond

In this chapter, we will cover "The Merge," the name given to the Ethereum upgrade that replaces **Proof-of-Work** (**PoW**) with **Proof-of-Stake** (**PoS**) consensus in the Ethereum blockchain. We will also cover what the future holds for Ethereum and how, through many novel technical innovations and upgrades, Ethereum is achieving its eventual goal of becoming a scalable world computer.

We'll cover the following topics in this chapter.

- Introduction
- Ethereum after the Merge
- The Merge
- Sharding
- The future of Ethereum

Introduction

The goal of Ethereum is to ultimately transition into a scalable, performant, and secure version of Ethereum that will serve as the **world computer**, which is Ethereum's original vision. The concept of a world computer was introduced with Ethereum, in which a global, decentralized network of interconnected nodes runs P2P contracts without the limitation of shutting down or being censored or hacked. This vision started in 2015 with the Ethereum PoW chain and gained a lot of traction; however, challenges such as scalability, privacy, and performance somewhat hindered mass adoption. So-called Ethereum 2.0 was expected to address these issues in pursuit of becoming a world computer, however, there is no Ethereum 2.0 now; but the vision remains and the first major milestone of adopting PoS instead of PoW has been achieved. There is a rich and innovative roadmap ahead, which will ensure further efficiency and improvement in order to realize the eventual goal of a scalable world computer. This work is being done in several upgrades, which we'll cover in this chapter.

Ethereum after The Merge

Note that Ethereum 1.0 and Ethereum 2.0 terminology is no longer used. Instead, the terminologies execution layer and consensus layer are now used to describe Ethereum 1 and Ethereum 2 respectively.



Remember that now the Ethereum vision looks like this:

Ethereum 1 → execution layer

Ethereum 2 → consensus layer

Execution layer + consensus layer = Ethereum

Eth 1 and Eth 2 terminologies are no longer used, and there is only one terminology now, i.e., Ethereum, which is the current and future version of Ethereum. Individual improvements and features like the Beacon Chain, The Merge, and sharding are now called upgrades. The Ethereum network will be upgraded through several upgrades over a period of a few years to achieve its eventual goal of scalable Ethereum becoming a World Computer. We can see this as a logical division in the protocol where execution deals with the processing of the transactions inside a block and maintains the world state of the chain by creating and executing smart contracts and transfer transactions. Consensus, on the other hand, chooses which blocks to include in the canonical chain.

Execution clients are new versions of Ethereum clients already in use like Geth, Nethermind, and Besu. These clients no longer have the PoW component (ETHASH) and now solely rely on consensus clients to achieve consensus.

Consensus clients are new client software that is developed specifically for PoS Ethereum. Some examples include Lodestar, Lighthouse, Prysm, and Teku. These clients are responsible for the validation of blocks and consensus and ensuring that valid blocks make it to the canonical chain.



A full Ethereum node after The Merge is now a combination of an execution and consensus client.

The major goals behind the upgrades of Ethereum can be divided into the following different dimensions:

- **Staking**: Ethereum, from its early days, aimed for a transition to PoS. This was achieved in September 2022.
- **Sharding**: One of the main aims of Ethereum is sharding, which will improve scalability, efficiency, and performance.
- **Energy efficiency**: Ethereum became 99.95% efficient due to the move to PoS.

- **Increased security**: With the advent of quantum cryptography and to mitigate the threats faced by cryptography in the post-quantum world, it is envisaged that in Ethereum, quantum-resistant cryptography (or at least easy pluggability of quantum-resistant elements) will be introduced, so that the blockchain remains secure even in a post-quantum world. Also due to the proof-of-stake mechanism, compromising the network would be difficult.
- **Increased participation**: It is also expected that greater participation of validators will help with improved security and overall participation.
- **Increased performance**: Instead of requiring specialist hardware or high-end CPUs and GPUs for performing validation activities, in Ethereum, it is expected that validation activities will be performed by using typical consumer computers with the usual resources. With the introduction of **PoS**, a faster block time is expected. This also results in increased performance.

The following list of Ethereum's main releases shows how Ethereum has evolved over time, along with its vision of eventually achieving a world computer—a scalable, decentralized, and secure blockchain network:

- July 2015: Frontier
- March 2016: Homestead
- October 2017: Byzantium
- February 2019: Constantinople
- December 2019: Istanbul
- Late 2020: Berlin
- December 2020: Beacon Chain
- August 2021: London upgrade
- September 2022: The Merge
- 2023 and beyond: Sharding
- April 2023: Shanghai upgrade/Shapella update

As indicated here, the development of Ethereum is divided into upgrades. The most recent relevant upgrades related to the new vision of Ethereum are:

- **Beacon Chain**: Introduced staking to Ethereum and provides a foundation for future upgrades.
- **The Merge**: The switch from PoW to PoS
- **Sharding**: Consists of multiple upgrades to improve the scalability and capacity of Ethereum.

Let's first discuss the Beacon Chain in more detail.

The Beacon Chain

The introduction of the Beacon Chain marked the integration of the PoS mechanism into the Ethereum ecosystem. The Beacon Chain introduced a consensus logic and block gossip protocol, which serves to secure the Ethereum network.

The **Beacon Chain** is the core system chain and manages the **Gasper PoS** consensus mechanism (explored later in the chapter). It is also used to store and maintain the registry of validators. Validators are nodes that participate in the PoS consensus mechanism. The Beacon Chain has a number of features, listed as follows:

- Production of good-quality randomness, which will be used for selecting block proposers and attestation committees without bias. Currently, a RANDAO-based scheme is prominent; however, other options such as BLS-based and STARK-based schemes have been considered in the past. RANDAO is a pseudorandom process that selects proposers for each slot in every epoch and rearranges the validators in the committees.
- Provision of attestation, which essentially means availability votes for a block in a shard chain. An adequate number of attestations for a shard block will result in the creation of a **cross link**, which provides confirmation for shard blocks in the Beacon Chain.
- Validator and stake management.
- Provision of validator sets (committees) for voting on proposed blocks.
- Enforcement of the consensus mechanism and the reward and penalty mechanisms.
- Serving as a central system chain where shards can write their states so that cross-shard transactions can be enabled.

In order to participate in the Beacon Chain, a Beacon Chain client (or node) is required.

Beacon nodes

The **beacon node** is the primary link in the Beacon Chain that forms the central core of the Ethereum blockchain. The beacon node's responsibilities include the synchronization of the Beacon Chain with other peers. It performs the attestation of blocks and runs an RPC server to provide critical information to the validators regarding assignments and block attestation. In addition, it also handles state transition and acts as a source of randomness for the validator assignment process.



Note that we use node and client interchangeably. However, there is a subtle difference between the terms client and node: when we say client, it actually means the software client that performs the functions of consensus or execution, whereas a node can be seen as a combination of client software and the hardware (computer) that it is running on. Generally, however, client and node are used interchangeably.

The beacon node also serves as a listener for deposit events in the Ethereum chain, as well as creating and managing validator sets. A beacon node keeps a synchronized clock with other nodes on the Beacon Chain to ensure the application of penalty rules (slashing). In a beacon node, various services based on the responsibilities mentioned previously are implemented. These include the blockchain service, synchronization service, operations service, Ethereum service, public RPC server, and the P2P networking service.

To participate in the Beacon Chain, a Beacon Chain client, or consensus client, is required. Based on the Ethereum specification, this client is developed by a number of different teams. A non-exhaustive list is presented here: `https://ethereum.org/en/developers/docs/nodes-and-clients/#consensus-clients`.

Consensus client

The primary responsibility of the consensus client is to maintain synchronization with the Ethereum network by implementing the necessary logic. This process involves receiving blocks from other nodes and using the fork choice algorithm to ensure that the node always follows the chain with the most attestations. In addition, the consensus client also participates in its P2P network for block and attestation sharing.

However, the consensus client does not propose or attest blocks. Instead, this task is reserved for validators, an optional component added to the consensus client if the user stakes at least 32 ETH.

Without a validator, the consensus client only tracks the head of the Beacon Chain, ensuring that the node remains synchronized.

The consensus client mainly performs blocks and attestations gossiping over its P2P network, tracking the chain, running the fork choice algorithm, and beacon state management.

Execution client

The execution client acts as an entry point for Ethereum users into the Ethereum blockchain network. The execution client handles transactions, manages the state, and contains the **Ethereum Virtual Machine** (**EVM**), state, and transaction pool. The execution client generates execution payloads, which comprise transactions, updated state tries, and similar data. These execution payloads are included in every block by consensus clients. Additionally, the execution client, using the EVM, re-executes transactions from the blocks it receives to ensure transaction validity. Moreover, the execution client offers users an interface to interact with Ethereum through RPC calls, enabling them to query the Ethereum blockchain, submit transactions, and deploy smart contracts. RPC calls are typically managed by a library like web3.js or via a cryptocurrency wallet. Execution clients allow interaction with Ethereum via the JSON RPC API and gossip transactions over the P2P network, execute transactions, verify state changes, manage state tries and receipts tries, and create the execution payload and send it to the consensus client.

In the next section, we introduce validator nodes, which serve as block proposers and attesters on the network as part of the PoS mechanism in Ethereum.

Validator client

A **validator node**, also referred to as a validator client, actively participates in the consensus mechanism to propose blocks and provide attestations. A user can stake a minimum of 32 ETH to get the ability to verify and attest to the validity of blocks. As validators participate in the consensus mechanism to secure the protocol, they are incentivized financially for their effort. The validators earn **ether** as a staker by creating and validating new blocks on the chain.



If you want to become a staker, follow the instructions at the link here: `https://launchpad.ethereum.org/en/`.

To add a validator to their consensus client, node operators must deposit 32 ETH in to the deposit contract. The validator client is included with the consensus client and can be added to the node whenever desired. The validator is responsible for attesting and proposing blocks, and it allows the node to receive rewards, penalties, or slashing. Additionally, running the validator software makes the node a candidate to be chosen as the block proposer for a slot.

Validators do block proposals, receive rewards and penalties, and block attestations.

How to stake on Ethereum

There are four main ways to stake on Ethereum: solo staking, staking as a service, pooled staking, and staking on centralized exchanges:

- Solo staking is the most impactful, providing complete control and rewards while being trustless. However, it requires technical knowledge and a dedicated computer connected to the internet.
- Staking as a service allows you to delegate the difficult part to a service provider while you earn rewards, but you need to entrust the node operation to a provider.
- Pooled staking is a simple option that allows users to stake any amount, even if it's below 32 ETH, and earn rewards proportionally. Several solutions exist, including "liquid staking," which involves an ERC-20 liquidity token representing the staked ETH. However, pooled staking is not native to the Ethereum network and carries risks.
- Finally, centralized exchanges provide staking services but require the highest level of trust assumption and can create a single centralized target for hacking and a single point of failure, making the network more vulnerable and less secure.

A validator has to perform a number of configurations before they can deposit ether as a stake to become a validator. Once they are accepted, they become part of the validator registry that is stored and maintained on the Beacon Chain.



Instructions on how to set up an Ethereum node including various consensus layer and execution layer clients are available here: `https://ethereum.org/en/developers/docs/nodes-and-clients/run-a-node/`.

The key functions that a validator client performs are listed as follows:

- New block proposals.
- Attestation provision for blocks proposed by other validators.
- Attestation aggregation.
- Connection with a trusted beacon node to listen for new validator assignments and shuffling.

A validator is assigned a status based on its current state. It can have one of six statuses: **deposited**, **pending**, **activated**, **slashed**, **exited**, and **withdrawable**. The deposited status means that a validator has made a valid deposit and is registered in the Beacon Chain state, pending means that the validator is eligible for activation, activated means that the activator is active, slashed means the validator has lost a portion of their stake, exited means the validator has exited from the validator set, and withdrawable indicates that the validator is able to withdraw funds.

A withdrawable state occurs after a validator has exited and roughly 27 hours have passed. A validator can be in multiple states simultaneously—for example, a validator can be in both the activated and slashed states at the same time.

Honest validator behavior is incentivized as part of the PoS mechanism, and dishonest behavior results in what is referred to as **slashing**. Slashing results in the removal of a validator from the validator committee (the active validator set) and the burning of a portion of its deposited funds. Slashing serves two purposes. First, it makes it prohibitively expensive to attack Ethereum and, secondly, it encourages (enforces) validators to actively perform their duties (functions). For example, a validator going offline when it is supposed to be actively participating in the consensus is penalized for its inactivity.

There are three scenarios when slashing can occur. The first is proposer slashing, which is when a validator signs two different blocks on the Beacon Chain in the same epoch. The second case where slashing can occur is when a validator signs two conflicting attestations. Thirdly, slashing can also occur when a validator, when attesting, signs an attestation that surrounds another attestation. In other words, it means that this scenario occurs when a validator first attests to one version of the chain and then later attests to another version of the chain, resulting in confusion as to which chain the validator actually supports. When any of the three preceding scenarios occur, the offending node is reported by a **whistleblowing validator**. The whistleblowing node creates a message containing evidence of the offense, sends it to a proposer to include it in a block, and propagates it on the network. In Phase 0, the total slashing reward is taken by the proposer and the whistleblowing validator does not get any reward. This has the potential to change, where both actors might be rewarded.

Also, remember that penalization and slashing are two related but different concepts. Penalization results in a decrease in the balance of a validator as a result of, for example, being inactive or going offline. On the other hand, slashing results in a forceful exit from the Beacon Chain along with the responsible validator's deposit being penalized in every epoch for as long as it has to wait for its turn in the exit queue to leave the chain.

Slashing and penalty calculations are based on several factors with various variables such as the length of validator inactivity and the type of offense that triggered the slashing. Moreover, a penalty is applied at various points in the slashing process. For example, a minimum penalty of *effective balance of slashed validator / 32* is applied when a validator proposer includes the message reporting the offense from the whistleblower in a block. After that, at the beginning of each epoch, a penalty calculated as *3 * base reward* is applied. Another penalty is applied between the time of inclusion of the whistleblowing message in a block, and the time when the slashed validator is able to withdraw.

Earning rewards by staking ether in the deposit contract also depends on several factors. A simple example is that if someone stakes 32 ETH with a current price of, for example, USD 240 per ETH, with a validator uptime of 80%, the annual interest earned will be around 8%. The base reward is calculated as per the following formula from the Phase 0 specification:

```
base_reward = effective_balance * (base_reward_factor / (base_rewards_per_epoch
* sqrt(sum(active_balance))))2
```

Here, `base_reward_factor` is set at the default value of 64, the `base_rewards_per_epoch` value is 4, and `sum(active balance)` is the total staked ether across all active validators.

The reward received by a validator is determined by two factors: their effective balance and the number of validators in the network. As the number of validators increases, the overall reward issued by the network also increases, but the base reward given to each validator decreases proportionally. These factors significantly influence the APR for a staking node. The total reward is calculated by combining five distinct components, each with a specific weight that determines the extent to which it contributes to the overall reward.

These five components are source vote, target vote, head vote, sync committee reward, and proposer reward. Votes are related to the timely voting of the consensus mechanism for the source, target checkpoints, and the head vote is related to the timely voting of the correct head block. The sync committee reward is a reward for participating in a sync committee, and the proposer reward is for proposing a block in the correct slot. All these components have assigned weights that sum up to 64.

The total reward received by a validator is determined by calculating the sum of the weightings of the five reward components and dividing it by 64. A validator who has made timely source, target, and head votes, proposed a block, and participated in a sync committee can receive `64/64 * base_reward`.

The effective balance is used to calculate the proportion of rewards and penalties applied to a validator. It is based on the validator's balance and the previous effective balance. The maximum effective balance can always be only up to 32 ETH. Even if the actual balance is 1,000 ETH, the effective balance will still be 32 ETH.

Validators can earn rewards by performing actions shown in the table below.

Action	Reward frequency
`New block proposal`	Each epoch
`Block attestation`	Each epoch
`Sync committee participation`	Every 27.3 hours (256 epochs). However, for example with 300,000 validators, the likelihood of a validator being chosen is very low and can happen roughly every 22 months on average as the number of validators grow the likelihood of selection will decrease accordingly.

Overall, there are three main components: the Beacon Chain, beacon nodes, and validator nodes. There are some key differences between a beacon node and a validator node, which are listed in the following table.

In the Ethereum Beacon Chain (consensus layer), there are two distinct types of nodes: Beacon Chain nodes and validator nodes. The key differences between these node types are presented in the following table.

Feature	Beacon nodes	Validator nodes
`Networking`	Connected via P2P to other beacon nodes	Dedicated connection with a single beacon node
`Staking`	No staking requirements to participate in the network	Ether staking required to participate in the network
`Block creation`	Attest validations and propagate blocks across the beacon chain	Propose and sign blocks
`Operation`	Read	Write

Being a PoS-based blockchain, Ethereum must have the ability to take deposits from users as a stake. This requirement is addressed by a deposit contract.

Let's now explain PoS in a bit more detail.

Proof-of-stake

In the PoS mechanism, validators stake their funds in a smart contract on the Ethereum chain. This stake serves as collateral that can be lost if the validator behaves dishonestly or negligently. Validators are responsible for checking the validity of new blocks and occasionally creating and propagating new blocks themselves.

PoS has several advantages over the PoW mechanism. It is more energy-efficient, as it does not require significant energy consumption for computational purposes. It also has lower barriers to entry, as high-end, expensive specialized hardware like ASICs is not required to participate in the consensus process. PoS also reduces the risk of centralization by encouraging more nodes to secure the network. Furthermore, a low amount of ether issuance is needed to incentivize participation because it requires low energy. In the event of a 51% attack, PoS imposes economic penalties for misconduct, making it much more costly for attackers than PoW. The community can also use social recovery mechanisms to recover an honest chain if a 51% attack occurs and overwhelms the crypto-economic defenses.

To become a validator in Ethereum's PoS system, a user must deposit 32 ETH into the deposit contract and use three software components: an execution client, a consensus client, and a validator. Upon deposit, the user enters an activation queue, which also serves as a mechanism to limit the number of new validators joining the network. Once activated, validators receive new blocks from their peers on the Ethereum network. The role of a validator here is to check the validity of the transactions in the received block and signature before voting in favor of the block and announcing the attestation to the network.

Unlike PoW, where the timing of blocks is determined by mining difficulty, PoS has a fixed block production frequency. Time is divided into slots of 12 seconds, and 32 slots make up an epoch of 6.4 minutes.

A period of 2,048 epochs, roughly 9.1 days, is called an eek or "Ethereum week." It can be used to measure processes that take a long time.

We can visualize this in *Figure 13.1* below:

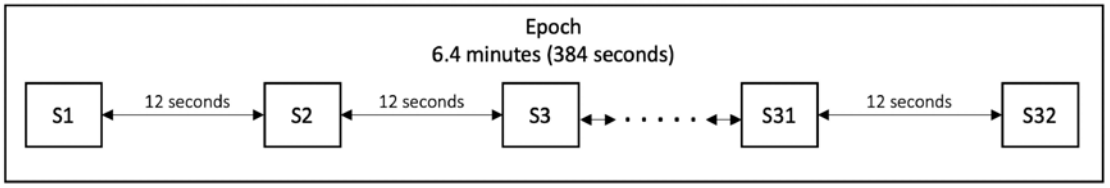


Figure 13.1: An epoch consists of 32 slots (S1 to S32)

In each slot, a single validator is chosen randomly through RANDAO to propose a new block and broadcast it to other nodes. Additionally, a committee of validators is randomly selected in each slot, and their votes are used to determine the validity and eventual acceptance of the proposed block.

Note that not every validator participates in voting for every block. Instead, they are assigned to a "committee" for each epoch, and each committee is randomly assigned a slot to attest to the proposed block's validity. These committees are distributed among the 32 slots in an epoch, with a maximum of 2,048 validators per committee. This means that each validator attests to only one block per epoch, specifically the one assigned to their committee's slot. Consequently, each slot or block is attested by 1/32 of the entire validator set.

We can visualize this in *Figure 13.2* below:

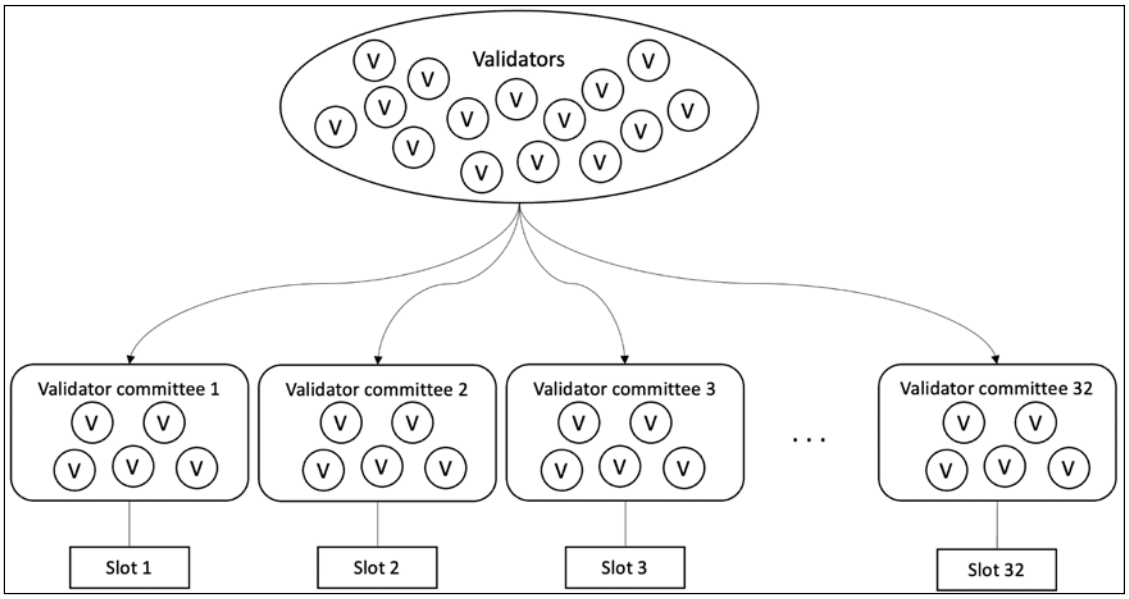


Figure 13.2: One committee assigned to one slot

In addition to attesting to the current head block, validators also attest to two other blocks referred to as "checkpoint" blocks. Each epoch features a single checkpoint block that designates the most recent block at the epoch's start.

During every epoch, validators validate two checkpoints: the "source" and "target" blocks.

Other than head, source, and target blocks, validators also include some other information in the attestation message, described below:

- `aggregation_bits`: This is a bit vector of validators where the bit position maps to the validator index in their committee. The value `0` or `1` indicates whether the validator is active and in agreement with the block proposer. In other words, whether the validator has signed (attested) the message or not.
- Data: This field contains several elements including the slot, index, beacon block root, source, and target:
    - Slot: Attested slot number
    - Index: The validator's committee number in a given slot
    - Beacon block root: Root hash of the recent block (at the head of the blockchain) observed by the validator
    - Source: Validator's view of the most recent justified block
    - Target: Validator's view of the first block in the current epoch
    - Signature: A **Boneh-Lynn-Shacham** (**BLS**) aggregate signature of the signatures of the validators.

After creating the data, the validator can change the bit in the `aggregation_bits` that corresponds to their own validator index from `0` to `1` to indicate their participation in the vote. To complete the process, the validator uses a private key to sign the attestation and sends it out to the network.

Not every validator listens to every other validator on the network. Passing attestation data by every validator to every other validator in the network would be too noisy and could cause congestion on the network. To address this issue, attestations are aggregated within the so-called "subnets" before broadcasting. These subnets can be seen as smaller networks within the larger network created by dividing the larger network. These subnets make communication efficient by reducing the distance that data needs to travel. Each committee is divided into 64 subnets to allow aggregation.

At every epoch, an aggregator is chosen from each subnet, whose role is to gather all the attestations that contain equivalent data to their own. The `aggregation_bits` field records the sender of each matching attestation. This means that all the validators who agree with the aggregator's data aggregate their signatures into a single signature. Finally, the aggregator broadcasts the aggregated attestations to the wider network. Once a validator is chosen as a block proposer, they assemble the aggregated attestations from the subnets up to the most recent slot and include them in the new block.

We can visualize the concept of subnets and an aggregator in *Figure 13.3* below:

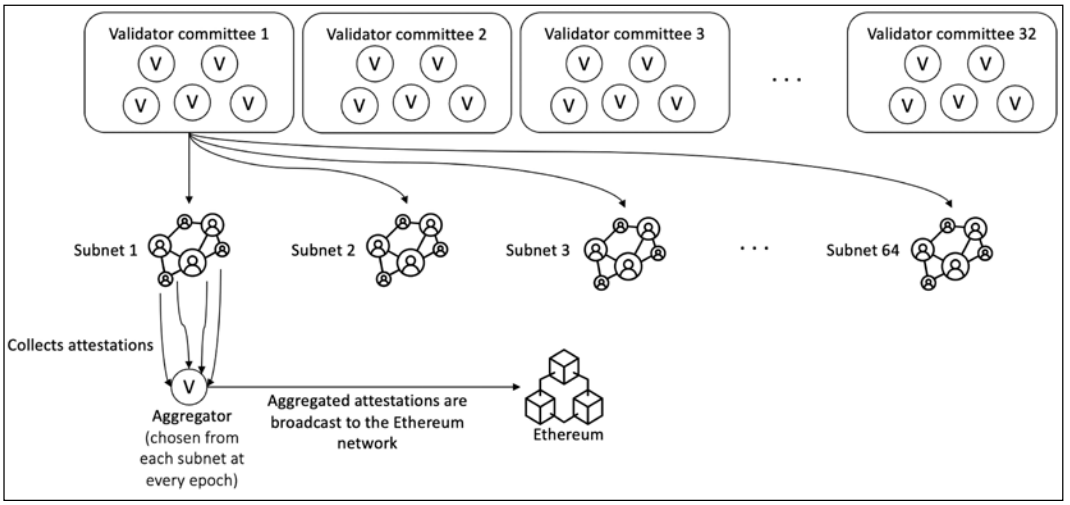


Figure 13.3: Subnets and aggregator

Now let's see how the transaction flow works in the PoS Ethereum, i.e., Ethereum after The Merge.

The PoS transaction execution flow can be broken down into several steps, as described below:

1. A transaction is composed and signed by a user's private key. Typically, this is done using a library like ether.js or web3.js. The user specifies the gas amount they will pay to incentivize validators to include the transaction in a block.
2. This transaction is submitted to an Ethereum execution client, which verifies that the sender's ETH balance is enough and the correctness of their signatures. If the transaction is valid, it's added to the client's mempool.
3. This transaction is broadcast to other nodes via the execution layer's P2P gossip network.
4. A random node on the network is chosen using RANDAO as the block proposer for the current slot. This node's role is to construct and broadcast the subsequent block to be appended to the blockchain and update the global state. Note that this node is not chosen as part of the transaction execution process; it's chosen randomly by the underlying protocol for each slot.
5. The execution client gathers transactions from the mempool and batches them into an "execution payload," which is then executed locally on the EVM to produce the new state.
6. The execution client sends this execution payload to the consensus client.
7. The consensus client (sometimes called the beacon client) encapsulates it within a "beacon block." Beacon blocks also contain information regarding rewards, penalties, attestations, and other similar information.
8. Other consensus nodes receive the beacon block via the consensus layer's P2P gossip network.
9. These clients send the payload to their respective execution clients, where the transactions are executed again locally on the EVM to validate the proposed state change.

10. The validator client determines and verifies that the block is valid and that it is the correct logical next block based on the heaviest weight of attestation defined in the fork choice rule, i.e., **Latest Message Driven Greediest Heaviest Observed Subtree** (**LMD GHOST**).
11. The block is then added to the local database of each node that has verified it.
12. A transaction is deemed "finalized" once it becomes a part of a chain with a supermajority link between two checkpoints.
13. Once an adequate number of validators has attested a block, it is appended to the head of the chain and finalized.

We can visualize the transaction flow in *Figure 13.4* below:

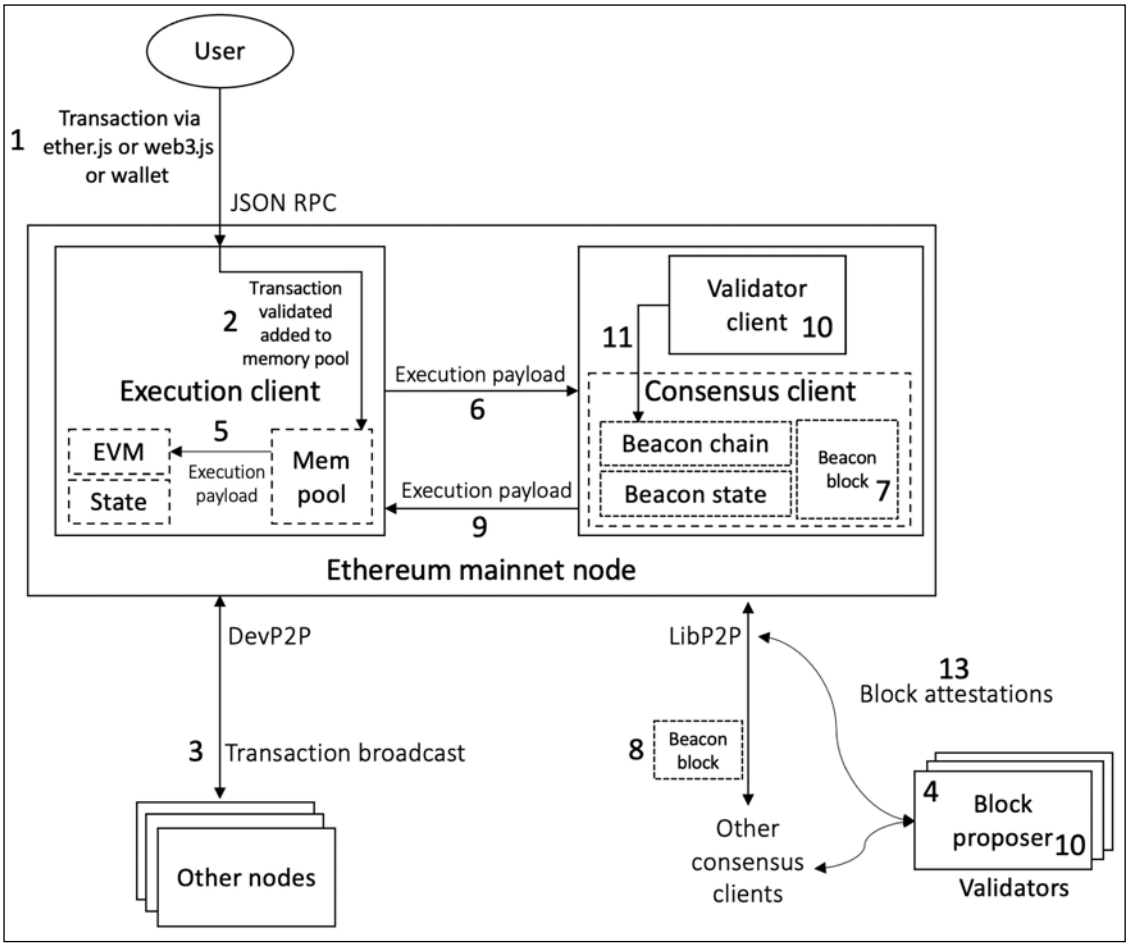


Figure 13.4: Ethereum transaction flow after The Merge

Read *Figure 13.4* using the numbers shown, which match the steps described in the transaction flow above.

Note that checkpoints are established at the start of each epoch and require a 66% attestation from the network's total staked ETH to qualify as a supermajority link.

A transaction is considered "final" when it is included in a block that cannot be changed without a significant amount of ETH being spent. Ethereum's PoS protocol achieves this through the so-called "checkpoint" blocks. The first block in each epoch serves as a checkpoint, and validators vote for pairs of checkpoints they deem valid. For example, suppose a pair of checkpoints receive votes representing at least two-thirds of the total staked ETH. In that case, the checkpoints are upgraded, with the more recent block becoming "justified" and the earlier block already justified from the previous epoch becoming "finalized." To reverse a finalized block, an attacker must commit to losing at least one-third of the total staked ETH, which is prohibitively expensive.

As finality requires a two-thirds majority, an attacker could prevent the network from reaching finality by voting with only one-third of the total stake. As a defense against this threat, an "inactivity leak" mechanism activates when the chain fails to finalize for more than four epochs. The inactivity leak gradually drains away the staked ETH from validators as a penalty for voting against the majority, enabling the majority of validators to regain a two-thirds majority and finalize the chain.

Finality, can be defined as the assurance that a block, once finalized, will not revert back. To achieve this, a mechanism called **Casper the Friendly Finality Gadget** (**Casper-FFG**) is implemented in the consensus layer. A paper on the topic is available at the following URL: `https://ethresear.ch/uploads/default/original/1X/1493a5e9434627fcf6d8ae62783b1f687c88c45c.pdf`.

In Gasper, finality is achieved using the following logic:

- If at least two thirds of the total staked ether cast their votes in favor of a block, it becomes "justified." While justified blocks are typically stable, they can still be reverted under certain conditions.
- If another block is justified on top of a justified block, it is then "finalized." Once a block is finalized, it is committed to the canonical chain and cannot be reverted unless an attacker manages to destroy a substantial amount of ether.

It is worth noting that not every slot leads to a "justified" or "finalized" state for blocks. These states are only reached by the first block of each epoch, which is referred to as the "checkpoint" block. When a checkpoint block is upgraded to justified, it must be linked to the previous checkpoint. This means that at least two thirds of the total staked ether must vote that checkpoint B is the rightful descendant of checkpoint A. As a result, the previous checkpoint block is then finalized, and the more recent block is justified.

Being a validator on the network is a serious commitment that requires maintaining proper hardware and internet connectivity to participate in the block validation and proposal process. In exchange for their services, validators are rewarded with ether, which increases their staked balance.

However, opening up participation publicly on the network as a validator also comes with risks, as it can expose the network to attacks by malicious users. As a prevention mechanism against hacking attacks, validators are penalized by slashing their stake if they fail to participate in the network as and when expected. In addition, they risk losing their entire stake if they engage in dishonest behavior. The two primary forms of dishonest behavior are:

- Equivocating, which means proposing more than one block in a single slot.
- Contradictory attestations, where a validator submits conflicting attestations.

The amount of slashed ETH depends on the number of validators being penalized simultaneously, known as the "correlation penalty." This slashing can range from a minor penalty of around 1% of the stake to losing 100% of their stake in a mass slashing event. The penalty is imposed halfway through a forced exit period. There's an immediate penalty of up to 0.5 ether on the first day. On day 18, the correlation penalty is enforced, and expulsion from the network occurs in roughly 5 weeks (36 days). In addition, validators who are present on the network but do not submit votes receive minor attestation penalties daily. As a result, a coordinated attack on the network would be prohibitively expensive for the attacker.

The fork choice rule in Ethereum is called **LMD GHOST**. Remember we discussed the **Greediest Heaviest Observed SubTree** (**GHOST**) in the context of Ethereum mining in *Chapter 9*, *Ethereum Architecture*. LMD GHOST is a variant of GHOST that was implemented in Ethereum with some modifications. More information can also be found at `https://ethereum.org/en/whitepaper/#modified-ghost-implementation`.

LMD GHOST governs a fork-handling mechanism to ensure that in the case of a fork, the correct honest chain is automatically chosen. As a general rule, the honest chain is the one that has the most attestations and stake (weight). In the event of a fork, the clients will use this fork choice rule to select the correct honest chain. Forks may occur due to the actions of colluding participants; however, the Beacon Chain's random selection of validators mitigates that to some extent, because validators do not know in advance when they will be selected. The paper on the topic is available at the following URL: `https://arxiv.org/pdf/2003.03052`.

In the next section, we introduce another important element that deals with all the networking requirements of Ethereum: the P2P interface.

P2P interface (networking)

This element deals with the networking interfaces and protocols for the Ethereum blockchain. There are three main dimensions addressed by the development of the Ethereum P2P/networking elements:

- The gossip domain
- The discovery domain
- The request/response domain

Currently, libP2P is used in various clients for this purpose. More details on this topic are available at `https://libp2p.io`.

The networking specification also covers the essentials of a test network where multiple clients can run simultaneously, that is, the interoperability of the test network and mainnet launch. In order to achieve interoperability, all Ethereum client implementations must support the TCP libp2p transport. This must also be enabled for both incoming and outgoing connections. Incoming connections are also called inbound connections or listening connections. Outgoing connections are also called outbound connections or dialing connections. Implementors may choose not to implement an IPv4 addressing scheme for the mainnet. Ethereum may implement an inbound connectivity feature only for IPv6, but the clients must support both inbound and outbound connections for both IPv4 and IPv6 addresses.

Simple Serialize, or **SSZ** for short, is the algorithm standard for providing serialization abilities for all data structures in Ethereum client software. SSZ has support for many data types, such as Boolean (`bool`), unsigned integers (`uint8`, `uint16`, `uint32`, and `uint64`), slices, arrays, structs, and pointers.

BLS cryptography (BLS12-381) is used extensively in Ethereum to provide security and efficient verification of digital signatures. **BLS** signatures, allows the aggregation of cryptographic signatures to contribute to the scalability of the network. BLS is used by validator clients to sign messages, which are then aggregated and eventually verified efficiently in the distributed network, thus increasing the overall efficiency of the network. A Go implementation of the BLS12-381 pairing is available at `https://github.com/phoreproject/bls`.

This completes our introduction to the Beacon Chain. In the next section, we explore features of The Merge.

The Merge

The Beacon Chain – the consensus layer, has been functioning independently since December 2020 and implements PoS by coordinating a network of validators using data from the Ethereum network. The Merge combined these networks, allowing Ethereum to transition to PoS as both the execution and consensus clients work together to verify the state of the network. In *Figure 13.5* below, we can visualize what Ethereum looked like before The Merge and what it looks like after The Merge:

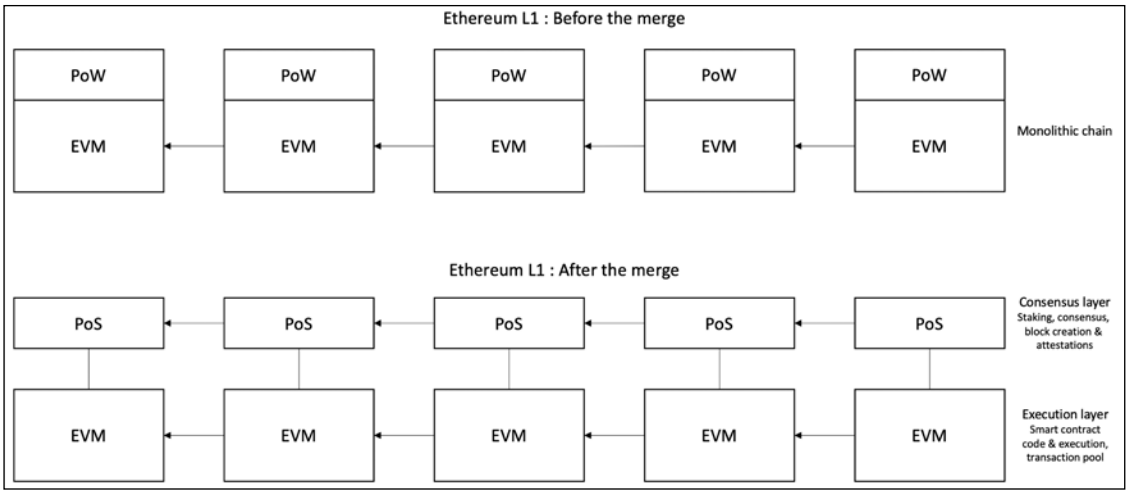


Figure 13.5: Ethereum before The Merge vs. after The Merge

Note that it is no longer possible to run an execution client alone. Following The Merge, both execution and consensus clients must be run together to allow access to the Ethereum network as shown below in *Figure 13.6*:

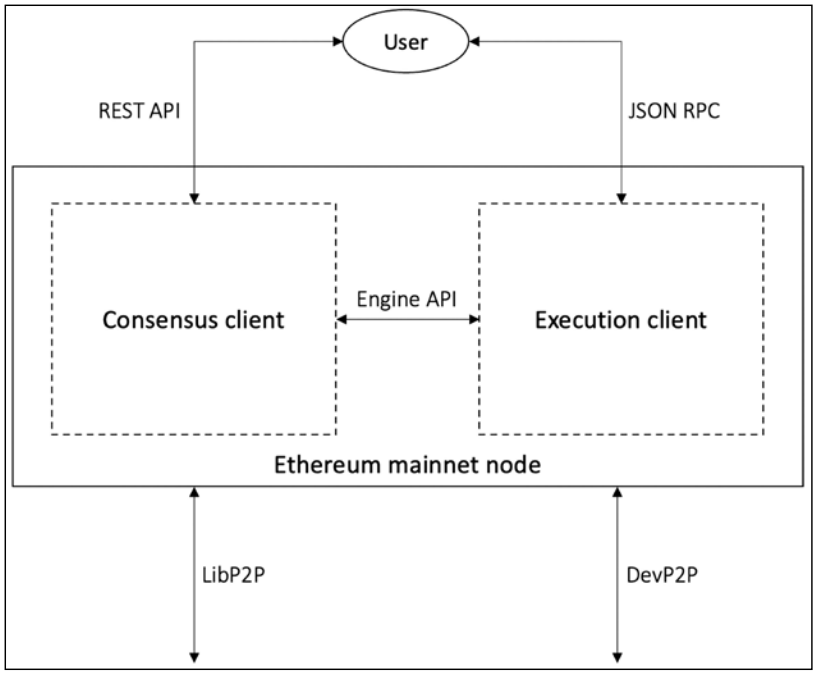


Figure 13.6: Ethereum full-node architecture after The Merge

As shown in the preceding diagram, after The Merge, two separate clients run in the full node:

- **Execution client**: The original PoW Ethereum clients that will continue to handle execution and process blocks, maintain memory pools, and synchronize blocks. The PoW component is removed.
- **Consensus client**: This is the Beacon Chain client responsible for performing PoS consensus, ensuring canonical chain emergence, block gossiping, block attestation, and receiving validator rewards.

These clients communicate with each other via the Engine API.

The diagram below shows all three node types and their relationships:

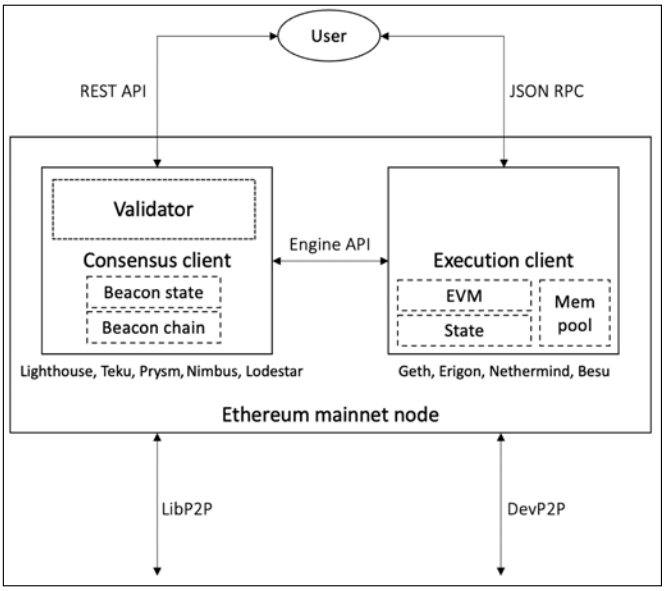


Figure 13.7: Different node types in Ethereum and their relationship

There is also a change in the block architecture after The Merge, i.e., the merge of the Beacon Chain and the old PoW chain, which is shown below:

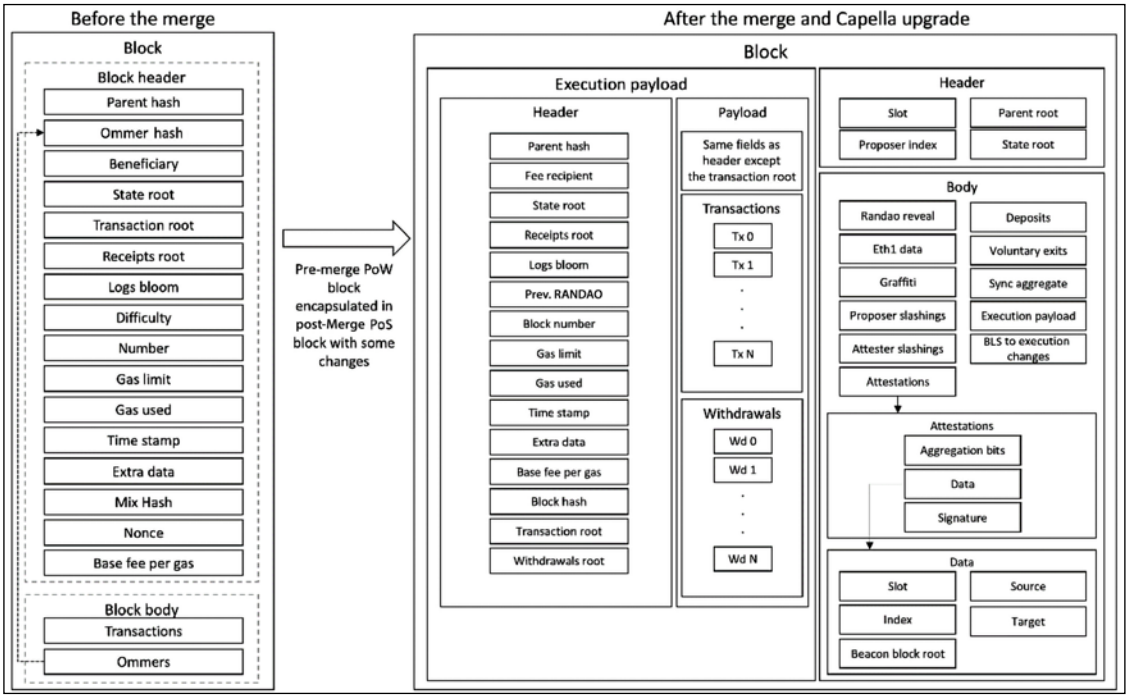


Figure 13.8: Ethereum block architecture before and after The Merge (showing main fields)

As shown in the preceding diagram after The Merge, the network no longer has PoW blocks. Instead, the contents of the old PoW blocks are integrated into blocks produced on the Beacon Chain. This change makes the Beacon Chain the new primary consensus layer for Ethereum, taking over from the previous PoW consensus layer.

The block production frequency is now 12 seconds, instead of the average 13 seconds in PoW.

The consensus layer Beacon Chain blocks contain `ExecutionPayloads`, which serve as the post-Merge equivalent of blocks on the old PoW chain. For users, interactions with Ethereum occur through these `ExecutionPayloads`.

Transactions on this layer are still processed by execution layer clients.



All changes in the block structure are described here in EIP-3675: `https://eips.ethereum.org/EIPS/eip-3675#block-structure`.

Some fields that were present in PoW block headers are no longer relevant to PoS and are therefore unused. To avoid causing too much disruption to existing tools and infrastructure, these fields are not removed from the data structure entirely but are instead set to 0 or their equivalent in the data structure. These fields include ommers, ommer hash, difficulty, and nonce.

The `mixHash` field now contains the RANDAO value.

The BLOCKHASH opcode remains available after The Merge, but its pseudo randomness is significantly weaker since it is no longer generated through the PoW hashing process. The DIFFICULTY opcode (0x44) has been updated and renamed PREVRANDAO. It returns the output of the randomness beacon provided by the Beacon Chain, making it a stronger source of randomness than BLOCKHASH, although it can still be biased. The value exposed by PREVRANDAO is stored in the `ExecutionPayload` where `mixHash`, a value associated with PoW computation, used to be stored. The `mixHash` field of the payload is renamed PREVRANDAO.

The target size of each block is set at 15 million gas units. However, the actual size of the blocks can vary based on the network demands as long as they stay within the block limit of 30 million gas units. To maintain a suitable block size, the total amount of gas used by all transactions in a block must be less than the block gas limit. This restriction is essential because it prevents blocks from becoming excessively large. If blocks could grow without limits, low-power nodes would be unable to keep up with the network due to their hardware limitations.

The larger the block, the more computing power is required to process it. Lower power nodes would miss out as a block must be processed within the next designated time slot. These requirements for more computing power would ultimately centralize the network. Therefore, the block size restriction helps prevent centralization, which could occur when only those users with high-end hardware could process the blocks, resulting in the network being used only by a select few. Limiting the block size helps to decentralize the network.

An Ethereum blockchain block contains information about the consensus, identification, execution, and other metadata necessary for the execution of the protocol. The block contains the following fields.

Field	Description
`slot`	The number of the slot the block is in
`proposer_index`	Proposing validators, ID
`parent_root`	Hash of the previous block
`state_root`	Root hash of the state tree
`body`	Contains several fields, described next

The body contains several fields as shown below.

Field	Description
`randao_reveal`	A verifiable random value provided by the proposer validator to be mixed with the RANDAO. Used for the next validator selection
`eth1_data`	Deposit contract information including deposit root, the deposit count, and blockhash
`graffiti`	Arbitrary data field – can be used for writing any information
`proposer_slashings`	The list of validators to be slashed
`attester_slashings`	The list of validators to be slashed
`attestations`	The list of attestations for the current block
`deposits`	The list of new deposits to the deposit contract
`voluntary_exits`	The list of validators exiting the network
`sync_aggregate`	A signature and a bit vector representing the sync committee (subset of validators)
`execution_payload`	Transactions sent from the execution client
`bls_to_execution_changes`	Contains a message to modify old BLS credentials to the new withdrawal credentials in execution address format to facilitate withdrawing balances

The Beacon Chain relies on a mechanism to generate in-protocol randomness called **RANDAO**. RANDAO serves as an accumulator that continuously accumulates randomness contributions from various sources. The proposer includes a random contribution to the current RANDAO value with each new block. The RANDAO value maintained by the Beacon Chain is updated with each new block added to the chain. This update occurs by mixing in the `randao_reveal` value provided by the block proposing validator. Thus, the RANDAO value gradually accumulates randomness from all block proposers over time.

The attestations field in a block consists of all the attestation associated with the block. The next validator is selected every epoch based on the RANDAO randomness.

The attestation data structure contains several fields as shown below.

Field	Description
`aggregation_bits`	A list of contributing attesting validators
`data`	A field consisting of various fields (described next)
`signature`	The aggregate signature of all attesting validators

The data field present in the attestation consists of the elements shown below.

Field	Description
`slot`	The slot number that the attestation is associated with
`index`	The indices for attesting validators
`beacon_block_root`	The root hash of the beacon block containing the data field
`source`	The last justified checkpoint
`target`	The latest epoch target block

The execution payload consists of a header and payload, both of which contain several fields. The payload contains transactions and withdrawals to be processed. When transactions in the execution payload are executed, they result in updating the global state.

All execution clients execute the transactions using the EVM as soon as they receive it from the consensus client to ensure the validity of the transactions. Practically, this means ensuring that the new state resulting from the transaction execution matches the one in the new block's state root field. This verification enables clients to verify that the new block is valid. If it is valid, it's added to the blockchain.

The execution payload header contains several fields, which we describe next.

Field	Description
`parent_hash`	The hash of the parent block
`fee_recipient`	The account address of the beneficiary for receiving transaction fees
`state_root`	The root hash of the global state after applying changes to this block
`receipts_root`	The hash of the transaction receipts trie
`logs_bloom`	The data structure containing event logs
`prev_randao`	A value used for random validator selection
`block_number`	The number of the current block
`gas_limit`	The maximum gas allowed in this block
`gas_used`	The amount of gas used in this block
`timestamp`	The block creation time
`extra_data`	Arbitrary additional data as raw bytes

Field	Description
base_fee_per_gas	The base fee value
block_hash	Hash of execution block
transactions_root	Root hash of the transactions in the payload
withdrawals_root	Root hash of the withdrawals in the payload

The execution payload contains the same fields as the header including parent_hash, fee_recipient, state_root, receipts_root, logs_bloom, prev_randao, block_number, gas_limit, gas_used, timestamp, extra_data, base_fee_per_gas, and block_hash are same as the header except that instead of the root hashes of transactions and withdrawals, it contains the actual transactions and withdrawals to be executed.

Field	Description
transactions	List of transactions to be executed
withdrawals	List of withdrawals to be executed

There is no shared code between consensus clients and execution clients. They are two separately developed artifacts. However, there are some common tasks that they perform that differ only in terms of the technology stack used. For example, networking and blockchain-related tasks are the same. Under **networking functions**, such as peer discovery, peer management, gossiping, Sybil resistance, and DoS prevention are common in both clients. Under blockchain functionality are functions such as blockchain integrity, e.g., chain reorganization handling, block synchronization, memory pools, state transition handling and relevant functions, and blockchain management.

While the functions they perform are somewhat similar and in some cases the same, the technology that has been used to achieve these is significantly different in many ways. The differences mean that, currently at least, users must run both clients. These differences are shown in the table below.

Attribute	Execution client	Consensus client
Network layer protocol	DEVP2P	LIBP2P
Discovery protocol	Discovery V4	Discovery V5
Serialization format	RLP	SSZ
Hash functions	SHA256	KECCAK
Signature scheme	ECDSA signature scheme	BLS signature scheme

This means that in the future, there is a possibility that the two could be merged, and only a single client will emerge that deals with both consensus and execution.

Some possible solutions that have been proposed are:

- A light consensus client built into the execution clients. This approach would make it easier for end users to run a basic node and reduce the system requirements. However, this approach could result in more architectural complexity and security vulnerabilities.

- Combine consensus and execution clients. This approach would make it simple for end users to run a node as it would be just a single executable. However, it could increase the architecture complexity and code complexity, reduce security, lead to no separation of concerns, and cause development complexity.

We may eventually see a single client; however, for now, it is two executables users need to run. Whether with combined or separate clients, the Ethereum ecosystem is set on growth and the future roadmap looks quite promising.

Post-Merge, Ethereum does not use PoW anymore; it uses Gasper – a combination of Casper-FFG and LMD GHOST. Gasper is an integration of **Casper the Friendly Finality Gadget** (**Casper-FFG**) and the LMD- GHOST fork choice algorithm, serving as the consensus mechanism that safeguards PoS Ethereum. The function of Casper-FFG is to promote blocks to a "finalized" state, providing assurance to new participants in the network that they are synchronizing with the authoritative (canonical) blockchain. The fork choice algorithm, on the other hand, leverages accumulated voting to facilitate the selection of the correct blockchain in case of forks. Gasper operates in a PoS environment where nodes (stakers) stake ether as collateral, which can be forfeited if they act sluggishly or dishonestly when proposing or verifying blocks. Gasper defines the rules for rewarding and penalizing validators, determining which blocks to accept or reject, and selecting which branch of the blockchain to continue building more blocks on.

Despite the many positive outcomes of The Merge, there are some issues with Ethereum's PoS, which we introduce next:

- First, there is a growing concern about centralization. This problem arose because Ethereum PoS required at least 32 ETH to run a validator, which means that users holding less than 32 ETH cannot become validators. They have to resort to using staking services, which leads to centralization due to more and more users relying on these services. There are some top service providers who make up more than 50% of the total stake. This means that, potentially, the control is in the hands of the top few stake service providers, which results in centralization. Also, a single central authority in a worldwide financial network isn't ideal, and giving someone control over your private keys who could be prone to making mistakes, vulnerable to hacking, penalized for unacceptable behavior by the network protocol, or even censored could be quite detrimental for the investor and the overall Ethereum ecosystem.
- Solo validators could be susceptible to losing keys if not managed appropriately, especially if users are inexperienced. Private key custody is a critical issue, as a solo validator might be vulnerable to malware attacks or lax user behavior. Private keys could be kept with a staking service provider, but that means giving up control of the keys, and stake service providers are also not immune to malware attacks and hacking.

In order to address the post-Merge centralization issue, **Distributed Validator Technology** (**DVT**) has been proposed. It is a technique to distribute the responsibilities of a validator between a network of multiple nodes, which will result in significantly enhancing the overall resilience of the system. In comparison to the traditional method of relying solely on a single machine to run the validator client, the utilization of DVTs offers improved safety and liveness, which not only makes the system less vulnerable to potential problems but also ensures a higher degree of stability and robustness overall.

To understand DVT better, let's first understand that there are two key risks that validator operators face on Ethereum: key theft and node failure. In the validation process, there are two private keys involved, the signing key and the withdrawal key, which can be stolen if not secured properly. Additionally, node failures can occur due to software bugs, viruses, or internet connectivity issues, resulting in the slashing of the stake. These risks can discourage some users from running a validator node, especially given the high cost of staking, i.e., 32 ETH. As a solution to this financial barrier, several centralized exchanges started to offer custodial staking solutions that allow users with less than 32 ETH to participate as validators.

However, this convenience comes at the cost of trusting the exchange and giving them full control over the ETH deposited. This approach is not in line with the decentralized ethos of blockchain, resulting in centralization. Moreover, validators are single points of failure.

Ethereum validators participate in the **PoS** protocol by signing blocks or attestations, using their private keys. These keys can only be accessed by the validator client software, which is responsible for scheduling the creation and signing of messages in accordance with the duties assigned to the validator. This traditional setup involves several risks, including:

- The centralization of the staking private key in a single location, which increases the likelihood of a potential adversary gaining access to the key and causing conflicts through the creation of conflicting messages, resulting in the slashing of the validator's deposit.
- The necessity for stakers who do not operate their own validator to entrust their staking private key to a third-party stake service provider, creating a dependency on the operator to ensure the security of the key.
- If the network experiences software crashes, loss of network connection, hardware faults, and other similar problems, the validator's stake is reduced due to "sluggish behavior." The failure of the validator client software to create timely messages results in an inactivity leak, which reduces the stake as a penalty.
- There is also a risk to a validator of following a minority fork. This can happen due to a fault in the beacon node to which the validator client is connected, which would make the validator appear offline to the rest of the PoS network.

This is where DVT technology helps as it allows an Ethereum validator to run across multiple nodes instead of only one, resulting in better security, no single point of failure, transparency, and accountability due to the **Istanbul Byzantine Fault Tolerant** (**IBFT**) consensus mechanism running between nodes and improving decentralization. The **Distributed Validator Protocol** provides a way to alleviate the risks and concerns mentioned above and other issues present in traditional validator client setups.

DVT makes use of four key components:

- Distributed key generation to split the private validator key into multiple parts so that it's split across multiple participants to take control of the key away from a single party.
- Shamir's secret sharing scheme to reconstruct the key from multiple parts.
- Multi-party computation, which allows participants to sign messages without rebuilding the entire private key on a single node, thus improving decentralization and the risk of key theft.

- The IBFT consensus algorithm is used to agree on a block by using a beacon node as a proposer and if the majority approves, then the block is added to the chain.

The diagram below shows the high-level architecture of DVT:

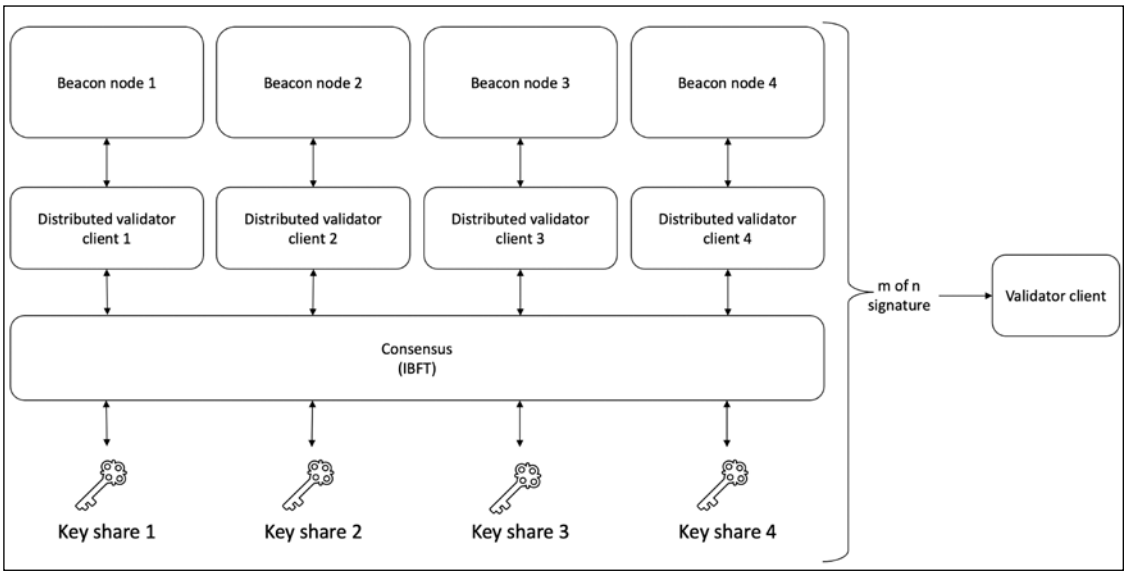


Figure 13.9: Distributed validator technology architecture

As shown in the diagram in *Figure 13.9*, DVT can be seen as a **Multi-Signature** (**M of N**) scheme to run a validator node.

An **M of N** threshold signature scheme allows a validator's staking key to be divided into **N** parts and each of the co-validators keeps a share. When **M** of the **N** co-validators reach an agreement via the consensus mechanism, the message is signed and, to the beacon nodes, it would simply be a standard and valid signed message by a validator. However, here, the signature is applied after co-validators have agreed via consensus protocol to sign the message.

The IBFT consensus mechanism allows splitting the responsibilities of a single validator into multiple co-validators who coordinate together to achieve an agreement before signing any message. The IBFT consensus protocol also allows the meeting of the safety and liveness requirements of the DV network. These requirements include safety against key theft, safety against slashing, and protection against deadlock, ensuring eventual production of a new attestation or block under a partially synchronous network with the possibility of Byzantine faults.

This combination of consensus and a threshold signature scheme ensures that consensus is secured through cryptography and that decisions are made only when at least the required threshold of co-validators agree.

After this introduction to The Merge and other innovations, let's now have a look at sharding.

Sharding

Sharding is the main scalability feature in Ethereum. It is a multi-phase enhancement to the Ethereum network to improve its scalability and capacity. This upgrade involves making the Ethereum chain amenable to rollups, makes rollups even more economical, and simplifies the operation of nodes. Sharding enables layer 2 solutions to offer low transaction fees while retaining the security of Ethereum.

The original plan was to introduce 64 separate EVM data shard chains connecting to the Beacon Chain, but this idea is no longer being pursued due to its design complexity, bad user experience, and security challenges.

The new approach is called danksharding. This is the latest scaling solution proposed by Dankrad Feist. The idea here is to shard data instead of many separate EVM shard chains. There is an Ethereum improvement proposal, EIP-4844, that proposes this, however, it is not the complete danksharding that's proposed in this EIP; instead, it proposes proto-danksharding, which allows for achieving a more feasible scaling solution in the short term before all technical challenges are addressed, before progressing with full danksharding. Proto-danksharding implements some elements of full danksharding and paves the path for full danksharding.

Rollups are the only viable trustless scaling solution for Ethereum in the near and medium-term future, and possibly even longer. High transaction fees on the Ethereum main chain, i.e., layer 1, pose a significant barrier to the adoption of new users and applications. The implementation of EIP-4844 will play a crucial role in fostering a shift toward rollups across the entire Ethereum ecosystem. This is also in line with Ethereum's rollup-centric future. Once EIP-4844 is implemented, it will make layer 2 rollup solutions more amenable to the Ethereum chain and will achieve huge scalability.

Another relevant EIP is EIP-4488, which aims to significantly reduce the gas cost of transaction calldata and also limits the total amount of transaction calldata in a block. This means that the transaction cost of Ethereum layer 2s such as Optimism or zk-rollups like zkSync will also reduce. This is, however, a short-term solution as the eventual solution to this issue will come through sharding. Sharding has not been implemented in the April 2023 Shanghai upgrade.

In later Ethereum upgrades, EIP-4844 implements proto-danksharding, which introduces blob-carrying transactions, which marks the first step toward implementing full sharding on Ethereum. This EIP introduces a new transaction format for "blob-carrying transactions," which can contain a large amount of data that is not executed by the EVM but whose commitment can be accessed. This format is designed to be fully compatible with the format that will be used in full sharding.

Rollups are a promising scaling solution for Ethereum and have already significantly reduced fees for many Ethereum users. However, they are still too expensive for some users. The ultimate solution to the limitations of rollups is data sharding, which would add 16 MB of dedicated data space to the chain per block for rollups to use. However, full data sharding is likely to take a long time to implement completely. This EIP provides a solution in the interim by implementing the transaction format that will eventually be used in full danksharding, but without actually implementing the sharding of transactions. Instead, the data from these transactions will be part of the Beacon Chain and will be fully downloadable by all consensus nodes but can be deleted after a relatively short delay.

Rollups involve posting a large amount of data, which could overburden nodes if they were required to download all of it. This would increase resource requirements and negatively impact decentralization. To address this, **Data Availability Sampling** (**DAS**) enables nodes, including light clients, to easily and securely confirm that all data has been made available without the need to download it all:

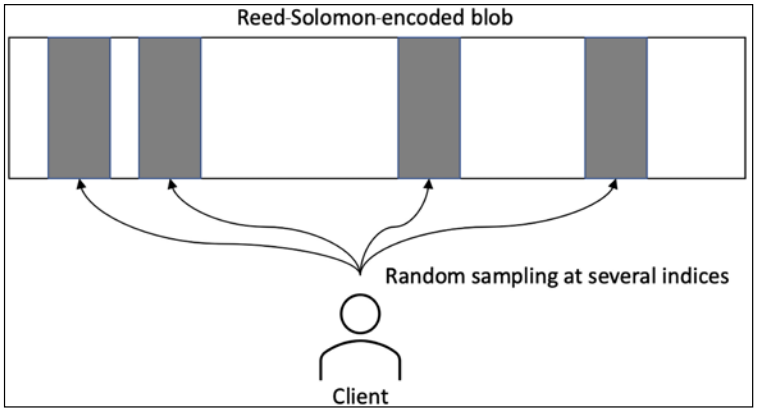


Figure 13.10: Data availability sampling

Danksharding solves the data availability problem by providing anyone with the ability to download the full data should the need arise. It uses a technique called *DAS* where a client randomly samples an erasure-coded (using Reed-Solomon codes) blob of data to ascertain the availability of data. In other words, the data is erasure-coded and extended using the Reed-Solomon codes, which allows for representing the data as a polynomial and evaluating it at various points. This technique means that now only 50% of the erasure code data is enough to ascertain the full data availability, as the other 50% can be reconstructed.

For an attacker to successfully deceive the DAS mechanism to make nodes think that the full data is available, they would have to conceal more than half of the block. With many random samples taken, the probability of less than half of the data being available becomes negligible. This technique works but the challenge here is how we ensure that the erasure coding is done correctly. What if the block producer extended the block with some useless data and the sampling came back fine, but the actual data was just junk, which cannot be used to reconstruct the data? This is where the KZG commitment scheme helps.

The KZG commitment scheme is a polynomial commitment scheme where a prover can commit to a polynomial. This means that the prover can reveal the value of the polynomial at any given point and prove that it is equal to a claimed value. Once the prover has committed to a value, the prover cannot change the polynomial later. The prover can only provide valid proofs for this specific polynomial and won't be able to produce a proof altogether or, if somehow they do, it won't be accepted by the verifier.

Sharding is also known as a phase named "The Surge," after The Merge.

The upgrades are divided into various phases, as listed below:

- **The Merge**: Already done on 15th September 2022, which switched from PoW to PoS.

- **The Surge**: Achieve scalability by making Ethereum rollup-friendly.
- **The Scourge**: Solve centralization and **Maximal Extractable Value** (**MEV**), and achieve neutral transaction inclusion.
- **The Verge**: Make block verification quick and easy by utilizing SNARKs.
- **The Purge**: Remove excess historical data, simplify the protocol, eliminate technical debt, and reduce the cost of participation.
- **The Splurge**: Solve any other remaining issues.



Note that some of these phases and terminologies could change as the roadmap is evolving; however, any new phases or efforts will be geared toward achieving the eventual goal of scalable Ethereum and making working with the layer 2 protocol easy.

Currently, in layer 2 solutions, the batch of transactions is submitted to layer 1 via *calldata*. However, with the introduction of EIP-4844, a new type of transaction called a "blob-carrying transaction" has been added, which carries "blob" data and is responsible for paying the transaction fee. The "blob-carrying transaction" includes a commitment to prove the existence of data in the blob. It is important to note that the additional content, which is the blob data, is separate from the "blob transaction" and can be considered a *sidecar*.

Danksharding is a simpler sharding design as compared to previous approaches such as multiple shard chains. The key idea here is that instead of increasing the space for transactions, sharding provides more space for data blobs, which are not interpreted by the Ethereum protocol at all. Blob verification simply involves checking that the blob is available and downloadable, which is achieved by DAS. These blobs will be used by layer 2 rollups to enable faster transaction processing and thus scalability.

The lifecycle of a blob transaction is described below:

1. The layer 2 transaction is submitted by the layer 2 users.
2. A rollup provider then creates a transaction batch and submits that to layer 1.
3. This transaction resides in the layer 1 transaction pool as a blob transaction containing blob data.
4. The blob transaction is encapsulated in the execution payload consensus client (Beacon Chain), which processes this and separates out the blob transaction in the execution payload for layer 1 execution and blob data.
5. The execution payload is executed on layer 1 and gets stored there.
6. The blob data is stored on beacon nodes for a month and layer 2 persists this data for future use.

We can visualize this flow at a network level in *Figure 13.11* below:

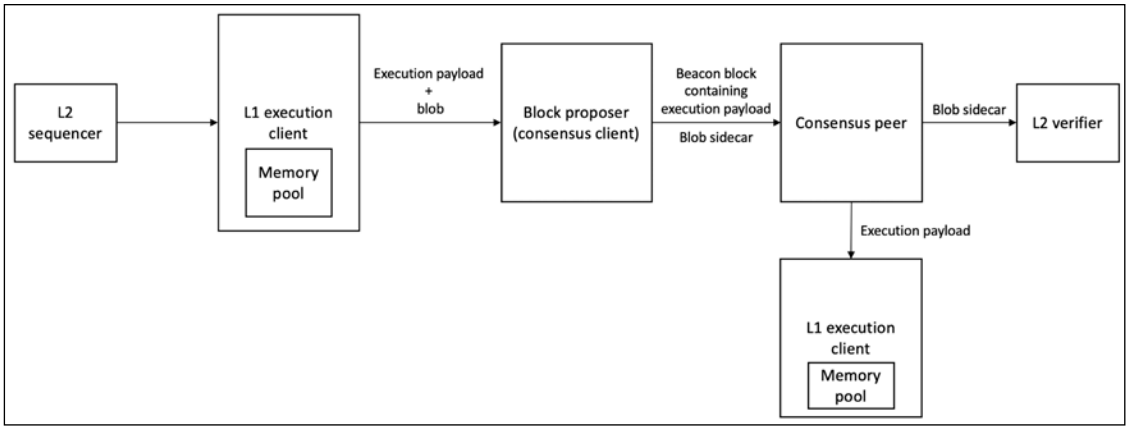


Figure 13.11: Blob transaction network flow

Figure 13.11 shows several steps, which are explained below:

1. The layer 2 sequencer submits regular transactions and blob-carrying transactions to layer 1, which ends up in the layer 1 transaction pool.
2. The layer 1 execution engine picks up the transaction and reads the execution payload and blobs.
3. The block proposer (consensus client) picks this up and creates a beacon block containing the execution payload. It also separates the blob sidecar from the blob transaction.
4. The consensus peer receives it and the execution payload is sent to the layer 1 execution engine. The execution payload contains transactions for execution.
5. The blob sidecar goes to the layer 2 verifier, which downloads these sidecars and synchronizes the layer 2 nodes.

Note that there is a distinction between the lifetimes of calldata and blob data. Calldata is present in the "execution payload" of a standard layer 1 transaction, whereas blob data is kept in the consensus layer. This means the "blob" is stored in a consensus layer client like Lighthouse or a similar node rather than in the execution layer's Geth or a similar client. The consensus layer nodes discard the blob data after a month.

The primary innovation introduced by danksharding is the **merged fee market**, where a single proposer selects all transactions and data for a specific slot, rather than multiple shards with different block proposers. However, this could lead to high system requirements for validators. To address this issue, a separate set of actors called *block builders* bid for the right to choose the contents of the slot. The *block proposer* only selects the valid header with the highest bid, and the *block builder* processes the entire block. This approach allows other validators and users to efficiently verify the blocks through DAS while reducing the computational burden on validators.

This separation of builders and proposers is called **Proposer-Builder Separation** (**PBS**). It is a promising solution to address the issue of censorship and MEV attacks in blockchain technology. With PBS, the responsibilities of constructing blocks and proposing blocks are separated, with each being assigned to distinct entities/roles within the network. Currently, in Ethereum, the block proposer has the discretion to select which transactions to include in the candidate block by evaluating the transaction in the memory pool and selecting those that have paid the highest fees.

This freedom of the block proposer allows it i.e., the block proposer to employ different sophisticated approaches for unfairly prioritizing transactions or even to include their own, to exploit opportunities such as DEX arbitrage and liquidations, to undeservedly maximize their profits. This is called MEV. Usually, due to the complexities and high cost of executing these strategies, individual validators do not perform these activities; instead, centralized pools see this as an advantage and opportunistically perform this on behalf of their pool participants. To address this issue, PBS has been proposed, which disassociates the role of block construction from that of block proposal. In this scheme, there is a new category of entities referred to as "builders" who:

1. Create units of executable block bodies, which consist of a sequence of transactions.
2. The builders then place bids, and the role of the proposer is simply to select the highest bid and accept the corresponding block body.

It's crucial to note here that the proposer and all other parties remain uninformed about the details of the block body until after the header has been selected. This pre-confirmation confidentiality is the key to preventing MEV exploits.

We can visualize the proto-danksharding EIP 4844 big-picture view in *Figure 13.12* below:

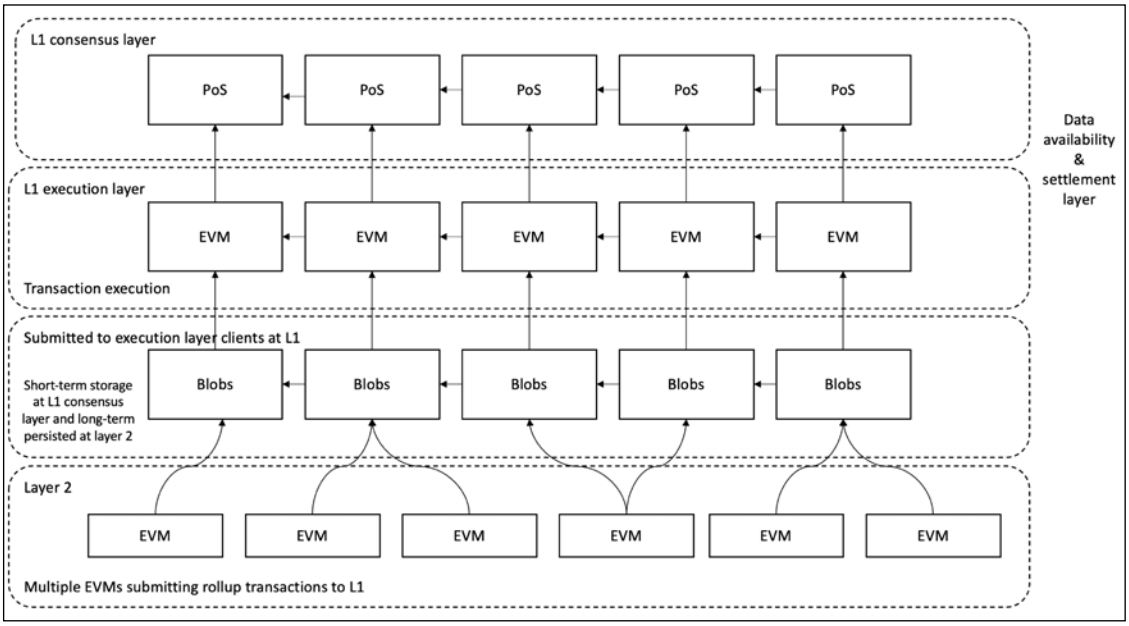


Figure 13.12: Proto-danksharding overview

As rollups post large amounts of data, it's not desirable to force nodes to download the entire data to ensure data availability. This enforcement will increase the load on the nodes, resulting in high-end hardware requirements and thus centralization risk as high-end hardware could be within reach of only a few users.

To resolve this, **DAS** has been proposed. A straightforward approach to DAS is randomly checking some parts of the block, and if they are valid, then the decision is made to accept the block. However, what if only that single chunk that contained rogue information is not checked, e.g., an illegal transfer of funds to another party? This would be detrimental to blockchain integrity.

The solution is to use error correcting codes, specifically Reed-Solomon erasure codes, which expand the data and allow data recovery by error correction. This is not a new technique; Reed-Solomon codes are used for error correction in many systems, including communication and storage technologies such as CDs, DVDs, barcodes, satellite communication, and DSL technology.

Using Reed-Solomon erasure codes, the entire block can be reconstructed as long as 50% of the erasure-coded data is available for a block. As we can see, this immediately reduces the storage requirements. From a data availability point of view, we can conclude with a very high probability, after several random samplings, that the complete block is available without needing to download the entire block. So far, so good; however, think about a scenario where the block producer is malicious and produces a rogue block with junk data. This means that while the sampling is okay, the actual data within the block is junk and not erasure-coded correctly.

To ensure that the proposer extended the block correctly with the Reed-Solomon codes, the KZG commitment scheme, a polynomial commitment scheme, is used. Polynomial commitments are discussed in *Chapter 17*, *Scalability*, in more detail.

Mathematically, we want to ensure that the original and extended data are on the same polynomial. The KZG polynomial commitment scheme allows us to commit to the original block data and its extension (from Reed-Solomon codes) and prove that they are on the same low-degree polynomial. Simply, it means the data using Reed-Solomon codes is extended correctly. So, in summary, DAS ensures the availability of erasure-coded data, and the KZG commitment proves that the original data was correctly extended.

As the first step in proto-danksharding, Ethereum validators must download shard blobs fully; however, when full danksharding is finally implemented, the validator only has to do data availability sampling.

Currently, in Ethereum, validators do two tasks: build and propose blocks, which leads to MEV. **PBS** splits these two tasks and assigns block building to builders, and proposers select the blocks. The idea here is that builders can be specialized high-power nodes or even a single powerful node that provides censorship resistance and liveness. Still, proposers are general-purpose and lightweight, meaning even users with commodity hardware can participate, thus improving decentralization. Builders in PBS have more power, which could lead to transaction censorship. The crList or censorship resistance list solution is proposed to address this issue. The idea is that proposers publish transactions from the memory pool, and builders are algorithmically mandated to include those in a block.

The proposed scheme is a hybrid PBS and comprises several steps described below:

1. The proposer publishes a censorship resistance list (crList) and its summary with all qualified transactions from the memory pool.
2. The builder proposes a block body and submits a bid that includes the hash of the crList summary, which proves that they have observed it.

3. The builder wins the bid and submits the block body. Note that the builder has yet to see the block body; it is not aware of any transactions in the block body.
4. The builder posts their block with proof that they have included all the transactions from the crList.
5. If the proof is valid, the LMD GHOST fork choice rule accepts the block, and attesters validate the published block body.

In summary, PBS allows for a scenario where, effectively, even if block production becomes centralized due to high-end hardware requirements, block verification (validation) is still decentralized and trustless while preventing transaction censorship. This scheme's overall effect still decentralizes the network, even though builders are centralized. The key idea to remember here is that block verification decentralization is more important than block production decentralization. Ethereum is eventually expected to evolve into such a mechanism by adopting danksharding and PBS. With danksharding, Ethereum will become a unified data availability and settlement layer.

Danksharding creates a tighter integration of the Beacon Chain execution block and shards. A single builder creates the entire block, and a proposer and a committee vote on it at a time. PBS is required for danksharding, as regular validators cannot handle a block full of blobs of data. With this tighter integration of execution blocks and shards, data availability can be ensured in a single go. With the introduction of large data blobs, effectively, an identical effect has been achieved, as if block size has been increased; however, the EVM does not touch the data blob at all.

We introduced LMD GHOST earlier. It is part of the Gasper consensus protocol. Gasper is the new consensus mechanism that secures the Ethereum PoS blockchain after The Merge. It is the combination of Casper-FFG, and the fork choice algorithm called LMD GHOST. The Casper-FFG mechanism finalizes blocks, ensuring new network joiners synchronize to the correct chain.

On the other hand, the fork choice algorithm uses accumulated votes to assist nodes in selecting the correct chain from forks. Gasper determines the incentives and penalties for validators, chooses which blocks to approve or decline, and picks the correct blockchain fork to use and build on. As post-Merge Ethereum is PoS-b based, the validators provide ether as a security deposit, which can be forfeited if inactive or dishonest in their block validation or proposal duties.

It's worth noting that there is no need for a fork choice rule in normal conditions where there's a single block proposer for each slot, and honest validators confirm it. However, the fork choice algorithm becomes critical in cases of significant network asynchrony or when a block proposer behaves dishonestly by making inconsistent claims. In those cases, the fork choice algorithm is essential in securing the correct blockchain.

As the first step to achieving danksharding, proto-danksharding is introduced in EIP-4844. It implements the logic, transaction formats, and verification rules that are in the full danksharding specification but does not implement the actual sharding yet. This means users still must directly validate the complete data availability. A key feature introduced by this EIP is a new transaction type called a "blob-carrying transaction". This transaction is similar to a regular Ethereum transaction, but it also has an extra piece of data called a "blob," which is roughly 125 kilobytes in size.

The key advantage here is that instead of using *calldata* to write data from rollups, these blobs can contain data from rollups, which can be a much cheaper option compared to *calldata*. Note that the EVM doesn't have access to this blob; it can only view a commitment to the blob. This approach results in scalability improvement because blobs themselves do not consume any gas and rollups can use them for data.

So, in summary, just a few key points to remember:

- Danksharding focuses primarily on data availability instead of code execution.
- The eventual aim of danksharding is to reduce the cost of rollups significantly for layer 2s and establish Ethereum's data settlement and availability.
- Proto-danksharding is the first step toward achieving danksharding, implementing many foundational functions.
- EIP-4488 implements a reduction in calldata cost from 16 gas units to 3 gas units per byte and puts a limit on the maximum calldata per block.
- EIP-4844 implements proto-danksharding, which introduces a new type of transaction called a blob-carrying transaction, which serves a similar purpose as calldata, i.e., submitting rollup batches to layer 1 for data availability, but by using blobs of data instead of using calldata, almost like a large-sized transaction, but without impacting the execution layer, as blobs are stored in beacon nodes.

In future, full danksharding will be implemented, which will make Ethereum a decentralization-and security-focused settlement and data availability layer.

We can visualize this in *Figure 13.13* below:

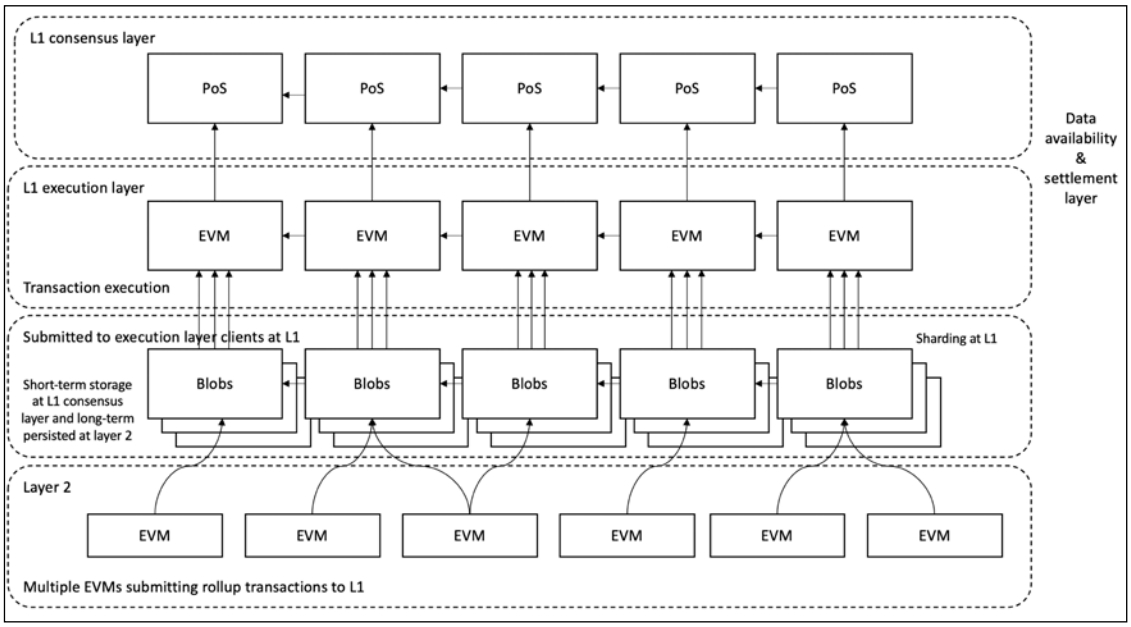


Figure 13.13: Danksharding – what Ethereum looks like after full danksharding

As shown in the preceding diagram, eventually sharding will be achieved at layer 1, with layer 2 EVMs writing blobs of data to layer 1, while Ethereum layer 1 becomes the base settlement and data availability layer, consisting of the layer 1 consensus and execution layer.

Let's now focus on what is the future road map of Ethereum and what to expect next.

The future roadmap of Ethereum

The Shanghai hard fork upgrade (also called Shapella due to combining the Capella upgrade on the Beacon Chain and the Shanghai upgrade on the execution layer) is scheduled for April 2023 and is the most important upgrade since The Merge. This includes several updates to reduce gas costs on Ethereum.

As we know, The Merge occurred in September 2022 and the Ethereum network transitioned to a PoS consensus mechanism from PoW. With the switch to PoS, users participate in validation by staking 32 ETH instead of using specialized mining hardware. However, after The Merge with the PoS Beacon Chain, users did not have any option to withdraw their staked funds. This changes with the Shanghai update (EIP-4895 – `https://eips.ethereum.org/EIPS/eip-4895`), which adds the much-needed withdrawal functionality.

It is important to acknowledge that eventually some degree of centralization may be required to attain high scalability. However, the Ethereum community is trying to avoid it as much as possible. Even if decentralization is sacrificed a bit, it is crucial to maintain the property of trustless validation and censorship resistance in the blockchain. The implementation of concepts such as PBS and weak statelessness facilitates a division between the building and validation processes, thereby facilitating scalability while preserving security and decentralization.

Current problems that Ethereum face are network congestion and excessive disk space usage. Both these problems mean that there is a sheer need to increase the speed of the network and reduce disk space usage. A simple approach to addressing these issues would be to increase centralization; however, decentralization is critical as it provides impartiality, resistance to censorship, openness, data ownership, and robust security. The aim is to be more scalable and secure but also maintain decentralization, which is not easy to achieve and has been called the "scalability trilemma." However, Ethereum appears to be on the right path, and its upgrades aim to resolve the scalability trilemma by making Ethereum more scalable, secure, and decentralized.

Scalability is achieved by sharding, which increases transaction throughput and will also reduce the validator node power requirements. Moreover, staking lowers the barriers to participation, which will result in creating a more decentralized network. We will discuss the scalability trilemma in detail in *Chapter 17*, *Scalability*.

For rollups to function optimally, cost-effective storage at the layer 1 level is required. Post-Merge upgrades, especially danksharding, will provide this and create ample room for storage, which will allow Ethereum to grow by maximizing the efficiency of rollups, enabling exponential speedups. More on rollups will be covered in *Chapter 17*, *Scalability*.

Summary

In this chapter, we looked at Ethereum after The Merge and its roadmap. We explored various aspects of its architecture and the overall vision behind the Ethereum upgrades to solve the scalability trilemma. We covered the Beacon Chain, The Merge, and sharding. We also touched upon some relevant concepts such as danksharding, PBS, and the distributed validator protocol, all leading to a more scalable and secure future for Ethereum. In the next chapter, we'll introduce Hyperledger, which is a very active and popular blockchain project upon which multiple blockchain projects are being built.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

14

Hyperledger

Hyperledger is not a blockchain but a project that was initiated by the Linux Foundation in December 2015 to advance blockchain technology. This project is a collaborative effort by its members to build an open source distributed ledger framework that can be used to develop and implement cross-industry blockchain applications and systems. The principal focus is to create and run platforms that support global business transactions. The project also focuses on improving the reliability and performance of blockchain systems.

This chapter will, for the most part, discuss Hyperledger projects, and their various features and components. As Hyperledger is so vast it's not possible to cover all projects in detail in this short chapter. As such, we will introduce a variety of the projects available, and then provide detailed discussions on Hyperledger Fabric. Projects under the Hyperledger umbrella undergo multiple stages of development, going from proposal, to incubation, and eventually graduating to an active state. Projects can also be deprecated or put in an end-of-life state where they are no longer actively developed. For a project to be able to move into the incubation stage, it must have a fully working code base along with an active community of developers. The Hyperledger project currently has more than 300 member organizations and is very active, with many contributors and meet-ups and talks organized around the globe.

We will first cover some of the various projects under Hyperledger. Then we will move on to examine the design and architecture of Hyperledger Fabric in more detail, covering the following topics on the way:

- Projects under Hyperledger
- Hyperledger reference architecture
- Hyperledger Fabric
- Fabric 2.0

Projects under Hyperledger fall into different categories, and we'll start with the introduction of these categories in the next section.

Projects under Hyperledger

There are four categories of projects under Hyperledger. Under each category, there are multiple projects. The categories are:

- Distributed ledgers
- Libraries
- Tools
- Domain-specific

We will look at five distributed ledger projects under the Hyperledger umbrella: **Fabric**, **Sawtooth**, **Iroha**, **Indy**, **and Besu**. Under libraries, we will explore the **Aries**, **Transact**, **Ursa**, and **AnonCreds** projects. Under the tools category we will look at projects such as **Cello, Caliper**, along with a domain-specific project called Hyperledger **Grid**. There are various other projects underway, (they can be reviewed at `https://www.hyperledger.org`) but for brevity, we are including only the more relevant and interesting here.

Also note that this landscape keeps changing due to developments and improvements, which you can keep an eye on here: `https://landscape.hyperledger.org/`

A brief introduction to all these projects follows.

Distributed ledgers

Distributed ledgers, as we covered in *Chapter 1*, *Blockchain 101*, are a broad category of distributed databases that are decentralized, shared among multiple participants, and updateable only via consensus.

Often, the terms *blockchain* and *distributed ledger* are used interchangeably. However, one key difference between distributed ledgers and blockchains is that blockchains are expected to have an architecture in which transactions are handled in batches of **blocks**, whereas distributed ledgers have no such requirements. All other attributes of distributed ledgers and blockchains are broadly the same. In Hyperledger, *distributed ledger* is a generic term used to denote both blockchains and distributed ledgers.

Fabric

Hyperledger Fabric is a blockchain project that was proposed by **IBM** and **Digital Asset Holdings** (**DAH**). It is an enterprise-grade permissioned distributed ledger framework for the development of blockchain solutions and applications. Fabric is based on a modular and pluggable architecture. This means that various components, such as the consensus engine and membership services, can be plugged into the system as required. Currently, its status is **active** and it is the first project to graduate from **incubation** to **active** state. We'll cover Fabric in more detail later in this chapter.



The source code is available at `https://github.com/hyperledger/fabric`

Sawtooth

Hyperledger Sawtooth is a blockchain project proposed by Intel in April 2016. It was donated to the Linux Foundation in 2016. Sawtooth introduced some novel innovations focusing on the decoupling of ledgers from transactions, flexible usage across multiple business areas using transaction families, and pluggable consensus. It is an enterprise-grade blockchain with a focus on privacy, security, and scalability.

Ledger decoupling can be explained more precisely by saying that the transactions are decoupled from the consensus layer, by making use of a new concept called **transaction families**. Instead of transactions being individually coupled with the ledger, transaction families are used, which allows more flexibility, rich semantics, and the open design of business logic. Transactions follow the patterns and structures defined in the transaction families.

One of the innovative features that Intel has introduced with Hyperledger Sawtooth is a novel consensus algorithm called **Proof of Elapsed Time** (**PoET**). It makes use of the **Trusted Execution Environment** (**TEE**) provided by **Intel Software Guard Extensions** (**Intel SGX**) to provide a safe and random leader election process. It supports both permissioned and permissionless setups.



This project is available at `https://github.com/hyperledger/sawtooth-core`

Iroha

Iroha was contributed by Soramitsu, Hitachi, NTT Data, and Colu in September 2016. Iroha aims to build a library of reusable components that users can choose to run on their own Hyperledger-based distributed ledgers.

Iroha's primary goal is to complement other Hyperledger projects by providing reusable components written in C++ with an emphasis on mobile development. This project has also proposed a novel consensus algorithm called **Sumeragi**, which is a chain-based Byzantine fault-tolerant consensus algorithm.



Iroha is available at `https://github.com/hyperledger/iroha`

Various libraries have been proposed and are being worked on by Iroha, including, but not limited to, a digital signature library (ed25519), a SHA-3 hashing library, a transaction serialization library, a P2P library, an API server library, an iOS library, an Android library, and a JavaScript library.

Indy

This project has graduated status under Hyperledger. Indy is a distributed ledger developed for building decentralized identities. It provides tools, utility libraries, and modules, which can be used to build blockchain-based digital identities. These identities can be used across multiple blockchains, domains, and applications. Indy has its own distributed ledger and uses **Redundant Byzantine Fault Tolerance** (**RBFT**) for consensus.



The source code is available at `https://github.com/hyperledger/indy-node`

Besu

Besu is a Java-based Ethereum client. It is the first project under Hyperledger that can operate on a public Ethereum network.



The source code of Besu is available at `https://github.com/hyperledger/besu`

With this, we have completed an introduction to distributed ledger projects under Hyperledger. Let's now look at the next category, libraries.

Libraries

A number of libraries are currently available under the Hyperledger project. This category includes projects that aim to support the blockchain ecosystem by introducing platforms for interoperability, identity, cryptography, and developer tools. We will now briefly describe each one of these.

Aries

Aries is not a blockchain; in fact, it is an infrastructure for blockchain-rooted, **peer-to-peer** (**P2P**) interactions. The aim of this project is to provide code for P2P interactions, secrets management, verifiable information exchange (such as verifiable credentials), interoperability, and secure messaging for decentralized systems. The eventual goal of this project is to provide a dynamic set of capabilities to store and exchange data related to blockchain-based identity.



The code for Aries is available at `https://github.com/hyperledger/aries`

Transact

Transact provides a shared library that handles smart contract execution. This library makes the development of distributed ledger software easier by allowing the handling of scheduling, transaction dispatch, and state management via a shared software library. It provides an approach to implement new smart contract development languages named **smart contract engines**. Smart contract engines implement virtual machines or interpreters for smart contract code. Two main examples of such engines are **Seth** (for EVM smart contracts) and **Sabre** (for web assembly-based smart contracts).



Transact is available at `https://crates.io/crates/transact`

Ursa

Ursa is a shared cryptography library that can be used by any project, Hyperledger or otherwise, to provide cryptography functionality. Ursa provides a C-callable library interface and a Rust crate (library). There are two sub-libraries available in Ursa: **LibUrsa** and **LibZmix**. LibUrsa is designed for cryptographic primitives such as digital signatures, standard encryption schemes, and key exchange schemes. LibZmix provides a generic method to produce zero-knowledge proofs. It supports signature **Proofs of Knowledge** (**PoK**), bullet proofs, range proofs, and set memberships.



Ursa is available at `https://github.com/hyperledger/ursa`

AnonCreds

AnonCreds stands for **Anonymous Credentials**. This is another very interesting project under incubation at Hyperledger. This is a type of verifiable credential that enables privacy protection based on zero-knowledge proofs.

We've now covered three library projects currently available under Hyperledger. In the next section, we'll explore different tools that can be used in Hyperledger projects.

Tools

There are several projects under the tools category in Hyperledger. Tools under Hyperledger focus on providing utilities and software tools that help to enhance the user experience. For example, visualization tools such as blockchain explorers, deployment tools, and benchmarking tools fall into this category. We'll describe some of these briefly.

Cello

The aim behind Cello is to allow the easy deployment of blockchains. This will provide an ability to allow **as a service** deployment of a blockchain service. Currently, this project is in the incubation stage.



The source code for Cello is available at `https://github.com/hyperledger/cello`

Caliper

Caliper is a benchmarking framework for blockchains. It can be used to measure the performance of any blockchain. There are different performance indicators supported in the framework. These include **success rate**, **transaction read rate**, **throughput**, **latency,** and **hardware resource consumption**, such as CPU, memory, and I/O. Ethereum, Fabric, Sawtooth, Burrow, and Iroha are currently the supported blockchains in Caliper.



Caliper is available at `https://github.com/hyperledger/caliper`

With this, we complete the introduction to the Hyperledger project's category of tools. Next, we'll introduce domain-specific projects.

Domain-specific

Under Hyperledger, there are also some domain-specific projects that are created to address specific requirements. The project we will explore here is called Grid.

Grid

Grid is a Hyperledger project that provides a standard reference implementation of supply chain-related data types, data models, and relevant business logic-encompassing smart contracts. It is currently in the incubation stage.



The code for Grid is available at `https://github.com/hyperledger/grid`

Each of the projects is in various stages of development. This list is expected to grow as more and more members join the Hyperledger project and contribute to the development of blockchain technology.

Now, in the next section, we will examine the reference architecture of Hyperledger, which provides general principles and design philosophies that can be followed to build new Hyperledger projects.

Hyperledger reference architecture

Hyperledger aims to build new blockchain platforms that are driven by industry use cases. As there have been many contributions made to the Hyperledger project by the community, the Hyperledger blockchain platform is evolving into a protocol for business transactions. Hyperledger is also evolving into a specification that can be used as a reference to build blockchain platforms, as compared to earlier blockchain solutions that address only a specific type of industry or requirement.

In this section, a reference architecture model is presented that has been published by the Hyperledger project. This architecture can be used by a blockchain developer to build a blockchain that is in line with the specifications of the Hyperledger architecture.

Hyperledger has published a whitepaper that presents a reference architecture model that can serve as a guideline to build permissioned distributed ledgers. The reference architecture consists of various components that form a business blockchain.



Various whitepapers related to the creation and design of Hyperledger are available here:
`https://www.hyperledger.org/learn/white-papers`

These high-level components are shown in the reference architecture diagram here, which has been drawn from the whitepaper:

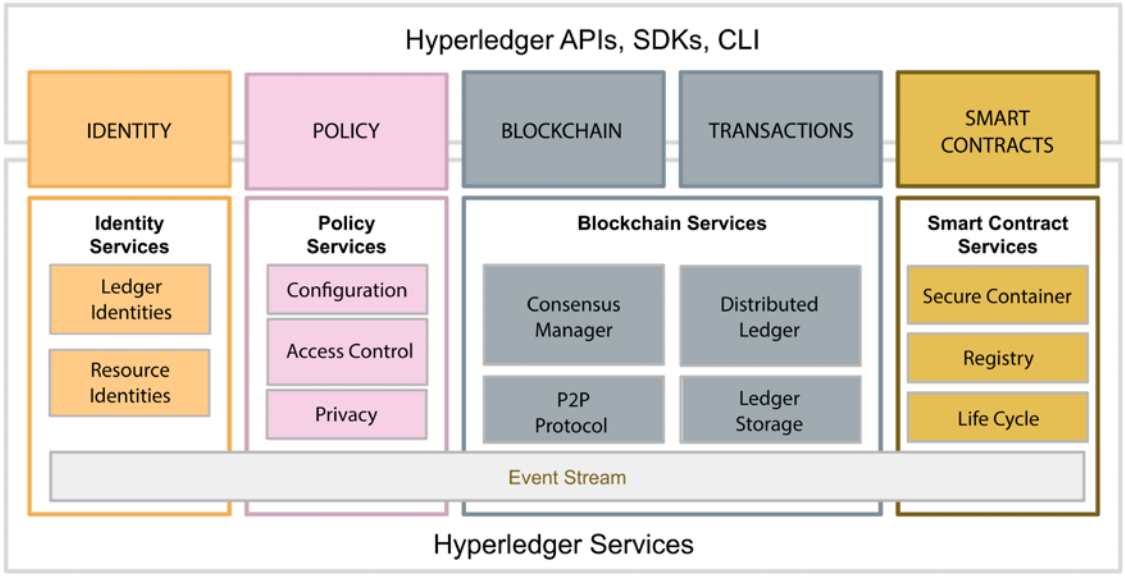


Figure 14.1: Reference architecture

In the preceding diagram, starting from the left, we see that we have five top-level components that provide various services. The first is **identity**, which provides authorization, identification, and authentication services under membership services. Then, we have the **policy** component, which provides policy services.

After this, we see **blockchain** and **transactions**, which consist of the **Consensus Manager**, **Distributed Ledger**, the network **P2P Protocol**, and **Ledger Storage** services. The consensus manager ensures that the ledger is updateable only through consensus among the participants of the blockchain network.

Finally, we have the **smart contracts** layer, which provides chaincode services in Hyperledger and makes use of **Secure Container** technology to host smart contracts. Chaincode can be considered the Hyperledger equivalent of smart contracts.



Chaincode is a program that implements a specific interface. It is usually written in Java, Node.js, or Go. Even though the terms *smart contract* and *chaincode* are used interchangeably in Hyperledger Fabric, there is a subtle difference between chaincode and smart contract. A smart contract can be defined as a piece of code that defines the transaction logic, which controls the transaction lifecycle and can result in updating the world state. Chaincode is a relevant but slightly different concept—it is a deployable object that contains smart contracts packaged within it. A single chaincode can contain multiple smart contracts and after deployment, all smart contracts contained within the chaincode are available for use. Generally, however, the terms are used interchangeably.

We will see all these in more detail in the *Hyperledger Fabric* section shortly.

From a components perspective, Hyperledger contains various elements, as described here:

- **Consensus**: These services are responsible for facilitating the agreement process between the participants on the blockchain network. Consensus is required to make sure that the order and state of transactions are validated and agreed upon in the blockchain network.
- **Smart contracts**: These services are responsible for implementing business logic as per the requirements of the users. Transactions are processed based on the logic defined in the smart contracts that reside on the blockchain.
- **Communication**: This component is responsible for message transmission and exchange between the nodes on the blockchain network.
- **Security and crypto**: These services are responsible for providing the capability to allow various cryptographic algorithms or modules to provide privacy, confidentiality, and non-repudiation services.
- **Data store**: This component provides an ability to use different data stores for storing the state of the ledger. This means that data stores are also pluggable, allowing the usage of any database backend, such as `couchdb` or `goleveldb`.
- **Policy services**: This set of services provides the ability to manage the different policies required for the blockchain network. This includes endorsement policy and consensus policy.
- **APIs and SDKs**: This component allows clients and applications to interact with the blockchain. An SDK is used to provide mechanisms to deploy and execute chaincode, query blocks, and monitor events on the blockchain.

In the next section, we are going to discuss the design goals of Hyperledger.

Hyperledger design principles

There are certain requirements for a blockchain service. The reference architecture is driven by the needs and requirements raised by the participants of the Hyperledger project after studying the industry use cases. There are several categories of requirements that have been deduced from the study of industrial use cases and are seen as principles of design philosophy. We'll describe these principles in the following sections.

- **Modular structure**: The main requirement of Hyperledger is a modular structure. It is expected that a cross-industry blockchain will be used in many business scenarios, and as such, functions related to storage, policy, chaincode, access control, consensus, and many other blockchain services should be modular and pluggable. The specification provided in the Hyperledger reference architecture suggests that the modules should be "plug and play" and users should be able to easily remove and add a different module that meets the requirements of the business.
- **Privacy and confidentiality**: This is one of the most critical factors. As traditional blockchains are permissionless, in a permissioned model it is of the utmost importance that transactions on the network are visible to only those who are allowed to view it. The privacy and confidentiality of transactions and contracts are of absolute importance in a business blockchain. As such, Hyperledger's vision is to provide support for a full range of cryptographic protocols and algorithms. We discussed cryptography in *Chapter 3*, *Symmetric Cryptography*, and *Chapter 4*, *Asymmetric Cryptography*.

  It is expected that users will be able to choose appropriate modules according to their business requirements. For example, if a business blockchain needs to be run only between already-trusted parties and performs very basic business operations, then perhaps there is no need to have advanced cryptographic support for confidentiality and privacy. Therefore, users should be able to remove that functionality (module) or replace it with a more appropriate module that suits their needs.

  Similarly, if users need to run a cross-industry blockchain, then confidentiality and privacy can be of paramount importance. In this case, users should be able to plug an advanced cryptographic and access control mechanism (module) into the blockchain, which can even allow the usage of **hardware security modules** (**HSMs**). The blockchain should also be able to handle sophisticated cryptographic algorithms without compromising performance. In addition to the previously mentioned scenarios, due to regulatory requirements in business, there should also be a provision to allow the implementation of privacy and confidentiality policies in conformance with regulatory and compliance requirements.
- **Identity**: To provide privacy and confidentiality services, a flexible **Public Key Infrastructure** (**PKI**) model that can be used to handle the access control functionality is also required. The strength and type of cryptographic mechanisms are also expected to vary according to the needs and requirements of the users. In certain scenarios, it might be required for a user to hide their identity, and as such, Hyperledger is expected to provide this functionality.
- **Scalability**: This is another major requirement that, once met, will allow reasonable transaction throughput, which will be sufficient for all business requirements and also a large number of users.

- **Deterministic transactions**: This is a core requirement in any blockchain. If transactions do not produce the same result every time they are executed, then regardless of who and where the transaction is executed, achieving consensus is impossible. Therefore, deterministic transactions become a key requirement in any blockchain network. We discussed these concepts in *Chapter 8*, *Smart Contracts*.
- **Auditability**: Auditability is another requirement of businesses. It is expected that an immutable audit trail of all identities, related operations, and any changes is kept.
- **Interoperability**: Currently, there are many blockchain platforms available, but they cannot communicate with each other easily. This can be a limiting factor in the growth of a blockchain-based global business ecosystem. It is envisaged that many blockchain networks will operate in the business world for specific needs, but it is important that they are able to communicate with each other. There should be a common set of standards that all blockchains can follow to allow communication between different ledgers. There are different efforts already underway to achieve this, not only under the Hyperledger umbrella, such as Hyperledger Quilt, but also other projects such as Cosmos and Polkadot.
- **Portability**: The portability requirement is concerned with the ability to run across multiple platforms and environments without the need to change anything at the code level. Hyperledger Fabric, for example, is envisaged to be portable, not only at the infrastructure level, but also at the code, library, and API levels, so that it can support uniform development across various implementations of Hyperledger.
- **Rich data queries**: The blockchain network should allow rich queries to be run on the network. This can be used to query the current state of the ledger using traditional query languages, such as SQL, and supporting statements like SELECT, which will allow wider adoption and ease of use.

All the points describe the general guiding principles that allow the development of blockchain solutions that are in line with the Hyperledger design philosophy.

In the next section, we will look at Hyperledger Fabric in detail, which is the first project to graduate to active status under Hyperledger.

Hyperledger Fabric

Hyperledger Fabric, or **Fabric** for short, is the contribution made initially by IBM and digital assets to the Hyperledger project. This contribution aims to enable a modular, open, and flexible approach toward building blockchain networks.

Key concepts

Various functions in the Fabric are pluggable, and it also allows the use of any language to develop smart contracts. This functionality is possible because it is based on container technology (Docker), which can host any language.

Chaincode is sandboxed in a secure container, which includes a secure operating system, the chaincode language, runtime environment, and SDKs for Go, Java, and Node.js. Other languages could be supported too in the future, if required, but this needs some development work.

This ability is a compelling feature compared to domain-specific languages in Ethereum, or the limited scripted language in Bitcoin. It is a permissioned network that aims to address issues such as scalability, privacy, and confidentiality. The fundamental idea behind this is modularization, which would allow flexibility in the design and implementation of the business blockchain. This can then result in achieving scalability, privacy, and other desired attributes and fine-tuning them according to requirements.

Transactions in Fabric are private, confidential, and anonymous for general users, but they can still be traced back and linked to the users by authorized auditors. As a permissioned network, all participants are required to be registered with the membership services to access the blockchain network. This ledger also provides an auditability functionality to meet the regulatory and compliance needs required by the user. There are six core capabilities of the Hyperledger Fabric blockchain:

- Modular architecture that makes Hyperledger Fabric resilient to changes and allows any industry to adopt as required.
- Privacy and confidentiality that allows sharing of data only between specific subsets of network members by using private channels.
- Efficiency and scalability in the Hyperledger Fabric network is provided due to Fabric's ability to assign roles to nodes, which results in concurrency and parallelism. Transactions can be threaded for faster processing in Hyperledger Fabric. Also, as execution can only be required at a specific subset of nodes, some network resources can remain free for performing other tasks, thus resulting in a more efficient network.
- Business logic in Fabric is provided by smart contracts called chaincode.
- Identity management is provided using a **membership service provider** (**MSP**), which manages user IDs and provides authentication services for users on the network.
- Governance is an important capability that enables policy building and enforcement to ensure access control security of a system. For example, a policy is required to identify the participants that are allowed to deploy a chaincode or make configuration changes.



Note that the Hyperledger reference architecture described earlier is not necessarily strictly followed in every Hyperledger project. It is there as a reference and guideline to follow; however, not all elements of the reference architecture are implemented in a Hyperledger project.

Hyperledger Fabric is composed of several modules that provide various services. Now we discuss these services in detail.

Membership service

This service is used to provide access control capability for the users of the Fabric network. It provides identity management capabilities and allows for custom and complex access control policies. Membership services perform the following functions:

- User identity verification
- User registration

- Assign appropriate permissions to the users depending on their roles

Membership services make use of a **certificate authority** (**CA**) to support identity management and authorization operations. This CA can be internal (such as Fabric CA, which is a default interface in Hyperledger Fabric), or an organization can opt to use an external certificate authority. Fabric CA issues **enrolment certificates** (**E-Certs**), which are produced by an **enrolment certificate authority** (**E-CA**). Once peers are issued with an identity, they are allowed to join the blockchain network. There are also temporary certificates issued called **T-Certs**, which are used for one-time transactions.

All peers and applications are identified using a certificate authority. An authentication service is provided by the certificate authority. MSPs can also interface with existing identity services like the **Lightweight Directory Access Protocol** (**LDAP**). The default implementation of MSP in Hyperledger Fabric is based on a widely used internet standard specifying the use and structure of public key certificates, the X.509 standard. This means that Hyperledger Fabric can support any PKI where certificates are issued by standard certificate authorities on the internet.

Hyperledger Fabric is also fortified with a cryptographic protocol called **identity mixer**, which provides strong authentication while providing identity privacy.

A closely associated concept is **policies**. Policies control who is allowed to do what and on which component of the Hyperledger network. While MSPs recognize valid identities, policies implement permissions for these identities.

This section covered the membership services implemented in Hyperledger Fabric. Next, we'll introduce some of Fabric's blockchain services.

Blockchain services

Blockchain services are at the core of Hyperledger Fabric. Components within this category are as follows.

Consensus services

A consensus service is responsible for providing the interface to the consensus mechanism. This serves as a module that is pluggable and receives the transaction from other Hyperledger entities and executes them under criteria, according to the type of mechanism chosen.

Consensus in Hyperledger Fabric is implemented as a peer called **orderer**, which is responsible for ordering the transactions in sequence into a block. An orderer does not hold smart contracts or ledgers. Consensus is pluggable and currently ordering services in Hyperledger Fabric are based on a consensus mechanism called RAFT, which is a crash fault-tolerant and leader-follower-based protocol for achieving distributed consensus.



We discussed consensus mechanisms in appropriate detail in *Chapter 5*, *Consensus Algorithms*. Readers can refer to this chapter to review the RAFT and PBFT mechanisms.

In addition to these mechanisms, other mechanisms may become available in the future that can be plugged into Hyperledger Fabric.

Distributed ledger

Blockchain and world state are two main elements of the distributed ledger. Blockchain is simply a cryptographically linked list of blocks (as introduced in *Chapter 1*, *Blockchain 101*) and world state is a key-value database that stores the current values of a set of ledger states. World state makes it easier to get this latest state instead of programs being required to traverse the entire blockchain database. This database is used by smart contracts to store relevant states during execution of the transactions. A blockchain consists of blocks that contain transactions. These transactions contain chaincode, which runs transactions that can result in updating the world state. Each node saves the world state on disk in LevelDB or CouchDB, depending on the implementation. As Fabric allows pluggable data stores, you can choose any data store for storage.

A block consists of three main components called **block header**, **block data (transactions)**, and **block metadata**. The following diagram shows a blockchain depiction with the block and transaction structure in Hyperledger Fabric, with the relevant fields:

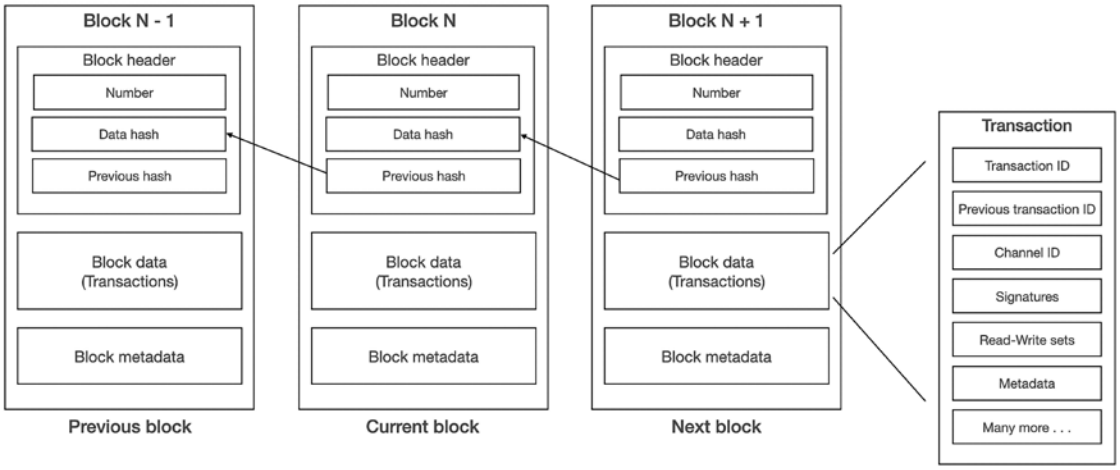


Figure 14.2: Blockchain and transaction structure

Block header consists of three fields, namely **Number**, **Data hash**, and **Previous hash**.

Block data contains an ordered list of transactions. A **Transaction** is made up of multiple fields such as **Transaction ID**, **Previous transaction ID**, **Channel ID**, **Signatures**, **Read-Write sets**, **Metadata**, and many more.

Block metadata mainly consists of the creator identity, the time of creation, and relevant signatures.

The peer-to-peer protocol

The P2P protocol in Hyperledger Fabric is built using **Google RPC** (**gRPC**). It uses protocol buffers to define the structure of the messages.

Messages are passed between nodes to perform various functions. There are four main types of messages in Hyperledger Fabric: **discovery**, **transaction**, **synchronization**, and **consensus**. Discovery messages are exchanged between nodes when starting up to discover other peers on the network. Transaction messages are used to deploy, invoke, and query transactions, and consensus messages are exchanged during consensus. Synchronization messages are passed between nodes to synchronize and keep the blockchain updated on all nodes.

Ledger storage

To save the state of the ledger, by default, LevelDB is used, which is available at each peer. An alternative is to use CouchDB, which provides the ability to run rich queries.

Now that we've established some of the core blockchain services used in Hyperledger Fabric, let's look at some of the chaincode, or smart contract, services available, along with an overview of the APIs and CLIs available in Fabric.

Smart contract services

These services allow the creation of secure containers that are used to execute the chaincode. Components in this category are as follows:

- **Secure container**: Chaincode is deployed in Docker containers that provide a locked-down sandboxed environment for smart contract execution. Currently, Golang is supported as the main smart contract language, but any other mainstream languages can be added and enabled if required.
- **Secure registry**: This provides a record of all images containing smart contracts.
- **Events**: Events on the blockchain can be triggered by endorsers and smart contracts. External applications can listen to these events and react to them if required via event adapters. They are similar to the concept of events introduced in Solidity in *Chapter 11*, *Tools, Languages, and Frameworks for Ethereum Developers*.

APIs and CLIs

An application programming interface provides an interface into Fabric by exposing various REST APIs. Additionally, command-line interfaces that provide a subset of REST APIs and allow quick testing and limited interaction with the blockchain are also available.

In the preceding sections, we have covered the main services of Hyperledger Fabric. Next, let's consider Hyperledger Fabric's components from a network perspective.

Components

There are various components that can be part of the Hyperledger Fabric blockchain network. These components include, but are not limited to, the peers, clients, channels, world state database, transactions, membership service providers, smart contracts, and crypto-service provider components.

Peers/nodes

Peers participate in maintaining the state of the distributed ledger. They also hold a local copy of the distributed ledger. Peers communicate via gossip protocol. There are three types of peers in the Hyperledger Fabric network:

- **Endorsing peers** or **endorsers**, which simulate the transaction execution and generate a read-write set. **Read** is a simulation of a transaction's reading of data from the ledger and **write** is the set of updates that would be made to the ledger if and when the transaction is executed and committed to the ledger. Endorsers execute and endorse transactions. It should be noted that an endorser is also a committer too. Endorsement policies are implemented with chaincode and specify the rules for transaction endorsement.
- **Committing peers** or **committers**, which receive transactions endorsed by endorsers, verify them, and then update the ledger with the read-write set. A committer verifies the read-write set generated by the endorsers along with transaction validation.
- **Orderer nodes** receive transactions from endorsers along with read-write sets, arrange them in a sequence, and send them to committing peers. Committing peers then perform validation and committing to the ledger.

All peers make use of certificates issued by membership services. A peer can assume two roles. A peer can be a leader peer or an anchor peer. In a channel when an organization has multiple peers, a leader peer is responsible for transaction distribution from the orderer node to the other committer nodes in the organization. An anchor peer is used when a peer needs to communicate with a peer in another organization. A peer can assume committer, endorser, leader, and anchor roles at the same time.

Clients

Clients are software that makes use of APIs to interact with Hyperledger Fabric and propose transactions.

Channels

Channels allow the flow of confidential transactions between different parties on the network. They allow the use of the same blockchain network but with separate overlay blockchains. Channels allow only members of the channel to view the transactions related to them; all other members of the network will not be able to view the transactions. We can think of a channel as a private subnet for communication between two or more network members.

World state database

World state reflects all the committed transactions on the blockchain. This is essentially a key-value store that is updated because of transactions and chaincode execution. For this purpose, either LevelDB or CouchDB is used. LevelDB is a key-value store whereas CouchDB stores data as JSON objects, which allows rich queries to run against the database. A related concept is private data, which enables privacy in the Hyperledger blockchain.

We can see this concept visually in the diagram below:

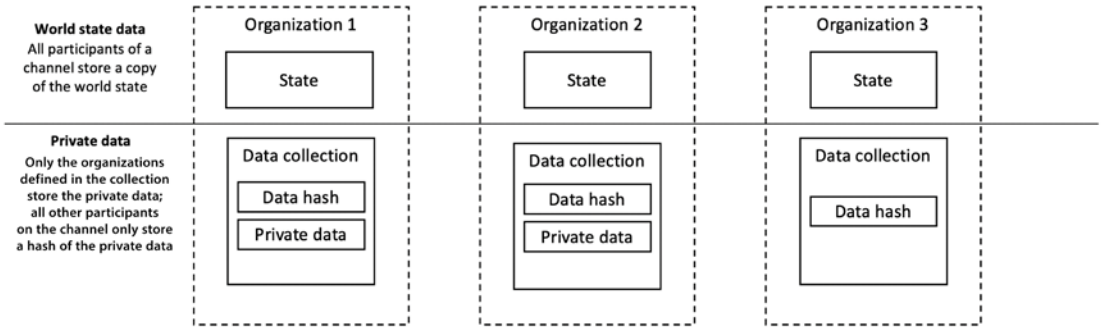


Figure 14.3: World state and private data

The preceding diagram shows a channel shared between multiple organizations. Note that while the "state" is shared between all organizations, the private data is only available to those peers and organizations that are party to the transaction. Peers that do not need to know the transaction only have a hash of the private data. This concept leads us to **private data collections** (**PDCs**).

Private data collections

PDCs are associated with organizations. They define how private data should be disseminated and endorsed, and are part of the chaincode definition. They are composed of a private section and a public section. The private section consists of data that is disseminated through a P2P protocol between the organizations' peers that belong to the PDC. The public section contains hashes of the private data, which are endorsed and committed to the ledger. PDCs are useful in scenarios where an ordering service cannot be trusted for confidentiality and when data in a single ledger or a Fabric channel is made visible to only specific parties, i.e., restricted private transactions.

Transactions

Transaction messages can be divided into two types: **deployment transactions** and **invocation transactions**. The former is used to deploy a new chaincode to the ledger, and the latter is used to call functions from the smart contract. Transactions can be either public or confidential. Public transactions are open and available to all participants, while confidential transactions are visible only in a channel open to its participants.

Membership Service Provider

The **MSP** is a modular component that is used to manage identities on the blockchain network. This provider is used to authenticate clients who want to join the blockchain network. A **certificate authority** (**CA**) is used in the MSP to provide identity verification and binding services.

Smart contracts

We discussed smart contracts in detail in *Chapter 8*, *Smart Contracts*. In Hyperledger Fabric, the same concept of smart contracts is implemented, but they are called chaincode instead of smart contracts. They contain conditions and parameters to execute transactions and update the ledger. Chaincode is usually written in Golang or Java.

Crypto service provider

As the name suggests, this is a service that provides cryptographic algorithms and standards for use in the blockchain network. This service provides key management, signature and verification operations, and encryption-decryption mechanisms. This service is used with the membership service to provide support for cryptographic operations for elements of blockchain, such as endorsers, clients, and other nodes and peers.

After this introduction to some of the components of Hyperledger Fabric, we will next see what an application looks like when on a Hyperledger network.

Applications

A typical application on Fabric is simply composed of a user interface, usually written in JavaScript/HTML, that interacts with the backend chaincode (smart contract) stored on the ledger via an API layer:

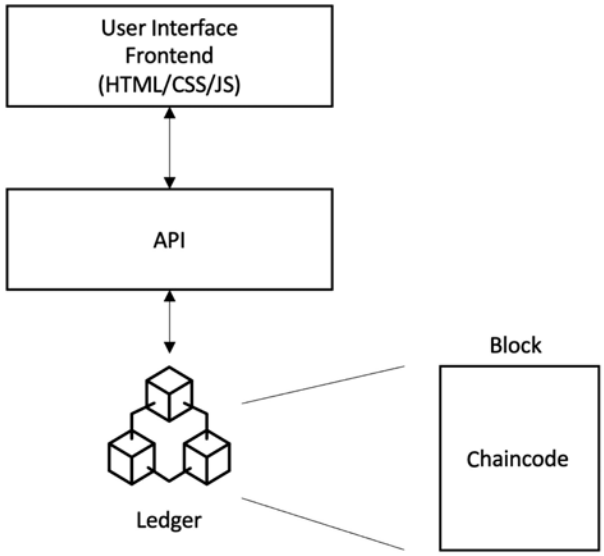


Figure 14.4: A typical Fabric application

Hyperledger provides various APIs and command-line interfaces to enable interaction with the ledger. These APIs include interfaces for identity, transactions, chaincode, ledger, network, storage, and events.

In the next section, we'll see how chaincode is implemented.

Chaincode implementation

Chaincode is usually written in Golang or Java. Chaincode can be public (visible to all on the network), confidential, or access controlled. These code files serve as a smart contract that users can interact with via APIs. Users can call functions in the chaincode that result in a state change, and consequently update the ledger.

There are also functions that are only used to query the ledger and do not result in any state change. Chaincode implementation is performed by first creating the chaincode shim interface in the code. Shim packages provide APIs for accessing state variables, the transaction context of chaincode, and call other chaincodes. They can either be in Java or Golang code.

The following four functions are required in order to implement the chaincode:

- `Init()`: This function is invoked when the chaincode is deployed onto the ledger. This initializes the chaincode and results in making a state change, which updates the ledger accordingly.
- `Invoke()`: This function is used when contracts are executed. It takes a function name as parameters along with an array of arguments. This function results in a state change and writes to the ledger.
- `Query()`: This function is used to query the current state of a deployed chaincode. This function does not make any changes to the ledger.
- `Main()`: This function is executed when a peer deploys its own copy of the chaincode. The chaincode is registered with the peer using this function.

The following diagram illustrates the general overview of Hyperledger Fabric. Note that peer clusters at the top include all types of nodes: **Endorsers**, **Committers**, and **Orderers**:

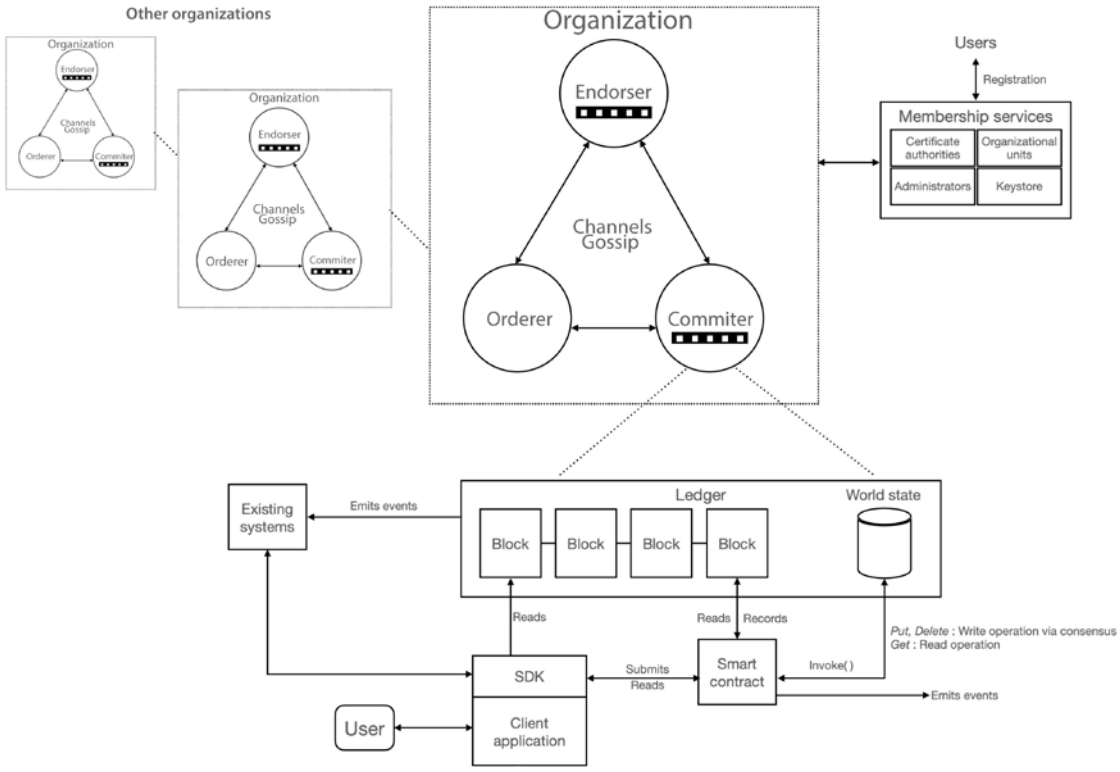


Figure 14.5: A high-level overview of a Hyperledger Fabric network

The preceding diagram shows that peers (shown center-top) communicate with each other, and each node has a copy of the blockchain. In the top-right corner, the membership services are shown, which validate and authenticate peers on the network by using a CA. At the bottom of the diagram, a magnified view of the blockchain is shown whereby existing systems can produce events for the blockchain and can also listen for the blockchain events, which can then optionally trigger an action. At the bottom right-hand side, a user's interaction with the application is shown. The application in turn interacts with the smart contract via the `Invoke()` method, and smart contracts can query or update the state of the blockchain.

Chaincode in Hyperledger Fabric doesn't need to be deterministic and can be developed in any mainstream language, such as JavaScript, Java, or Golang.

In this section, we saw how chaincode is implemented and examined the high-level architecture of Hyperledger Fabric. In the next section, we will discuss the application model.

The application model

Any blockchain application for Hyperledger Fabric follows the MVC-B architecture. This is based on the popular MVC design pattern. Components in this model are **View**, **Control**, **Model** (the data model), and **Blockchain**:

- **View logic**: This is concerned with the user interface. It can be a desktop, web application, or a mobile frontend.
- **Control logic**: This is the orchestrator between the user interface, data model, and APIs.
- **Data model**: This model is used to manage the off-chain data.
- **Blockchain logic**: This is used to manage the blockchain via the controller and the data model via transactions.



The IBM Cloud service offers sample applications for blockchain under its blockchain as a service offering. It is available at `https://www.ibm.com/blockchain/platform/`. This service allows users to create their own blockchain networks in an easy-to-use environment.

After this brief introduction to the application model, let's move on to another important topic, consensus, which is important not only in Hyperledger Fabric but is also central to blockchain design and architecture.

Consensus mechanism

The consensus mechanism in Hyperledger Fabric consists of three steps:

- **Transaction endorsement**: This process endorses the transactions by simulating the transaction execution process.
- **Ordering**: This is a service provided by the cluster of **orderers that run the consensus algorithm**, which takes endorsed transactions and decides on a sequence in which the transactions will be written to the ledger.
- **Validation and commitment**: This process is executed by committing peers, which first validate the transactions received from the orderers and then commit that transaction to the ledger.

These steps are shown in the following diagram:

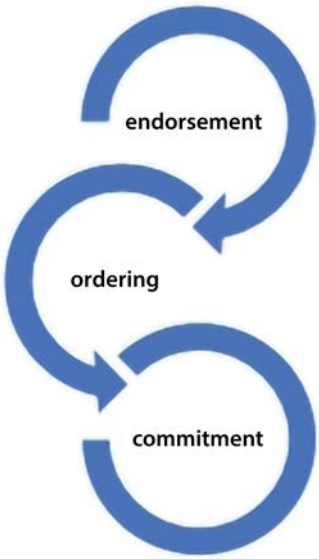


Figure 14.6: The consensus flow

The ordering step is also pluggable. There are several ordering services available for the Fabric consensus mechanism. Initially, in some versions of Fabric 1.0, only simple ordering services such as Apache Kafka and SOLO were available. However, now, other advanced consensus algorithms such as RAFT are also available.

In the next section, we'll describe how a transaction flows through various stages in Hyperledger Fabric before finally updating the ledger.

Transaction lifecycle

There are several steps that are involved in a transaction flow in Hyperledger Fabric. The steps are described in detail as follows:

1. Transaction proposal by clients. This is the first step where a transaction is proposed by the clients and sent to endorsing peers on the distributed ledger network. All clients need to be enrolled via membership services before they can propose transactions.
2. The transaction is simulated by endorsers and generates a **read-write** (**RW**) set. This is achieved by executing the chaincode, but instead of updating the ledger, only an RW set depicting any reads or updates to the ledger is created.

3. The endorsed transaction is sent back to the application.
4. The endorsed transactions and RW sets are submitted to the ordering service by the application.
5. The ordering service assembles all endorsed transactions and RW sets in order in a block and sorts them by channel ID.
6. The ordering service broadcasts the assembled block to all committing peers. Committing peers take the following actions:
    1. Validate the endorsement policy.
    2. Validate read set versions in the state DB.
    3. Commit the block to the blockchain.
    4. Commit the valid transaction to the state DB.
7. The validation ensures two conditions are met: first, transactions are executed as per the given transaction logic and secondly, there are no state conflicts in the submitted transactions, e.g., a scenario where two transactions are updating the same state.
8. Finally, notification of success or failure of the transaction by committing peers is sent back to the clients/applications.

The following diagram represents the steps and the Fabric architecture from a transaction flow perspective:

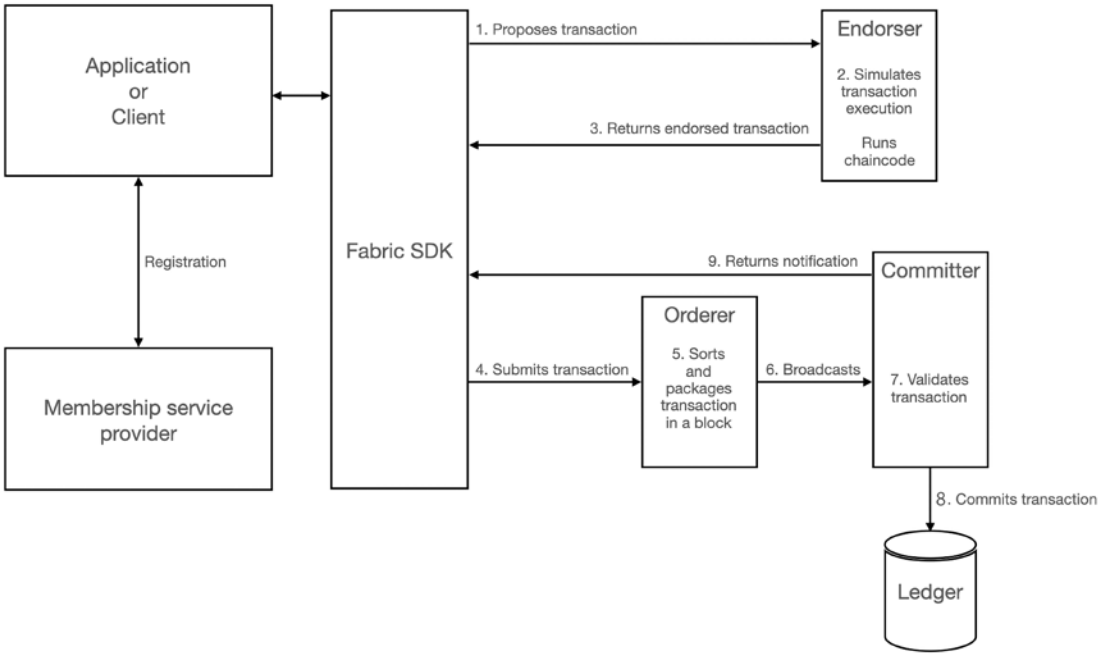


Figure 14.7: The transaction flow

As you can see in the preceding diagram, the first step is to propose transactions, which a client does via an SDK. Before this, it is assumed that all clients and peers are registered with the membership service provider.

Hyperledger Fabric uses an execute-order-validate pattern that allows transactions to execute before the blockchain achieves consensus, which enables better efficiency. In Fabric every transaction is executed and endorsed only by a subset of peers, which allows parallel execution. Also, the final state of a transaction is only written to the blockchain.

So far, we have covered the core architecture of Hyperledger Fabric, which is the most popular Hyperledger project.

In the next section, we will introduce the new features introduced with Fabric 2.0.

Fabric 2.0

This is a major upgrade to the protocol. The fundamentals remain the same, but some very interesting new features and improvements have been made, which are introduced in the following sections.



Hyperledger Fabric 2.0 was released on January 30, 2020. The latest release is available at `https://github.com/hyperledger/fabric/releases/latest`

New chaincode lifecycle management

In Hyperledger Fabric 2.0, new decentralized chaincode lifecycle management is implemented, which is the main notable difference between `Fabric 1.x` and `Fabric 2.x`. In the former, an organization administrator can install the chaincode in its own organization, and other organizations' administrators have to unconditionally agree to instantiate (deploy or upgrade) the chaincode. In other words, there is no agreement process between organizations to agree on the installation of a chaincode. This problem is resolved by introducing checks, which enable peers to only participate in a chaincode if their organizational administrators have agreed to it.

First, let's look at the chaincode lifecycle in `Fabric 1.0`:

1. Each organization administrator installs chaincode on their peers.
2. An administrator instantiates the chaincode by deploying or upgrading.
3. After results are received, the transaction is submitted to the orderers.
4. After ordering, it is submitted to all peers and committed.

Now, we'll look at Fabric 2.0's lifecycle, which highlights the differences between it and Fabric 1.0:

1. Install chaincode by putting the package on the filesystem. Each organization administrator installs chaincode on their peers.
2. Provide approval by executing the `approveformyorg` function. This verifies that the organization agrees with the chaincode definition.

3. Send the ordering service to the orderer.
4. Commit to the organization collection. The collection is a private data collection that provides a mechanism to allow private data sharing only between those organizations that are party to the transaction, even if they are within the same channel. In other words, collections allow private data sharing between a subset of organizations on a channel. Even though other organizations are members of the same channel, they cannot see the data shared between a subset of organizations that are sharing data using the collection, also called a private data collection.
5. Other organizations' administrators perform the same *steps 1* to *4*.
6. After these steps are executed, there is a record in each organization indicating the agreement on the chaincode definition.
7. Define the chaincode as per the agreement in *step 6*.
8. Submit to ordering.
9. Commit to the definition of all peers.
10. The organization administrator who committed the chaincode definition now invokes the `init` function on its own peers and other organizations' peers.
11. Submit to ordering.
12. Commit all the peers.



Note that Fabric 2.0 supports both legacy and new lifecycle patterns, but eventually the legacy lifecycle will become deprecated.

In addition to the chaincode lifecycle management changes in `Fabric 2.0`, there are also new chaincode application patterns. We will introduce these next.

New chaincode application patterns

There are also new chaincode application patterns introduced in Fabric 2.0 for collaboration and consensus. These are automated checks that can be added by an organization to enable additional validation before a transaction is endorsed. In addition, a decentralized agreement is also supported where multiple organizations can propose conditions for an agreement, and when those conditions are met, a collective agreement can be made on a transaction.

- **Enhanced data privacy**: `Fabric 2.0` enhances data privacy by introducing collection-level endorsement policies, per-organization collections, and the flexible sharing and verification of private data.
- **External chaincode launcher**: In Hyperledger `Fabric 2.0`, by default, Docker APIs are used to build and run chaincode. In addition, external builders and launchers can be used if required. In previous versions, there was a requirement to have access to the Docker daemon to build and run chaincode, but in `Fabric 2.0` there is no such dependency. There is also no requirement to run chaincode in Docker containers anymore; chaincode can be run in any environment deemed appropriate by the network operator.

- **RAFT consensus:** RAFT is a **crash fault-tolerant** (**CFT**) ordering service for achieving consensus. It is based on the `etcd` library and a leader-follower model. In this model, a leader is elected whose decision is then replicated by its followers.
- **Better performance:** In `Fabric 2.0`, caching has been introduced at the CouchDB state database level to improve performance by reducing the read latency, introduced due to expensive lookup operations in previous versions.

With all that we have learned so far, the Hyperledger application architecture can be depicted as follows:

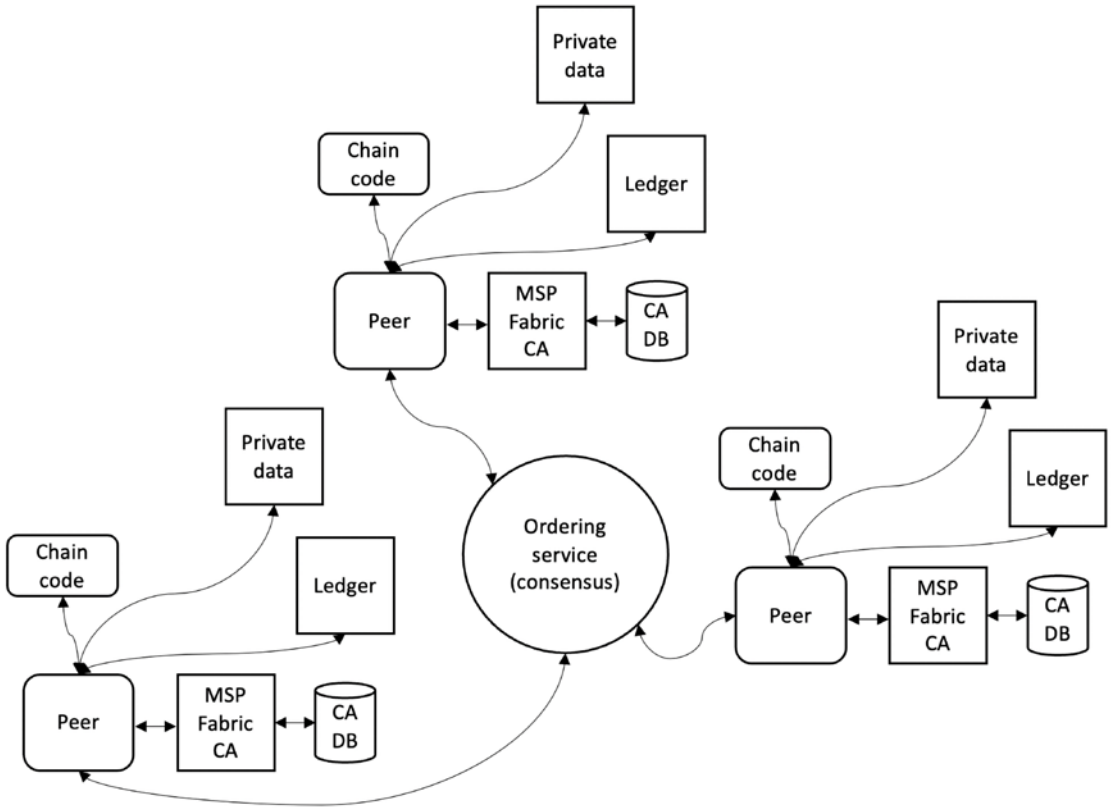


Figure 14.8: Hyperledger Fabric application architecture



A list of cross-industry projects built with Hyperledger Fabric and other Hyperledger projects is being maintained here: `https://www.hyperledger.org/learn/blockchain-showcase`. Readers can explore this further to understand how Hyperledger Fabric is used in different use cases.

With this section, our introduction to Hyperledger Fabric is complete. Hyperledger Fabric is the flagship project of Hyperledger and is used in a wide variety of applications. Hyperledger Fabric is continually evolving, and more features and changes are expected in future releases. However, no major design changes are expected since the introduction of version 2.0.

Summary

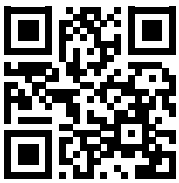
In this chapter, we have gone through an introduction to the Hyperledger project. Firstly, the core ideas behind the Hyperledger project were discussed, and a brief introduction to all projects under Hyperledger was provided. A key Hyperledger project was discussed in detail, namely, Hyperledger Fabric. All these projects are continuously improving, and changes are expected in the next releases. However, the core concepts of all the projects mentioned previously are expected to remain unchanged or change only slightly. Readers are encouraged to visit the relevant links provided in the chapter to see the latest updates.

It is evident that a lot is going on in this space, and projects like Hyperledger from the Linux Foundation are playing a pivotal role in the advancement of blockchain technology. Each of the projects introduced in this chapter has novel approaches toward solving the issues faced in various industries, and any current limitations within blockchain technology are also being addressed, such as scalability and privacy. It is expected that more projects will soon be proposed for the Hyperledger project, and it is envisaged that with this collaborative and open effort, blockchain technology will advance tremendously and collectively benefit the community.

In the next chapter, we will discuss tokenization, which is one of the most prominent applications of blockchain technology and is disrupting many industries, especially the financial industry.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

15

Tokenization

A token is a representation of an object. We use tokens in many different fields, including economics and computing. In daily life, tokens have been used to represent something of value, such as a gift voucher that is redeemable in exchange for items. In computing, different types of tokens are used, which can be defined as objects that represent eligibility to perform some operation. For example, an access token is used to identify a user and its privileges on a computer system. A hardware security token is used in computer systems to provide a means to authenticate a user (verify their identity) to a computer system. Tokens are also used in computer security mechanisms to substitute sensitive data with non-sensitive equivalents to prevent direct access to sensitive information. For example, in mobile payment systems, tokens are used to safeguard credit card information, which is represented by a token on the mobile device instead of the actual credit card data.

Just like the general use of tokens in day-to-day life, in the blockchain world, tokens also represent something of value. However, the critical difference is that the token exists digitally on a blockchain and is operated cryptographically, which means that they are generated, protected, and transferred using cryptographic protocols. Now that we understand what a token is, we can define tokenization as a process that converts an asset to a digital token on a blockchain. Specifically, it is a process of converting the ownership rights of a real-world asset into a cryptographic/digital token on a blockchain.

In this chapter, we will cover blockchain tokens (or crypto-tokens) and tokenization. We will also cover the following topics:

- Tokenization on a blockchain
- Types of tokens
- Process of tokenization
- Token offerings
- Token standards
- Building an ERC-20 token
- Emerging concepts

Now, we'll begin our discussion on tokenization in the context of blockchain, which is the main aim of this chapter.

Tokenization on a blockchain

Tokenization, in the context of blockchain, is the process of representing an asset digitally on a blockchain. It can be used to represent commodities, real estate, ownership of art, currency, or anything else of value.



Remember, for the rest of the chapter when we refer to tokenization, it is in the context of blockchain.

After this brief definition, let's explore how tokenization can be beneficial.

Advantages of tokenization

Tokenization provides several benefits. The following are some of the most important of these benefits:

- **Faster transaction processing**: As transactions and all relevant parties are present on the blockchain and readily available, there is no need to wait for a response from a counterparty or to wait for clearing and settlement operations. All these operations can be performed efficiently and quickly on a blockchain.
- **Flexibility**: Due to the worldwide adoption of systems that use tokens, such as payment systems, tokenization becomes simple due to easier cross-border use.
- **Low cost**: In comparison with traditional financial products, tokenization requires a lower cost to implement and incurs a lower cost to the end user due to digitization. More recently, the introduction of digital payments has revolutionized the financial industry. In the same spirit, tokenization using blockchain can be considered another step toward achieving further efficiency and cost reduction. In fact, tokenization gives rise to a better, more efficient, and more democratic financial system. For example, just being able to share customer data recorded on one blockchain across all financial institutions reduces costs. Similarly, the possibility of converting illiquid assets into liquid assets is another way of increasing efficiency and profits.
- **Decentralization**: Tokens are presented on a public blockchain, leveraging the decentralization offered by blockchain technology. However, in some cases, a level of somewhat acceptable centralization is introduced due to the control and scrutiny required by investors and exchanges and other involved and interested parties.



We discussed the benefits of decentralization in detail in *Chapter 2, Decentralization*. Refer to that chapter to review decentralization.

- **Security**: As tokens are cryptographically protected and produced using a blockchain, they are secure. However, note that proper implementation must use good practice and meet established security industry standards, otherwise security flaws may result in exploitation by hackers, leading to financial loss.
- **Transparency**: Because they are on a blockchain, tokens are more transparent than traditional financial systems, meaning that any activity can be readily audited on a blockchain and is visible to everyone.
- **Trust**: As a result of the security and transparency guarantees, more trust is naturally placed in tokenization by investors.
- **Fractional ownership**: Imagine a scenario in which you own a painting by Vincent van Gogh. It is almost impossible in traditional scenarios to have multiple owners of the painting without immense legal, financial, and operational challenges. However, on a blockchain using tokens, any asset (such as our van Gogh painting) can be fractionalized in such a way that it can be owned in part by many investors. The same situation is true for a property: if I wanted to have shared ownership with someone, it requires complicated legal procedures. However, with tokenization, fractional ownership of any asset, be it art, real estate, or any other off-chain real-world asset, is quick, easy, efficient, secure, and a lot less complicated than traditional methods.
- **Low entry barrier**: Traditional financial systems require traditional verification mechanisms, which can take a long time for a customer. While they are necessary and serve the purpose in traditional financial systems, these processes take a long time due to the necessary verification processes and the involvement of different entities. However, on a blockchain, this entry barrier is lowered because there is no need to go through the long verification checks. This is because not only is tokenization based on cryptographic guarantees provided by the blockchain, but for many **decentralized applications** (**DApps**) in this ecosystem, it is basically just a matter of downloading a DApp on your mobile device, depositing some funds if required, and becoming part of the ecosystem.
- **Innovative applications**: Tokenization has resulted in many innovative applications, including novel lending systems on blockchain, insurance, and other numerous financial applications, including decentralized exchanges. Securities can now be tokenized and presented on blockchain, which results in client trust and satisfaction because of the better security and faster processing times.
- **More liquidity**: This is due to the easy availability and accessibility of the tokens for the general public. By using tokens, even illiquid assets such as paintings can be tokenized and traded on an exchange with fractional ownership.

With all these advantages, there are, however, some issues that must be addressed in the tokenization ecosystem. We'll discuss these next.

Disadvantages of tokenization

In this section, we present some of the disadvantages of tokenization. At the top of the list we have regulatory requirements:

- **Regulatory issues**: Regulation is a crucial subject of much debate. It is imperative that the tokens are regulated so that investors can have the same level of confidence that they have when they invest using traditional financial institutions. The big issue with tokenization and generally any public blockchain technology is that they are mostly decentralized and in cases where no single organization is in control, it becomes quite difficult to hold someone responsible if something goes wrong. In a traditional system, we can go to the regulatory authorities or the relevant ombudsman services, but who is held responsible on a blockchain?

  Some of this situation has changed with recent security tokenization standards and legislation, which consider tokens as securities. This means that security tokens will then be treated as securities, and the same legal, financial, and regulatory implications will be placed on these tokens that are applicable in the currently established financial industry. Refer to `https://www.sec.gov/answers/about-lawsshtml.html`, where different laws that govern the securities industry are presented. This helps to increase customer confidence levels and trust in the system; however, there are still many challenges that need to be addressed.

  A new type of financial crime might be evolving with this tokenization ecosystem where, instead of using well-known and researched traditional financial fraud methods, criminals may choose to launch a technically sophisticated attack. For an average user, this type of attack is difficult to spot and understand as they are entirely on a blockchain and digitized. New forms of front running and market skewing on decentralized finance platforms is increasingly becoming a concern.

- **Legality of tokens**: This is a concern in some jurisdictions where tokens and cryptocurrency are illegal to own and trade.
- **Technology barrier**: Traditional financial systems have been the norm with brick-and-mortar banks. We are used to that system, but things have changed and are expected to change rapidly with tokenization. Tokenization-based financial ecosystems are easier to use for a lot of people, but for some, technological illiteracy can become an issue and could become a barrier. Sometimes the interfaces and software applications required to use tokenization platforms such as trading platforms are difficult to use for the average user.
- **Security issues**: The underlying blockchain technology is considered solid from a security point of view, and it is boasted sometimes that due to the use of cryptography, it is impossible to launch attacks and commit fraud on a blockchain. However, this is not entirely true, even with the firm security foundation that blockchains provide. The way tokenization platforms and DApps are implemented on the blockchain result in security issues that can be exploited by hackers, potentially leading to financial loss. In other words, even if the underlying blockchain is relatively secure, the tokenization DApp implemented on top may have vulnerabilities that could be exploited. These weaknesses could have been introduced by poor development practices or inherent limitations in the still-evolving smart contract languages.

In this section, we have looked at some of the pros and cons of tokenization. Next, let's look at some of the many types of tokens.

Types of tokens

With the rapid development of blockchain technology and the related applications, there has been a tremendous increase in the development of various types of token and relevant ecosystems. First, let's clarify the difference between a coin and a token. Is a Bitcoin a token? Or are tokens and coins the same thing?

A coin is a native token of a blockchain. It is the default cryptocurrency of the blockchain on which it runs. Common examples of such a token are Bitcoin and ether. Both tokens or coins have their own native blockchain on which they run: the Bitcoin blockchain and the Ethereum blockchain.

On the other hand, a token is a representation of an asset that runs on top of a blockchain. For example, Ethereum not only has its own ether cryptocurrency as a native token (or coin) but also has thousands of other tokens that are built on top of Ethereum for different applications. Thanks to its support of smart contracts, Ethereum has become a platform for all sorts of different types of tokens ranging from simple utility tokens, which are usually non-mineable and allow users to perform some specific services on products on the blockchain network/ecosystem, to game tokens (used for gaming) and high-value, application-specific tokens.

Now that we understand the difference between a coin and a token, let's have a look at the usage and significance of different types of tokens. Tokens can be divided broadly into two categories based on their usage: **fungible** tokens and **non-fungible** tokens.

Fungible tokens

From an economics perspective, fungibility is the interchangeability of an asset with other assets of the same type. For example, a ten-pound note is interchangeable with a different ten-pound note or two five-pound notes. It does not matter which specific denominations they are, as long as they are of the same **type** (pound) and have the same **value** (the sum of two five-pound notes), the notes are acceptable.

Fungible tokens work on the same principle. They are:

- **Indistinguishable**: Tokens of the same type are indistinguishable from each other. In other words, they are identical.
- **Interchangeable**: A token is fully interchangeable with another token of the same value.
- **Divisible**: Tokens are divisible into smaller fractions.

Non-fungible tokens

Let's consider **non-fungible tokens** (**NFTs**), also called *nifty*, with the help of an example. We saw that fungibility allows the same types of token to be interchangeable, but NFTs are not interchangeable. For example, a collectible painting is a non-fungible asset, as it cannot be replaced with another painting of the same type and value. Each painting is unique and distinguishable from others.

NFTs are:

- **Unique**: NFTs are unique and different from other tokens of the same type or in the same category.
- **Non-interchangeable**: Since they are unique and represent specific attributes, these tokens are not interchangeable with tokens of the same type. For example, a rare painting is unique due to its attributes and is not interchangeable with another, even an exact-looking replica. We can also think about the certificate of authenticity that comes with a rare painting: that is also non-interchangeable with another due to its unique attributes representing the rare art.
- **Indivisible**: These tokens are available only as a complete unit. For example, a college diploma is a unique asset that's distinguishable from other diplomas of the same type. It is associated with a unique individual and is thus not interchangeable and it is not rational for it to be divided into fractions, making it indivisible.



One of the prime examples of NFTs is the game CryptoKitties—`https://www.cryptokitties.co`. CryptoKitties played a big role in the popularity of NFTs, in that it was the first game (DApp) that was based on NFTs. From this, the community grew, as did users' interest in the underlying NFT mechanism. A popular term, "crypto collectibles," also emerged with this development. Some other projects that use NFT and offer crypto collectibles include Gods Unchained, Decentraland, Skyweaver, and My Crypto Heroes.

Since the previous edition of this book, NFT popularity has skyrocketed. We will discuss them in more detail in *Chapter 21*, *Decentralized Finance*.

In the next section, we introduce another interesting topic, stable tokens, which are tokens with interesting properties that make them more appealing to some investors.

Stable tokens

Stable tokens or stable coins are a type of token that has its value pegged to another asset's value, such as fiat currencies or precious metals. Stable tokens maintain a stable price against the price of another asset. Common examples of stable tokens are USDT, developed in 2014 by Tether Limited, and USDC, introduced in 2018 by Coinbase and Circle.

Bitcoin and other similar cryptocurrencies are inherently volatile and experience dramatic fluctuations in their price. This volatility makes them unsuitable for usual everyday use. Stable tokens emerged as a solution to this limitation in tokens.

The price stability is maintained by backing the token up with a stable asset. This is known as collateralization. Fiat currencies are stable because they are collateralized by some reserve, such as **foreign exchange** (**forex**) reserves or gold. Moreover, sound monetary policy and regulation by authorities play a vital role in the stability of a currency. Tokens or cryptocurrencies lack this type of support and thus suffer from volatility.

There are four types of stable coin:

- **Fiat collateralized**: These stable tokens are backed by or pegged to a traditional fiat currency such as US dollars. Fiat collateralized coins are the most common type of stable coin. Some of the common stable coins available today are **Gemini Dollar** (**GUSD**) (`https://gemini.com/dollar`), **Paxos** (**PAX**) (`https://www.paxos.com/pax/`), **USDT** (**Tether**) (`https://tether.to/en/`), and **Libra** (`https://libra.org/`).
- **Commodity collateralized:** As the name suggests, these stable coins are backed up by fungible commodities (assets) such as oil or gold. Common examples of such tokens are Digix gold tokens (`https://www.crunchbase.com/organization/digix-global`) and Tether gold (`https://tether.to`).
- **Crypto collateralized:** This type of stable coin is backed up by another cryptocurrency. For example, the Dai stable coin (`https://makerdao.com/en/`) accepts Ethereum-based assets as collateral in addition to soft pegging to the US dollar.
- **Algorithmically stable:** This type of stable token is not collateralized by a reserve but stabilized by algorithms that keep track of market supply and demand. For example, Basecoin (`https://basecoin.cc`) can have its value pegged against the US dollar, a basket (a financial term for group, or collection of similar securities) of assets, or an index, and it also depends on a protocol to control the supply of tokens based on the exchange rate between itself and the peg used, such as the US dollar. This mechanism helps to maintain the stability of the token.

Security tokens

Security tokens derive their value from external tradeable assets. For example, security tokens can represent shares in a company. The difference is that traditional security is kept in a bank register and traded on traditional secondary markets, whereas security tokens are available on a blockchain. Being securities, they are governed by all traditional laws and regulations that apply to securities, but due to their somewhat decentralized nature, in which no middleman is required, security tokens are seen as a more efficient, scalable, and transparent option.

Now that we have covered different types of token, let's see how an asset can be tokenized by exploring the process of tokenization.

Process of tokenization

In this section, we'll present the process of tokenization, discuss what can be tokenized, and provide a basic example of the tokenization of assets.

Almost any asset can be tokenized and presented on a blockchain, such as commodities, bonds, stocks, real estate, precious metals, loans, and even intellectual property. Physical goods that are traditionally illiquid in nature, such as collectibles, intellectual property, and art, can also be tokenized and turned into liquid tradeable assets.

A generic process to tokenize an asset or, in other words, offer a security token, is described here:

1. The first step is to onboard an investor who is interested in tokenizing their asset.

2. The asset that is presented for tokenization is scrutinized and audited, and ownership is confirmed. This audit is usually performed by the organization offering the tokenized security. It could be an exchange or a cryptocurrency start-up.
3. The process of tokenized security starts, which leads to the **security token offering** (**STO**).
4. The physical asset is placed with a custodian (in the real world) for safekeeping.
5. The **derivative token**, representing the asset, is created by the organization offering the token and issued to investors through a blockchain.
6. Traders start to buy and sell these tokens using trading exchanges in a secondary market and these trades are settled (the buyer makes a payment and receives the goods) on a blockchain.

Common platforms for tokenization include Ethereum, Solana, and EOS. Tezos is also emerging as another platform for tokenization due to its support for smart contracts. In fact, any blockchain that supports smart contracts can be used to build tokens; however, the most common are Ethereum and Solana.

At this point, a question arises about how all these different types of tokens reach the public for investment. In the next section, we examine how this is achieved by first explaining what token offerings are and examining the different types.

Token offerings

Token offerings are mechanisms to raise funds and profit. There are a few different types of token offerings. We will introduce each of these separately now. One main common attribute of each of these mechanisms is that they are hosted on a blockchain and make use of tokens to facilitate different financial activities. These financial activities can include crowdfunding and trading securities.

Initial coin offerings

Initial coin offering or **initial currency offering** (**ICO**) is a mechanism for raising funds using cryptocurrencies. ICOs have been a very successful but somewhat controversial mechanism for raising capital for new cryptocurrency or token projects. ICOs are somewhat controversial sometimes due to bad design or poor governance, but at times some of the ICOs have turned out to be outright scams. A list of fraudulent schemes is available at `https://dfpi.ca.gov/crypto-scams/`. These incidents have contributed to the bad reputation and controversy surrounding ICOs in general. While there are some scams, there are also many legitimate and successful ICOs. The **return on investment** (**ROI**) for quite a few of ICOs has been quite big and has contributed toward the unprecedented success of many of these projects. Some of these projects are Ethereum, NXT, NEO, IOTA, and QTUM.

A common question is asked here regarding the difference between traditional **initial public offerings** (**IPOs**) and **ICOs**, as both mechanisms are fundamentally designed to raise capital. The difference is simple: ICOs are blockchain-based token offerings that are usually initiated by start-ups to raise capital by allowing investors to invest in their start-up. Usually in this case, contributions by investors are made using already existing and established cryptocurrencies such as Bitcoin or ether. As a return, when the project comes to fruition, the initial investors get their return on the investment.

On the other hand, IPOs are traditional mechanisms used by companies to distribute shares to the public. This is done using the services of underwriters, which are usually investment banks. IPOs are only usually allowed for those companies that are already well established, but ICOs on the other hand can be offered by start-ups. IPOs also offer dividends as returns, whereas ICOs offer tokens that are expected to rise in value once the project goes live.

Another key comparison is that IPOs are regulated, traditional mechanisms that are centralized in nature, while ICOs are decentralized and unregulated.



There are some examples where traditional means such as signing paper contracts have been used to provide some level of legal protection for the parties involved, but ICOs are largely unregulated. This is especially true from the perspective of large financial regulatory bodies and ombudsman services providing investor protection and handling consumer complaints in the traditional financial services industry.

Because ICOs have been unregulated, which has resulted in a number of scams and people losing their money, another form of fundraising known as security token offerings was introduced. We'll discuss this next.

Security token offerings

Security token offerings (**STOs**) are a type of offering in which tokenized securities are traded at cryptocurrency exchanges. Tokenized securities or security tokens can represent any financial asset, including commodities and securities. The tokens offered under STOs are classified as securities. As such, they are more secure and can be regulated, in contrast with ICOs. If an STO is offered on a traditional stock exchange, then it is known as a tokenized IPO. Tokenized IPO is another name for an STO that is used when an STO is offered on a regulated stock exchange, which helps to differentiate between an STO offered on a traditional regulated exchange and one that is offered on cryptocurrency exchanges. STOs are regulated under the Markets in Financial Instruments Directive—MiFID II—in the European Union.

Initial exchange offerings

Initial exchange offering (**IEO**) is another innovation in the tokenization space. The key difference between an IEO and an ICO is that in an ICO, the tokens are distributed via crowdfunding mechanisms to investors' wallets, but in an IEO, the tokens are made available through an exchange.

IEOs are more transparent and credible than ICOs due to the involvement of an exchange and due diligence performed by the exchange.

Equity token offerings

Equity token offerings (**ETOs**) are another variation of ICOs and STOs. ICOs offer utility tokens, whereas equity tokens are offered in ETOs. When compared with STOs, ETOs offer shares of a company, whereas STOs offer shares in any asset, such as currencies or commodities. From this point of view, ETOs can be regarded as a specific type of STO, where only shares in a company or venture are represented as tokens.

Decentralized autonomous initial coin offering

Decentralized Autonomous Initial Coin Offering (**DAICO**) is a combination of **decentralized autonomous organizations** (**DAOs**) and ICOs that enables investors to have more control over their investment and is seen as a more secure, decentralized, and automated version of ICOs.

Other token offerings

Different variations of ICOs and new concepts are being introduced quite regularly, and innovation is only expected to grow in this area.

A comparison of different types of token offering is presented in the following table:

Name	Concept	First introduced	Scale of decentralization	How to invest	Regulation
ICO	Crowdfunding through a utility token	In July 2013 with Mastercoin	Semi-decentralized	Investors send cryptocurrency to a smart contract released by the token offerer	Low regulation
STO	Tokenized security such as bonds and stocks	2017	Somewhat centralized	Use the exchange-provided platform	Regulated under established laws and directives in many jurisdictions
IEO	Tokens are made available through an exchange	2018	Somewhat centralized	Use the exchange-provided platform	Low regulation
ETO	Offers shares of any asset	December 2018 with the Neufund ETO	Somewhat centralized	Use the exchange-provided platform	Mostly regulated
DAICO	Combination of DAO and ICO	May 2018 with ABYSS DAICO	Mostly decentralized	Investors send cryptocurrency to the DAICO smart contract	Low regulation

With all these different types of tokens and associated processes, a need to standardize naturally arises so that more adoption and interoperability can be achieved. To this end, several development standards have been proposed, which we discuss next.

Token standards

With the advent of smart contract platforms such as Ethereum, it has become quite easy to create a token using a smart contract. Technically, a token or digital currency can be created on Ethereum with a few lines of code, as shown in the following example:

```
pragma solidity ^0.8.0;
contract token {
    mapping (address => uint) public coinBalanceOf;
    event CoinTransfer(address sender, address receiver, uint amount);

  /* Initializes contract with initial supply tokens to the creator of the
contract */
  function  tokenx(uint supply) public {
        supply = 1000;
        coinBalanceOf[msg.sender] = supply;
    }
  /* Very simple trade function */
    function sendCoin(address receiver, uint amount) public returns(bool
sufficient) {
        if (coinBalanceOf[msg.sender] < amount) return false;
        coinBalanceOf[msg.sender] -= amount;
        coinBalanceOf[receiver] += amount;
        emit CoinTransfer(msg.sender, receiver, amount);
        return true;
    }
}
```



This code is based on one of the early codes published by Ethereum as an example on `ethereum.org`.

The preceding code works and can be used to create and issue tokens. However, the issue is that without any standard mechanism, everyone would implement tokenization smart contracts in their own way based on their requirements. This lack of standardization will result in interoperability and usability issues, which obstruct the widespread adoption of tokenization.

To remedy this, the first tokenization standard was proposed on Ethereum, known as ERC-20.

ERC-20

ERC-20 is the most famous token standard on the Ethereum platform. Many token offerings are based on ERC-20 and there are wallets available, such as MetaMask, that support ERC-20 tokens.

ERC-20 was introduced in November 2015 and since then has become a very popular standard for fungible tokens. This standard has been used in many ICOs and has resulted in valuable digital currencies (tokens) over the last few years, including EOS, Golem, and many others. Famous tokens, such as USDT, BNB, USDC, and Matic, are all based on the ERC-20 standard. There are almost 1,000 ERC-20 token projects listed on Etherscan (`https://etherscan.io/tokens`), which is a clear indication of this standard's popularity.

While ERC-20 defined a standard for fungible tokens and was widely adopted, it has some flaws, which result in some security and usability issues. For example, a security issue in ERC-20 results in a loss of funds if the tokens are sent to a smart contract that does not have the functionality to handle tokens. The effectively "burned" tokens result in a loss of funds for the user.

To address these shortcomings, ERC-223 was proposed.

ERC-223

ERC-223 is a fungible token standard. One major advantage of ERC-223 as compared to ERC-20 is that it consumes only 50% of ERC-20's gas consumption, which makes it less expensive to use on Ethereum's main net. ERC-223 is backward compatible with ERC-20 and is used in a number of major token projects such as LINK and **CNexchange** (**CNEX**).

ERC-777

The main aim of ERC-777 is to address some of the limitations of ERC-20 and ERC-223. It is backward compatible with ERC-20. It defines several advanced features to interact with ERC-20 tokens. It allows sending tokens on behalf of another address (contract or account). Moreover, it introduces a feature of "hooks," which allows token holders to have more control over their tokens.



The **Ethereum Improvement Proposal** (**EIP**) is available here: `https://eips.ethereum.org/EIPS/eip-777`

ERC-721

ERC-721 is an NFT standard. ERC-721 mandates several rules that must be implemented in a smart contract for it to be ERC-721 compliant. These rules govern how these tokens can be managed and traded. ERC-721 was made famous by the **CryptoKitties** project. CryptoKitties is a blockchain game that allows players to create (breed) and trade different types of virtual cats on the blockchain. Each "kitty" is unique and tradeable for a value on the blockchain.

ERC-884

This is the standard for ERC-20-compliant share tokens that are conformant with Delaware General Corporations Law.



The legislation is available here: https://legis.delaware.gov/json/BillDetail/GenerateHtmlDocument?legislationId=25730&legislationTypeId=1&docTypeId=2&legislationName=SB69

The token standard EIP is available here: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-884.md

ERC-1400

This is a security token standard that defines how to build, issue, trade, and manage security tokens. Under ERC-1400, there are a few other standards, which are as follows:

- **ERC-1410:** A partially fungible token standard that defines a standard interface for grouping an owner's tokens into partitions
- **ERC-1594:** This defines a core security token standard
- **ERC-1643:** This is the document management standard
- **ERC-1644:** This is the token controller operations standard
- **ERC-1066:** This standard specifies a standard way to design Ethereum status codes

The aim of ERC-1400 and the standards within it is to cover all activities related to the issuance, management, control, and processing of security tokens.

ERC-1404

This ERC standard allows the issuance of tokens with regulatory transfer restrictions. These restrictions enable users to control the transfer of tokens in different ways. For example, users can control when, to whom, and under what conditions the tokens can be transferred. For example, an issuer can choose to issue tokens only to a whitelisted recipient or check whether there are any timing restrictions on the senders' tokens.

ERC-1404 introduces two new functions in the ERC-20 standard to introduce restriction and control mechanisms. These two functions are listed as follows:

```
contract ERC1404 is ERC20 {
  function detectTransferRestriction (address from, address to, uint256 value)
public view returns (uint8);
  function messageForTransferRestriction (uint8 restrictionCode) public view
returns (string);
}
```



More information on ERC-1404 is available at https://erc1404.org

ERC-1155

This is the multi-token standard. This standard allows bundling of transactions to save costs and more efficient trades. It can be used to create utility tokens and NFTs. The idea is to create a smart contract interface that can handle and represent any number of fungible and NFT types. This means that ERC-1155 can function as ERC-20 and ERC-721, even at the same time. It also improves the efficiency and correctness of ERC-20 and ERC-721.



More information is available in EIP-1155-Multi token standard at `https://eips.ethereum.org/EIPS/eip-1155`

ERC-4626

This is a tokenized vault standard that aims to optimize and unify the technical parameters of tokenized yield-bearing vaults.



Tokenized yield-bearing vaults are smart contracts that accumulate yield as the token holders lock tokens inside the vault. In other words, they help to maximize the yield tokens holders can earn from their deposited tokens.

Tokenized vaults are used on Ethereum in different protocols, such as Yearn Finance, Tulip Garden, and Pickle Finance. These vaults allow a token holder to deposit their tokens into the vault and earn a yield on those tokens. When a token holder deposits their tokens into a vault, they get a vault token. This vault token can appreciate over time. The token holders can get their original tokens back when they return these vault tokens.

The problem is that there is no single unifying standard for implementing standard operations such as depositing and withdrawing. This difference in implementations creates friction between different protocols and results in integration attempts, which may or may not correctly work. For example, if you want to write a DApp that interacts with all these different tokenized vaults, then you have to write adapters that can talk to all these different vaults, making it difficult for integrators and aggregators. ERC-4626 aims to solve this problem by providing a standard way to interact with these vaults. It provides an interface that standardizes deposits, withdrawals, reading balance operations, and other parameters. This standard will result in interoperability, composability, and improvement of the overall decentralized finance and blockchain ecosystem.



More details on the standard are available here: `https://eips.ethereum.org/EIPS/eip-4626`

With this, we have completed our introduction to ERC standards. Now after all this theoretical background, let's see how a token can be built on Ethereum. We will build our own ERC-20-compliant token.

Building an ERC-20 token

In this section, we will build an ERC-20 token. In previous chapters, we saw several ways of developing smart contracts, including writing smart contracts in Visual Studio Code, and compiling them and then deploying them on a blockchain network. We also used the Remix IDE, Truffle, and MetaMask to experiment with various ways of developing and deploying smart contracts. In this section, we will use a quick method to develop, test, and deploy our smart contract on the Sepolia test network. We will not use Truffle or Visual Studio Code in this example as we have seen this method before; we are going to explore a different and quicker method to build and deploy our contract.

In this example, we will see how quickly and easily we can build and deploy our own token on the Ethereum blockchain network. The aim of this exercise is to understand how MetaMask can be used to deploy our new token smart contract on an Ethereum network. We will also see how we can import ERC-20 tokens in MetaMask and use it to transfer funds from one account to another.

We will use the following components in our example, which were both introduced in *Chapter 10*, *Ethereum in Practice*:

- **Remix IDE**, available at `http://remix.ethereum.org`. Note that in the following example, the user interface and steps required might be slightly different depending on the stage of development. This is because Remix IDE is under heavy development, and new features are being added to it at a tremendous pace—as such, some changes are expected in the user interface too. However, the fundamentals are not expected to change, and no major changes are expected in the user interface.
- **MetaMask**. We will use the network we set up in *Chapter 10*, *Ethereum in Practice*, but you can create a new account if required.

Now let's start writing our code.

Building the Solidity contract

First, we explore what the ERC-20 interface looks like, and then we will start writing our smart contract step by step in the Remix IDE.

The ERC-20 interface defines a number of functions and events that must be present in an ERC-20-compliant smart contract. Some more rules that must be present in an ERC-20-compliant token are listed here:

- `totalSupply`: This function returns the number of the total supply of tokens:

```
function totalSupply() public view returns (uint256)
```

- `balanceOf`: This function returns the balance of the token owner:

```
function balanceOf(address _owner) public view returns (uint56 balance)
```

- `transfer`: This function takes the address of the recipient and a specified value as a number of tokens and transfers the amount to the address specified:

```
function transfer(address _to, uint256 _value) public returns (bool
success)
```

- `transferFrom`: This function takes `_from` (sender's address), `_to` (recipient's address), and `_value` (amount) as parameters and returns `true` or `false`. It is used to transfer funds from one account to another:

```
function transferFrom(address _from, address _to, uint256 _value) public
returns (bool success)
```

- `approve`: This function takes `_spender` (recipient) and `_value` (number of tokens) as parameters and returns a Boolean, `true` or `false`, as a result. It is used to authorize the spender's address to make transfers on behalf of the token owner up to the approved amount (`_value`):

```
function approve(address _spender, uint256 _value) public returns (bool
success)
```

- `allowance`: This function takes the address of the token owner and the spender's address and returns the remaining number of tokens that the spender has approval to withdraw from the token owner:

```
function allowance(address _owner, address _spender) public view returns
(uint256 remaining)
```

There are three optional functions, which are listed as follows:

- `name`: This function returns the name of the token as a `string`. It is defined in code as follows:

```
function name() public view returns (string)
```

- `symbol`: This returns the symbol of the token as a `string`. It is defined as follows:

```
function symbol() public view returns (string)
```

- `decimals`: This function returns the number of decimals that the token uses as an integer. It is defined as follows:

```
function decimals() public view returns (uint8)
```

Finally, there are two events that must be present in an ERC-20-compliant token:

- `Transfer`: This event must trigger when tokens are transferred, including any zero-value transfers. The event is defined as follows:

```
event Transfer(address indexed _from, address indexed _to, uint256 _
value)
```

- `Approval`: This event must trigger when a successful call is made to the `approve` function. The event is defined as follows:

```
event Approval(address indexed _owner, address indexed _spender, uint256
_value)
```

Now let's have a look at the source code of our ERC-20 token. This is written in Solidity, which we are familiar with and explored in detail in *Chapter 11*, *Tools, Languages, and Frameworks for Ethereum Developers*.

The source code for our ERC-20 token is as follows—we will explain it step by step before writing it into the Remix IDE.

First is the SPDX license identifier:

```
// SPDX-License-Identifier: GPL-3.0
```

Second, we have the Solidity compiler and language version, which specifies the compiler version using the `pragma` directive for which our program is written:

```
pragma solidity ^0.8.0;
```

Then we create the `contract` object:

```
contract MyERC20Token {
```

After this, the `contract` object is defined with the name `MyERC20Token`.

Then we have the mappings:

```
mapping (address => uint256)  _balances;
mapping (address => mapping(address => uint256)) _allowed;
```

These are the two mappings used in the ERC-20 smart contract. The first one is for keeping balances and the other one is used for allowances.

These are the state variables:

```
string public name = "My ERC20 Token";
string public symbol = "MET";
uint8 public decimals = 0;
uint256 private _totalSupply = 100;
```

They describe the name, symbol, and decimal precision points and the total supply of our token.

This is the `Transfer` event:

```
event Transfer(address indexed _from, address indexed _to, uint256 _value);
```

It has three parameters: `from`, `to`, and `value`. `from` represents the address from which the tokens are coming, `to` is the account to which tokens are being transferred, and `value` is the number of tokens.

This is the `Approval` event:

```
event Approval(address indexed _owner, address indexed _spender, uint256 _
value);
```

This has three parameters: owner address, recipient address, and `value`. The `indexed` keyword allows us to search for a specific log item instead of searching through all logs. It enables log filtration to search and extract only the required data instead of returning all logs.

This is the constructor that is executed when the contract is created:

```
constructor(){
  _balances[msg.sender] = _totalSupply;
   emit Transfer(address(0), msg.sender, _totalSupply);
}
```

It is optional in Solidity and is used to run initialization code. In our example, the initialization code contains the statements to transfer the entire balance, `_totalSupply`, to the creator of the smart contract; in our case, it is the sender account. It also then emits the `Transfer` event, indicating that the transfer has taken place from `address(0)` to `msg.sender` (our contract creator) and `_totalSupply`, which is 100 in our case, just to keep things simple.

This is the `totalSupply` function:

```
function totalSupply() public view returns (uint) {
    return _totalSupply - _balances[address(0)];
}
```

This function returns the total amount of tokens after deducting it from the balance of the account.

This is the `balanceOf` function:

```
function balanceOf(address _owner) public view returns (uint balance) {
    return _balances[_owner];
}
```

The `balanceOf` function returns the balance of the token owner.

The `allowance` function is as follows:

```
function allowance(address _owner, address _spender) public view returns (uint
remaining) {
    return _allowed[_owner][_spender];
}
```

The `allowance` function returns the total remainder of the tokens.

This is the `transfer` function, which returns `true` or `false` depending on the result of the execution:

```
function transfer(address _to, uint256 _value) public returns (bool success) {
    require(_balances[msg.sender] >= _value,"value exceeds senders balance");
    _balances[msg.sender] -= _value;
    _balances[_to] += _value;
    emit Transfer(msg.sender, _to, _value);
    return true;
}
```

The `require` convenience function is used to check for certain conditions and throw an exception if the conditions are not met. In our example, `require` checks whether the value exceeds the sender's balance, and if it does, an error message will be generated stating that `value exceeds senders balance`. If this check passes, the transfer occurs, and after emitting the `Transfer` event, the function returns `true`, indicating the successful transfer of tokens.

This is the `approve` function, which returns `true` or `false` depending on the result of the execution of the function:

```
function approve(address _spender, uint256 _value) public returns (bool
success)
{
    _allowed[msg.sender][_spender] = _value;
    emit Approval(msg.sender, _spender, _value);
    return true;
}
```

This function takes `_spender` (the user) and `_value` (number of tokens) as arguments and serves as a mechanism to provide approval to the user to acquire the allowed number of tokens from our ERC-20 contract.

This function is the `transferFrom` function, which can be used to automate the transfer of tokens from one address to another:

```
function transferfrom(address _from, address _to, uint256 _value) public
returns (bool success)
    {
        require(_value <= _balances[_from],"Not enough balance");
        require(_value <= _allowed[_from][msg.sender],"Not enough allowance");
        _balances[_from] -= _value;
        _balances[_to] += _value;
        _allowed[_from][msg.sender] -= _value;
        emit Transfer(_from, _to, _value);
        return true;
    }
```

It takes three parameters: `_from`, `_to`, and `_value`. It returns `true` or `false` depending upon the execution of the function. First, with the `require` functions, the balance, and allowances are checked to ensure that enough balance and allowance are available. After that, the transfer occurs, and eventually the `Transfer` event is emitted, followed by a `true` Boolean value returned by the function indicating the successful transfer.

Now that we understand what our source code does, the next step is to write it in the Remix IDE and deploy it.

Deploying the contract on the Remix JavaScript virtual machine

In this step, we simply take the source code and write or simply paste it in the Remix IDE. To achieve this, take the following steps:

1. Open the Remix IDE.
2. When the Remix IDE starts up, select **SOLIDITY** under **Environments**, as we are going to write smart contracts using the Solidity smart contract language.
3. After selecting the Solidity environment, create a new file by choosing the **FILE EXPLORERS** option from the list of icons on the left-hand side and add a new file by clicking the + sign.
4. Create a new file in the Remix IDE named `erc20example.sol`.
5. When we have created the new file, simply write the source code in the IDE, or paste it directly.
6. Compile the source code. To do this, click on **Compile erc20example.sol**. Optionally, **Auto compile** can also be selected under **COMPILER CONFIGURATION**, which will compile the code automatically as soon as it is written into the IDE:

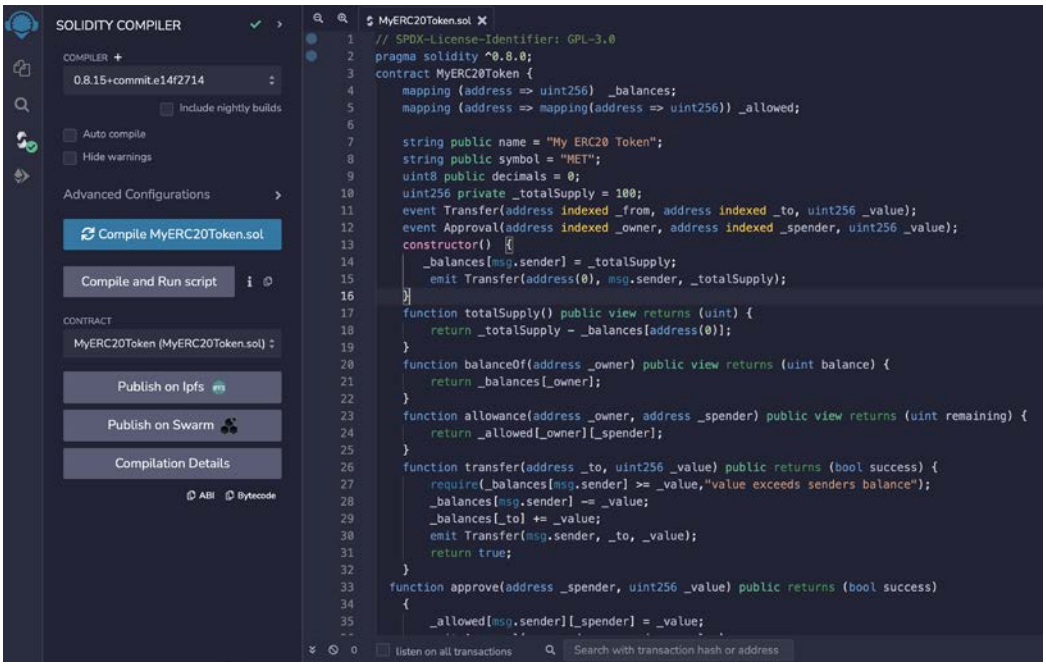


Figure 15.1: Solidity compiler in Remix

7. Once compiled successfully, it is ready to be deployed. First, we will deploy it on JavaScript VM available with the Remix IDE to ensure that everything works. Once we have tested that it can be deployed correctly and works as per our expectations, we could deploy it on main net using MetaMask. In our example, however, we will deploy it on the Sepolia (`https://sepolia.dev`) test net using MetaMask.



To deploy on main net, simply choose main net from MetaMask after ensuring that enough funds are available to deploy it on main net.

Remember that we have some ether left from our exercise in *Chapter 13*, *The Merge and Beyond*, and have a test account on the Sepolia network. We can use the same account for this example or create a new account and fund it using the process described in the aforementioned chapter. As an alternative, you can use faucets available for the Sepolia network at `https://sepolia-faucet.pk910.de`.

8. After compilation, we deploy the smart contract using the **Deploy & Run Transactions** interface available within the Remix IDE. Make sure the environment JavaScript VM is selected and an account is selected from which to deploy, and then select **Deploy**:

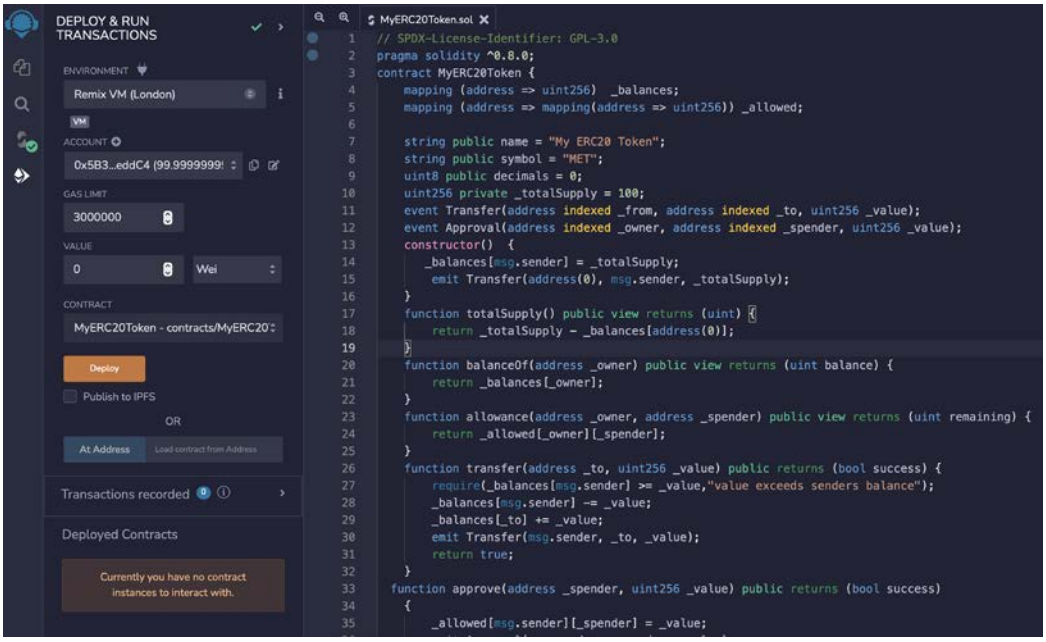


Figure 15.2: Deploying and running transactions in Remix

Once it's deployed, the contract will become available under **Deployed Contracts** and, in the logs, we can see relevant details regarding the deployment of the contract.

Most importantly, we can see in the logs that when the contract is created, the first event emitted is `'Transfer'`. Here, we can see that all 100 tokens have been transferred to the owner account, `0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8`:

```
logs
[
    {
        "from": "0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8",
        "topic":
"0xddf252ad1be2c89b69c2b068fc378daa952ba7f163c4a11628f55a4df523b3ef",
        "event": "Transfer",
        "args": {
            "0": "0x0000000000000000000000000000000000000000",
            "1": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
            "2": "100",
            "_from": "0x0000000000000000000000000000000000000000",
            "_to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
            "_value": "100"
        }
    }
]
```

Once it's deployed, we will see under **Deployed Contracts** all the functions exposed by the contract in the Remix IDE:

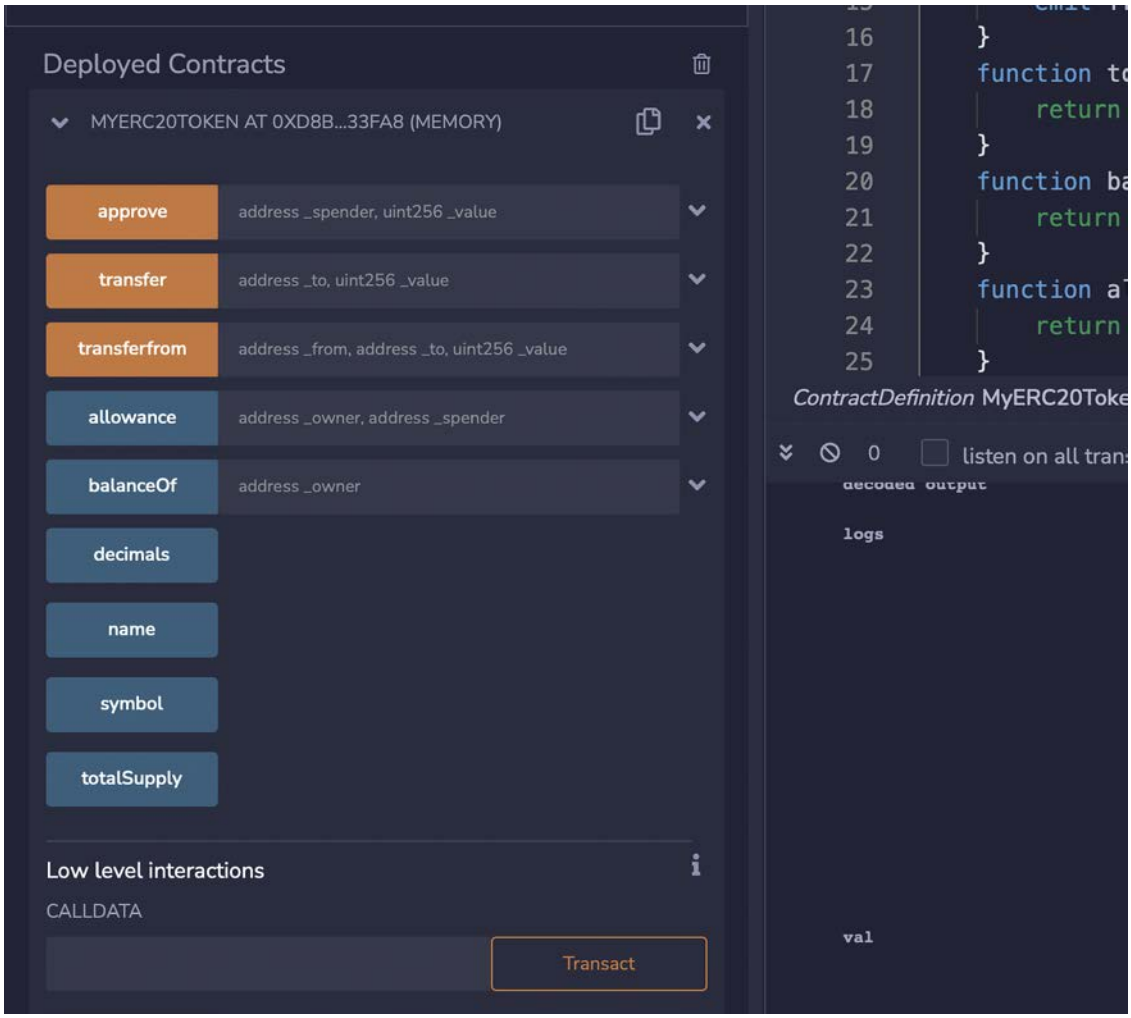


Figure 15.3: Deployed Contracts—exposed functions in Remix for our contract

We can test the functionality of our contract using this interface. For example, calling `totalSupply` shows a value of `100`, which indicates the number of tokens, and calling `symbol` shows the string `MET`, our token's symbol.

At this point, our contract works, and we have tested it locally. Now we can deploy it onto the Sepolia test network. We will use MetaMask for this purpose:

1. In the Remix IDE, under **DEPLOY & RUN TRANSACTIONS**, select the **Injected Web3** option from the ENVIRONMENTS list. This is the execution environment that is provided by enabling web3 within the browser using the MetaMask plugin.
2. Next, confirm that MetaMask is running and connected to the Sepolia test network. This should be easy to check, as we used this same network in *Chapter 13, The Merge and Beyond*.

   If everything is working in MetaMask, it should display a screen similar to the following screenshot. Note that you may have to log in again to MetaMask. Once you're logged in, select the Sepolia network and an account that has some ether in it.

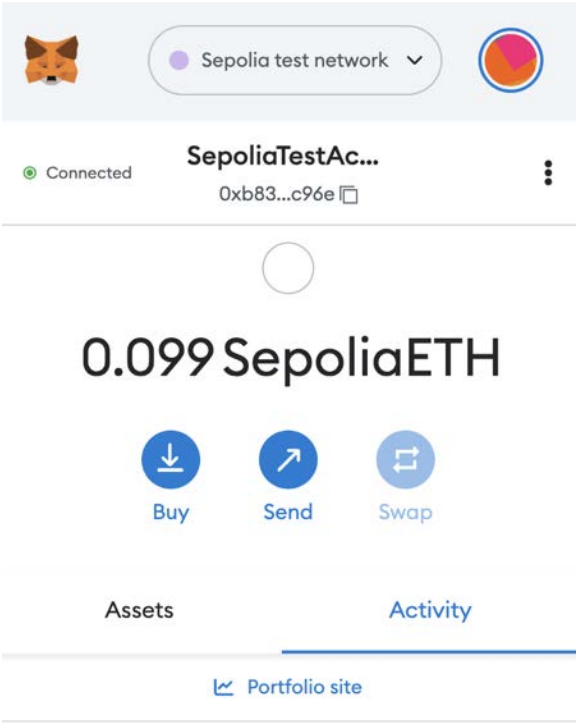


Figure 15.4: Sepolia test network in MetaMask

3. In the Remix IDE, choose **Injected Provider - MetaMask** in the **ENVIRONMENT** field under **DEPLOY & RUN TRANSACTIONS**. Once connected you will notice that the **Sepolia network** (**Sepolia (11155111) network**) is shown in remix IDE just under the **Environment** box.

Note that MetaMask also shows that it's connected to the Sepolia network, and it also shows the account that we have set up in MetaMask. Now we are all set to deploy this on the Sepolia test network.

4. Click **Deploy**, which will open the MetaMask window to confirm the transaction.
5. Click **Confirm**, and the contract will be deployed. We can see this in the history in MetaMask. Also, in the Remix IDE logs, we can see if it has been deployed successfully.

Like the tests that we did earlier in this example when we deployed our contract on the JavaScript VM, we can invoke different functions exposed by our new ERC-20 token directly from the Remix IDE. We can also see our token on the Etherscan token tracker. This is shown via `https://sepolia.etherscan.io/tx/0xfd427145a393fb6dbb08bf9f3e4c945acb1039404503380692b81b366b0f23e6`.

The Etherscan page also shows detailed information about out token contract such as total supply, contract address, etc, which can be accessed here : `https://sepolia.etherscan.io/token/0x07c152a6ab577e8f78e3bede502d79652787a9fc`.

So here we have it, our own MET token deployed on the Sepolia test network. Now, if desired, we could deploy this to the Ethereum main net if we have some ether available. We can perform the same steps to deploy it, with the only difference being that in MetaMask, we will choose the Ethereum main net as the network.

Adding tokens in MetaMask

Once we have deployed our contract and our ERC-20 token is now on the blockchain (so to speak), unless we are able to view it and perform operations on it, the token on its own is of no real use.

To perform operations on the token, we can manually create commands using the Web3.js `sendRawTransaction` method and use the JavaScript and command-line `geth` console to interact with it. Alternatively, we can use an easier option and simply add the token to a wallet. Wallets abstract away the complexities associated with transaction creation and management and provide an easy-to-use interface to perform transfers and similar tasks.

MetaMask can serve as a wallet for tokens and provides an interface to add tokens. In this section, we'll see how we can add our MET token to MetaMask and perform some operations on it:

1. Open MetaMask and find the **Add Token** option on the main user interface where it says *Don't see your token? Import tokens* and click on **import token.** Alternatively, open the account menu and it will display the account import option.
2. Click on **Add Token** and select **Custom Token**.

3. Enter the contract address, `0x07C152A6ab577E8F78e3bedE502D79652787A9FC`, which will automatically display the **Token Symbol** and the decimal points, as shown in the following screenshot:

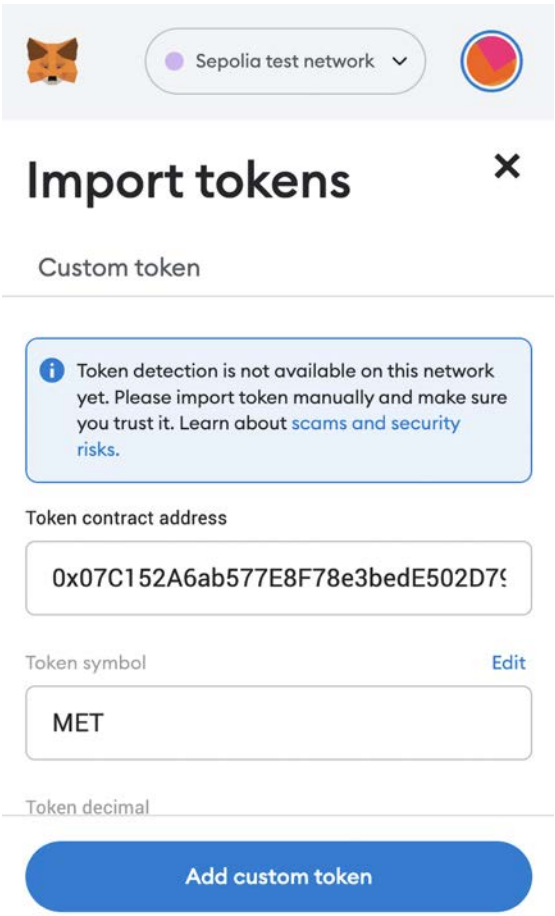


Figure 18.5: Custom token in MetaMask

4. Press **Add custom token**, and then press **Import tokens** on the next screen. Now we have **100 MET** in our MetaMask wallet! Now let's send 10 MET to another account on the Sepolia network.
5. Click on **Send**, then enter the target account's details.
6. Enter **10 MET** and click **Next**.
7. Then click **Confirm**, and the transaction will be processed.
8. Now, coming back to the Remix IDE, we can see that our balance has been reduced by 10 MET. We can see this by passing the address of the token holder, `0xb838042b89ebdfb1d5ed0de32323f798eb5dc96e`, and clicking the **balanceof** button. Similarly, we can check the balance of the target account, `0xb838042b89ebdfb1d5ed0de32323f798eb5dc96e`, to which we transferred 10 MET, which is now **10 MET**. You can see this by passing `0xb838042b89ebdfb1d5ed0de32323f798eb5dc96e` to the `balanceof` function.

We can see all the transfers of our ERC-20 token, MET, on Etherscan: `https://sepolia.etherscan.io/token/0x07c152a6ab577e8f78e3bede502d79652787a9fc`.

With this, we have covered how to create an ERC-20 token from scratch and deploy it on the Ethereum blockchain.



Note that OpenZepplin is an open source framework for building secure smart contracts and has many pre-built token standards that can be easily used by simply importing them in your Solidity code. More information is available here: `https://docs.openzeppelin.com/contracts/`

Let's now have a look at some of the novel concepts that are emerging due to the remarkable success of tokenization and the blockchain ecosystem in general.

Emerging concepts

With the advent of blockchain and tokenization, several new concepts have emerged over the last few years. We will introduce some of them now.

Tokenomics/token economics

Tokenomics or token economics is an emerging discipline that is concerned with the study of economic activity, economic models, and the impact of tokenization. It deals with the goods and assets that have been tokenized and the entities that are involved in the entire process of token issuance, sale, purchase, and investment.



L You might have heard another term, cryptoeconomics, which is a related but slightly different term. Cryptoeconomics is concerned with the same topics, but it is a superset of tokenomics. In other words, tokenomics is a subset of cryptoeconomics. Tokenomics is only concerned with tokens and tokenization ecosystems, but does not include the broader blockchain networks, protocols, and cryptocurrencies.

With the use of the **proof of work** (**PoW**) mechanism in Bitcoin, it was demonstrated for the first time that computer protocols can be designed in such a way that attacking a system does not result in achieving an uneven advantage or commercial benefit. This concept then further matured into what we call today cryptoeconomics. This can also be understood as a combination of economics, game theory, and cryptography.

Token engineering

Token engineering is an emerging concept that is looking at tokenization from an engineering perspective and is striving to apply the same rigor, systems thinking, and mathematical foundations to tokenization and blockchain in general that a usual engineering discipline has. This subject is still in its infancy; however, good progress has been made toward the development of this new discipline.



More information on token engineering can be found at `https://tokenengineeringcommunity.github.io/website/`

Token taxonomy

There is a lack of consistent taxonomy for tokens. As such, there are no clear standards defined on how to design and manage tokens. Also, it is not clear how to reuse an already existing and working token design, or if any exist at all.

Therefore, there is a need for a classification system that categorizes all different types of tokens according to the different attributes they have. There is also no universal classification of different attributes of tokens such as type, value, and economic attributes. Such a system would benefit the tokenization ecosystem tremendously.

Don't confuse this with the ERC standards we explained earlier; those are development standards, and they are certainly useful. But they are specific in scope and are not the universal classification of tokens.

Some work on this was started by Interwork Alliance by producing a **Token Taxonomy Framework** (**TTF**). More details on this work can be found on their website: `https://interwork.org`.

Summary

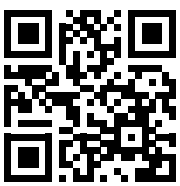
In this chapter, we covered tokenization and relevant concepts and standards. We also covered different types of tokens and related token standards. Moreover, a primer on trading and finance was also provided to familiarize readers with some standard finance concepts, which help us to understand the decentralized finance ecosystem too, as most terminology is borrowed from traditional finance.

Also, we introduced a practical example of how to create our own ERC-20-compliant token using the Ethereum platform. Finally, we introduced some emerging ideas related to tokenization.

In the next chapter, we will explore how blockchain can be used in business contexts, in addition to the cryptocurrency and contract deployment related contexts we have reviewed so far.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

16

Enterprise Blockchain

In this chapter, we'll investigate **enterprise blockchains**. What is the standard architecture of enterprise blockchains? Why are they required? We will answer these questions and will also try to answer the big question of why current public blockchains are not necessarily a suitable choice for enterprise use cases.

We will also introduce several enterprise blockchain platforms, including Quorum. Along the way, we'll cover the following topics:

- Enterprise solutions and blockchain
- Limiting factors
- Requirements
- Enterprise blockchain versus public blockchain
- Use cases of enterprise blockchains
- Enterprise blockchain architecture
- Designing enterprise blockchain solutions
- Blockchain in the cloud
- Currently available enterprise blockchains
- Enterprise blockchain challenges
- Quorum
- VMware Blockchain
- Setting up Quorum with IBFT
- Other Quorum projects

We discussed several different blockchain types in *Chapter 1*, *Blockchain 101*, including permissioned, public, private, and consortium blockchains. Enterprise blockchains are a permissioned **consortium** chain type that primarily tackles enterprise requirements.

Permissioned doesn't mean private. Permissioned blockchains can also be public and allow access only to known participants. Enterprise blockchains are usually private and permissioned and are run between consortium members.

While public blockchains provide integrity, consistency, immutability, and security guarantees, they lack certain features, which makes them less suitable for enterprise usage. We'll discuss these limitations in detail shortly.

First, we'll look into what enterprise solutions are and how blockchain can fit into an enterprise. Second, we'll see what questions should be answered before introducing an enterprise blockchain solution to a business.

Enterprise solutions and blockchain

Enterprise solutions integrate different fragments of a business and enable it to achieve its goals by providing business-critical information to the stakeholders. Here, we'll consider the question: *Does blockchain fit this definition and help achieve enterprise business goals?*

To find out if a blockchain is suitable for an enterprise or not, we can ask some questions:

- How can business/enterprise processes be improved using blockchain technology?
- Do I really need an enterprise blockchain? For this, we can ask some questions to rationalize whether we need an enterprise blockchain solution or not. Does my use case:
    - Need shared data between participants?
    - Have participants from other organizations that are not necessarily trustworthy and have conflicting interests?
    - Need strict auditing? A blockchain can provide this, as it is an immutable and tamper-proof chain of records that can provide a definite audit trail of activities performed on the chain (enterprise system).
    - Need to ensure the confidentiality of transactions?
    - Need to ensure the anonymity of participants?
    - Need controlled but transparent updates to the ledger?
    - Not need a single trusted authority? Instead, the decisions on the network (updates to the ledger) should be consortium member-driven and agreed upon between members.

If the answer to any of the preceding questions is *yes*, then using an enterprise blockchain solution could be a good option. Otherwise, a traditional database might give a better alternative. In addition to the questions mentioned here, when proposing an enterprise blockchain solution, we also need to answer some other questions from a business perspective that help establish the overall vision and strategy of the implementation of the blockchain solution:

- What is the overall objective of the proposed blockchain solution? Does it align with the business goals of the enterprise? Is it a **Proof of Concept** (**PoC**) project, intended just to demonstrate an idea, or a production project with real business deliverables?
- Where would it be deployed? Cloud or local hosting? Who will manage the blockchain solution once it is deployed?

- Risk management considerations—does the solution follow any established guidelines for risk management? For example, NIST 800-37 (`https://csrc.nist.gov/publications/detail/sp/800-37/rev-2/final`) is a risk management framework that provides a process used to manage security and privacy risk for IT systems and organizations.
- Are there any other projects that this organization may have implemented already using enterprise blockchain? Can we learn from previous experiences and leverage some of the resources and best practices that may have been developed previously?

These questions should be—and are usually—asked when developing any other enterprise solution, but in relation to blockchain, these questions become even more critical due to the nascent and immature nature of enterprise blockchain technology. A clear definition of the objectives, along with a clear alignment with business requirements, will result in an implementation that meets business goals.

Next, we'll envisage some success factors that can help enterprise blockchains to become successful and adopted in the enterprise.

Success factors

There are some business-oriented factors that should be addressed for a successful enterprise blockchain solution.

Of the utmost importance is the requirement that the blockchain solution should bring some economic value and help to achieve real business goals. Moreover, enterprise solutions should be aligned with business goals.

The primary value of enterprise blockchains is in the property of being a sharable, replicable, permissioned ledger between organizations that immediately results in cost reduction by eliminating the need for data exchange. In doing so, we also eliminate the need for infrastructure and tools to support such exchanges. Also, due to the security, immutability, and auditing provided as inherent features of a blockchain, there is no need to invest separately for these requirements.

Blockchain solutions that integrate easily/seamlessly with existing systems provide better value because already mature legacy systems (at least at this stage) provide a mechanism to connect with the blockchain, read its data, and save it in a known format that the enterprise is already familiar with. Moreover, just a blockchain network on its own is not entirely useful in enterprises if it is not integrated with existing back-office systems such as **Enterprise Resource Planning** (**ERP**), backend databases, record reconciliation systems, or other organization reporting tools.

An enterprise blockchain solution should be seen as a complete end-to-end enterprise solution as part of the larger enterprise architecture, instead of only a siloed blockchain network. We'll explore this topic later in this chapter, under the *Designing enterprise blockchain solutions* section.

Enterprise-grade governance, control, and security are also desirable features from a business perspective, as they allow business stakeholders to apply already-established organization rules and policies to the blockchain. This can also help to achieve regulatory and compliance requirements.

Now, the following question arises: *With the availability of many different public blockchain platforms, why has the adoption of blockchain technology in enterprises still not been fully achieved?* The reason is that there are several factors that make public blockchains unsuitable for enterprise use cases. In the next section, we'll answer this question and explore why public blockchains are not quite suitable for enterprise use cases.

Limiting factors

We discussed several benefits of blockchain technology in general in *Chapter 1*, *Blockchain 101*. While all those benefits are attainable using public blockchains, several features are lacking in public blockchains, which makes them unsuitable for use in enterprise use cases. The interest in enterprise blockchain arises from these limitations in public blockchains, along with specific requirements in any business.

We'll describe some of the most common concerns next:

- Slow performance: Public blockchains are slow and can process only a few transactions per second. Bitcoin processes 3-4 transactions per second, while Ethereum processes around 14. This low transaction rate is not suitable for businesses that usually require high transaction speed. For example, card payment businesses typically need to process thousands of transactions per second.
- Lack of access governance: Public blockchains are available for anyone to join, which makes it easier for investors and cryptocurrency enthusiasts to join and helps with the network effect. However, in an enterprise, all participants must be known so that everyone knows who they are dealing with. This lack of an identification and access control mechanism makes public blockchains rather unsuitable for businesses.
- Lack of privacy: Public blockchains are inherently transparent, and everything on the ledger is visible to everyone. This means anyone can easily view transaction details and participants involved in a transaction. Lack of privacy results in a strategy leak problem, which we describe next.
- Strategy leak problem: As public blockchains are open to everyone, it is possible that by using traffic analysis, one can figure out which party is communicating with which other party, and thus can infer who is doing more business. Moreover, even a simple transaction count from an address can be used to infer the scale of the business, and thus the revenue an entity might be generating.
- Probabilistic consensus: Traditional public blockchains usually use a **Proof of Work** (**PoW**) type of consensus mechanism, which is inherently a probabilistic protocol that provides probabilistic finality. Even though the confirmations mechanism, as discussed in *Chapter 6*, *Bitcoin Architecture*, does give a certain level of confidence that a transaction is irrevocable, it is still possible that the chain may fork and transactions can be lost. This issue is particularly a concern for businesses where once a transaction commits, it is deemed final. Imagine receiving a property ownership document on a chain from someone only to discover later that the blockchain has forked, and that you are no longer the owner of the property.
- To address this limitation, enterprise blockchains use deterministic consensus algorithms, which provide immediate finality.

- Note that some newer public blockchains such as Avalanche and Cosmos do have deterministic finality. However traditionally public chains have been and some still are PoW, e.g., Bitcoin.
- Transaction fees: In Ethereum or other similar blockchains, the transaction fee is charged in the native cryptocurrency for each transaction execution. While this mechanism provides incentives for miners and protects against spam, there is no requirement for such an arrangement in enterprise blockchains. If an organization were to use public blockchains for their business transactions, then they'd need cryptocurrency in reserve to pay for operations on the blockchain. This extra cost might be undesirable for some businesses. Moreover, when the network is busy, the gas fees go up significantly, which results in even more costs.
- Network congestion: It is possible on public blockchain networks that, due to busy dApps, my own dApp is impacted because the network is busy. If the whole network is congested due to too many transactions, then all dApps would run slow, and this is the risk that can prove detrimental to a business. While the concerns mentioned here are considered limitations in public blockchains, based on these concerns and limitations, we can derive and define several requirements, or features, of a blockchain that will enable it to become an enterprise blockchain. In other words, the limitations in public blockchains can be seen as requirements of enterprise blockchains.

The next section talks about several features that a blockchain should possess to become suitable for enterprises.

Requirements

In addition to *integrity* and *consistency*, which are also provided by public blockchains, there are several other requirements specifically for enterprise blockchains that make them suitable for enterprise use cases. In some cases, some requirements become even stricter in enterprise blockchains compared to public blockchains. For example, in public blockchains, eventual consistency is acceptable. However, in enterprise blockchains, the moment a transaction is committed, it should immediately finalize and irrevocably become part of the global record (state). Thus, we'll begin by briefly defining integrity and consistency before introducing specific requirements for enterprise blockchains:

- **Integrity**: This attribute of a blockchain is provided by the use of hash functions and digital signatures, and plays a vital role in the overall security of the blockchain. Hash functions allow us to check for any data modifications, whereas digital signatures ensure that the messages that originated from a sender have not been altered.
- **Consistency**: This attribute of a blockchain ensures that all honest nodes agree on the same sequence of blocks. To achieve this, various mechanisms are used, such as PoW or **Byzantine Fault Tolerance** (**BFT**) consensus protocols.

You can refer to *Chapter 1*, *Blockchain 101*, for a refresher on this, as we introduced these properties there in more detail.

Now, we'll introduce three fundamental requirements that should be met for a blockchain to become suitable for enterprises. These requirements are **privacy**, **performance**, and **access governance**.

Privacy

Privacy is of paramount importance in enterprise blockchains. Privacy has two facets: first, *confidentiality*, and second, *anonymity*.

Confidentiality is a fundamental requirement in an enterprise. It is anticipated that, in enterprise blockchains, all transactions hide their payloads so that the value of the transactions is not revealed to anyone who is not privy to the transaction.

Private transactions can be defined as transactions that meet the privacy requirements of an enterprise use case. There are two types of private transactions, as defined by **Enterprise Ethereum Alliance** (**EEA**):

- **Restricted private transactions**: This type of private transaction is transmitted only to those parties on the blockchain network who are privy to the transaction.
- **Unrestricted private transactions**: Under the unrestricted private transaction paradigm, private transactions are transmitted to all participants on the network, regardless of whether they are privy to the transaction or not. The payload is still encrypted and confidential, but the transaction itself is broadcast to the entire network.

There are many approaches to achieving privacy in enterprise blockchains and blockchain in general. These methods range from an off-chain mechanism like privacy managers (used in Quorum and some other enterprise chains) to utilizing **zero-knowledge proofs** (**ZKPs**), trusted hardware, and **secure multiparty computation** (**SMPC**). We'll cover some of these techniques in the next chapter, *Chapter 17*, *Scalability*, but in this chapter, we will mainly focus on privacy manager-based privacy, where an off-chain component is used to provide privacy services. We will discuss the privacy manager-based approach in the *Quorum* section later in this chapter.

Anonymity in enterprise blockchains might not seem a strict requirement at first, because all participants are known and identified. However, it is necessary for scenarios with some competing participants and conducting business on-chain. It could be essential, for example, that in a scenario where parties *X* and *Y* are transacting together, party *Z* doesn't find out which parties are transacting together. Even though the transaction values are not visible, they can still reveal details about the business that the two parties might be doing. If that information is available publicly, then other parties on the consortium chain will gain market intelligence and may try to influence that process via marketing or other methods.

Performance

Due to the high-speed requirements of businesses, enterprise blockchains must be able to process transactions at a high rate.

Performance has two facets: scalability and speed. **Speed** deals with how many transactions can be processed in a given amount of time. It deals with the ability of a system to handle a large volume of transactions at an acceptable speed.

On the other hand, we have the number of participants in a system. Public blockchains can support a large number of users. This is especially true in cryptocurrency blockchains such as Bitcoin and Ethereum.

This is possible due to the PoW or **Proof of Stake** (**PoS**) algorithms for consensus used in these public blockchains. However, in enterprise blockchains, a different class of consensus algorithms (most commonly, BFT) are used, which do not work well with a large number of nodes. This results in limiting the number of users on a consortium chain.

Here, we have to consider a tradeoff. Ideally, an enterprise blockchain should be able to perform well with a large number of users; however, with the tradeoff, enterprise blockchains should be able to process transactions at a high rate. This is what most enterprise blockchains focus on since, in many use cases, the number of nodes is not that high, and speed can be prioritized over scalability.

In public blockchains, sometimes, network congestion caused by high volumes of traffic can cause performance issues and can increase transaction processing times. This is detrimental to **enterprise dApps**, which need faster transaction processing and response times. Network congestion also results in increased gas prices, which can also be a concern for businesses from a cost perspective.

Access governance

From another angle, being a permissioned blockchain, enterprise-grade access control (in the form of either a new mechanism on the chain or control driven by an enterprise SSO already in place) is a fundamental requirement in enterprise blockchains. As all participants must be identifiable on a consortium chain, it is essential to build an access control mechanism that facilitates that process. This feature can be achieved by using enterprise-grade access control mechanisms such as **Role-Based Access Control** (**RBAC**).



RBAC is an ANSI standard. More information is available in the ANSI INCITS 359-2004 document. You can find more information on the document and the RBAC standard here: `https://csrc.nist.gov/projects/role-based-access-control`.

The access control mechanism also can address **Know Your Customer** (**KYC**) requirements.

In addition to privacy, performance, and access governance, which are usually identified as three fundamental requirements of enterprise blockchains, there are also some other requirements that are highly desirable but can be considered somewhat optional, as they are more use case dependent. We'll describe these next.

Further requirements

In this section, we'll present some further requirements that are very useful and can increase the suitability/efficacy of enterprise blockchain solutions.

Compliance

A common concern is compliance. Public blockchains are not suitable for enterprise use cases due to strict regulatory and law requirements in almost all sectors such as finance, health, and government. Compliance challenges mainly include regulatory compliance, standards compliance, data sovereignty, and liability concerns.

We'll define these briefly next:

- **Compliance with standards and regulations**: Often, in enterprise use cases, compliance with a technical standard or laws is required. For example, compliance with GDPR is mandatory in the European Union and the European Economic Area. Another example is compliance with **Financial Conduct Authority** (**FCA**) regulations in the UK. Moreover, it might be necessary, in certain use cases, to comply with technical standards such as cryptography standards published by NIST. One such example is the use of NIST-approved curves, as described here: `https://csrc.nist.gov/Projects/elliptic-curve-cryptography`.
- **Data sovereignty**: Data sovereignty is a broad topic that mainly subjects data to the laws of the country in which it is located. For example, under GDPR, transferring personal data outside the EU is subject to adequacy decisions (Article 45) and the appropriate safeguards (Article 46).



More on this GDPR guidance can be found here: `https://gdpr.eu/tag/chapter-5/`.

  As public blockchains are borderless and geographically dispersed systems, compliance with such regulations can be challenging.

- **Liability**: In traditional business and IT systems, legal responsibility often lies with a party who provides a specific service; for example, a cloud service provider is responsible for handling data in accordance with local laws and regulations. In a decentralized public blockchain, the data is on a public blockchain, and it, therefore, becomes challenging to keep a single party responsible for data management or providing services. In case of malicious incidents, again, it is not possible to hold any single party responsible.

This limitation poses a significant challenge in enterprise settings where, usually, a responsible party is in control of service provision and is held accountable. From another angle, we know that, in traditional systems, in case of any problems, a legal system can help. However, in a decentralized blockchain, it can become quite challenging to blame any single party for their actions. Therefore, in enterprise blockchains, the ability to comply with regulations and laws becomes a very sought-after attribute and is achievable by appropriate identification, KYC, and access control mechanisms. In an enterprise chain, the participants are known and can be held accountable for their actions.

Interoperability

As the enterprise blockchain ecosystem evolves, the need to be able to exchange data between disparate enterprise and public blockchains also arises. Lack of standardization also alleviates this problem; however, standardization efforts are underway, such as EEA (discussed in the *Enterprise blockchain challenges* section). There are also interoperability solutions being developed and available for blockchains that allow interoperability between chains, such as **ION**, **IBC protocol**, **Polkadot**, and **Interledger**.

Integration

No enterprise blockchain is an isolated end-to-end solution. It has to integrate with existing enterprise systems or other off-chain systems that are part of the whole enterprise solution to fulfill business goals. Therefore, enterprise blockchains must provide interfaces for integration. This can be as simple as providing RPC endpoints, or as complex as building blockchain-specific connectors and plugins to integrate with enterprise service buses or other legacy systems—more on this in the *Enterprise blockchain architecture* section.

Integration with security devices such as **Hardware Security Modules** (**HSMs**) is also quite desirable for many enterprise use cases where strict security is required, or due to regulatory or compliance requirements.

Ease of use

Usually, enterprise systems are easy to deploy and use. Deployment in enterprises is easy and quick and often relies on mature frameworks and tools, such as Ansible and other proprietary tools, but this is not the case with blockchain.

The deployment of enterprise systems is a well-studied, understood, and mature area. With enterprise orchestration tools and established techniques over the years, enterprise deployment has become easy. However, blockchains are not as easy to deploy as other enterprise systems. With the availability of **Blockchain as a Service** (**BaaS**) and deployment automation tools, this is changing. However, there is still some work that needs to be done.

Monitoring

Monitoring using visualization tools plays a vital role in any enterprise solution. Without the ability to monitor and visualize a system, it is almost impossible to ensure the health of the enterprise system. It is also a desirable feature in enterprise blockchain solutions to be able to visualize the blockchain network. Monitoring a blockchain allows an administrator to keep an eye on the network's health and operations. It allows an administrator to monitor and respond to the events of interest, such as a node going down, communication link slowness, a node being unable to sync with the blockchain, and many other scenarios.

Secure off-chain computation

In some scenarios, it is desirable to be able to offload some intensive computation to off-chain systems; for example, if there is a requirement to do some computation that requires **High-Performance Computing** (**HPC**) resources.

This somewhat overlaps with integration, but it's mentioned here separately because of specific security requirements that the off-chain computations must be provably correct with integrity and authenticity guarantees.

Better tools

Usually, in enterprise systems, there are many supporting tools and utilities packaged with the main product to operate the software. For example, these include administration tools, deployment tools, developer utilities, visualization tools, management tools, and end user tools. Blockchain platforms with better tooling are much more desirable because of better user support. Tools such as block explorers, user administration modules, and dApps to manage smart contracts are quite useful. They are gradually becoming more mature as the whole blockchain ecosystem is growing.

Now that we understand the features of enterprise blockchains, we will present a comparison between public and enterprise blockchains to help understand the main differences.

Enterprise blockchain versus public blockchain

In this section, we'll provide a comparison between public and enterprise blockchains. Consider the table shown here, which assesses some points of comparison between the two blockchain types. This is not an exhaustive list; however, we'll touch on the most major points:

Aspect	Public chains	Enterprise chains
Confidentiality	No.	Yes.
Anonymity	No.	Yes.
Membership	Permissionless.	Permissioned via voting, KYC, usually under an enterprise blockchain.
Identity	Anonymous.	Known users.
Consensus	PoW/PoS.	BFT.
Finality	Mostly probabilistic.	Requires immediate/instant finality.
Transaction speed	Slower.	Faster (usually, should be).
Scalability	Better.	Not very scalable, usually due to consensus choice. Usually, a much smaller number of nodes compared to public chains.
Regulatory compliance	Not usually required - difficult to achieve.	Required at times - comparatively easier to achieve.
Trust	Fully decentralized.	Semi-centralized and managed via consortium and voting mechanisms.
Smart contracts	Not strictly required; for example, in the Bitcoin chain.	Strictly required to support arbitrary business functions.

The preceding table compares several key aspects of public and enterprise platforms. Next, we'll briefly consider some of the use cases of enterprise blockchains.

In the next section, we'll describe enterprise blockchain architecture, which can help us communicate with stakeholders, promote early design decisions, serve as a reusable model for development, build shared understanding, and understand how the system is structured.

Enterprise blockchain architecture

A typical enterprise blockchain architecture contains several elements. We saw a generic blockchain architecture in *Chapter 1*, *Blockchain 101*, and we can expand and modify that a little bit to transform it into an enterprise blockchain architecture that highlights the core requirements of an enterprise blockchain. These requirements are mostly driven by enterprise needs and use cases:

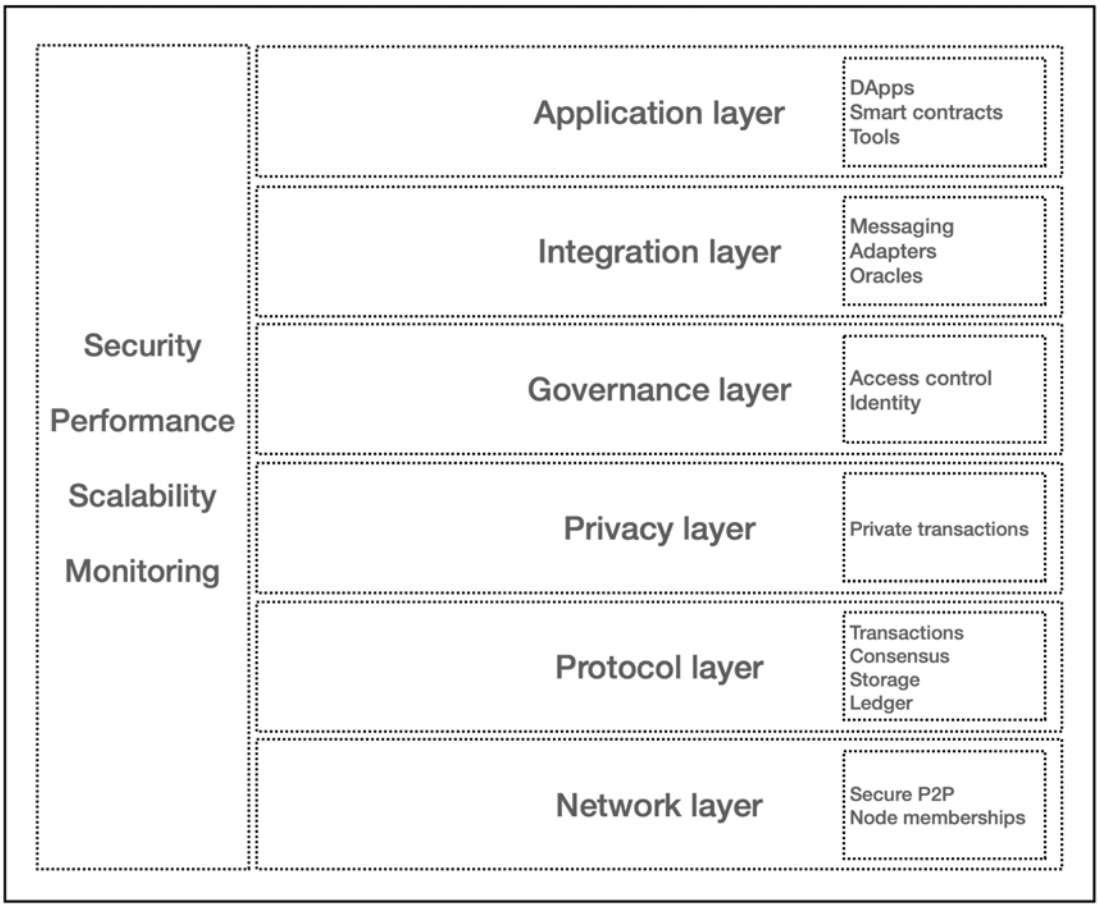


Figure 16.1: Enterprise blockchain layered architecture

We'll discuss each of these layers as follows:

- **Network layer**: The network layer is responsible for implementing network protocols such as **peer-to-peer** (**P2P**) protocols used for information dissemination.

- **Protocol layer**: This is the actual ledger layer, or blockchain layer, where the core consensus, transaction management, and storage elements are implemented.
- **Privacy layer**: This layer is responsible for providing one of the core features of enterprise blockchain: privacy. There are a number of ways to achieve privacy, including off-chain privacy managers, zero knowledge, and hardware-assisted privacy. Hardware-based privacy is usually supported using **trusted execution environments** (**TEEs**). On the other hand, software or algorithmic privacy such as ZKPs are also quite common in enterprise blockchains.
- **Governance layer**: This layer is responsible for providing the enterprise-grade access control mechanism that controls the consortium network membership. This can either be controlled via an on-chain permissioning system implemented in smart contracts, as part of the software client, or can be integrated with existing off-chain enterprise permissioning systems such as **Single Sign-On** (**SSO**) or **Active Directory** (**AD**).
- **Integration layer**: This layer provides APIs and a mechanism used to integrate with the legacy or existing back-office systems. This can be as simple as an RPC layer providing APIs over RPC or can constitute built-in connectors and plugins for integrating with the enterprise service bus.
- The integration layer is responsible for ensuring integration with back-office, legacy, and existing off-chain systems. It is not part of the core protocol but, as part of the holistic view of the blockchain end-to-end solution, this layer is vital for delivering business results. While there are many techniques available, a common integration framework used in the enterprise environment is **Apache Camel**. This can also be used in blockchain solutions as it comes with the Ethereum Web3J library component.



Apache Camel is an open source enterprise integration framework. It enables easy integration between various systems by utilizing enterprise integration patterns. It has hundreds of components for different systems such as databases, APIs, and MQs for easy integration between systems.

For example, the Web3J connector (Apache Camel Ethereum connector) is a feature-rich connector available in Apache Camel that enables integration between Ethereum chains and other systems. More details on the Web3J component are available here: `https://camel.apache.org/components/latest/web3j-component.html`.

The Apache Camel Ethereum connector works with Ethereum Geth nodes, Quorum, Parity, and Ganache. It supports JSON RPC API over HTTP and IPC with implementations for different blockchain operations such as `net`, `eth`, `shh`, and so on. It has support for Ethereum filters, **Ethereum Name Service** (**ENS**), and JSON RPC API, and is also a fully tested (unit and integration) solution.

A generic high-level design using Apache Camel is shown in the following diagram:

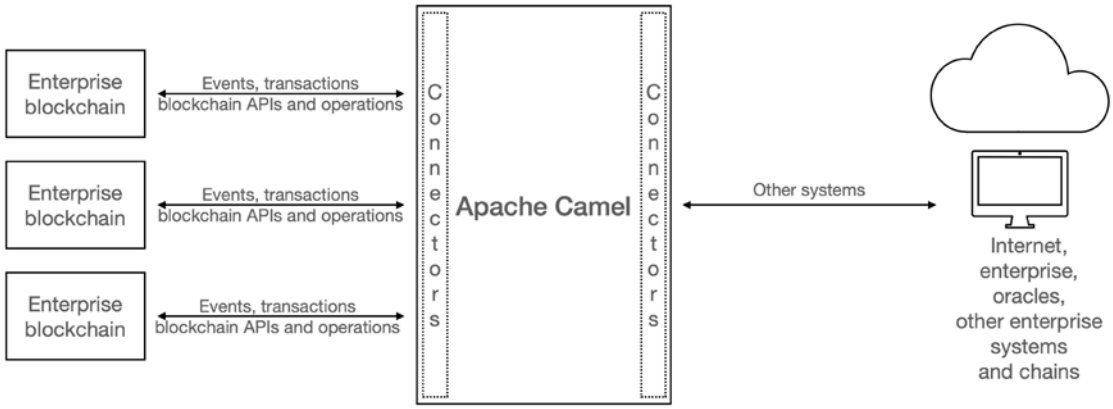


Figure 16.2: High-level design of Apache Camel, which enables blockchain integration



In the preceding diagram, we can see several enterprise blockchains connecting to Apache Camel using Apache Camel Ethereum connectors. Also, other systems connect to Apache Camel via their respective connectors. Apache Camel via connectors is responsible for integrating all these systems. For example, it is entirely possible to use Apache Camel to take data from enterprise blockchains and store it in traditional SQL databases in the enterprise via suitable connectors.

- **Application layer**: The application layer, as the name suggests, consists of **dApps**, **smart contracts**, tools, and other relevant software to support enterprise use cases.
- **Security, performance, scalability, monitoring**: On the left-hand side of the architecture diagram (*Figure 16.1*), security, scalability, performance, and monitoring are shown. As each layer in the enterprise blockchain benefits from (and indeed, requires) security, scalability, and performance, this layer is shown as encompassing all layers. Monitoring also plays a vital role in any enterprise solution. Without effective visualizations in such complex networks, it is almost impossible to keep track of everything. Therefore, this layer is shown as relevant to all enterprise blockchain layers.

Now that we understand the architecture of enterprise blockchains in general, let's dive a little bit deeper into the mechanics of designing enterprise blockchain solutions. In the next section, we'll see which tools and frameworks we can use to build enterprise solutions.

Designing enterprise blockchain solutions

An isolated blockchain in an enterprise is not sufficient to solve business problems. In addition to choosing a blockchain platform, there are some other factors to consider while introducing a blockchain in an enterprise. On top of the list of these factors is integration with the existing back-office and legacy systems.

Without any architecture framework, it becomes quite challenging to have a holistic view of an enterprise and see how all business processes fit together. As such, a question arises regarding whether we can leverage existing frameworks to address enterprise blockchain architectural needs. The answer is *yes*. There are already established and mature frameworks to facilitate enterprise architecture development.

The purpose of enterprise architecture frameworks is to enable an organization to execute its business strategy effectively. It allows an organization to see an organization from different perspectives, including business, information, process, and technology, and make effective business decisions to achieve business goals.

Let's now explore the popular enterprise architecture frameworks, **The Open Group Architecture Framework** (**TOGAF**) and Zachman Framework.

TOGAF

TOGAF stands for **The Open Group Architecture Framework**. It is developed by the Open Group.



The official TOGAF website can be found here: `https://www.opengroup.org/togaf`.

It is a framework that enables organizations to systematically design, plan, and implement enterprise solutions in businesses. It has four architectural domains:

- Business architecture domain: This domain defines the business strategy, organization structure, business processes, and governance of the organization.
- Data architecture domain: This domain describes the structures of an organization's data assets and relevant data management resources.
- Application architecture domain: This domain describes the enterprise applications, deployment blueprints, relationships, and interactions between applications, along with the relationships these applications have with business processes within the enterprise.
- Technology architecture domain: This domain defines the requirements of the technical architecture implementation of the enterprise applications. This includes the description of hardware, software, middleware, and network infrastructure required for the implementation of the enterprise applications.
- Now, re-examining each domain with blockchain in mind, we can include blockchain at each layer and make decisions such as business requirements, the application architecture, logical and physical data assets, and finally, the infrastructure required for implementing the enterprise blockchain solution.

After this basic introduction to TOGAF, let's dive a bit deeper to understand the method TOGAF uses to develop an IT architecture.

Architecture development method (ADM)

TOGAF ADM is a method for developing IT architecture that meets organizational business needs. It describes the process of moving from the foundational TOGAF architecture to an organization-specific architecture. This is an iterative process with continuous requirements management, which results in the development of an architecture that is specific to an organization and addresses specific needs. Once the architecture development is complete, the architecture can be published throughout the organization to develop a common understanding.

The ADM is a tested and repeatable process for developing enterprise architectures. The process follows the following steps: establish an architecture framework, develop architecture content, architecture governance, and implementation of architectures.

The ADM has nine phases and each phase can be further divided into multiple steps. The ADM model is shown here:

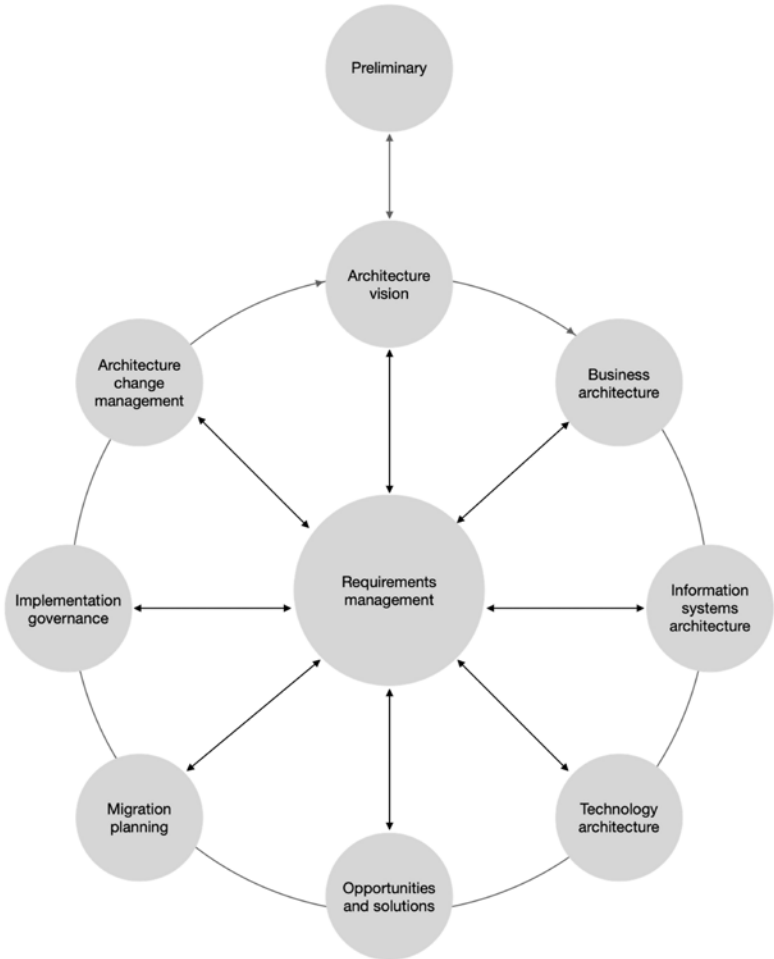


Figure 16.3: ADM model



The original diagram can be found at `https://pubs.opengroup.org/architecture/togaf9-doc/arch/chap05.html`.

Now, we'll describe each phase in more detail:

- Preliminary phase: This phase defines the groundwork of the architecture. It defines methodologies, architecture principles, the architecture scope, and assumptions about the architecture. It also defines the responsible parties for architecture delivery. From a blockchain perspective, we can define the overall blockchain strategy in this phase.
- Architecture vision: This phase validates business principles, goals, and business strategy. It defines key business requirements, as well as a high-level description of the business value expected from the architecture work. This phase also analyzes the impact of the new architecture on other processes.
- Business architecture: This phase proposes the baseline business architecture and develops a target business architecture, along with gap analysis. In a blockchain scenario, we can define the current state architecture and target state architecture at this stage.
- Information systems architecture: This phase defines the data architecture and application architecture. The data architecture includes building the baseline data architecture, the data architecture description, building data architecture models, doing impact analysis review reference models, and viewpoints. Application architecture defines the baseline application architecture, builds application architecture models, and proposes applications. Both of these activities also perform gap analysis to validate the architecture being developed, as well as to find any shortfalls between the baseline architecture and the target architecture. Similar to the business architecture, for an enterprise blockchain solution, we can define the current state architecture and target state architecture.
- Technology architecture: This phase is primarily concerned with reviewing the baseline business, data, and application architecture and building a baseline description of the current technology architecture in the enterprise. It also proposes the target technology architecture. In this phase, we can propose a target enterprise blockchain platform and the target solution.
- Opportunities and solutions: This phase performs evaluations and selects various proposed target architectures. Also, an implementation strategy and plan are proposed at this stage. We can apply this to blockchain in a similar fashion, by evaluating or selecting the target architecture encompassing enterprise blockchain solutions. When evaluating blockchain solutions, a number of features that we presented earlier in the comparison between enterprise blockchain and public blockchain can also be used; for example, confidentiality, scalability, and finality.
- Migration planning: This phase creates and finalizes the comprehensive implementation plan for migrating to the target architecture from the current architecture.

- Implementation governance: This phase deals with the implementation of the target architecture. A strategy to govern the overall deployment and migration is developed here. We can also perform blockchain solution testing and devise a deployment strategy during this phase.
- Architecture change management: This phase is responsible for creating change management guidelines and procedures for the newly implemented target architecture. From a blockchain perspective, this phase can provide change management procedures for the newly implemented enterprise blockchain solution.

With this, we've completed our introduction to TOGAF and explored the idea that enterprise blockchain solutions should be viewed through the lens of enterprise architecture.

The fundamental idea to understand here is that enterprise blockchain solutions are not merely a matter of quickly spinning up a network, creating a few smart contracts, creating a web frontend, and hoping that it will solve business problems. This setup may be useful as a PoC or for an incredibly simple use case but is certainly not an enterprise solution. We suggest that an enterprise blockchain solution must be looked at through the lens of the enterprise architecture and regarded as a full-grade enterprise solution. This is so that we can effectively achieve the business goals intended to be solved by enterprise blockchain solutions.

Next, let's explore how we could implement a blockchain business solution in an organization that has moved its operations to the cloud.

Blockchain in the cloud

Cloud computing provides excellent benefits to enterprises, including efficiency, cost reduction, scalability, high availability, and security. Cloud computing delivers computing services such as infrastructure, servers, databases, and software over the internet. There are different types of cloud services available; a standard comparison is made between **Infrastructure as a Service** (**IaaS**), **Platform as a Service** (**PaaS**), and **Software as a Service** (**SaaS**). A question arises here: where does blockchain fit in?

Blockchain as a Service, or **BaaS**, is an extension of SaaS, whereby a blockchain platform is implemented in the cloud for an organization. The organization manages its applications on the blockchain, and the rest of the software management, infrastructure management, and other aspects, such as security and operating systems, are managed by the cloud provider. This means that the blockchain's software and infrastructure are provided and maintained by the cloud provider. The customer or enterprise can focus on their business applications without worrying about other aspects of the infrastructure.

The following is a comparison of different approaches:

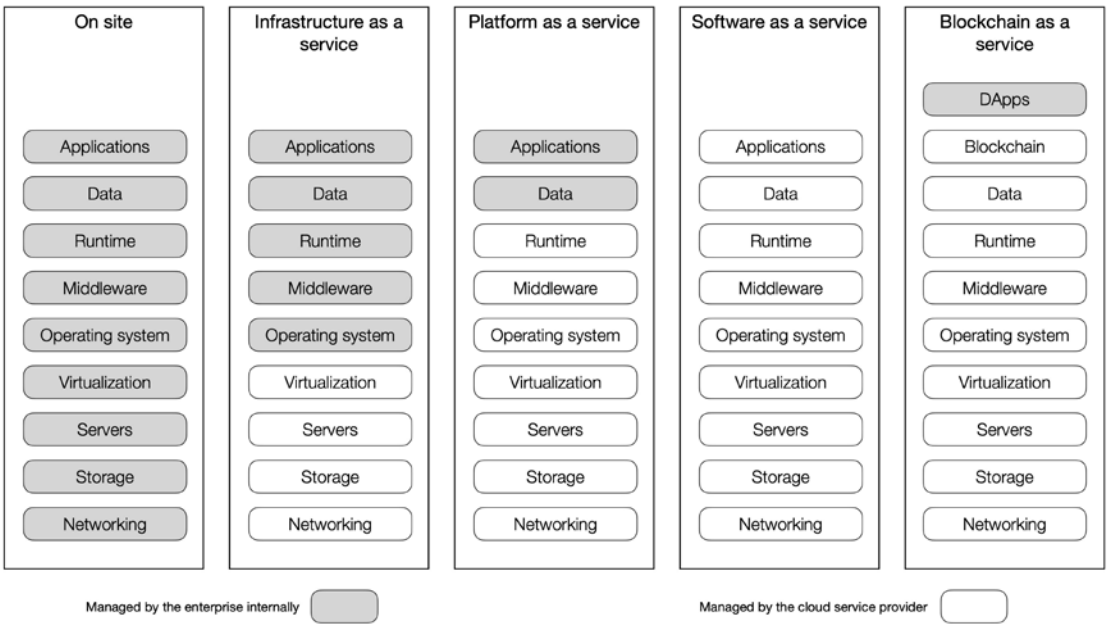


Figure 16.04: Cloud solutions

BaaS can be thought of as a type of SaaS, where the software is a blockchain. In this case, just like in SaaS, all services are externally managed. In other words, customers get a fully managed blockchain network on which they can build and manage their own dApps. Note that in the preceding diagram, under the **Blockchain as a Service** column, **Applications** have been replaced with **Blockchain**, as a differentiator between other cloud services and BaaS. Here, blockchain is the software (application) provided and managed by the cloud service provider. Also, note that dApps have been added on top, which are managed by the enterprise internally.

There are many BaaS providers. A few of them are listed as follows:

- **AWS**: `https://aws.amazon.com/blockchain/`
- **Azure**: `https://azure.microsoft.com/en-gb/solutions/blockchain/`
- **Oracle**: `https://www.oracle.com/uk/application-development/cloud-services/blockchain-platform/`
- **IBM**: `https://www.ibm.com/uk-en/cloud/blockchain-platform`

In the next section, we'll introduce some enterprise blockchain platforms.

Currently available enterprise blockchains

As this is a very ripe area for research, and indeed the market is very active in this space, there have been several enterprise blockchain solutions developed and made available in the last few years. Notably, the year 2019 has been called "the year of the enterprise blockchain" as many startups and enterprises focused on this area emerged.

In this section, we will not cover all these platforms in detail but will provide a brief introduction and links to more details on these chains:

- **Quorum** is an open source enterprise blockchain platform. It aims to support enterprise business needs and allows a business to achieve business goals (and unlock economic value) by leveraging blockchain technology. It addresses crucial enterprise requirements including privacy, performance, and enterprise permissioning. We will also explore Quorum in greater detail later in this chapter.
- **Fabric** is a **Hyperledger** project. It is an enterprise-grade distributed ledger that allows the development of blockchain solutions with a modular architecture. It has a permissioned architecture, support modularization, and pluggable consensus, and supports smart contracts. We discussed Hyperledger and other projects under it, such as **Sawtooth**, in detail in *Chapter 14, Hyperledger*.

Let's consider a comparison of leading enterprise blockchain platforms. We'll cover most of the desirable features of enterprise blockchains. This can be used as a reference for a quick comparison between these platforms. This comparison is based on the current implementations of these platforms available at the time of writing. However, as this is a very rapidly changing area, some features may change over time, or new features might be added or improved:

Feature	Quorum	Fabric	Corda
Target industry	Cross-industry	Cross-industry	Cross-industry
Performance (approximate transactions per second, or TPS)	700 (*)	560 (∞)	600 (@)
Consensus mechanism	Pluggable multiple Raft, IBFT, PoA	Pluggable Raft	Pluggable, notary-based
Tooling	Rich enterprise tooling	SDKs	Rich enterprise tooling
Smart contract language	Solidity	Golang	Kotlin/Java
Finality	Immediate	Immediate	Immediate
Privacy	Yes (restricted private transactions)	Yes (restricted private transactions)	Yes (restricted private transactions)

Access control	Enterprise-grade permissioning mechanism	Membership service providers/certificate based	Doorman service/KYC. Certificate-based
Implementation language	Golang, Java	Golang	Kotlin
Node membership	Smart contract and node software managed	Via membership service provider	Node software managed using configuration files, certificate authority controlled
Member identification	Public keys/ addresses	PKI-based via membership service provider, supports organization identity	PKI-based, support organization identity

Cryptography used	`SECP256K1`
	`AES`
	`CURVE25519 + XSALSA20 + POLY13050`
	`PBKDF2`
	`SCRYPT`
		`SECP256R1`	`ED255519`
			`SECP256R1`
			`RSA – PKCS1`

Smart contract runtime	EVM	Sandboxed in Docker containers	Deterministic JVM
Upgradeable smart contract	Possible with some patterns, not inherently supported	Allowed via upgrade transactions	Allowed via administrator privileges and auto-update allowed under administrative checks
Tokenization support	Flexible—inherited from public Ethereum standards	Programmable	Corda token SDK

* TPS results for Quorum are based on `https://arxiv.org/pdf/1809.03421.pdf`.

∞ TPS results for Hyperledger Fabric are based on `https://hyperledger.github.io/caliper-benchmarks/fabric/performance/2.0.0/nodeContract/nodeSDK/submit/empty-contract/`.

@ TPS results for Corda are based on `https://www.r3.com/corda-enterprise/`.

This comparison may also be used for the quick evaluation of these platforms and their suitability for an enterprise use case.

Now that we've examined some requirements, architectures, and use cases of enterprise blockchains, let's briefly describe some of the challenges that enterprise blockchains face. These can be attributed to public chains as well, but enterprise chains suffer from these specific issues too.

Enterprise blockchain challenges

While enterprise blockchains have addressed core enterprise requirements (privacy, performance, and governance) to some extent, there are still more challenges that need to be addressed. There is significant progress being made toward solving these issues. However, there is still a lot of work required to be done to adequately address these limitations. Some of these limitations are listed as follows.

Interoperability

Blockchain solutions are built by different development teams with different target requirements. As a result, there are now many different types of blockchains, ranging from cryptocurrency public blockchains to application-specific blockchains, which are developed for a single business application. Data exchange between these chains is a crucial concern. While blockchain networks continue to grow independently, integration and interoperability between these chains remain a big concern.

This problem also stems from a lack of standardization. If a standard specification is available, then all chains following that standard will become compatible automatically. However, what about those chains that are already deployed in production, including public chains? How do we achieve interoperability between them? Many business use cases have the requirement to get data from one organization to another or from one blockchain network to another. This is also true in the case of data exchange requirements between a public blockchain and a consortium network.

Lack of standardization

Lack of standardization is a commonly highlighted concern in enterprise blockchains and generally in the blockchain ecosystem. Traditional systems are mostly developed in line with standards defined by standards bodies, such as NIST FISMA standards for information security.

The **Federal Information Security Management Act** (**FISMA**) of 2002 is a United States federal law that requires all federal agencies to develop, document, and implement an agency-wide information security program. FISMA was amended in 2014 as the Federal Information Security Modernization Act. More on FISMA here: `https://www.congress.gov/bill/113th-congress/senate-bill/2521`.

Standards are also essential to achieving interoperability. Several initiatives have been taken to address the challenges mentioned here and also to standardize enterprise blockchain platforms. A leading organization in this field is EEA, a standards organization run by its members that aims to develop enterprise blockchain specifications.

EEA regularly releases technical specifications that can be downloaded at the link provided here:



`https://entethalliance.org/technical-specifications/`.

EEA's official website is `https://entethalliance.org`.

Compliance

There are various compliance requirements in almost all industries—especially finance, law, and health—where strict guidelines and rules have been mandated by regulatory authorities. Enterprise systems are expected to conform to these requirements. Blockchain initially started with a different focus of building an electronic cash system with an egalitarian philosophy. Still, surely enterprises have different visions and mindsets. Some of the regulations include GDPR, SOX, and PCI, which define different requirements for an enterprise to conform to. Also, compliance with technical standards such as the NIST FISMA standard for information security is necessary.

There are also jurisdiction issues. A distributed network with geographically dispersed locations may require different legal requirements to be met in various jurisdictions. For example, a specific type of cryptography may be allowed by law in the US but not in Cuba.

Business challenges

From a business perspective, cost, funding, and governance are some of the challenges that need to be met. If building and implementing an enterprise blockchain solution is prohibitively costly, then the business stakeholder might prefer traditional enterprise systems. Also, from an operational perspective, training staff to operate blockchain solutions might be a concern. We can categorize all blockchain-related costing, funding, and economics under an umbrella term that we call **Enterprise Blockonomics**. Enterprise Blockonomics (blockchain economics) can be defined as a study of cost and cost models in the context of enterprise blockchains.

With that, we have covered a lot of background material and developed an understanding of the enterprise blockchain requirements, architecture, and challenges. Let's now dive into some examples of enterprise blockchain platforms. First, we'll discuss VMware, and then we'll cover Quorum, a popular enterprise blockchain platform, in detail.

A recent addition to the enterprise blockchain world is **VMware Blockchain** (**VMBC**), which we introduce next.

VMware Blockchain

VMBC is an enterprise-grade blockchain platform that provides features suitable for enterprise use cases. It has introduced many innovations that result in better performance, flexibility, security, and scalability of blockchain dApps.

Components

The core components of VMware Blockchain are:

- Modular framework – a modular framework such that various smart contract execution engines (such as DAMLe or EVM) can be used. DAMLe, or the DAM execution engine, basically converts application layer actions into events on distributed ledger layer.
- Replica network – a set of replica nodes that present a single and consistent state machine to the client nodes.
- Replica nodes:
    - Replica nodes use a consensus mechanism to implement the SBFT state machine replication protocol, which forms a replica network composed of replica nodes.
    - Each replica node has a unique identifier, stores a replicated state machine, and makes use of the local key-value database (RocksDB). One replica is selected as a leader during a consensus cycle. Moreover, some replica nodes have collector roles. There are two types of collectors: C-collector and E-collector. C-collector collects commit messages and a combined signature is sent back to replicas to confirm the commit. E-collector collects all execution messages, and a combined signature is sent back to replicas and clients as a proof certificate of their request's execution.
- Group of client nodes:
    - Regular client nodes – provide an interface between applications and the replica network. Client nodes incorporate the BFT client and application interfaces such as the DAML ledger API server.
    - Full copy client node – receives all the updates in contrast to the regular client node, which receives a filtered view of the data it is permitted to access.
    - Duplicate client node – used for providing high availability and load balancing.
- VMWare Blockchain Orchestrator – a standalone virtual appliance that deploys on-premises VMware Blockchain nodes.
- There are also tools available for node deployment and monitoring and fleet management.
- An object store is used for archiving the data.

VMware, as with other blockchains, relies on a consensus protocol to ensure agreement among its participants, which we discuss next.

Consensus protocol

Consensus in VMware Blockchain is a PBFT variant, called SBFT. It is more scalable and decentralized as compared to previous PBFT variants. It guarantees safety, liveness, and linearity. Linearity, in simple words, means constant size messages, which result in a constant cost of operations on the chain. The leader election mechanism in SBFT ensures that a different group of collectors is chosen for each block, which spreads the load of primary leadership and collectors among all replicas. SBFT has reported roughly around 70 transactions per second on a geographically dispersed heavy-load network.

There are three core improvements including:

- Reduced communication because of using collectors, which enables linear communication and avoids quadratic all-to-all communication. Moreover, a single confirmation is sent to the clients.
- Implementation of faster and smaller sized (33 bytes) BLS signatures, which reduces proof length. Threshold signatures also reduce client communication from linear to constant.
- One round optimistic path consensus, which is important for reducing latency and results in faster agreement. There is also a faster optimistic agreement path for executions in normal cases.

With all the components we introduced so far, we can now describe the architecture of VMware Blockchain.

Architecture

We can visualize the high-level architecture of VMware Blockchain in the diagram shown below:

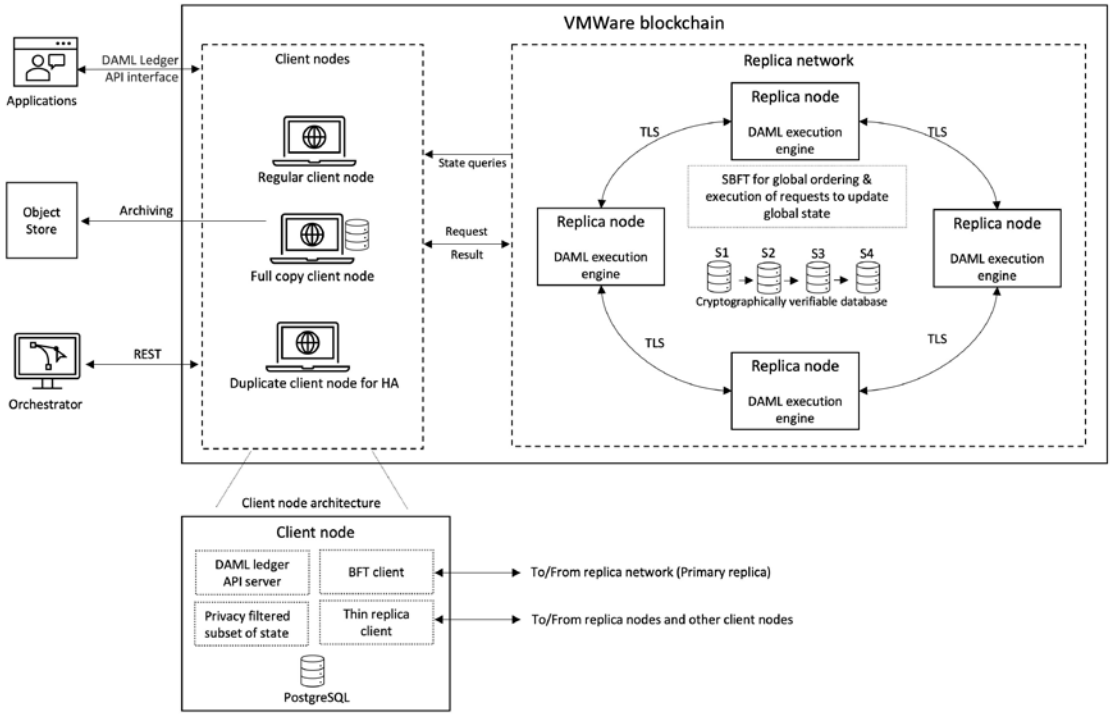


Figure 16.5: VMware Blockchain high-level architecture

The diagram shows a replica network composed of four nodes and a cryptographically verifiable database. The final state is achieved after running the SBFT consensus algorithm. The client nodes communicate with the replica network via TLS-secured communication channels. Client nodes (shown on the left side of the diagram) can make state queries and request information form the primary replica node in the replica network.

A client node consists of a DAML ledger API server and a privacy-filtered subset of state, as a node only sees what it is authorized to see. The node also consists of a BFT client that speaks to the primary replica in the replica network, and a thin replica client, which runs the thin replica protocol.

The client node accepts requests coming from applications and sends them to replicas, receives results from the replica network, and sends them back to the applications. The dApp (application) sends commands to the DAML ledger API server component of the client node. The DAML API components forward the command to a pool of BFT clients, which sends the requests to the replica network for execution and waits to collect the results from the replica nodes. The BFT client collects and ensures the validity of the reply (results) it receives from the replica network and forwards the results back to the DAML ledger API. The DAML ledger API receives the results of the execution using the PostgreSQL notification. Using the thin replica protocol, a replica node sends the results to other client nodes that are subscribed to receive these updates. The client node using the thin replica protocol receives the update and writes it into the local client PostgreSQL database. A notification indicating the update in the local DB is sent to the DAML ledger API. The Postgres SQL DB stores a materialized view of the replica network, which is the data that the specific client node is permitted to view in a processed format. Each replica node contains a **replicated state machine** (**RSM**), authenticated local key-value store (RocksDB), and DAMLe – DAML execution engine. The database is cryptographically verifiable. It supports checkpoints for archiving. Request ordering is achieved via consensus.

Other components of the blockchain include applications that talk to the client nodes via the DAML ledger API interface. An object store also exists, which is used for archiving data via a full copy client node. Finally, an orchestrator appliance is used to create, deploy, and manage VMware Blockchain via a REST interface.

VMBC supports DAML and EVM. Parallel execution is supported by the DAML engine.



More information on this blockchain is available here: `https://www.vmware.com/uk/products/blockchain.html`

DAML is a statically typed functional smart contract language – a DSL, influenced by Haskell. It is a concise, English-like, easy-to-understand and write, business-oriented language. It is designed to build a multi-party composable application. It is also called "Enterprise Haskell." It is basically Haskell, with added primitives for smart contracts, authorization rules, and privacy. DAML aims to solve portability, interoperability, and privacy problems in the blockchain. It focuses on data privacy and the authorization of distributed applications. In this language, parties are first-class primitives. It is built with privacy in mind and enables tracking and authorization at each workflow step – i.e., each smart contract has its own defined privacy. By abstracting data privacy and authorization, DAML allows developers to focus on workflow logic instead of worrying about concrete cryptographic primitives. It is also storage layer agnostic, portable, and interoperable, which means that DAML code written for one platform will work for other platforms too, without any change. For example, the same DAML code will work for both Hyperledger and Ethereum, and even a SQL database without any change.



You can find out more about DAML here: `https://www.digitalasset.com/developers`. Note that DAML is developed by Digital Asset Holdings.

VMware Blockchain for Ethereum

Public Ethereum is unsuitable for enterprise use cases due to scalability, strict governance, enterprise-grade operational support, and privacy requirements. Nevertheless, it is the most used platform for smart contract development due to its mature development ecosystem. It can become suitable for enterprise use cases if we can address the limitations in the public Ethereum chain. VMware Blockchain for Ethereum addresses these issues and provides a platform that allows easy development, efficient operations, and scalable Ethereum blockchain networks. Ethereum support can be provided in addition to DAML support already available in VMware Blockchain. VMware Blockchain for Ethereum is a high-performance, high-throughput platform with ZKP-based programmable privacy capabilities, enterprise-grade operations, and governance features. Developers can use familiar tools like MetaMask, Remix, Hardhat, and Truffle to develop on this platform. Moreover, it supports standard Ethereum APIs providing a seamless developer experience.

In summary, we can say that with all the innovations, improvements, and smart choices of enterprise-friendly features, VMware Blockchain is a worthy choice for enterprise use cases.

Quorum

Quorum is an open source enterprise blockchain platform. It is a lightweight fork of **Ethereum**. Quorum not only benefits from the innovation and research being done in the upstream public Ethereum (Geth) project but also many excellent enterprise features have been introduced in Quorum. These enterprise features primarily focus on providing enterprise-grade privacy, performance, and permissioning (access control).

First, let's take a look at Quorum's architecture.

Architecture

Quorum addresses three fundamental issues in public blockchains, which makes it an excellent choice for enterprise use cases:

- Privacy
- Performance
- Enterprise governance

At a high level, the Quorum architecture consists of nodes and their associated privacy managers. The Quorum architecture can be visualized using the following diagram:

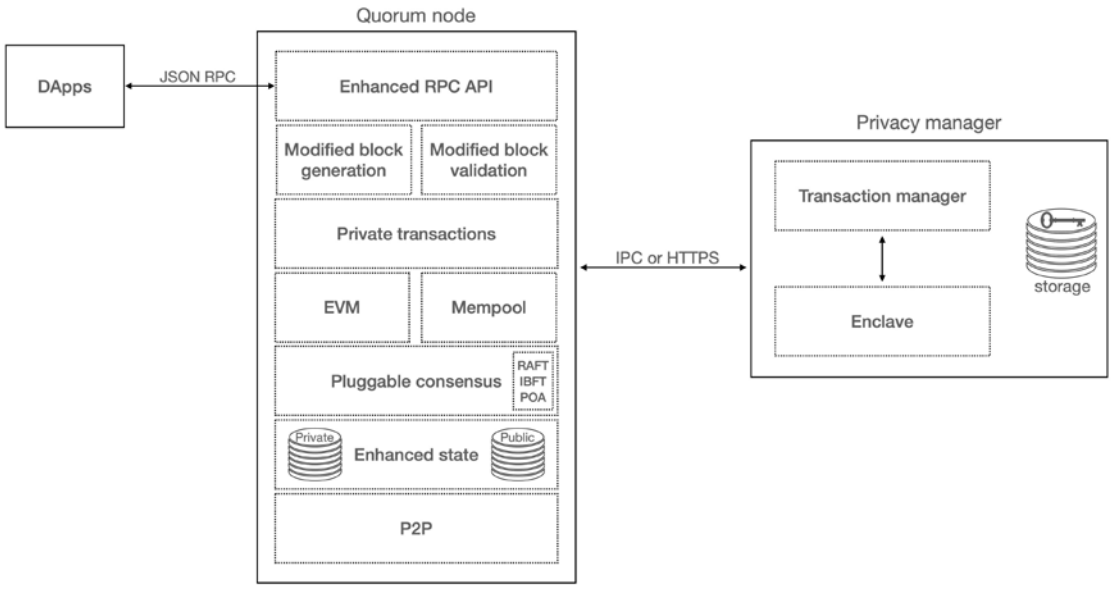


Figure 16.6: Quorum architecture

Nodes

The Quorum node is a modified and enhanced version of public `geth` that supports private transactions. Quorum nodes communicate via HTTPS with privacy managers, which, in turn, are responsible for providing privacy by storing the payload in encrypted format in their local storage. These Quorum nodes maintain the public and private states separately.

There are several changes that have been made to the Quorum client to make it suitable for enterprise use cases. We'll provide an overview of each of those changes here:

- **Enhanced P2P**: This layer is modified to allow connections only between authorized nodes.
- **Enhanced state (private and public)**: In addition to the public state trie, there is an additional Merkle Patricia state trie for the private state. In Ethereum, a modified **Merkle Patricia trie** (**MPT**) is a data structure that is used to provide a cryptographically authenticated key-value store. More technical details on this are available here: `https://eth.wiki/en/fundamentals/patricia-tree`.
- **Pluggable consensus**: Quorum supports multiple consensus algorithms such as IBFT, **Proof of Authority** (**PoA**), and Raft. Quorum, being an enterprise blockchain, does not need a PoW type of consensus.
- **No transaction fees**: The pricing of gas has been set to zero as there is no need for transaction fees in consortium networks.

- **Private transactions**: Private transactions are identified using `37` or `38` as the `V` value in the transaction. The transaction creation mechanism is modified to enable replacing transaction data (input) with the hash of the encrypted payload.
- Remember that in standard Ethereum, as we learned in *Chapter 9*, *Ethereum Architecture*, a transaction's `V` value can be either `27` or `28`, but in Quorum, that has been changed to `37` or `38` to indicate private transactions. Public transactions are still identifiable by a `27` or `28` `V` value, whereas private transactions are identifiable with a `V` value of either `37` or `38`.
- **Modified block generation mechanism:** The logic for generating blocks is modified with a new check for the global public state root instead of the global state root.
- **Modified block validation mechanism**: Block validation logic is modified to replace the global state root check with a global state root check *for public state*—otherwise known as a global public state root.
- **Enhanced RPC API:** Quorum supports additional RPC APIs that help us interact with the enhanced enterprise features of Quorum, such as permissioning and consensus mechanisms.

Now, we'll discuss **privacy manager**, which consists of two components: a transaction manager and an enclave.

Privacy manager

Privacy manager is an off-chain mechanism that provides transaction confidentiality. It is paired on a one-to-one basis with a Quorum node. It allows Quorum nodes to share the transaction payload securely between authorized participants.

The **transaction manager** is responsible for:

- Storing encrypted transaction payloads
- Managing access to encrypted transaction payloads
- Propagating encrypted payloads to other transaction managers on the network
- Discovering other transaction managers on the network

Quorum has developed two transaction managers. Initially, a Haskell implementation called **Constellation** was made available. However, it is no longer actively developed in favor of a more feature-rich, Java-based privacy manager called **Tessera**. Tessera is developed in Java. It is used for the storage, encryption, decryption, and propagation of private transaction data.

An **enclave** is an isolated and independent element that provides cryptography services for transaction payload encryption and decryption. It can only be associated on a one-to-one basis with its own transaction manager.

In order to achieve all the desired privacy features, several cryptographic protocols and primitives have been used in the Quorum platform. We'll provide a quick overview of these in the next section.

Cryptography

Quorum inherits its cryptography stack from public **Ethereum.** It includes standard ECDSA signatures, AES CTR cryptography for wallets and DEVP2P, and Keccak256 hash functions. Privacy manager makes use of Curve25519, **Elliptic-curve Diffie-Hellman** (**ECDH**), poly1305, and Xsalsa20 cryptographic operations to provide confidentiality guarantees required by private transactions in Quorum.

Now, we'll explore how each of the main facets—privacy (confidentiality), enterprise-grade membership control (access control, and the permissioning mechanism), and performance—are achieved in Quorum.

Privacy

Quorum supports both private and public transactions. For private transactions, it uses a mechanism called private transaction manager, which is an off-chain component to facilitate the confidentiality of transactions. We will now describe how private transactions work in Quorum.

Suppose there are three parties: *A*, *B*, and *C*. Parties *A* and *B* are privy to a transaction, while party *C* is not. We'll now explore how the private transaction is generated and flows between parties and is propagated on the network while maintaining confidentiality. Let's call this transaction *transaction AB*:

1. To begin private *transaction AB*, party *A* creates a transaction and signs it before sending it to their Quorum node, node *A*. The transaction is composed of a transaction payload and the public key of the intended recipient. This list of public keys of intended recipients is maintained in the `PrivateFor` list. It can be a single public key or multiple keys, depending on the requirements.
2. There are two signing mechanisms in Quorum. For public transactions, an EIP55-based mechanism is used, and for private transactions, an Ethereum Homestead signing mechanism is used. Transactions in Quorum can also be signed independently, without using Quorum's signing mechanism.
3. Quorum node *A* sends the transaction to transaction manager *A* for processing.
4. Transaction manager *A* makes an encryption request to its enclave, to encrypt the transaction payload.
5. Party *A*'s enclave encrypts the transaction payload and sends it to transaction manager *A*.
6. Transaction manager *A* stores the transaction payload and sends it to other transaction managers, in this case, *B*.
7. A transaction hash is returned to Quorum node *A* by transaction manager *A*, which replaces the original transaction payload with the hash and changes the value `V` of the transaction to `37` or `38` to indicate that the transaction is private.
8. The transaction propagates to other nodes via the normal Ethereum P2P protocol.
9. The block that contains *transaction AB* finalizes and propagates on the network between all nodes (*A*, *B*, and *C*).
10. When the block containing the transaction is received by the Quorum nodes, they recognize the transaction as private because the `V` value of the transaction is either `37` or `38`.

If the transaction is identified as private, the Quorum nodes will query their respective transaction managers to figure out if they are party to the transaction or not. This is achieved by finding out if there is an entry available in the database with the transaction hash. Here, parties *A* and *B* will have the transaction hash stored in the transaction manager, whereas party *C* will not.

11. The transaction managers of parties *A* and *B* make a transaction payload decryption request to their respective enclaves.
12. The enclaves of parties *A* and *B* decrypt the private transactions.
13. When the transaction managers receive the decrypted payload back from their enclaves, they send this data to their respective Quorum nodes. In our example, the transaction managers of parties *A* and *B* will send the transaction payload to their respective Quorum nodes, *A* and *B*. The Quorum nodes will execute the transaction (contract) via the EVM and update their private state database accordingly. The transaction manager of party *C* will return a message to its Quorum node, indicating that it is not privy to (or a recipient of) the transaction. This means that the Quorum node of party *C* will simply ignore the transaction.

This process can be visualized using the following diagram:

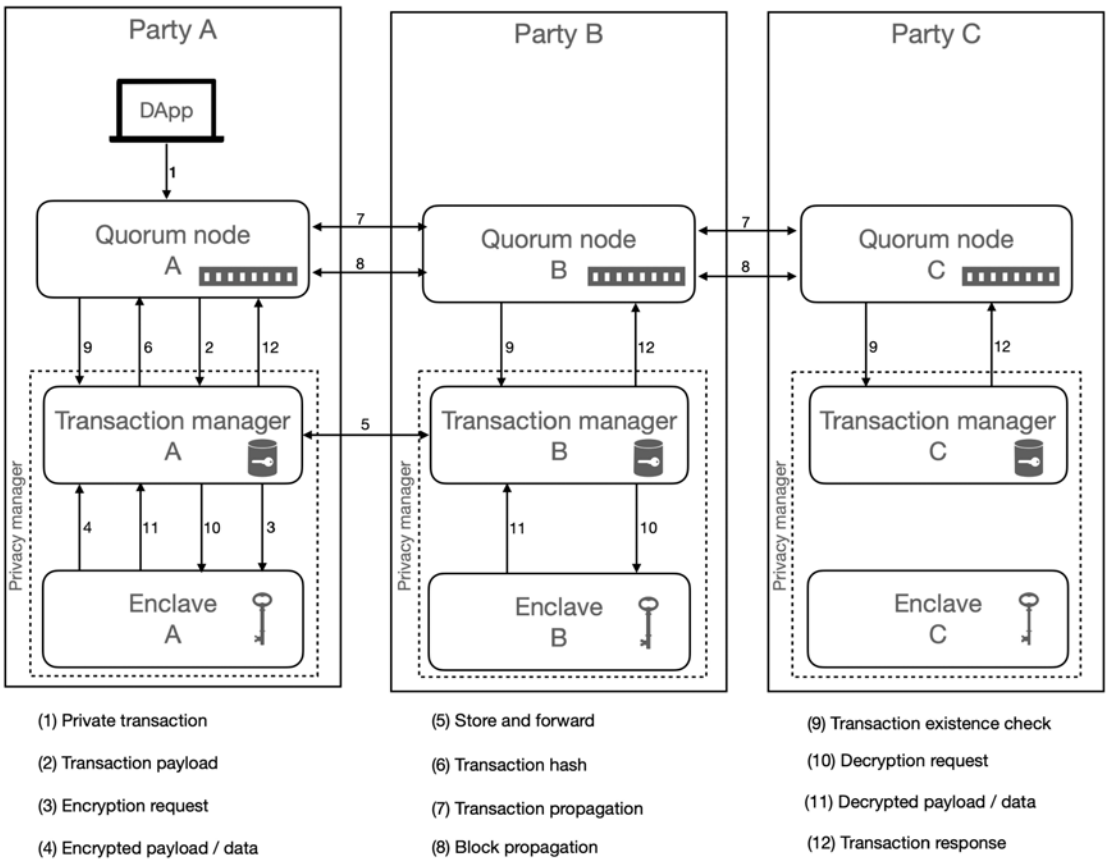


Figure 16.7: Quorum private transaction flow

Let's now expand on *Step 4* from the *achieving privacy* process, enclave encryption, as it consists of several sub-steps that are performed within the enclave.

Enclave encryption

When an **enclave** receives a transaction, it performs several steps to encrypt the transaction. All these steps are shown here and are performed by the enclave as part of the transaction payload encryption process:

1. Generates a symmetric key for the transaction.
2. Generates two random nonces.
3. Encrypts the transaction payload and one of the nonces with the symmetric key generated in the first step.
4. Encrypts the transaction key for transactions for each recipient. For this purpose, the following steps are performed:
    - Generates a shared symmetric key by utilizing ECDH. ECDH will use the sender's private key and the receiver's public key.
    - Encrypts the symmetric key for transaction for each receiver separately using the newly generated shared symmetric key and the second nonce.
    - This step will be repeated for each recipient.
5. Finally, the transaction manager receives the encrypted transaction payload, all the encrypted symmetric keys for transactions, both nonces, and the public keys of the senders and receivers.

Now, let's see how the transaction manager stores and propagates transactions to the other transaction manager. We'll now expand on *Step 5* of the *achieving privacy* process mentioned previously.

Transaction propagation to transaction managers

A **transaction manager** performs the following steps after it receives an encrypted response back from the **enclave**:

1. It generates the SHA3-512 hash of the encrypted payload received from the enclave.
2. It stores the encrypted payload, the hash and the encrypted symmetric key for the transaction, both nonces, and the public keys of the sender and receiver.

3. It sends, using HTTPS, the encrypted payload, the hash, and the encrypted symmetric key for the transaction using transaction manager *B*'s public key to transaction manager *B*.
4. Transaction manager *A* awaits an acknowledge message from transaction manager *B*. If an acknowledgment is not received, the transaction will not be propagated to the network.

Let's now discover how the decryption process works in enclaves. This corresponds to *Step 11* of the *achieving privacy* process.

Enclave decryption

The transaction payload decryption process starts when a transaction payload decryption request is made to an **enclave**. The enclave performs the following steps to decrypt a payload:

1. Derives the shared symmetric key. The key derivation process works as follows:
    - Party *A*, being the sender of the transaction, derives the shared symmetric key using its private key and the receiver's public key.
    - Party *B*, being the recipient of the transaction, derives the shared symmetric key using its private key and the sender's public key.
2. Decrypts the symmetric key for transactions with the shared symmetric key, along with the encrypted payload and nonce, fetched from the database.
3. Decrypts the transaction data with the symmetric key for transactions and the encrypted data and nonce fetched from the database.
4. Finally, the decrypted private transaction data is sent to the transaction manager.

The enclave, being separated from the transaction manager, addresses the **separation of concerns** and allows parallelization in order to improve performance. Separation of concerns is a form of abstraction that allows the segregation of different components of computer software into separate and distinct units so that each unit addresses a separate concern. This simplifies design and helps to modularize the software.

The overall process (enclave encryption, transaction manager storage and propagation to other nodes, and enclave decryption), corresponding to *Steps 4*, *5*, and *11* in *Figure 16.7*, can be visualized in the following diagram:

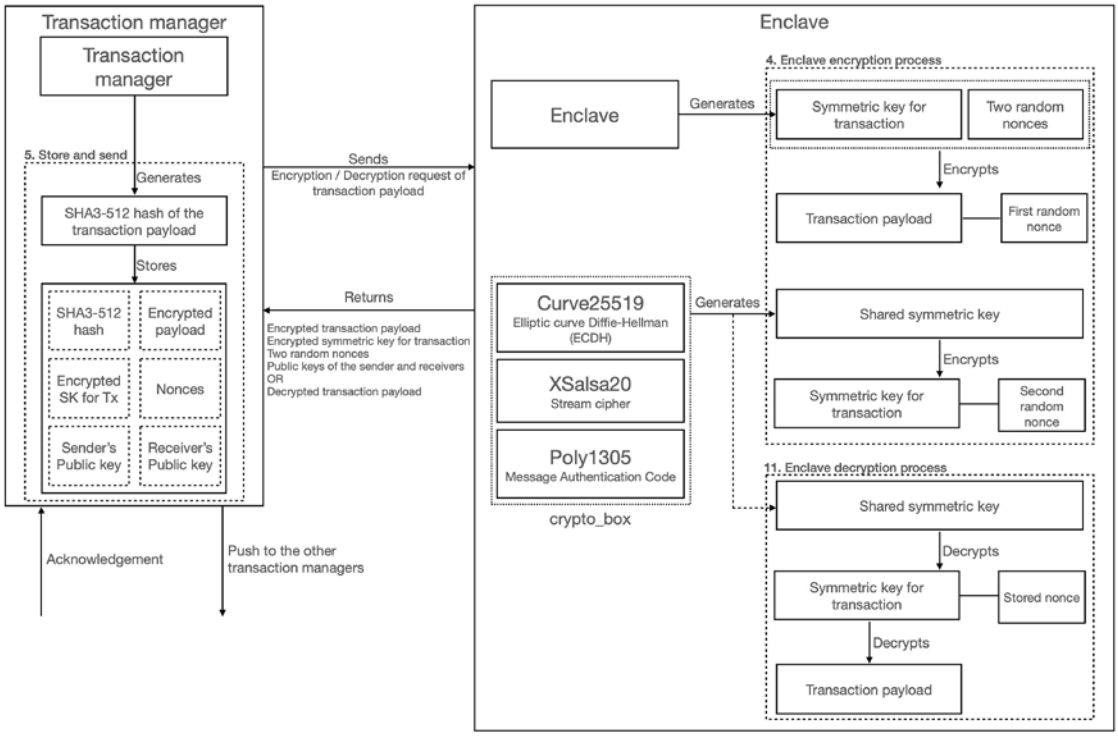


Figure 16.8: Enclave and transaction manager high-level architecture

We've covered quite a lot of theory here, but with this, we have now completed our introduction to Quorum private transactions. Next, we'll explore how access control works in Quorum.

Access control with permissioning

As an enterprise blockchain platform, **Quorum** comes with an enterprise-grade access control mechanism that manages network membership for consortium members. It is implemented in smart contracts written in the **Solidity** language. It supports many features that a typical enterprise-grade permissioning mechanism would. It includes features to support the management of nodes, account-level permissioning, and support for voting-driven decision-making for permissioning actions.

The overall architecture of Quorum's permissioning mechanism can be visualized using the following diagram:

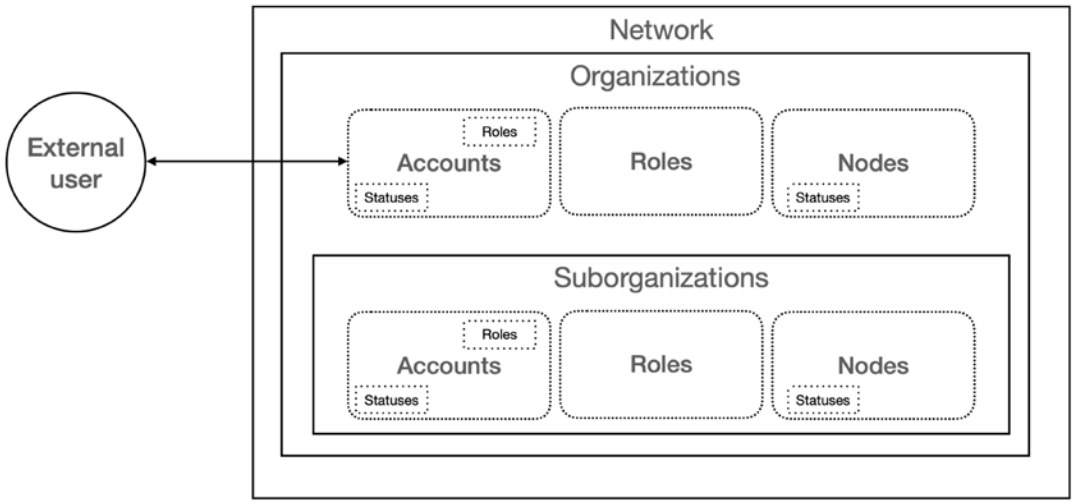


Figure 16.9: Quorum permissioning mechanism

This permissioning mechanism is inspired by a widely accepted industry standard called **Role-Based Access Control** (**RBAC**). It is standardized by the **American National Standards Institute** (**ANSI**) and is used in most enterprise-grade software, such as operating systems and cloud systems.

The RBAC standard is available here:

`https://standards.incits.org/apps/group_public/project/details.php?project_id=1658`.

The features that Quorum's permission mechanism supports are listed here:

- Role management
- Assignment of roles to accounts (subjects)
- Permissions management
- Node management

A role can be defined as a named job function in an organization.

At a fundamental level, the permissions management model supports granular control over:

- Funds transfer
- Smart contract creation
- Smart contract execution

It controls which account can transfer funds, create smart contracts, or execute smart contracts. These permissions are used to create roles, which are then assigned to accounts to enforce permissions.

Permissions are divided into two broad categories: *account permissions*, which control what functions an account can perform, and *node permissions*, which control the membership of nodes on the network.

Accounts can be assigned with different roles and statuses, whereas nodes can be assigned with statuses. Statuses for accounts include several statuses such as active, inactive, suspended, and blacklisted. Nodes can be in approved, deactivated, pending approval, or blacklisted states. The access types can be read-only, transact, contract deploy, and full access.

Now that we have covered how enterprise-grade permissioning is implemented in Quorum, let's now move on to another important aspect of enterprise blockchain: performance.

Performance

Performance is a vast subject and can mean different things in different scenarios. Mostly, it is merely a measure of how many transactions a system can perform in a given interval, usually measured in **transactions per second** (**TPS**). It can also mean the overall throughput of the system, the quality of the service, and the ability to scale. From a business perspective, it could mean business performance. However, here, we will only deal with the TPS scenario and present some studies that have been performed to evaluate the performance of the Quorum blockchain.

The core idea behind performance enhancement in the Quorum blockchain is the usage of deterministic and fast consensus algorithms. Quorum supports various consensus mechanisms suitable for consortium networks. As compared to other public blockchain consensus mechanisms, the algorithms used in Quorum provide better performance.

Several evaluations have been made and reported regarding Quorum's performance. These studies have reported Quorum's performance to be as high as 2,500 TPS.



These studies are available here:

`https://scandiweb.com/blog/jpmorgan-s-quorum-blockchain-performance-testing`

`https://arxiv.org/pdf/1809.03421.pdf`

Now, we'll introduce the consensus mechanisms supported by Quorum.

Pluggable consensus

Quorum supports several consensus algorithms that can be plugged in, based on use case requirements. For example, if the requirement is simple crash-tolerance, users can choose Raft. If more security is required and Byzantine faults need to be handled, then users can choose IBFT. We have listed the consensus mechanisms available in Quorum here:

- **Raft**: A crash fault-tolerant consensus algorithm
- **IBFT**: A PBFT-inspired BFT algorithm
- **Clique:** PoA, inherited from public Ethereum

We discussed all these algorithms in detail in *Chapter 5*, *Consensus Algorithms*. You can review these topics in detail there.

Now that we have gone through several features and have built a theoretical understanding of various features of the Quorum enterprise blockchain, let's now see how we can set up a Quorum network.

Setting up a Quorum network with IBFT

In this section, we will set up a **Quorum** network with four nodes, demonstrating how an IBFT network can be set up and carry out private transactions in Quorum. These steps can be performed manually, or we can use a tool called **Quorum Wizard** to spin up a Quorum network quickly and easily.

For details on the manual setup, you can refer to the detailed Quorum setup instruction on Quorum's official website here: `https://docs.goquorum.consensys.net/tutorials/quorum-dev-quickstart`. We will use Quorum Wizard to set up our network.

Installing and running Quorum Wizard

Quorum Wizard is a command-line tool written in **JavaScript** that enables users to create a local network of Quorum nodes quickly. It runs as an `npm` module and, as such, requires Node.js and npm to run.



Node.js can be installed from this address: `https://nodejs.org/en/`.

To check the current version installed, issue the following command:

```
$ npm -v
```

This command will produce the following output, indicating the installed version of the node package manager, `npm`:

```
6.14.4
```

This command will show the currently installed version of `node`:

```
$ node -v
```

This will produce an output specifying the version of `node` you have installed:

```
v12.18.0
```

If the expected version number is displayed after executing both of these commands, then we are all set to install Quorum Wizard.

Quorum Wizard can be installed using `npm install`. It's an excellent tool and cuts installation time down from hours to minutes:

```
$ npm install -g quorum-wizard
```

This will show an output similar to the one shown here:

```
/usr/local/bin/quorum-wizard -> /usr/local/lib/node_modules/quorum-wizard/
build/index.js
+ quorum-wizard@1.1.0
added 155 packages from 143 contributors in 27.897s
```

Once installed, we can run it with the following procedure.

We will create a 4-node IBFT network in this example. Simply run Quorum Wizard and follow the steps as prompted:

```
$ quorum-wizard
```

Running Quorum Wizard will show the following text:

```
Welcome to Quorum Wizard!

This tool allows you to easily create bash, docker, and kubernetes files to start up a quorum network.
You can control consensus, privacy, network details and more for a customized setup.
Additionally you can choose to deploy our chain explorer, Cakeshop, to easily view and monitor your network.

We have 3 options to help you start exploring Quorum:

  1.  Quickstart - our 1 click option to create a 3 node raft network with tessera and cakeshop

  2.  Simple Network - using pregenerated keys from quorum 7nodes example,
      this option allows you to choose the number of nodes (7 max), consensus mechanism, transaction manager, and the option to deploy cakeshop

  3.  Custom Network - In addition to the options available in #2, this selection allows for further customization of your network.
      Choose to generate keys, customize ports for both bash and docker, or change the network id

Quorum Wizard will generate your startup files and everything required to bring up your network.
All you need to do is go to the specified location and run ./start.sh

  (Use arrow Keys)
> Quickstart (3-node raft network with tessera and cakeshop)
  Simple Network
  Custom Network
  Exit
```

Figure 16.10: Quorum Wizard options

Select **Simple Network** and select all the options by using the up and down arrow keys. Press *Enter* after a choice is made, which will move us on to the next question in Wizard:

```
Simple Network
? Would you like to generate bash scripts, a docker-compose file, or a
kubernetes config to bring up your network? bash
? Select your consensus mode - istanbul is a pbft inspired algorithm with
transaction finality while raft provides faster blocktimes, transaction
finality and on-demand block creation istanbul
? Input the number of nodes (2-7) you would like in your network - a minimum of
4 is recommended 4
? Which version of Quorum would you like to use? Quorum 2.6.0
```

```
? Choose a version of tessera if you would like to use private transactions in
your network, otherwise choose "none" Tessera 0.10.2
? Do you want to run Cakeshop (our chain explorer) with your network? Yes
? What would you like to call this network? 4-nodes-istanbul-tessera-bash
```

Once all the options have been selected, as shown in the preceding code snippet, the tool will download and install dependencies. This process will produce an output similar to the one shown here. For brevity, the full output is not shown:

```
Downloading dependencies...
.
.
Building network directory...
Generating network resources locally...
Building qdata directory...
Writing start script...
Initializing quorum...
Done
```

Finally, the output shown here is produced, indicating the success of the operation:

```
Quorum network created
Run the following commands to start your network:
cd network/4-nodes-istanbul-tessera-bash
./start.sh
A sample simpleStorage contract is provided to deploy to your network
To use run ./runscript.sh public-contract.js from the network folder
A private simpleStorage contract was created with privateFor set to use Node
2's public key: QfeDAys9MPDs2XHExtc84jKGHxZg/aj52DTh0vtA3Xc=
To use run ./runscript private-contract.js from the network folder
```

Now, change to the `4-nodes-instabul-tessera-bash` directory, as shown here, and start the network:

```
$ cd network/4-nodes-istanbul-tessera-bash
$./start.sh
```

This will produce an output similar to the one shown here. Some of the output has been truncated for brevity:

```
Starting Quorum network...
.
.
.
All Tessera nodes started
Starting Quorum nodes
Starting Cakeshop
```

```
Cakeshop started at http://localhost:8999
Successfully started Quorum network.
```

And that's it! We now have a Quorum network with four nodes running on the IBFT protocol.

Now, let's do an experiment to see how private transactions can be created and run on the Quorum network. Quorum Wizard has already created the relevant scripts.

Running a private transaction

In this section, we will explore how private transactions can be created and executed on a Quorum network:

1. From within the `4-nodes-instabul-tessera-bash` directory, run the command shown here:

```
$ 4-nodes-istanbul-tessera-bash ./runscript.sh private_contract.js
```

2. Once the preceding command runs, it will produce an output similar to the one shown here, indicating that the transaction has been sent and that a transaction hash `0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4` has been produced:

```
Contract transaction send: TransactionHash:
0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4
waiting to be mined...
True
```

Here, we have basically executed a private transaction and deployed a private smart contract that is only visible to Node 1 and Node 2. Nodes 3 and 4 are not privy to this transaction and they shouldn't be able to view the contents of the code or the values (state) of the contract. We will demonstrate in this example that this is indeed the case.

Attaching Geth to nodes

Next, we'll attach Geth to each node, as shown using the commands in the following sections, and run the console commands as shown in the examples for each node.

Node 1: In order to interact with `geth` via IPC, enter the following command in the operating system's Terminal. This will open the Geth JavaScript console, where users can interact with the blockchain using several methods exposed by Geth:

```
$ geth attach qdata/dd1/geth.ipc
```

Once the `geth` console is open, enter the following command, which declares a variable named `abi`:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
```

```
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256"}],
"payable":false,"type":"constructor"}];
```

Now, create a new contract instance using the command shown here. We can use this object to access all methods and events of the contract:

```
> var simpleContract = eth.contract(abi)
```

Next, we use the instance object created previously to get a complete abstraction of our contract by using the command shown here:

```
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

Now, we can use the `simple` object to access all the methods in our smart contract. In the following example, we're using the `get()` method to return the stored value in our contract:

```
> simple.get()
42
```

This command outputs `42`, which is our stored value.

Note that after each statement we enter in the `geth` console, except the last statement, `simple.get()`, we also see that a message of `Undefined` is displayed. Simply ignore this; it is just a standard way in JavaScript of indicating uninitialized variables, non-existent object properties, or other similar scenarios.

Node 2: Similar to Node 1, open the `geth` console for Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

As we did for Node 1, in the `geth` console, enter the following commands:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256
"}],"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
42
```

Much like Node 1, the final command outputs `42`, which is our stored value.

Node 3: Again, similar to Node 1, open the `geth` console for Node 3 using the command shown here:

```
$ geth attach qdata/dd3/geth.ipc
```

In the geth console, enter the following:

```
> var abi =
[{"constant":true,"inputs":[],"name":"storedData","outputs":[{"name":"",
"type":"uint256"}],"payable":false,"type":"function"},{"constant":false,
"inputs":[{"name":"x","type":"uint256"}],"name":"set","outputs":[],
"payable":false,"type":"function"},{"constant":true,"inputs":[],"name":"get",
"outputs":[{"name":"retVal","type":"uint256"}],"payable":false,
"type":"function"},{"inputs":[{"name":"initVal","type":"uint256 "}],
"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
0
```

Note that the value returned is `0`, which indicates that Node 3 cannot access the value of `42`, which is expected as Node 3 is not privy to the transaction.

Node 4: Finally, we open the geth console for Node 4 using the command shown here:

```
$ geth attach qdata/dd4/geth.ipc
```

In the console, enter the following command as we did for nodes 1, 2, and 3:

```
> var abi = [{"constant":true,"inputs":[],"name":"storedData",
"outputs":[{"name":"","type":"uint256"}],"payable":false,
"type":"function"},{"constant":false,"inputs":[{"name":"x","type":"uint256"}],
"name":"set","outputs":[],"payable":false,"type":"function"},{"constant":true,
"inputs":[],"name":"get","outputs":[{"name":"retVal","type":"uint256"}],
"payable":false,"type":"function"},{"inputs":[{"name":"initVal",
"type":"uint256 "}],"payable":false,"type":"constructor"}];
> var simpleContract = eth.contract(abi)
> var simple = simpleContract.at("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
> simple.get()
0
```

As expected, Nodes 3 and 4, which are not privy to the transaction, will see `0` as the transaction value. On the other hand, Nodes 1 and 2 are able to see the transaction value, which is `42`.

Remember, as we discussed earlier in this chapter, we need to monitor and visualize enterprise blockchain networks. Cakeshop fulfills that need.

Viewing the transaction in Cakeshop

Cakeshop is a powerful visualization and administration tool with different features such as node management, block explorer, and contract management. Cakeshop is installed as part of the Quorum Wizard network creation process. Once the network runs successfully, we can browse to `http://localhost:8999`, where Cakeshop runs.

Cakeshop is shown in the following screenshot:

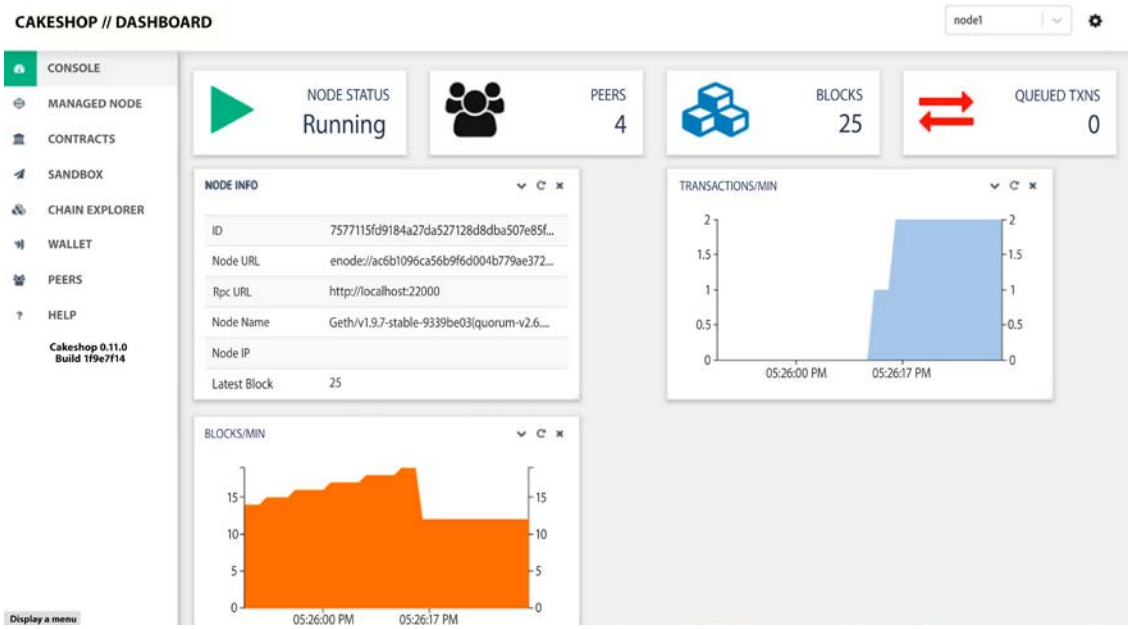


Figure 16.11: Cakeshop

In Cakeshop, we can see the transaction and relevant attributes. We simply browse to chain explorer and enter the transaction hash `"0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4"` from the *Running a private transaction* section. Cakeshop will display the status of the transaction, block ID, block number, contract address, and several other attributes, which makes it easy to explore the transaction process in detail:

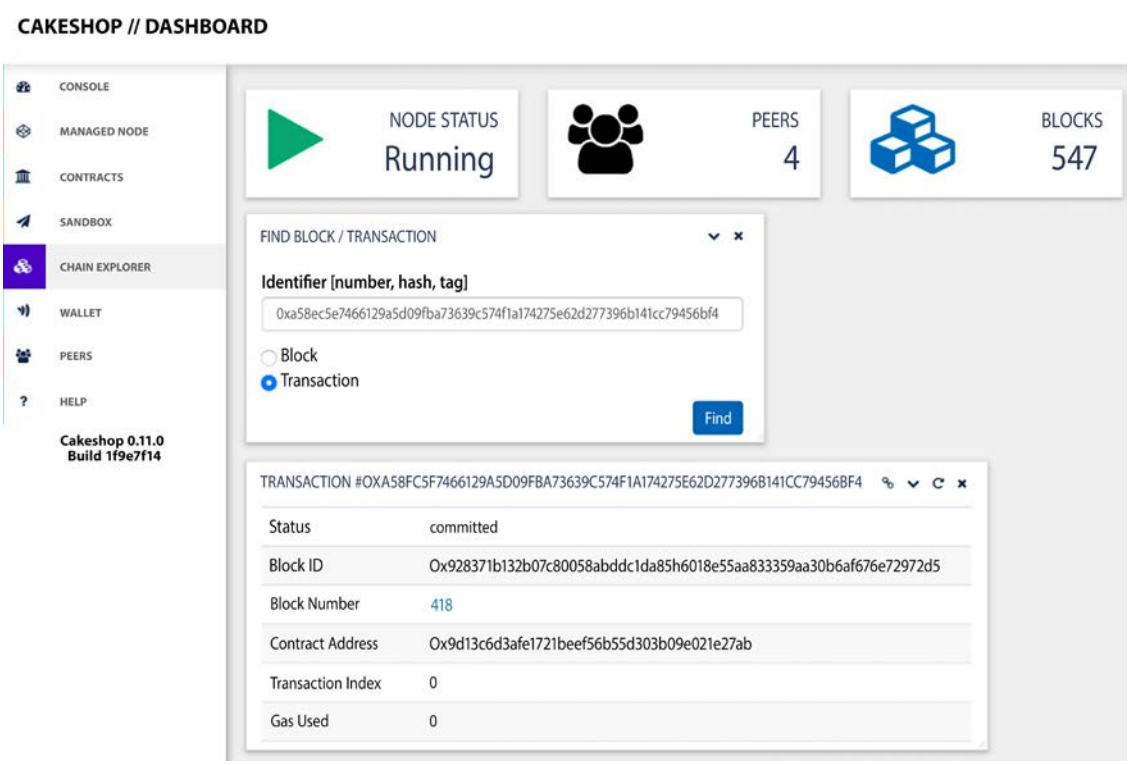


Figure 16.12: Cakeshop chain explorer

Further investigation with Geth

We can also see the transaction receipts and contract code using the `geth` console.

For this, we `attach` to `geth` using the following command:

```
$ geth attach
```

We will do this on each node, as follows.

Node 1: First, we open the `geth` console using the following command on Node 1:

```
$ geth attach qdata/dd1/geth.ipc
```

When the `geth` console opens, enter the following statement, which uses the transaction hash from our first step of contract deployment:

```
> eth.getTransaction("0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b14
1cc79456bf4")
```

The output shown here will be displayed:

```
{
  blockHash:
"0x92837fb132b07c80058abddc1da85b6018e55aa833359aa30b6af676e72972d5",
  blockNumber: 418,
  from: "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
  gas: 4700000,
  gasPrice: 0,
  hash: "0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4",
  input: "0x6d99da7776195f534b48d53e8a2e94231299d38967a874115fe329b0a8a236e83258
b04187674022ab7bb79847e45d02d266009ccb37a1011967e77da12b152b",
  nonce: 2,
  r: "0x3ca297d21fed45e3701d2e3ed9d4e3a144c1350ea5b981005ff734efc8467967",
  s: "0x6233d5d65445ae258e7c04efd085f1cdba42ca5f63c9982b4db779d45d098bae",
  to: null,
  transactionIndex: 0,
  v: "0x26",
  value: 0
}
```

In the preceding output, we can see that the `V` value is `0x26` in hexadecimal, which means `38` in decimal. This value of `38` indicates that the transaction is private.

Also, notice the input field, which is the encrypted payload of the transaction.

Similarly, we can get the transaction receipt of this transaction by issuing the call shown here:

```
> eth.getTransactionReceipt("0xa58ec5e7466129a5d09fba73639c574f1a174275e62
d277396b141cc79456bf4")
{
  blockHash:
"0x92837fb132b07c80058abddc1da85b6018e55aa833359aa30b6af676e72972d5",
  blockNumber: 418,
  contractAddress: "0x9d13c6d3afe1721beef56b55d303b09e021e27ab",
  cumulativeGasUsed: 0,
  from: "0xed9d02e382b34818e88b88a309c7fe71e65f419d",
  gasUsed: 0,
  logs: [],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000",
  status: "0x1",
  to: null,
  transactionHash:
"0xa58ec5e7466129a5d09fba73639c574f1a174275e62d277396b141cc79456bf4",
  transactionIndex: 0
}
```

We can see in the output that the status is `0x1`, indicating the success of the transaction. As we need to get the code of this transaction (smart contract), we need the contract address, which we can see in the `contractAddress` field. We copy that address into the RPC call shown here to fetch the code for this contract:

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
"0x60606040526000357c0100000000000000000000000000000000000000000000000000000000
00000900463ffffffff1680632a1afcd914605157806360fe47b11460775780636d4ce63c1
46097575b600080fd5b3415605b57600080fd5b606160bd565b604051808281526020019150
5060405180910390f35b3415608157600080fd5b6095600480803590602001909190505060
0c3565b005b341560a157600080fd5b60a760ce565b604051808281526020019150506040
180910390f35b60005481565b806000819055505b50565b6000805490505b905600a165627
a7a72305820d5851baab720bba574474de3d09d beaabc674a15f4dd93b974908476542c23
f00029"
```

As we can see in the preceding output, the contract code is visible as bytecode in hex.

Node 2, which is privy to the transaction: Similarly, to Node 1, open the `geth` console on Node 2 using the command shown here:

```
$ geth attach qdata/dd2/geth.ipc
```

In the `geth` console, enter the following statement. This method takes the address of the smart contract and returns the compiled smart contract code (bytecode):

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

The following output shows the bytecode of our contract at address `"0x9d13c6d3afe1721beef56b55d303b09e021e27ab"`:

```
"0x60606040526000357c0100000000000000000000000000000000000000000000000000000000
00000900463ffffffff1680632a1afcd914605157806360fe47b11460775780636d4ce63c1
46097575b600080fd5b3415605b57600080fd5b606160bd565b604051808281526020019150
5060405180910390f35b3415608157600080fd5b6095600480803590602001909190505060
0c3565b005b341560a157600080fd5b60a760ce565b604051808281526020019150506040
180910390f35b60005481565b806000819055505b50565b6000805490505b905600a165627
a7a72305820d5851baab720bba574474de3d09d beaabc674a15f4dd93b974908476542c23
f00029"
```

As expected, the code is also visible on Node 2, as Node 2 is a party to the transaction.

Node 3, which is not privy to the transaction: Now, we open the `geth` console on Node 3:

```
$ geth attach qdata/dd3/geth.ipc
```

In the console, enter the following statement with the address of our smart contract:

```
> eth.getCode("0x9d13c6d3afe1721beef56b55d303b09e021e27ab")
```

This will show the output shown here:

```
"0x"
```

This output indicates that the contract code is only available on the nodes that are party to the transaction. The contract code is not available on other nodes that are not a party to the transaction.

This experiment demonstrates that the contract is only available on the nodes that are party to the transactions. First, we deployed our private contract from Node 2 with Node 1 as a participant and used the `geth` console to interact with the contract. We saw that the value `42` is only available on the nodes that are party to the transaction. Also, we further experimented and noticed that not only the value but also the smart contract code is only available on the nodes that are party to our private contract (transaction).

This network is now available for further experiments. You can create your own private contracts with different nodes as parties. You can also explore how other methods such as `debug_traceTransaction` behave when interacting with private contracts. This network can also be used to build some PoCs for an enterprise dApp.

After this quick experiment of demonstrating how private transactions work in Quorum, let's explore other Quorum projects.

Other Quorum projects

As mentioned earlier, Quorum is a feature-rich platform and is under continuous improvement and development. As a result, new features and projects are introduced regularly. Some of the other projects under Quorum are listed here, along with brief descriptions.

Remix plugin

This is a plugin for the popular Remix IDE, which supports private smart contracts on the Quorum blockchain. More information on this is available here: `https://docs.goquorum.consensys.net/tutorials/quorum-dev-quickstart/remix#docusaurus_skipToContent_fallback`.

Pluggable architecture

Quorum supports a pluggable architecture that allows us to add new features to the Quorum client as plugins. This approach allows us to keep the core Quorum services separate from the new features, which allows greater modularity and extensibility without modifying the core client. More information on this feature is available here: `https://docs.goquorum.consensys.net/concepts/plugins`.

Quorum is a large project with many excellent enterprise features. It is not possible to cover all of them in this chapter. However, the material provided should get you started with the Quorum blockchain setup, plus an understanding of the concept of private transactions.

There is a wealth of information available in the official Quorum documentation at `https://docs.goquorum.consensys.net/en/latest/`. You are encouraged to go through the documentation to get a deeper understanding of Quorum.

Quorum has been used in many industries for different projects, including finance, supply chain, healthcare, media, and government sectors. A couple of example uses are:

- Tracking luxury goods: `https://www.coindesk.com/louis-vuitton-owner-lvmh-is-launching-a-blockchain-to-track-luxury-goods`
- Post-trade processing platform: `https://cointelegraph.com/news/oil-trading-blockchain-platform-vakt-launches-with-shell-bp-as-first-users`

Quorum is also available on different cloud platforms, including:

- Azure: `https://azure.microsoft.com/en-us/solutions/web3/#overview`
- Kaleido: `https://www.kaleido.io`

Summary

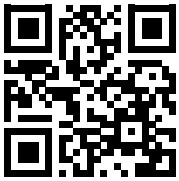
In this chapter, we started with an introduction to enterprise solutions and blockchain. We saw some limiting factors in public chains that make them unsuitable for enterprise use cases. We also looked at some requirements that, when met, will make a blockchain suitable for enterprise use cases.

We also covered how to design enterprise blockchain solutions and made a case to see the enterprise blockchain solutions in the context of the enterprise architecture. Next, we explored cloud computing and the definition of **Blockchain as a Service**. We briefly looked at the efforts being made to standardize enterprise Ethereum specifications. The last sections of this chapter covered enterprise blockchain platforms, including Quorum, and finally, we set up an IBFT network using Quorum to demonstrate how privacy is achieved using Quorum. Recent blockchains, such as VMware Blockchain, were also introduced.

In the next chapter, we will introduce scalability, security, and other challenges that blockchains face, how many of these limitations are being addressed, and what the future holds for blockchain technology.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

17

Scalability

Even though multiple use cases and Proof-of-Concept (PoC) systems have been developed using blockchains, and the technology works well for many scenarios, there is still a need to address some fundamental limitations present in blockchains to make the technology more adoptable.

At the top of the list of these issues comes **scalability**, which is a significant limitation. The lack of scalability is a general concern, where blockchains do not meet the adequate performance levels expected by users when the chain is used on a large scale.

There are several techniques to solve the scalability issue in blockchains, which will be discussed in detail in the following sections. Along the way, we'll cover the following topics:

- What is scalability?
- Blockchain scalability trilemma
- Methods to improve blockchain scalability
- Layer 0, 1, 2, and beyond

What is scalability?

This is the single most important problem in blockchain, which could mean the difference between the wider adaptability of blockchains or limited private use only by consortiums.

We can define scalability as the ability of a system to preserve or readjust its attributes to adapt to the increased demand for its resources as its utilization increases.

As a result of substantial research in this area, many solutions have been proposed, which are discussed in the following section.

From a theoretical perspective, the general approach toward tackling the scalability issue generally revolves around protocol-level enhancements. For example, a commonly mentioned solution to Bitcoin scalability is to increase its block size. This would mean that a larger number of transactions can be batched in a block, resulting in increased scalability.

Other proposals include solutions that offload certain processing to off-chain networks; for example, off-chain state networks.

Based on the solutions mentioned above, generally, the proposals can be divided into two categories: **on-chain solutions**, which are based on the idea of changing fundamental protocols on which the blockchain operates, and **off-chain solutions**, which make use of off-chain network resources to enhance the blockchain.

While there are many solutions to the scalability problem now, and the situation is not as bad as the first decade after Bitcoin's introduction, note that scalability is not easy to achieve. Some tradeoffs need to be made for a scalability solution to work. It has been conjectured that blockchain's three objectives, decentralization, scalability, and security, cannot be achieved simultaneously – the so-called blockchain trilemma.

Blockchain trilemma

Based on the problem presented by Vitalik Buterin, an Ethereum co-founder, it is understood that only two of the three main core properties of a blockchain can be utilized at a time. These three core properties are the following:

- **Decentralization** – This means that the system runs for participants with access only to normal resources, e.g., an entry-level computer with the usual hardware. In other words, no specialized hardware is required to participate in the network, a requirement that could tilt the scales in favor of those with greater resources.
- **Scalability** – This means that the overall system is able to process a larger number of transactions than the number of transactions an individual participant can process. In simpler words, the system is able to perform a high number of transactions.
- **Security (or consistency)** – This means that the system is secure against adversaries even with high levels of resources (though not unlimited).

The trilemma means that the three key goals of a scalable blockchain – security, scalability, and decentralization – cannot be fully achieved; we must sacrifice at least one of them to achieve the other two. In other words, the blockchain trilemma states that a simple blockchain architecture can only achieve two out of three properties, e.g. if you need a secure and decentralized blockchain, you need to sacrifice scalability. Similarly, if you want to achieve scalability and security, then decentralization must be sacrificed.

This is known as the **blockchain trilemma** and is seen as a fundamental problem that needs to be addressed before the global adoption of Ethereum and other blockchains can be achieved.

This concept is visualized in the following diagram:

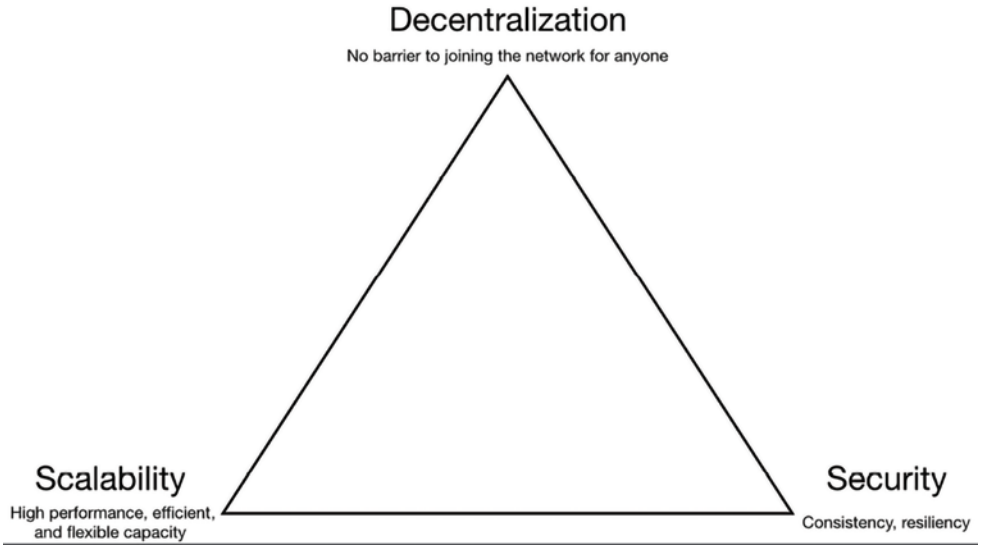


Figure 17.1: Blockchain trilemma

Traditional monolithic blockchains like Bitcoin perform all blockchain operations including data availability, consensus, settlement, and execution on-chain. Data availability ensures that block header data is publicly available so that anyone can recreate the state for verification purposes. Consensus, as we saw in *Chapter 5*, *Consensus Algorithms*, ensures consistency of the chain by achieving agreement between participants on the inclusion and ordering of the transactions. Settlement means finalization of the transactions on-chain. Finally, we have execution, which means the running (computation) of a transaction to transition it from the existing state to the new state.

There are three bottlenecks that adversely affect blockchain performance: execution, verification, and communication. By design, on monolithic blockchains, every participant has to accept, process, forward, and execute every transaction, which makes the network fundamentally slow.

Intensive research is underway and significant advancement has been made in this regard in the blockchain research community. With solutions such as layer 2, multichain networks, and layer 1 improvements, the scalability issue has been addressed to some extent, although there are still some tradeoffs to be made. Even with all the aforementioned advances, blockchain scaling is not straightforward. There is no one-size-fits-all solution due to differing requirements and use cases. In future, it is likely that multiple scalability solutions will coexist. Multiple solutions also lead to disparate heterogeneous architectures, which makes developer adoption somewhat challenging. Also, although good progress is being made, the layer 2 landscape undergoes constant evolution, which makes it difficult to adopt a single solution for the long term.

An important thing to understand is that due to the blockchain trilemma, when high performance and efficiency are achieved, either security or decentralization must be given up to some extent. All three properties cannot be achieved at the same time.

This is, however, somewhat debatable, as proponents of layer 1 scaling such as Solana are of the opinion that all three properties can be achieved, and Solana in particular claims to have demonstrated this with the Solana blockchain. However, there is some concern that this solution is not as decentralized as PoW networks, because the number of validators in the former is not enough to enable full decentralization. Roughly speaking, as long as more than 50% of the validator network is under the control of an honest majority, the network is expected to remain secure.

Note that in order to achieve performance and speed, it might be acceptable to give up some level of decentralization as a tradeoff, but it should not be so much that it compromises security and consequently results in transaction censorship.



We discussed the Nakamoto coefficient in *Chapter 2*, *Decentralization,* on decentralization. It turns out that chains like Solana have a good Nakamoto coefficient (around 31), but perhaps not enough to thwart cartel formation attacks. Other chains like Polygon have an even lower Nakamoto coefficient (about 4), which is not suitable. Here, too much decentralization is given up in favor of efficiency (speed), which is not ideal. Solana appears to be a better choice, with a Nakamoto coefficient of 31. These stats are tracked online at `https://nakaflow.io`.

There's a lot of research in this area, and the aim is to address this trilemma and find the right balance between all three properties instead of compromising and only choosing two of the three. In this regard, Algorand has proposed a key solution that claims to have solved the blockchain trilemma without sacrificing any of the three objectives. The trick is to enable random validator selection to pick up the next set of nodes to add blocks. The algorithm can choose anyone randomly to become the next block-adding validator, thus not giving up decentralization. The algorithm achieves scalability by randomly selecting a small set of representatives (committee) that run the protocol instead of using all nodes. This small set of block proposers and verifiers allows users to only have to receive a small, fixed number of messages to achieve consensus for the next block, thus achieving speed and scalability. This is called a pure proof-of-stake algorithm.



Algorand is a very important development and has in fact refuted the blockchain trilemma. With Algorand it is possible to achieve all three properties, i.e., scalability, decentralization, and security. We'll introduce Algorand in *Chapter 23*, *Alternative Blockchains*.

With the advent of **layer 2 protocols**, **innovative layer 1 enhancements**, **faster consensus mechanisms**, and other techniques like **sharding** and **parallelization**, a somewhat balanced combination of all three properties of decentralization, scalability, and security (consistency) can be achieved.

Now, we discuss some methods for improving scalability.

Methods for improving scalability

As this is a very active area of research, over the years many techniques and proposals have been made to address the blockchain scalability problem. In this section, we'll introduce many of these techniques.

We can divide approaches to solving the scalability issue into four main categories based on the *layer* in the blockchain stack they operate on.

We describe these categories here:

- **Layer 0** methods (also called *multichain methods*, *modular blockchains*, or *polylithic blockchain architecture*) involve a multi-chain ecosystem, enabling interoperability between chains and allowing developers to create custom chains.
- **Layer 1** methods are also called on-chain methods, where the blockchain and network protocol themselves are enhanced to improve scalability. This layer represents the blockchain and the network.
- **Layer 2** methods, or off-chain methods, are where mechanisms that are not part of the blockchain and exist outside of the main blockchain are used to improve the scalability of the blockchain.
- **Layer 3** methods (also called *multilayer scaling* or *hyperscaling*) are based on the fundamental observation that if layer 2, using rollups, can compress data up to n times, then it could be possible to add another layer on top and achieve data compression multiple (up to n x n) times. Note that there are some other views on what's beyond layer 2, such as proposals made by StarkWare – so-called fractal scaling, which introduces application-specific layer 3s that are built recursively over layer 2.

Scalability has two facets. One is increasing the processing speed of transactions to achieve better **transactions per second** (**TPS**), and the other is the increase in the number of nodes on the network. Both are desirable in many situations; however, transaction speed is more sought after on public networks, as node scalability is not an issue in public blockchain networks. This is evident from blockchain networks like Ethereum and Bitcoin where thousands of nodes operate on the network, but the transaction throughput is roughly around 7 and 15 TPS respectively.

Layer 0 – multichain solutions

We'll describe some of the layer 0, or network layer, solutions in the following sections.

Layer 0 solutions emerged because of multichain architectures (sometimes called a "blockchain of blockchains") where a central chain acts as a relay between multiple sidechains. Some prime examples of layer 0 chains are **Polkadot**, **Avalanche**, and **Cosmos**.

A brief introduction to Polkadot is given below.

Polkadot

Polkadot is a modern blockchain protocol that connects a network of purpose-built blockchains and allows them to operate together. It is a heterogenous multichain ecosystem with shared consensus and shared state.

Polkadot has a central main chain called the Relay Chain. This Relay Chain manages the parachains – the heterogenous shards connected to the Relay Chain. The Relay Chain holds the states of all parachains. All these parachains can communicate and share security, leading to a better and more robust ecosystem.

As the parachains are heterogenous, they can serve different purposes. For example, one chain can be specifically for smart contracts, another for gaming, another for providing some public service, and so on. The Relay Chain is secured by nominated proof of stake.

This Relay Chain and parachain-based sharding architecture with many blockchains running in parallel is how Polkadot achieves scalability. Sharding allows many computations to be executed in parallel. Polkadot can also connect with private chains, other public chains, consortium chains, and oracles, which enables interoperability and results in increased network scalability, improved "network effect," and overall throughput.

The Relay Chain's validators produce blocks, communicate with parachains, and finalize blocks. On-chain governance through stake-based voting schemes decides what the ideal number of validators should be:

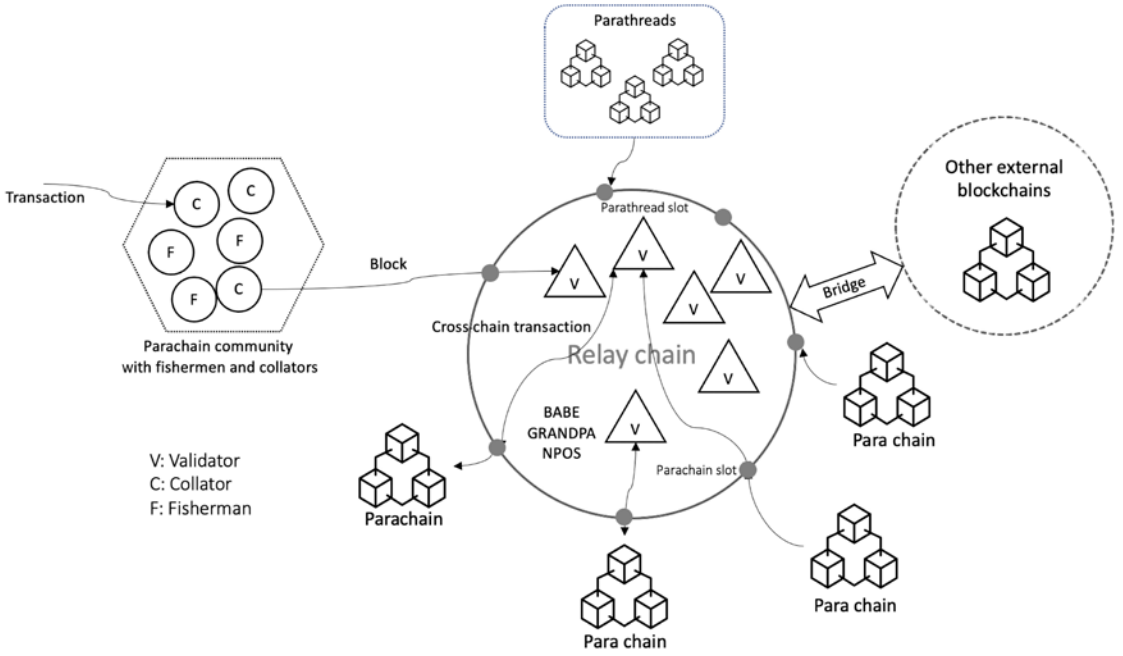


Figure 17.2: A depiction of the Polkadot network

Polkadot aims to be able to communicate with other blockchains as well. For this purpose, bridges are used that connect parachains to external blockchains such as Bitcoin and Ethereum.

There are several components in Polkadot. The Relay Chain is the main chain responsible for managing parachains, cross-chain interoperability, interchain messaging, consensus, and security.

As shown in *Figure 17.2*, the network consists of nodes and roles. Nodes can be light clients, full nodes, archive nodes, or sentry nodes. Light clients consist of only runtime and state. Full nodes are pruned at configurable intervals. Archive nodes keep the entire history of blocks, and sentry nodes protect validators and thwart DDoS attacks to provide security to the Relay Chain.

There are several roles that nodes can perform: validator, nominator, collator, and fisherman. Validators are the highest level in charge of the system. They are block producers, and to become block producers, they need to provide a sufficient bond deposit. They produce and finalize blocks and communicate with parachains. Nominators are stakeholders and contribute to the validators' security bond. They trust the validators to "be good" and produce blocks. Collators are responsible for transaction execution. They create unsealed but valid blocks for validators. Fishermen nodes are used to detect malicious behavior. Fishermen are rewarded for providing proof of the misbehavior of participants.

Parachains are heterogenous blockchains connected to the Relay Chain. These are fundamentally the execution core of Polkadot. Parachains can exist with their own runtimes, called application-specific blockchains. Another component called a parathread is a blockchain that works within the Polkadot host and connects to the Relay Chain. They can be thought of as pay-as-you-go chains. A parathread can become a parachain via an auction mechanism. Bridges are used to connect parachains with external blockchain networks like Bitcoin and Ethereum.

Next, let's move on to some core on-chain solutions, or *layer 1* solutions.

Layer 1 — on-chain scaling solutions

In this section, we'll describe layer 1 (network-level; on-chain), solutions, which target core blockchain elements such as blocks, transactions, and other on-chain data structures to address the scalability problem.

Kadcast

This is a new protocol that enables fast, efficient, and secure block propagation for the Bitcoin blockchain network.



More information is available in the paper: Rohrer, E. and Tschorsch, F., 2019, October. *Kadcast: A structured approach to broadcast in blockchain networks*. In Proceedings of the 1st ACM Conference on Advances in Financial Technologies (pp. 199-213).

`https://dl.acm.org/doi/pdf/10.1145/3318041.3355469`

bloXroute

Another network layer solution is bloXroute, which aims to address scalability problems by creating a trustless blockchain distribution network.



More information about bloXroute is available at `https://bloxroute.com` and in the following whitepaper:

`https://bloxroute.com/wp-content/uploads/2019/11/bloXrouteWhitepaper.pdf`

Another option that has been proposed to improve scalability is transaction parallelization, which we introduce next.

Transaction parallelization

Usually, in blockchain designs, transactions are executed sequentially. For example, in Ethereum, all transaction execution is sequential, which allows it to be safe and consistent. This also raises the question that if, somehow, transaction executions can be done in parallel without compromising the consistency and security of the blockchain, it would result in much better performance.

Some suggestions have already been made for Ethereum, for example:

- Easy parallelizability, which introduces a new type of transaction: `https://github.com/ethereum/EIPs/issues/648`.
- Transaction parallelizability in Ethereum, which proposes a mechanism to enable parallel transaction execution in Ethereum. The paper is available here: `https://arxiv.org/pdf/1901.09942.pdf`.

Parallel transactions are supported on several blockchain platforms including Hyperledger Sawtooth and Solana. Solana allows parallel execution of smart contracts using a runtime named *Sealevel*.

Increase in block size

This is the most debated proposal for increasing blockchain performance (transaction processing throughput). Currently, Bitcoin can process only about three to seven transactions per second, which is a major inhibiting factor in adapting the Bitcoin blockchain for processing microtransactions. Block size in Bitcoin is hardcoded to be 1 MB, but if the block size is increased, it can hold more transactions and can result in faster confirmation time. There are several **Bitcoin Improvement Proposals** (**BIPs**) made in favor of block size increase. These include `BIP 100`, `BIP 101`, `BIP 102`, `BIP 103`, and `BIP 109`.



An interesting account of historic references and discussion is available at `https://en.bitcoin.it/wiki/Block_size_limit_controversy`.

In Ethereum, the block size is not limited by hardcoding; instead, it is controlled by a gas limit. In theory, there is no limit on the size of a block in Ethereum because it's dependent on the amount of gas, which can increase over time. This is possible because miners are allowed to increase the gas limit for subsequent blocks if the limit has been reached in the previous block. Bitcoin **Segregated Witness** (**SegWit**) has addressed this issue by separating witness data from transaction data, which resulted in more space for transactions. Other proposals for Bitcoin include Bitcoin Unlimited, Bitcoin XT, and Bitcoin Cash. You can refer to *Chapter 6*, *Bitcoin Architecture*, for more details.



For more information on Bitcoin proposals, refer to the following addresses:

`https://www.bitcoinunlimited.info`

`https://www.bitcoincash.org`

Block interval reduction

This is another proposal about reducing the time between each block generation. The time between blocks can be decreased to achieve faster finalization of blocks, but it may result in less security due to the increased number of forks. Ethereum has achieved a block time of approximately 14 seconds.

This is a significant improvement from the Bitcoin blockchain, which takes 10 minutes to generate a new block. In Ethereum, the issue of high orphaned blocks resulting from shorter times between blocks is mitigated by using the **Greedy Heaviest Observed Subtree** (**GHOST**) protocol, whereby orphaned or stale blocks (also called uncles in the Ethereum chain) are also included in determining the valid chain. Once Ethereum moves to **Proof of Stake** (**PoS**), this will become irrelevant as no mining will be required and almost immediate finality of transactions can be achieved.

Invertible Bloom Lookup Tables

This is another approach that has been proposed to reduce the amount of data required to be transferred between Bitcoin nodes. **Invertible Bloom Lookup Tables** (**IBLTs**) were originally proposed by Gavin Andresen, and the key attraction of this approach is that it does not result in a hard fork of Bitcoin if implemented. The key idea is based on the fact that there is no need to transfer all transactions between nodes; instead, only those that are not already available in the transaction pool of the syncing node are transferred. This allows quicker transaction pool synchronization between nodes, thus increasing the overall scalability and speed of the Bitcoin network.

Sharding

Sharding is not a new technique and has long been used in distributed databases such as MongoDB and MySQL for scalability. The key idea behind sharding is to split up the tasks into multiple chunks that are then processed by multiple nodes. This results in improved throughput and reduced storage requirements. In blockchains, a similar scheme is employed, whereby the state of the network is partitioned into multiple shards. The state usually includes balances, code, nonce, and storage. Shards are loosely coupled partitions of a blockchain that run on the same network. There are a few challenges related to inter-shard communication and consensus on the history of each shard. This is an area of active research and has been extensively studied in the context of scaling Ethereum.

Private blockchains

Most private blockchains are inherently faster because no real decentralization is required and participants on the network do not need to mine using PoW; instead, they can only validate transactions. This can be considered as a workaround to the scalability issue in public blockchains; however, this is not the solution to the scalability problem. Also, it should be noted that private blockchains are only suitable in specific areas and setups such as enterprise environments, where all participants are known.

Block propagation

In addition to the preceding proposal, pipelining of block propagation has also been suggested, which is based on the idea of anticipating the availability of a block. In this scheme, the availability of a block is already announced without waiting for actual block availability, thus reducing the round-trip time between nodes.

Finally, the problem of long distances between the transaction originator and nodes also contributes toward the slowdown of block propagation. It has been shown in research conducted by Christian Decker et al. that connectivity increases can reduce the propagation delay of blocks and transactions. This is possible because, if at any one time the Bitcoin node is connected to many other nodes, it can speed up the information propagation on the network.

An elegant solution to scalability issues will most likely be a combination of some or all of the aforementioned general approaches. A few initiatives undertaken in order to address scalability and security issues in blockchains are now almost ready for implementation or have already been implemented. For example, Bitcoin SegWit is a proposal that can help massively with scalability and only needs a soft fork in order for it to be implemented. The key idea behind so-called SegWit is to separate signature data from the transactions, which resolves the transaction malleability issue and allows block size increase, thus resulting in increased throughput.

Bitcoin-NG

Another proposal, Bitcoin-NG, based on the idea of microblocks and leader election, has gained some attention recently. The core idea is to split blocks into two types, namely leader blocks (also called key blocks) and microblocks:

- **Leader blocks**: These are responsible for PoW, whereas microblocks contain actual transactions.
- **Microblocks**: These do not require any PoW and are generated by the elected leader every block-generation cycle. This block-generation cycle is initiated by a leader block. The only requirement is to sign the microblocks with the elected leader's private key. The microblocks can be generated at a very high speed by the elected leader (miner), thus resulting in increased performance and transaction speed.

On the other hand, the *Ethereum 2.0 Mauve Paper*, written by Vitalik Buterin and presented at Ethereum Devcon2 in Shanghai, describes a different vision of a scalable blockchain.



The Mauve Paper is available at `https://docs.google.com/document/d/1maFT3cpHvwn29gLvtY4WcQiI6kRbN_nbCf3JlgR3m_8/edit#`.

This proposal is based on a combination of sharding and an implementation of the PoS consensus algorithm. The paper defined certain goals such as gaining efficiency via PoS, minimizing block time, and ensuring economic finality, scalability, cross-shard communication, and censorship resistance. Some of the vision presented in the Mauve Paper has been implemented in Ethereum as "the Merge" upgrade. We discussed the Merge and the future of Ethereum in *Chapter 13*, *The Merge and Beyond*.

DAG-based chains

DAG stands for **Directed Acyclic Graph**. It is seen as an alternative to linear chain-based blockchain technology. A blockchain is fundamentally a linked list, whereas a DAG is an acyclic graph where links between nodes have only one direction. In other words, DAG-based blockchains do not look like a linear chain, but more like a graph that resembles a tree. A DAG consists of vertices and edges.

Directed means the graph moves only in one direction and acyclic means that there is the possibility of moving back to a node from the current node.

The DAG structure allows for parallel creation and confirmation of transactions and blocks, thus achieving high transaction throughput. The key idea behind DAG-based chains is that, as we know in normal blockchains, there is a linear sequence of blocks one after another and in the event of forks, only one fork survives due to the fork selection rule (e.g. the longest chain) where the rest of the forks are ignored (destroyed). This means that blocks/transactions can be generated in a limited fashion one after another. Now if somehow, we are able to keep those forks without compromising security, then it means that we can produce more blocks and much faster, even almost in parallel. So instead of only one child and parent in a linear sequence of blocks, in DAGs there are blocks with multiple children and multiple parents, thus improving the block/transaction production speed.

There are two types of DAG-based distributed ledgers:

- **Block-less DAGs**: Here, vertices are transactions, and no blocks exist. Examples of this type of chain are IOTA (`https://www.iota.org`) and Obyte (`https://obyte.org/`).
- **Block-based DAGs**: Here, vertices are blocks and blocks can refer to several predecessor blocks. Examples of this type of chain are the FANTOM, Ghost, and Spectre protocols. See more at `https://eprint.iacr.org/2018/104.pdf`.

Faster consensus mechanisms

Traditionally, blockchains are based on PoW or PoS algorithms, which are inherently slow in terms of performance. PoW especially can process only a few transactions per second. If a different consensus mechanism such as Raft or PBFT is used, then the processing speed can increase significantly. We examined these concepts in greater detail in *Chapter 5*, *Consensus Algorithms*.

PoS algorithm-based blockchains are fundamentally faster because PoS algorithms do not require the completion of time- and energy-consuming PoW.

So far, we have discussed layer 0 (multichain) and layer 1 (on-chain) solutions to the scalability issue. In the next section, we'll introduce a more popular approach that aims to solve the scalability problem by using off-chain or layer 2 components.

Layer 2 — off-chain solutions

Layer 2 solutions are based on the idea that instead of modifying the main chain to achieve scalability, the objective should be to offload some of the processing to faster mechanisms outside of the main underlying chain, do the processing there, and then write the result back on the main chain as an integrity guarantee. Several such techniques are described here.

Layer 2

Layer 2 is the name given to technologies that help to scale Ethereum using off-chain techniques. Blockchains such as Bitcoin and Ethereum currently can do up to 7 and 15 transactions per second respectively. This is much slower than traditional centralized methods of payment, such as Visa, which can process 100,000 transactions per second.

Unless blockchain transaction throughput can be significantly improved, it cannot be used for day-to-day business. A lot of research has been done in this regard. There are two ways to address this issue, either by improving the base layer (i.e., layer 1), or offloading some of the work to another layer (an off-chain system).

Note that there are some complex layer 1 chains that are trying to achieve all three properties at the same time with apparently good results, e.g. Solana. However, there are a few hurdles along the way and only time will tell if it succeeds – so far it's looking reasonable. So, if we cannot scale layer 1, then is there any other option? Yes – we offload the work to another off-chain system, called layer 2.

Layer 2 solutions are off-chain tools, mechanisms, and protocols that allow layer 1 to scale without requiring any changes in layer 1. Some changes can be made to layer 1 to make it more layer 2 friendly, however no changes are necessary. In a way, layer 2 complements layer 1 and results in a more scalable blockchain system. The L2 state can be verified by L1 through either validity proofs or fraud proofs. This mechanism is most important as it ensures that L2 validators cannot cheat and include invalid transactions in an L2 block, e.g., mint coins out of thin air or steal your coins. The second use of L1 is as a data availability layer (state validation) for L2 transactions so that, if there is a dispute, users can either independently re-create the L2 state and ensure data validity and continued system operation, or they can trustlessly exit to L1. Key advantages of layer 2 include lower fees, benefitting from L1 security while being able to execute transactions quicker, and the fact that more use cases are possible than on L1. Layer two solutions aim to achieve similar security and decentralization guarantees as those of the layer 1 chain by inheriting these security guarantees from layer 1.

State channels

Also called payment channels, state channels are another proposal for speeding up the transaction on a blockchain network. The basic idea is to use side channels for state updating and processing transactions off the main chain; once the state is finalized, it is written back to the main chain, thus offloading the time-consuming operations from the main blockchain.

State channels work by performing the following three steps:

1. First, a part of the blockchain state is locked under a smart contract, ensuring the agreement and business logic between participants.
2. Now, off-chain transaction processing and interaction are started between the participants that update the state (only between themselves for now). In this step, almost any number of transactions can be performed without requiring the blockchain. This is what makes the process so fast and arguably the best candidate for solving blockchain scalability issues. It could be argued that this is not a real on-blockchain solution such as, for example, sharding, but the end result is a faster, lighter, and more robust network that can prove very useful in micropayment networks, IoT networks, and many other applications.
3. Once the final state is achieved, the state channel is closed, and the final state is written back to the main blockchain. At this stage, the locked part of the blockchain is also unlocked.

This process is shown in the following diagram:

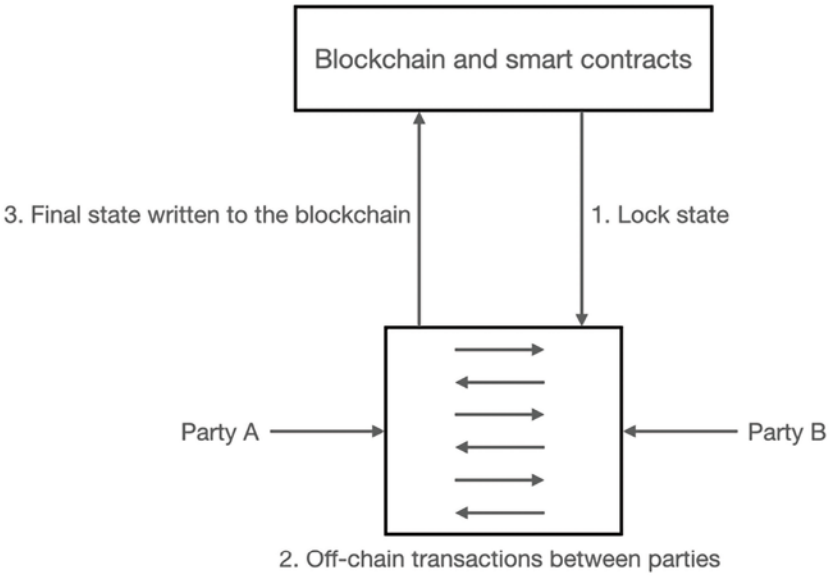


Figure 17.3: State channels

This technique has been used in the **Bitcoin Lightning** network and Ethereum's **Raiden**. The key difference between Lightning and Raiden is that Lightning only works for Bitcoin transactions, whereas Raiden supports all ERC20-compliant tokens, making the Raiden network a more flexible option.

Sidechains

Sidechains can improve scalability indirectly by allowing many sidechains to run along with the main blockchain, while allowing usage of perhaps comparatively less secure and faster sidechains to perform transactions but that are still pegged with the main blockchain. The core idea of sidechains is called a **two-way peg**, which allows the transfer of coins from a parent chain to a sidechain and vice versa.

Sub-chains

This is a relatively new technique recently proposed by Peter R. Rizun and is based on the idea of weak blocks, which are created in layers until a strong block is found.



Rizun's sub-chains research paper is available at `https://ledger.pitt.edu/ojs/ledger/article/view/40/55`. Rizun, P. R. (2016). *Subchains: A Technique to Scale Bitcoin and Improve the User Experience*. Ledger, 1, 38-52.

Weak blocks can be defined as those blocks that have not been able to be mined by meeting the standard network difficulty criteria but have done enough work to meet another, weaker, difficulty target. Miners can build **sub-chains** by layering weak blocks on top of each other unless a block is found that meets the standard difficulty target.

At this point, the sub-chain is closed and becomes the strong block. Advantages of this approach include a reduced waiting time for the first verification of a transaction. This technique also results in a reduced chance of orphaning blocks and a speeding up of transaction processing. This is also an indirect way of addressing the scalability issue. Sub-chains do not require any soft or hard fork to implement but do need acceptance by the community.

Tree chains

There are also other proposals to increase Bitcoin scalability, such as **tree chains**, which change the blockchain layout from a linearly sequential model to a tree. This tree is basically a binary tree that descends from the main Bitcoin chain. This approach is similar to sidechain implementation, eliminating the need for major protocol changes or block size increases. It allows improved transaction throughput. In this scheme, the blockchains themselves are fragmented and distributed across the network to achieve scalability.

Moreover, mining is not required to validate the blocks on the tree chains; instead, users can independently verify the block header. However, this idea is not ready for production yet and further research is required in order to make it practical.



The original idea was proposed in the following research paper: `https://eprint.iacr.org/2016/545.pdf`.

In addition to the aforementioned general techniques, some Bitcoin-specific improvements have also been proposed by Christian Decker in his book *On the Scalability and Security of Bitcoin*. This proposal is based on the idea of speeding up propagation time as the current information propagation mechanism results in blockchain forks. These techniques include minimization of verification, pipelining of block propagation, and connectivity increase. These changes do not require fundamental protocol-level changes; instead, these changes can be implemented independently in the Bitcoin node software.

With regard to verification minimization, it has been noted that the block verification process contributes toward propagation delay. The reason behind this is that a node takes a long time to verify the uniqueness of the block and transactions within the block. It has been suggested that a node can send the inventory message as soon as the initial PoW and block validation checks are completed. This way, propagation can be improved by just performing the first *difficulty check* and not waiting for transaction validation to finish.

Plasma

Another scalability proposal is **Plasma**, which has been proposed by Joseph Poon and Vitalik Buterin. This proposal describes the idea of running smart contracts on the root blockchain (Ethereum mainnet) and having child blockchains that perform high numbers of transactions to feed back small amounts of commitments to the parent chain. In this scheme, blockchains are arranged in a tree hierarchy with mining performed only on the root (main) blockchain, which feeds the proofs of security down to child chains. This is also a layer 2 system since, like state channels, Plasma also operates off-chain.



The research paper on Plasma contracts is available at `http://plasma.io`.

Plasma vs sidechains

Plasma chains are almost like sidechains, however, they sacrifice some functionality for extra security. Plasma chains can be seen as non-custodial sidechains.

A side chain is an alternate chain to the main parent chain, whereas Plasma is a framework for child chains. Sidechains run as a separate blockchain in parallel to a layer 1 blockchain such as Ethereum. These two chains can communicate with each other so that assets can be moved between the chains. Sidechains also have a consensus mechanism. Plasma chains do have a consensus mechanism used to produce blocks, however, unlike sidechains the Merkle root of each block of the Plasma chain is submitted to Ethereum. Block Merkle roots are used to prove the correctness of the blocks.

Trusted hardware-assisted scalability

This technique is based on the idea that complex and heavy computing can be offloaded to off-chain resources in a verifiably secure manner. Once the computations are complete, the verified results are sent back to the blockchain. An example of such a solution is Truebit: `https://truebit.io/`.

Commit chains

Commit chains are a more generic term for Vitalik Buterin's Plasma proposal. They are also referred to as non-custodial sidechains but without a new consensus mechanism, as is the case with sidechains. They rely on the main chain consensus mechanism, so they can be considered as safe as the main chain. The operator of the commit chain is responsible for facilitating communication between transacting participants and sending regular updates to the main chain.



More information on commit chains is available here: Khalil, R., Zamyatin, A., Felley, G., Moreno-Sanchez, P. and Gervais, A., 2018. *Commit-Chains: Secure, Scalable Off-Chain Payments*. Cryptology ePrint Archive, Report 2018/642, 2018. `https://eprint.iacr.org/2018/642.pdf`.

Rollups

In rollup solutions, transactions are submitted directly to layer 2 instead of layer 1. Submitted transactions are batched and eventually submitted to layer 1. Layer 2s are independent blockchains with nodes that are Ethereum-compatible. All states and executions are processed on layer 2, including signature verification and contract execution. L1 only stores the transaction data, hence the performance boost.

In other words, rollups provide an execution environment outside layer 1, i.e., layer 2, which results in faster execution due to the absence of layer 1 limitations. Once the execution is completed the proof and summary of data is posted to layer 1, where consensus is reached. Therefore, layer 1 is usually called the settlement layer.

This proof is a proof of computational integrity, proving that the state transition is valid after transaction execution. These proofs provide evidence of transaction validity in the batch, that internal application logic of the rollup is correctly followed, and of state transition.

Rollups are now quite commonly used and are the main scalability solution for blockchains.

Rollups can be divided into two categories based on their usage. First, we have application-specific rollups where a resource-hungry (expensive) part of an application (i.e. execution) is bundled up in a rollup. Secondly, we have general rollups, which help to scale EVM-based blockchain networks by their ability to generate validity proofs for any state transition on the EVM.

Layer 2 can improve TPS to tens of thousands of TPS without sacrificing decentralization and security because its security is inherited from the layer 1 chain, e.g., Ethereum.

While achieving scalability, it is also important to ensure the integrity of the chains and relevant data. There are two ways layer 1 can provide security (data integrity) for layer 2, *data validity* and *data availability*.

Data validity

Data validity refers to the requirement that the layer 2 state can be verified by layer 1 by using validity proofs or fraud proofs. This mechanism ensures that layer 2 validators cannot do malicious things such as including invalid transactions in layer 2 blocks, censoring transactions, creating money out of thin air, or stealing funds. Colloquially speaking, data validity ensures that "no one can spend funds that do not belong to them." A layer 2 blockchain can publish its state periodically to the layer 1 chain by writing the hash of its latest state root. This state root is provided as a cryptographic validity proof using zero knowledge and verified at layer 1 by a smart contract. Another technique is the use of fraud proofs, which are based on the paradigm where honest observers monitor the layer 2 chain and in the case of any suspected incorrect state root submission to layer 1, they can raise an alarm and provide the fraud proof, which will result in automatic chain rollback.

Data availability

Data availability refers to a requirement where in the case of disputes on a transaction, the users must be able to independently recreate the layer 2 state and can do a graceful trustless exit to layer 1. Colloquially speaking, data availability ensures that "anyone can spend their own funds." Data availability is useful when users need to prove to a layer 1 chain that they own the funds they are trying to withdraw or spend.

For this to happen, layer 1 needs to have access to all transactions on layer 2, or its latest (current) state. One common technique to achieve this is to record layer 2 transactions on layer 1 using *calldata*. Another technique is to store records on a separate external data availability layer where the provider guarantees the data availability via cryptoeconomic or cryptographic mechanisms. Note that posting data to Ethereum using calldata is expensive, however, this is the only practical technique available to post data easily on L1 Ethereum. Any other technique could be too expensive. Another technique is to store data totally off-chain, where another provider becomes the custodian of the data.

A simple way to solve the data availability problem is to download the full data on each entity on the network, i.e., every node keeps a full copy of the data, which is of course not sustainable due to the amounts of data involved and the replication overhead. The other more commonly used and practical option is data availability proofs, which allow clients to check that full data for a block has been published by only downloading a small part of the block.

Computation and data are the two main bottlenecks on the blockchain that are addressed by rollups. Rollups compress data so that only a small amount of data is posted onto layer 1, thus reducing the data footprint. Rollups use several techniques for data compression including signature aggregation (one signature per batch, instead of one signature per transaction).

Computation and execution are addressed by performing executions off-chain and submitting fraud proofs or validity proofs to ensure data validity.

Generally, we can say the key security goals of rollups are data availability, state transition integrity, and censorship resistance. Data availability can be summarized as the question of whether all state updates made to the layer 2 database are publicly available. State transition integrity questions whether all state updates made to the database are valid. Censorship resistance asks the question of whether a user is able to ensure that once the transaction is submitted, it will eventually execute and not be censored by an adversary.

How rollups work

Rollups operate using a smart contract (i.e., rollup contract) that exists on-chain. This smart contract is responsible for verifying and maintaining the state root. State root or batch root is in fact the Merkle root of the entire state of the rollup including account balances, smart contract code, transactions, etc.

A collection of transactions in a compressed format, usually called a batch, can be posted by anyone. The previous and new state root are published with the batch. The on-chain rollup contract checks the previous state root in the batch and matches it with its current state root and if both match, then it shifts the state root to the new state root.

A rollup system handles deposits, transfers, and withdrawals. We can visualize a generic rollup system in *Figure 17.4*:

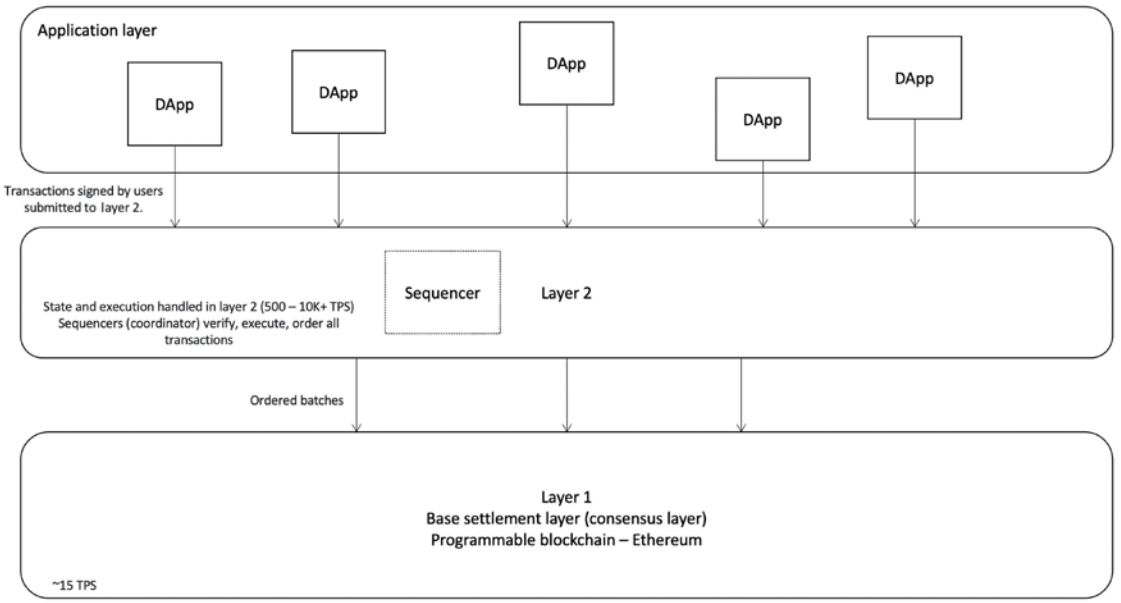


Figure 17.4: A rollup system

The core idea is as simple as described, but the key problem is to how make sure that the new states are valid and correct. This is where *fraud proofs* and *validity proofs* come in.

If someone observes that a batch had an incorrect post state root (new root), then a fraud proof is submitted as evidence to the chain via the rollup contract to refute the batch. The rollup contract maintains the complete history of state roots and hash of each batch of transactions. The rollup contract verifies the fraud proof, and if valid, reverts the batch in question and all batches that come after it.

Validity proofs are used by ZK-rollups, which are cryptographic proofs included with every batch that prove that the new state root is the correct result of executing the batch of transactions.

A transaction batch can be submitted by many types of users, depending on the security and architecture of the system. The core security of the system revolves around the rule that anyone wishing to submit batches should have a large stake/deposit in the system, which acts as a guarantee against fraudulent (malicious) behaviours. If the user ever submits a fraudulent batch that is proven to be invalid via a fraud proof, then the stake/deposit is burned according to the system rules. Anyone can submit a batch, but such an open approach is not efficient and secure. There are other options too, for example, a sequencer auction can be held at regular intervals to determine the next sequencer for a fixed period of time. The sequencer can also be randomly chosen from a set of stakeholders that are holding a stake in a PoS mechanism. An inadequately performing sequencer can be voted out by the token holders on the network, and a new auction can be held again.

Types of rollups

There are two types of rollups that vary depending on the security model they are based on: optimistic rollups and **zero-knowledge** (**ZK**) rollups.

Optimistic rollups

Under this model it is assumed that transactions are valid by default unless challenged by a fraud proof. The fundamental idea is that the rollup provider posts the transaction data on the chain and waits for a few days. If someone (an external validator) complains and proves via a fraud proof that the posted transaction is invalid, the rollup provider gets "slashed" (loses its stake) and the transactions are rolled back.

Optimistic rollups use a sidechain that runs in parallel to the main Ethereum chain. The key idea is to move computation and storage off layer 1. After a transaction is executed on L2, the optimistic rollup proposes a new state to the L1 or notarizes the transaction. The transaction data from L2 is written to L1 as *calldata.* The rollup system compresses many transactions into a batch to reduce the amount of data posted on the main L1 chain and thus the cost (in fees).



"Calldata" is a data area in Ethereum smart contracts used to pass arguments to a function. Its behavior is similar to memory. It persists on the Ethereum blockchain as history logs but is not part of Ethereum's state and is therefore an inexpensive method of storing data on-chain.

The system utilizes fraud proofs, meaning that if a verifier (someone on the network) notices a fraudulent transaction, the optimistic rollup network will execute a fraud proof and run the transaction's computation using the available state data (data validity). In this case, the gas consumed to run a fraud proof is reimbursed to the verifier.

Optimistic rollups are more suitable for general-purpose EVM computations because of their ready compatibility with EVM and low cost.

Transactions are processed off-chain but are posted on-chain to the rollup contract. The funds in an optimistic rollup system are stored in a smart contract on the Ethereum L1 main chain. This smart contract is responsible for handling fund deposits and withdrawals, signing up aggregators, and committing fraud proofs. Anyone can sign up as an aggregator by staking a bond in the smart contract.

This is how the process generally works:

1. A user deploys a smart contract/transaction off-chain to an aggregator (block producer or rollup provider).
2. The new smart contract/transaction is created locally at an aggregator. At this point, the local state of the aggregator has advanced to the new state.
3. The aggregator calculates the new state Merkle root.

4. The aggregator creates a new Ethereum transaction containing the new state root. The new state contains elements such as accounts, balances, and contract code. The operator submits both the old and new state root and if the old state root matches the existing state root, the existing one is discarded, and the new state is applied.
5. If an aggregator deploys an invalid transaction, an observer can challenge this by posting the valid state root and the Merkle state root as a proof (i.e., a fraud proof). If the fraud proof is correct, the aggregator is penalized along with any others that accepted the block or earned rewards due to the fraudulent transaction. This penalization can take the form of stake removal (slashing) and/or aggregator/node removal.
6. If the fraud proof is accepted, the layer 2 chain is rolled back to the last good configuration.

Optimistic rollup systems are composed of on-chain (layer 1) smart contracts and off-chain (layer 2) virtual machines. As the transaction data is posted and available on-chain, anyone can verify the correctness of state transitions and challenge via fraud proofs if they are found to be inaccurate.

As rollup providers are centralized due to aggregators/sequencers being standalone entities, there is some risk of transaction censorship and denial of service by withholding state data. This risk is, however, somewhat mitigated by layer 1's security guarantees and the fact that transaction data (state data) is available on the main chain. This means that if a rogue or otherwise faulty rollup provider goes offline, another node can take over that role and produce the rollup's last state to continue block production. On-chain data availability also enables users to construct Merkle proofs proving their ownership of the funds, which allows them to withdraw their funds from the rollup. Moreover, users have the option of submitting transactions (usually for a higher fee) directly to the L1 rollup contract instead of a sequencer, which forces the sequencer (aggregator) to include the transaction within a time limit to keep producing blocks. As L1 is the final settlement layer where a transaction is only considered final if the rollup block is accepted on the Ethereum main layer, users can be confident that L1 provides sufficient security guarantees for the safety of their funds.

Advantages of optimistic rollups include the fact that anything you can do on L1, you can do with optimistic rollups because optimistic rollups are EVM and Solidity compatible. Optimistic rollups are also secure because all transaction data is stored on the L1 main chain, which means that they are secure and decentralized due to the security properties of the main chain.

Disadvantages include long wait times for on-chain transactions due to potential fraud challenges. It can also be vulnerable to attacks if the value in an optimistic rollup exceeds the amount in an operator's deposited bond.

ZK-rollups

The key idea behind ZK-rollups is that a node on layer 2 processes and compresses transactions and produces a proof of correctness that is submitted to the blockchain. Verification of this proof is much faster than running and verifying all transactions. So, the insight here is that it doesn't matter if the main chain is slow and can handle only seven transactions per second. If each transaction contains thousands of transactions within it, then processing a single transaction on the main chain actually means processing thousands of transactions. This is how scalability is achieved.

The main difference between optimistic rollups and ZK-rollups is that ZK-rollups use zero-knowledge validity proofs whereas optimistic rollups use fraud proofs. The key idea is similar to optimistic rollups in that the rollup provider posts transaction data on the layer 1 chain, but with a zero-knowledge proof that proves the correct execution of the transactions. This zero-knowledge proof guarantees data validity. ZK-rollups compress a large amount of computation and verification into a single operation. ZK-rollups use this technique to bundle many transactions that are executed on the L2, into a single L1 mainnet transaction via a smart contract located on L1. From the data submitted, the smart contract verifies all the transfers that are included in the rollup. The key advantage here is that verifying the transactions doesn't require all of the relevant data, just the zero-knowledge proof. Transactions are written to Ethereum as *calldata* to reduce gas fees.

ZK-rollups consist of two types of actors. The transactors, which create and broadcast the transactions, and the relayers (validators), which collect transactions and create rollups. The relayers create the ZK-ZNARK or ZK-STARK proofs, which are basically a hash that represents the difference between the old and new state. Anyone can become a relayer as long as they meet the rollup-contract-specific requirements of the stake amount.

Zk-rollups are usually better for simple payments (token transfers) and simple exchanges because they do not perform general-purpose computations efficiently. This is because they are not readily compatible with the standard EVM and thus cannot perform general-purpose (Turing-complete) computations efficiently. However, great efforts are being made to achieve this, such as the development of ZK-EVM. Several projects working on building ZK-EVMs are zkSync, Applied ZKP, Scroll, Polygon Hermez, and Polygon Miden.

The key idea behind ZK-rollups is depicted in *Figure 17.5*. Instead of users sending their transactions to the layer 1 main blockchain, they send them to an entity called a rollup provider or rollup coordinator:

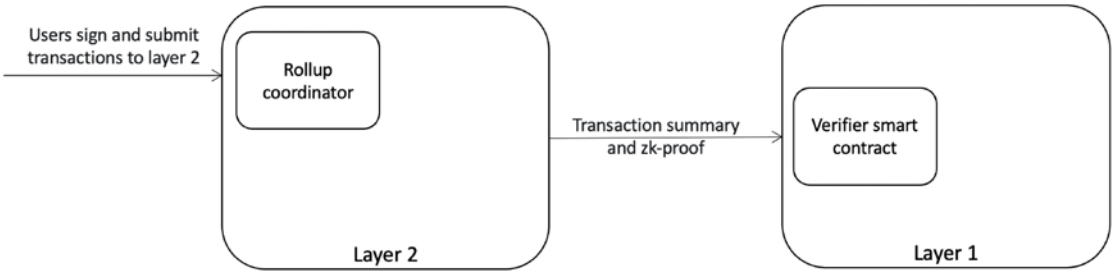


Figure 17.5: The idea behind the ZK-rollup-based scalability solution

The rollup coordinator verifies, executes, and compresses many (usually thousands of) transactions into a single proof that is published on-chain and is verified by the verifiers (miners). This proof is much quicker to verify compared to running the execution itself on-chain. The proof is computed using techniques such as SNARKs and STARKs.

Similar to optimistic rollups, the ZK-rollup system is composed of on-chain L1 rollup smart contracts and off-chain virtual machines. The transactions are finalized and settled on Ethereum layer 1 and are only settled if the L1 rollup contract accepts the validity proof submitted by the layer 2 coordinator (prover). As this proof proves the integrity of the transaction and all transactions must be approved by layer 1, this removes the risk of rogue operators trying to submit fraudulent transactions.

Technologies used for building ZK-rollups

There are two technologies that are used to build ZK-rollups, including ZK-SNARKs and ZK-STARKs.

ZK-STARKs are transparent, meaning no trusted setup is required. They are scalable, as proving time is quasi-linear and verifying time is logarithmic. The setup is succinct, requiring at most a logarithmic size. The proof size is somewhat large. It is also post-quantum secure. Key examples of STARK-based layer 2s are StarkNet and Polygon Miden. STARKs generate proofs faster and are post-quantum secure. As the proof is larger in size, they require more calldata space – roughly around 50KB. Thus, they also require more gas to verify and are more expensive for users. Also note that STARKs are based on 80-bit security, meaning a 1/1000 chance of proof forging, which can be a big problem, given that these systems are expected to be used for financial dApps potentially handling millions and billions of dollars. Therefore, this needs to be looked at carefully before choosing a STARK-based rollup solution.

ZK-SNARKs are non-interactive, meaning there is only one message posted by the verifier. They are succinct, meaning logarithmic verifying times. They require a trusted setup, which can take linear time or more to do. The proof size is smaller. They are not quantum secure. Some key examples of ZK-SNARK-based systems are Aztec, Loopring, zkSync, Hermez, and Polygon Zero.

The following table shows a comparison between SNARKs and STARKs:

	Proof size	**Prover time**	**Verification time**
SNARKs	Small	Medium	Small
STARKs	Large	Small	Medium

Let's now see how SNARKs and STARKs are used to create proofs.

Think about a program that always terminates – let's call it c. This program c takes a public input called x, and a private input called w. The SNARK/STARK system is composed of a prover, a verifier, the program c, and a proof π. The prover knows the program c, the public input x, and the private input w. The verifier only knows the program c and public input x. The prover produces a short proof that proves that it knows a witness (private input) w that makes the program output some expected value. The verifier runs a verification process on the proof, which convinces the verifier that the prover indeed knows a witness (private input) w that, when input into the program c with public input x and executed, outputs the expected value. The key point here from the scalability point of view is that the verifier's running time to verify the proof is a lot less than the execution of the program itself.

In other words, the prover time is linear to the program whereas the verifier time is sublinear:

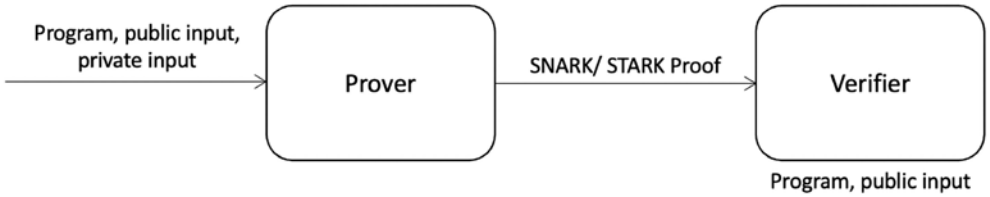


Figure 17.6: SNARK/STARK fundamental idea

What does a rollup provider do? Fundamentally, rollup providers maintain Merkle trees, which are comprised of accounts and consequently represent a state with a Merkle root. This Merkle root is posted onto the layer 1 main blockchain verified by the verifiers. When the rollup provider executes all the transactions submitted to it, it creates a new Merkle root as a new state. At this stage, the old Merkle root is no longer effective. The rollup provider creates a SNARK proof and submits that to the layer 1 blockchain along with the transaction summary. On layer 1, the verifiers (miners) verify the proof.

The new Merkle root is the result of applying a sequence of valid transactions to the old Merkle root. Here we can write program c (recall program c!) that checks for that. Here the public input x is the old Merkle root, new Merkle root, and the transaction summaries. The private input to program c is the account balances (state) of all the accounts that changed due to the transaction execution, the old account balances , and the digital signatures that signed those transactions. Program c will run all the checks, including signature validations, account balance checks, and transaction verification checks and ensure that the executions are correct. Basically, this checks that the new root is valid and consistent with the old root after the transaction execution was completed. Next, the rollup provider posts the updated new Merkle root to the layer 1 blockchain along with the SNARK proof that the new root is consistent with the old root according to the transaction the rollup provider received and executed. The data posted to the blockchain includes the Merkle root, SNARK proof, and a summary of the transaction data excluding transaction signatures. Miners now simply record the Merkle root and verify that the ZK-SNARK proof is valid. Now the verifiers have basically verified the new state root, which was created as a result of executing many (usually thousands of) transactions on layer 2, i.e., the rollup provider. This is how the massive scaling in transaction throughput is achieved.

ZK-rollups' architecture is largely based on two key components: on-chain contracts and off-chain virtual machines. The ZK-rollup protocol is managed by smart contracts running on the layer 1 Ethereum main chain. The ZK-rollup contracts are responsible for rollup block storage, tracking deposits and withdrawals, monitoring state updates, and verification of the zero-knowledge proofs submitted by the block producers from layer 2. The off-chain virtual machine on layer 2 is responsible for transaction execution and state storage. Here, the validity proofs verified on the Ethereum main chain provide a cryptographic guarantee of the correctness of the state transitions on the off-chain layer 2 VM.

ZK-rollups rely on Ethereum's main chain for data availability, transaction finality, and censorship resistance. Note that only the execution has been offloaded to layer 2 – decentralization and security are still based on the Ethereum main chain.

What happens if the rollup provider fails? As rollup providers are trustless entities, users can find new rollup providers. However, the issue is that these new rollup providers don't know anything about the state information (account balances, etc.) held by the previous rollup provider. So, how can the new rollup provider get the state of the old provider? More generally, the problem domain concerns how we transfer assets/state/account data to and from layer 2. If the old rollup provider (also called the coordinator) is no longer available, how do we make the old rollup provider's data available to the new rollup provider? For this purpose, there are two solutions, on-chain data availability and off- chain data availability. An example of an on-chain data availability solution is zkSync and an off-chain data availability solution is zkPorter. We introduce these solutions next.

In zkSync, all transaction summary data is stored on the layer 1 blockchain. When the new rollup provider starts up, it can read all transaction information from layer 1 that the old rollup provider processed before failing and can reconstruct the entire state (world state) from this data. Once the new rollup provider has the latest state, it can start working as the new rollup provider just as the old one was. This is a safe way to make data available again – a safe solution (due to the L1 security guarantees) to the data availability problem, however, the disadvantage here is that rollup providers have to write large amounts of transaction data to the layer 1 chain, resulting in higher transaction fees. As it's a safe but expensive method, it is recommended for high-value crypto digital assets.

In zkPorter, the transaction summary data is not posted to layer 1, but on a separate new blockchain (off-chain). This mechanism is much cheaper than zkSync. This seperate blockchain is there for storing transaction data only and is managed by coordinators under a proof-of-stake mechanism or some other cryptoeconomic incentive mechanism. This mechanism is less safe than zkSync because it doesn't have layer 1 security guarantees, but is cheap because it doesn't require posting and storing data on the layer 1 chain. This means that when we need transaction summary data, it may not be available. Even though the chances of this happening are minimal, the data availability guarantee is still not as strong as that of layer 1. When there is a need to reconstruct state data/world state/account state again, these coordinators on the network provide the transaction summaries, which are then used to reconstruct the state from scratch e.g., on a new coordinator, thus effectively solving the data availability problem.

There is a choice for the user here: they can use either on-chain data availability or off-chain data availability depending on the security and use case requirements.

In summary, ZK-rollups are tools of choice to enable high transaction throughput by executing the computations off-chain and using zero-knowledge proofs to verify these executions on-chain. As verification of transactions is faster than execution, such techniques provide a considerable increase in transaction throughput as measured in transactions per second (TPS). The future of Ethereum is rollup-centric and several upgrades are in the pipeline to remove performance bottlenecks and high costs and make Ethereum more rollup-friendly. We covered this rollup-centric roadmap in *Chapter 13*, *The Merge and Beyond*.



Technically, ZK-rollups could be (should be) called validity rollups because they do not use ZK for privacy, only for scalability, and the ZK acronym could give the incorrect impression. Rollups that provide privacy should be called ZK-rollups or private rollups. Remember that ZK-rollups use ZK for data integrity rather than for data confidentiality.

Pros of ZK-rollups: No delay, efficient, faster finality, cryptographically secure, and less vulnerable to economic attacks. Due to cryptographic validity proofs, the correctness of off-chain transactions can be guaranteed, in turn guaranteeing security, censorship resistance, and decentralization because it stores the data needed to recover the off-chain state on layer 1. Faster withdrawal time.

Cons of ZK-rollups: ZK-rollups are limited only to simple transfers as they are not compatible with EVM. This is due to the reason that validity proofs are computationally heavy and in order to efficiently process them, ZK-rollups have to build their own programming language. The introduction of a new custom language means ZK-rollups are not Solidity-compatible. However, to alleviate this issue, some work on building Solidity to ZKP language compilers (transpilers) is already in progress. For example Warp, which is a Solidity to Cairo (StarkNet ZKP language) transpiler. ZK-rollups are not efficient enough to run smart contracts. It is better to use ZK-rollups for applications with a simple, high-volume transaction rate such as exchanges. They are suitable for applications with little on-chain activity. Computation and verification of validity proof is costly, which can result in higher fees. EVM compatibility is not easy to achieve. Producing validity proofs can need special high-end hardware, which could result in centralization due to resource requirements, where only a few parties are able to afford such expensive hardware. Centralized operators could possibly censor or reorder transactions to their advantage. SNARKs require a trusted setup, which can raise concerns about security.

The key difference between rollups and sidechains is that sidechains are based on their own assumption of security and run their own validators, whereas rollups inherit their security and trust from layer 1. Sidechains can be seen as another layer 1 chain, thus suffering from the blockchain trilemma, trading off decentralization for scalability. Moreover, sidechains rely on trusted bridges (usually multisig) to connect with the main chain (Ethereum). Sometimes these bridges are also somewhat centralized where a majority of multisig signatures are controlled by members of the same group, e.g., foundational members. Rollups use trustless bridges with better trust assumptions.

ZK-rollups are good for basic transactions but are not general-purpose. They are good enough to do specific basic transactions within the rollup system and post proofs for verification on the main chain, but what about running general-purpose programs like Solidity smart contracts inside the rollup? It is possible, by using Zero-Knowledge **EVMs** (**ZK-EVMs**). This means that the rollup provider can produce proof of correct execution of a Solidity program. This is done the same way that standard Ethereum proof of balance updates are created, meaning that proof of state transitions that occurred due to a Solidity program execution resides on a leaf in the Merkle tree.

Proving off-chain deposits, transfers, and withdrawals is very useful, but it is not general-purpose. We still cannot prove an arbitrary execution like a smart contract. If we can somehow prove an off-chain EVM execution's correctness, then you can imagine how many different use cases it would enable. This is exactly what ZK-EVMs try to do, which we'll explore next.

ZK-EVM

A **ZK-EVM** is a specific type of ZK-rollup. Instead of generating proof of some application-specific logic, as is the case with normal ZK-rollups, ZK-EVMs produce proof of the correctness of an EVM bytecode execution. This means that standard Ethereum layer 1 EVM programs can run on a ZK-EVM and the ZK-EVM will produce the proof of correct execution of EVM code. A ZK proof of valid state transition on layer 2 is published on layer 1.

We can define a ZK-EVM as a layer 2 VM that executes smart contracts as zero-knowledge proof computations. It aims to replicate the Ethereum system as a ZK-rollup to scale Ethereum without requiring any changes to code or the loss of compatibility with existing Ethereum development tools.

The general high-level architecture of a ZK-EVM is shown in *Figure 17.7*:

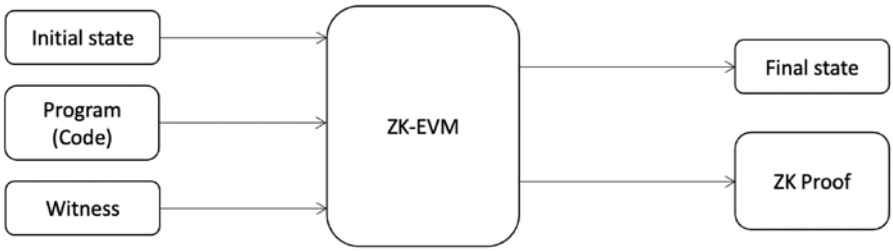


Figure 17.7: ZK-EVM high-level architecture

Remember, the key idea here is to prove EVM bytecode execution. The following are the steps involved:

1. First, layer 2 executes the transactions and as a result state changes occur.
2. The ZK-EVM extracts the previous state, executions, and new state and calculates the difference between the old and new state.
3. The ZK-EVM runs and produces a trace of execution. The trace of execution usually contains the execution steps, block header, transaction data, smart contract bytecode, and Merkle proofs.
4. The prover takes the trace and produces a proof. This is where arithmetic circuits are built, such as the EVM circuit, RAM circuit, and storage circuit. Each circuit produces a proof and is then aggregated into a smaller circuit. Eventually a final proof is produced.
5. This proof is published on the main chain.

This process allows all layer 2 transactions (possibly thousands) to be represented as a single transaction on the main chain.



The key idea here is that ZK-EVMs scale the transaction throughput by multiplexing (consolidating) layer 2 blocks and transactions into layer 1 as a transaction.

Designing a ZK-EVM is difficult. It requires not only refactoring of the EVM but also refactoring of the Ethereum state transition mechanism to make it zero-knowledge friendly.

There are two ways to do this. The first is **native ZK-EVM** or **bytecode compatible**, which creates circuits for all EVM opcodes. The other option is to build a new customized **Virtual Machine** (**VM**) and a programming language for the new VM t. It's not easy to build either the native ZK-EVM or a customized virtual machine, however, the native ZK-EVM is more difficult to build. The native ZK-EVM is more developer-friendly due to compatibility with Ethereum tools that already exist and that developers are familiar with, while customized EVMs require learning new programming languages and tools, which is not as much of a developer-friendly option. As native ZK-EVMs inherit the security model of the standard EVM as well as Solidity, a known, comparatively mature language, they are more secure, whereas customized EVMs may not offer the same level of security.

Scroll, ConsenSys, and Applied ZKP are examples of native (bytecode-compatible) EVMs whereas StarkNet and zkSync are two prime examples of customized VMs. In all these options, the key requirement is that proofs must be generated quickly and at minimal cost. The key benefit of ZK-EVMs is that we can reuse the already deployed bytecode on layer 1 without changing any code, which makes already deployed dApps immediately more scalable simply by redeploying them on layer 2. Moreover, we can use existing tools and languages like MetaMask, Truffle, and Solidity for writing smart contracts on layer 2, just as if we were on layer 1.

ZK-EVMs allow developers to write smart contracts using the same tools and languages used on layer 1, e.g., Truffle and Solidity, and also allow reuse of already deployed bytecode. Both features are allowed while still guaranteeing state integrity.

The diagram in *Figure 17.8* shows various approaches that native EVMs and customized EVM projects are taking to build ZK-EVMs:

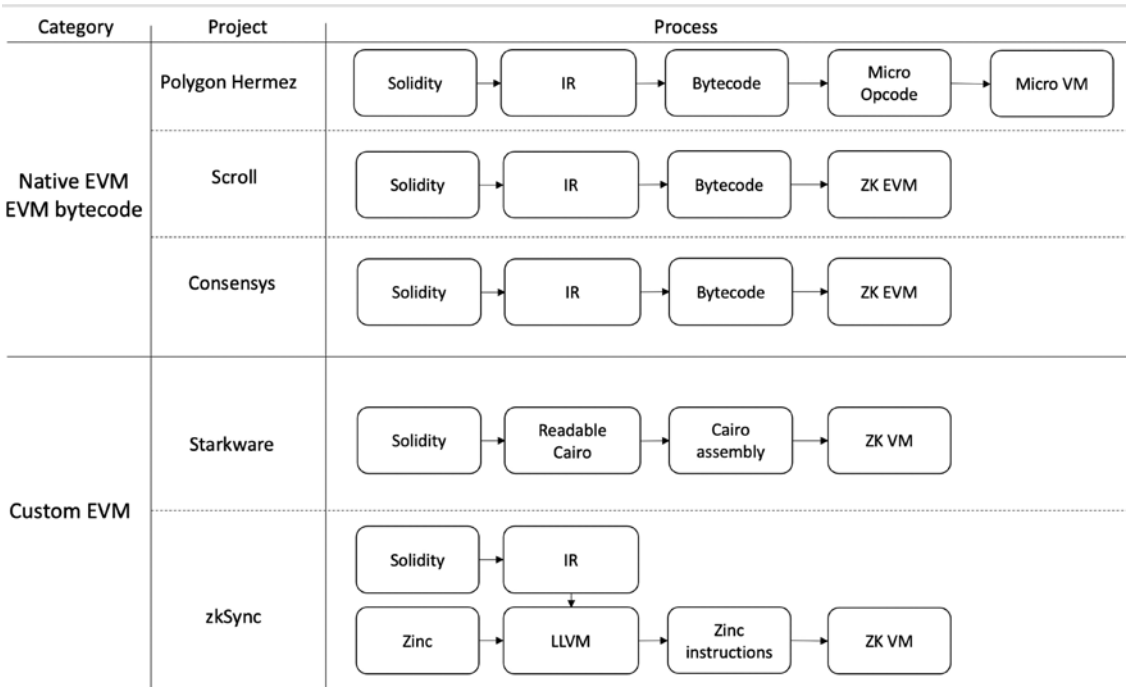


Figure 17.8: Different approaches to building ZK-EVMs

ZK-EVMs can be divided into three broad categories based on the technique they use to interpret the code and produce proofs:

- **Transpiler based** – In this type, an EVM language such as Solidity is trans-compiled (transpiled) into a SNARK-compatible VM. StarkWare and Matter Labs use this approach.
- **Direct interpretation based** – In this approach, EVM bytecode is interpreted directly to produce a proof of correct execution of bytecode. This technique is used by Hermez, Scroll, and ConsenSys ZK-EVMs.
- **Full EVM equivalence** – This approach aims for full EVM equivalence at the Ethereum layer 1 consensus level. This proves the validity of layer 1 Ethereum state roots. There is no concrete example of this variant yet.

There are also various types of ZK-EVMs based on different levels of performance and compatibility with the EVM. These types were proposed by Vitalik Buterin at `https://vitalik.ca/general/2022/08/04/zkevm.html`.

There is a dilemma here in terms of how much a ZK-EVM can be compatible with the EVM. We can call it the ZK-EVM tradeoff dilemma. The dilemma is that the more compatible a ZK-EVM is with the standard EVM, the less efficient it becomes. In other words, better performance (speed) means less compatibility with the standard EVM. This dilemma will become clearer by examining the types of ZK-EVM described as follows:

- **Type 1**: This type allows verification of Ethereum blocks natively. This type is fully equivalent (compatible) to Ethereum without any changes being made to any component of the Ethereum system, including state trees, any relevant data structures, compute logic, consensus logic, precompiles, or any other mechanism. EVM equivalence means that if the same set of transactions were input into both the EVM and ZK-EVM, the resultant states produced by both would be exactly the same, i.e., the same state root, storage, account state, balances, and any other data structures. As such, this type is compatible with all native Ethereum applications. However, it is not efficient, as proof generation takes a long time with this type.
- **Type 2**: This type is EVM-compatible but not Ethereum-compatible, meaning that it is equivalent to the EVM but differs from Ethereum core data structures like block structure, state tries, consensus, and other relevant mechanisms. It is fully compatible with existing applications but gives up Ethereum compatibility somewhat by modifying Ethereum's data structures to make proof generation faster and to make development easier. For example, Ethereum uses **Merkle Patricia Tree** (**MPT**), whereas a ZK-EVM could use sparse Merkle trees, which are more efficient data structures. Another example is that Ethereum uses the Keccak hash function, while the ZK-EVM could use the Poseidon hash function, which is more efficient. Another improvement could be to use only a single trie in the ZK-EVM system, instead of the multiple types of tries used in Ethereum. It is fully compatible with almost all Ethereum applications and can share most infrastructure, however proof generation is still slow.
- **Type 2.5**: This type modifies the EVM only by changing gas costs. It has fast proof generation times but introduces some incompatibilities.

- **Type 3:** This type is most equivalent to the EVM, but some tradeoffs are made to improve proof generation times and make development easier. These tradeoffs include some application incompatibility – for example, if a dApp uses precompiles, it would have to be rewritten because type 3 doesn't support these. Scroll and Polygon are two key examples of type 3 ZK-EVMs. Proof generation is fast in this type and is fully compatible with most Ethereum dApp.
- **Type 4:** This type of ZK-EVM compiles high-level language contracts written in Solidity or other languages into a specialized VM. It is not compatible with some Ethereum dApp and cannot share most of their infrastructure. However, this type has the fastest proof generation times, thus reducing costs and centralization risk.

Some key challenges that ZK-EVM implementations must solve are complexity and performance. ZK-EVM implementations are complex because creating and maintaining circuits (arithmetization) of EVM operations, i.e. opcodes can be challenging. Also, proofs must be generated as fast as possible with minimal cost, making achieving a high level of performance difficult. However, it is doable, and many reasonable examples are available. Proof generation time is usually linear to the program size, whereas verifier time is much lower than the program execution itself.

With ZK-EVMs, it is easy to migrate dApps from layer 1 to layer 2 zk-rollups. It also results in a reduction of transaction fees and increased program execution throughput because programs are now running on layer 2.

There are different ZK-EVM projects including Scroll ZK-EVM (`https://scroll.io/`), ConsenSys ZK-EVM (`https://consensys.net/blog/news/consensys-launches-private-beta-zkevm-testnet-to-scale-ethereum/`), zkSync (`https://zksync.io`), and Polygon ZK-EVM.

ZK-ZK-rollups

This is another type of rollup, fundamentally still a ZK-rollup, but providing privacy along with scalability. A prime example of this type is Aztec, who introduced the term ZK-ZK-rollup, meaning fully private rollups. .

Optimistic rollups vs ZK-rollups

The following table summarizes the differences between optimistic rollups and ZK-rollups:

Property	Optimistic	ZK-rollups	ZK-ZK-rollups
Confidentiality	No	Not inherently; ZK only used for scaling	Yes
Anonymity	No	Not inherently; ZK only used for scaling	Yes
Complexity	Low	High	High
General-purpose execution–EVM compatibility	Yes, easier to do	Hard to do	Hard to do

Withdrawal time	Approx. 1 week as network participants must be given time to verify transactions/ batches	Fast, as soon as the next batch is submitted by the rollup provider, usually less than 10 mins	Fast, as soon as the next batch is submitted by the rollup provider, usually less than 10 mins
Performance / efficiency	High	Lower comparatively	Lower comparatively
Verification efficiency	Fraud proof is quick to verify once submitted, however, the system relies on verifiers to monitor the chain to detect fraud and submit fraud proofs, which can be a slow process.	Quick to verify	Quick to verify
Security	Crypto-economic incentives	Cryptographic proof	Cryptographic proof
Finality	Days (single confirmation)	Minutes	Minutes
Usability	Easy to migrate apps from L1 to L2	Not easy to build apps on due to mathematical complexities, but compilers and DSLs are available to create circuits, e.g., Cairo	Not easy to build apps on due to mathematical complexities, but compilers and DSLs are available to create circuits, e.g., Noir
Privacy	No	No	Full
Transaction costs	Lower as no high-end hardware is usually required, and limited data is posted on-chain. However, note that at scale, e.g. more than 100 TPS, optimistic rollups tend to start to cost more and ZK-rollups start to cost less.	Higher due to on-chain proof verification and high-end hardware required to generate proofs	Higher due to on-chain proof verification and high-end hardware required to generate proofs
Trusted setup	No	Yes	Yes

Fraud and validity proof-based classification of rollups

From another angle, depending on whether validity proofs or fraud proofs are used and what mechanism is used for data availability, we can broadly divide L2s into the following four categories:

- **ZK-rollups** – These use validity proofs with data kept on the L1 Ethereum chain and utilize ZK-SNARKs/STARKs.
- **Optimistic rollups** – These use fraud proofs with data posted on the L1 Ethereum chain.
- **Validium (validia)** – Validia (validiums) are scaling solutions that use validity proofs such as ZK-rollups utilizing ZK-SNARKs or ZK-STARKs to guarantee transaction integrity and do not store transaction data on the Ethereum L1 chain. If data is kept off-chain, it can result in data availability issues, but this is seen as a tradeoff to achieve a high level of scalability – usually around 10K transactions per second. Moreover, multiple chains can run in parallel to achieve further increases in throughput and scalability.
- **Plasma** – This is a separate blockchain anchored with the layer 1 Ethereum chain. It executes transactions off-chain using its own block validation mechanism and transaction data is kept off-chain. It uses fraud proofs to mediate disputes.



In the future, we expect to see hybrid solutions using a mix of the above techniques. Some are good for certain specific use cases, and some for others.

We can visualize this classification in the following table:

Where is data	SNARK/STARK	Fraud proofs
Data on-chain	ZK-rollup	Optimistic rollup
Data off-chain	Validium	Plasma



Even though **ZK-ZK-rollups** are basically **ZK-rollups**, they also provide confidentiality, which is why they are categorized separately here.

Remember: the key difference between the security models of rollups and sidechains is that rollups inherit security guarantees from Ethereum by posting state changes and transactions to the main Ethereum chain, whereas sidechains are independent and do not post state changes and transaction data to the Ethereum chain.

There are many solutions that can be categorized as layer 2. A selection is given in the following list, with a brief description of each:

- **Arbitrum** – An optimistic rollup aiming to feel exactly like interacting with Ethereum, but with a fraction of the L1 transaction cost.

- **Aztec** – An open source L2 network offering scalability and privacy for Ethereum. It enables inexpensive and fully private crypto payments using zero-knowledge proofs.
- **Base** – A layer 2 chain introduced by Coinbase based on the Optimism stack (OP stack).
- **Boba Network** – An L2 Ethereum scaling and augmenting solution based on optimistic rollups, forked from Optimism.
- **DeversiFi** – Enables access to DeFi on Ethereum. Users can invest, trade, and send tokens without paying gas fees.
- **Fuel v1** – An optimistic rollup with low transaction costs, high speed, and high throughput.
- **Gluon** – A layer 2 scalable trading engine built on top of Ethereum and offering low fees and high-frequency trading.
- **Immutable X** – A layer 2 solution for NFTs on Ethereum with zero gas fees, instant trades, and scalability for games, applications, and marketplaces.
- **Layer2 Finance** – Enables access to DeFi protocols for everyone, where users can aggregate their DeFi usage and save on Ethereum fees.
- **Loopring** – The same security guarantees as L1 Ethereum, but with scalability and throughput increasing by 1000x and transaction costs reduced to 0.1% of L1.
- **Metis Andromeda** – An EVM-equivalent scaling solution originally forked from Optimism.
- **OMG Network** – A value-transfer network for ETH and ERC20 tokens. Scales through centralized transaction processing and guarantees safety by decentralizing security.
- **Optimism** – Optimistic Ethereum is an EVM-compatible optimistic rollup chain. It aims to be fast, simple, and secure.
- **Polygon** – This is a layer 2 solution consisting of various sidechains that run concurrently with the main Ethereum chain. Several techniques are utilized by these sidechains to provide scalability, including ZK-rollups, Plasma chains, and optimistic rollups. Polygon has a native token called MATIC, which is an ERC-20 token and is used for governance, staking, and fees. Polygon is a suite consisting of the following scalability solutions:
    - **Polygon PoS** – A public PoS network achieving high transaction throughput and low fees by running transactions on sidechains. PoS ensures security through the Plasma bridging framework and decentralization through a PoS validator network.
    - **Polygon Edge** – Polygon Edge is a framework that allows the building of Ethereum-compatible public and private networks.
    - **Polygon zkEVM** – This is an EVM-compatible ZK-rollup.
    - **Polygon Avail** – This is a data availability solution that allows the ordering and storing of transaction data for other blockchains.
    - **Polygon Miden** – This is a STARK-based ZK-rollup.
    - **Polygon Zero** – This is an Ethereum-compatible ZK-rollup based on recursive proofs.
    - **Polygon Nightfall** – This is an optimistic rollup utilizing zero-knowledge proofs.
    - **Polygon Hermez** – This is a ZK-rollup optimized for secure, low-cost, and usable token transfers on Ethereum.

    - **Polygon supernets** – Allow the creation of custom, high-performance blockchains.

- **STARKs, StarkNet, and StarkEx** – StarkNet and StarkEx are two solutions provided by StarkWare to scale Ethereum. StarkNet is a general-purpose, permissionless, and decentralized layer 2 based on STARK-based ZK-rollups. StarkEx is an application-specific scaling engine developed using Cairo and SHARP. StarkNet, StarkEx, and related technologies are developed by StarkWare. These scaling technologies use STARKs to enable blockchain scaling by proving the computational integrity of the computations performed off-chain on layer 2.
- **ZKSpace** – A platform consisting of three key parts: 1) a layer 2 Automated Market Maker Decentralized Exchange – AMM DEX based on ZK-rollups called ZKSwap v3; 2) a payment service called ZKSquare; 3) an NFT marketplace called ZKSea
- **ZKSwap** – It is a fork of zkSync with added AMM functionality.
- **zkSync** – A user-centric ZK-rollup platform providing scaling for Ethereum. It supports payments, token swaps, and NFT minting.
- **zkSync 2.0** – This is a new version of zkSync that combines zk-rollup and zkporter.

In summary, rollups are the most promising and actively developed and used technology for scaling layer 1 blockchains.

Example

In this section, we'll see how Polygon works and how can we deploy a contract to the Polygon layer 2 network.

Polygon PoS

Polygon is a suite of scalability solutions. Polygon PoS is a proof-of-stake commit chain anchored to Ethereum layer 1 through checkpoints.

Polygon PoS is composed of two core components, a Heimdall chain and Bor chain. The Heimdall chain works in combination with the stake manager contract deployed on Ethereum layer 1. The stake manager contract is responsible for validator and stake management. By having a stake manager contract on Ethereum, the security of the mechanism improves, because now there is no need to trust the validator running on the Heimdall chain, and you can rather rely on the security of layer 1. The Heimdall chain creates checkpoints on the Ethereum mainchain periodically. This checkpointing ensures finality on the Ethereum main chain.

The Bor chain component is responsible for block production by collecting transactions into each block. Block production is carried out by a subset of PoS validators from the Heimdall network. A subset of validators is selected for producing blocks for a set period of time or number of blocks, called a span. After the span elapses, the validator selection process starts again. Validators are selected based on the amount of MATIC tokens they have staked. The greater the stake, the greater the number of slots the validator is allocated to. However, the shuffling of the validator set coupled with a Tendermint-based proposer selection mechanism allows the algorithm to select validators in such a way that everyone gets a chance in a span as block producer, even if their stake amount is lower than others.

Once the blocks are produced by the Bor layer, the Heimdall layer aggregates them into a single Merkle root and posts it onto the Ethereum layer 1 chain at regular intervals. The proposers for checkpoints are selected via Tendermint in a weighted round-robin manner.

Polygon is thus secured by its own proof-of-stake mechanism and with its own native token called MATIC.

A high-level architecture of Polygon PoS is shown in *Figure 17.9*:

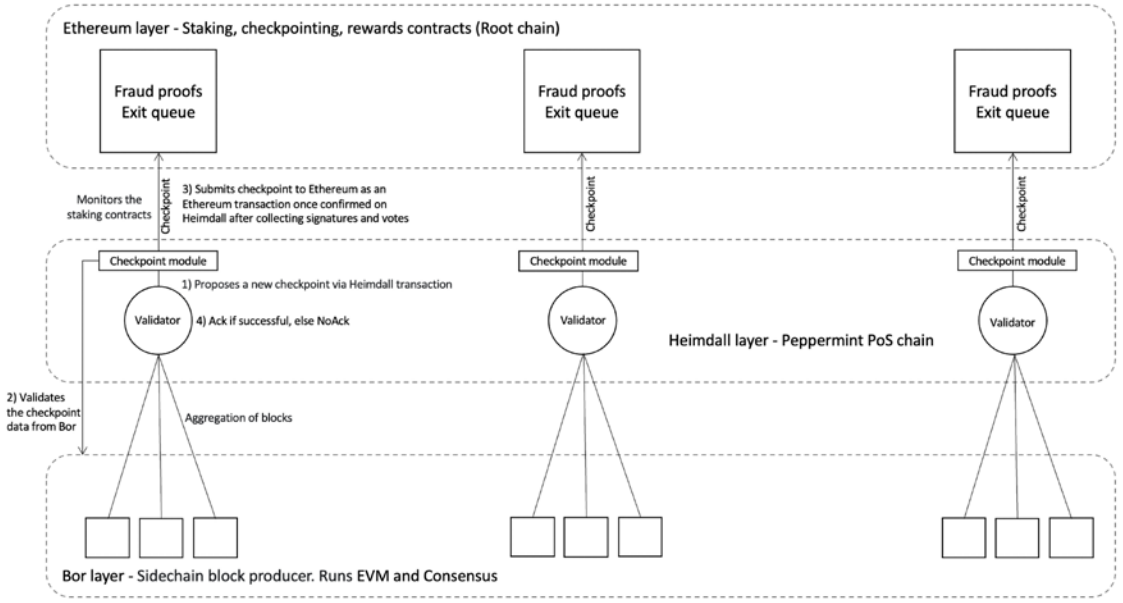


Figure 17.9: High-level Polygon architecture

The preceding diagram shows the Ethereum main chain layer 1 at the top with PoS nodes in the middle and the Polygon sidechain at the lowest layer.

As an example, we can implement on Polygon the ERC-20 token called MET that we built in *Chapter 15*, *Tokenization*. As the Ethereum standard toolchain works with Polygon, all we need to do is simply connect MetaMask to the Polygon test network, and then deploy it through the Remix IDE. I won't go through these instructions step by step, as it is quite straightforward and we've seen already how to set up, configure, and use MetaMask in earlier chapters. All we need to do is to add the Polygon network details under the custom RPC option in MetaMask and deploy the contract as you would normally on Ethereum. The details of the Polygon test network are shown below:

```
Network name: Mumbai
Network URL: https://matic-mumbai.chainstacklabs.com
Chain ID: 80001
Currency symbol: MATIC
```

As an alternative, you can browse to `https://chainlist.org/`, which hosts a list of EVM networks and enables users to add networks easily to their MetaMask browser, without requiring these details to be added manually via a custom RPC in MetaMask.

You will also need some test tokens to be able to deploy the smart contracts. In order to get test MATIC tokens, visit `https://faucet.polygon.technology/` and enter your MetaMask wallet account address.

Once MetaMask is configured and connected and you have sufficient funds from Faucet, open the Remix IDE from `https://remix-project.org/` and choose **Injected provider – MetaMask** as the environment under the **DEPLOY & RUN TRANSACTIONS** item. Once Remix is connected to MetaMask, make sure the account is set up correctly. Once verified, simply compile and deploy the same way you did in *Chapter 15*, *Tokenization*, using the Remix IDE. As MetaMask is now connected to Polygon, deployment will be made on the Polygon Mumbai test net. Once deployed, you can explore further by using the block explorer at `https://mumbai.polygonscan.com`. With this example, we complete our introduction to layer 2 technologies.

A question arises here – with all these scalability solutions of various types, what will the future of blockchain scalability look like? Is it going to be rollup-centric, or multichain-based, or is layer 1 scaling the future? Only time will tell, but for now rollups are ahead in the race! At the same time, remember that there is no single best solution that is suitable for all use cases. It is likely that the future will contain all these scalability solutions working together to form a high-performance and efficient global ecosystem of blockchains.

Layer 3 and beyond

Multilayer solutions are expected to exist in future where there is a public layer 1 chain at the core of the ecosystem and a public layer 2 providing scalability. There is the possibility of another layer on top, the so-called layer 3, where multiple application-specific chains can run that inherit the security of the "core chain" (i.e., layer 1) through layer 2 – for example, one chain that provides privacy, another chain that's another rollup, another chain that's used for payments, and so forth.

Beyond that, we can imagine another layer, also an application-specific layer or another rollup layer, and this ecosystem can grow layer by layer. Layer 3 is conceived as an additional off-chain execution layer on top of existing layer 2 networks. There can even be an additional layer 4 and further layers to provide even more scalability, dubbed "hyperscalability."

Such a configuration could enable an interoperable ecosystem of chains and layers that could give rise to a heterogenous multichain layered network of networks, which would create an interoperable inclusive environment of services and economy. Therefore, layer 3 can also be imagined as an interoperable layer between different blockchains.

Ideas such as fractal scaling or hyperscaling have been proposed by StarkWare and others with different variations, but the general consensus is that it is possible to scale beyond layer 2 and the future of the blockchain ecosystem will largely be based on this vision.

We can visualize this concept as in *Figure 17.10*:

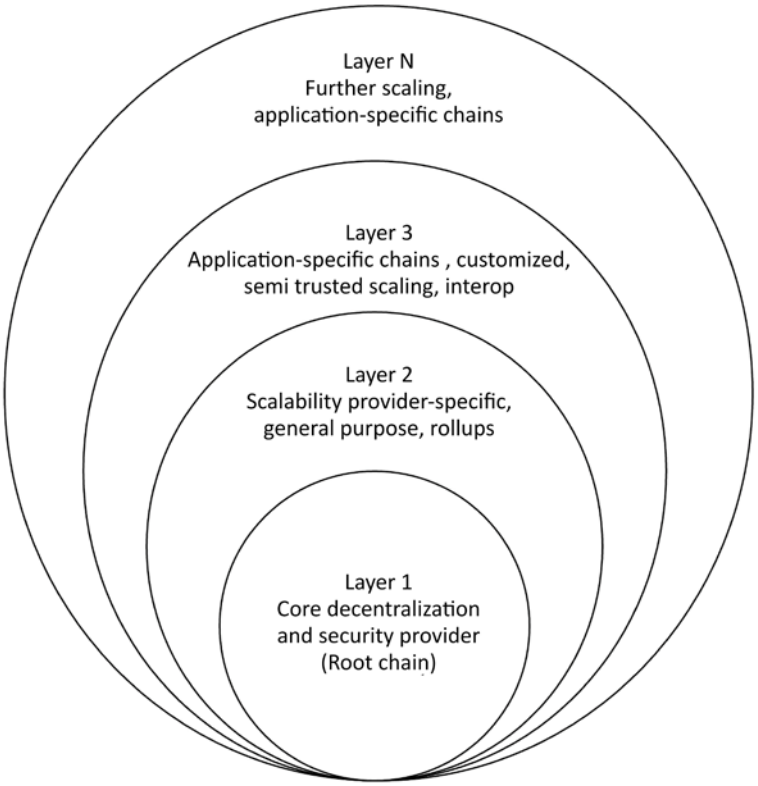


Figure 17.10: A vision of a scalability ecosystem

With this, we have completed our discussion of some of the many existing and potential scalability improvements for blockchain. In the next chapter, we'll discuss privacy and some techniques to achieve privacy in blockchain.

Summary

In this chapter, we introduced a major challenge that blockchain faces: limited scalability, along with several methods available to achieve greater scalability. Fundamentally, there exist layer 0, layer 1, and layer 2 solutions. Layer 2 can be divided into Plasma, state channels, rollups, and validiums. Layer 1 solutions are on-chain solutions such as DAG-based chains and blockchains based on faster consensus mechanisms such as Solana. Layer 0 solutions are multichain solutions such as Polkadot.

Note that while layer 2 solutions are quite promising and a lot of research and development is going on to make these even better, there are several security issues and relevant challenges that need to be addressed to ensure that these protocols are safe. We'll cover some key security challenges of layer 2 protocols in *Chapter 19*, *Blockchain Security*. In the next chapter, we will explore another very interesting topic – blockchain privacy.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

18

Blockchain Privacy

In this chapter, we'll cover privacy, which is an important topic. Data privacy or information privacy is not a new concept. Data privacy refers to the protection of personal and sensitive information and the rights of individuals to control how their personal information is collected, stored, used, and shared. While privacy has always been important, with the advent of the internet and online services, protecting personal information has become even more crucial than ever. In the established digital world, like the internet and other online services, there are many laws and regulations, along with strict information security policies and technical controls protecting users.

However, in the blockchain world, there is a sheer lack of such rigor so far. Achieving the same level of security that the typical digital world enjoys is still an issue that needs to be addressed. In this chapter, we'll see what privacy in the context of blockchain is and how we can introduce it using different protocols. Along the way, we'll look at the following topics:

- Privacy and its types
- Layer 0, Layer 1, and Layer 2 protocols for privacy on blockchain
- Zero-knowledge proofs, their various types, polynomial commitment schemes, and relevant protocols
- A practical example

So, let's begin; first we'll discuss privacy, why it's important, and its various types.

Privacy

Privacy in blockchain can be divided into two main categories based on the type of service required. These categories are *the anonymity of the users* and *the confidentiality of the transactions*. Anonymity is concerned with hiding the sender's or receiver's identity, whereas confidentiality addresses the requirements of hiding transaction values.

The fundamental reason why blockchains are not privacy-preserving is that every transaction in a blockchain needs to be verified and executed by every participant on the network. While this property of executing all transactions by all nodes gives blockchains their powerful integrity property and execution guarantee, this is also a weakness.

This weakness is the result of inherent transparency in blockchains. As all transaction data, including account details, inputs, outputs, and states are visible to anyone on the blockchain, privacy cannot be preserved.

One solution that comes to mind is that we could somehow encrypt the data, but if the values are hidden, then the transactions cannot be verified. The key requirement is to somehow combine public verifiability and confidentiality so that even if the data is encrypted in some form, it still can be publicly verified.

Let's now explain anonymity and confidentiality, before we delve deeper into what can we do to solve this problem.

Anonymity

Anonymity is desirable in situations where the identity of users is required to be hidden from other participants on a network. This can be due to regulatory requirements, enterprise requirements, or just due to the sensitive nature of the transactions. For example, in a business network, it might be desirable to hide the dealings between different participants on the network from other competitors to avoid friction or unjustified business competitive advantage.

The properties of *unlinkability* and *being untraceable* are used to achieve anonymity. The property of untraceability allows us to hide the trace of a transaction from one party to another in a network. On the other hand, unlinkability means that an observer is unable to deduce the link between transactions and their participants, or the relationships between transaction participants (senders and receivers). This means that if a third party looks at the transaction flow, they cannot see which entities are engaged in transactions with which other entities.

There are several blockchains that support privacy inherently and are built into the design of a system. The most popular is Zcash, which uses **zero-knowledge proofs** (**ZKPs**) to achieve anonymity. Other examples include Monero (`https://web.getmonero.org`), which makes use of ring signatures to provide anonymity services.

Confidentiality

Confidentiality is an absolute requirement in many industries such as finance, law, and health. Similarly, the privacy of transactions is a much-desired property of blockchains. However, due to its very nature, especially in public blockchains, everything is transparent, thus inhibiting its usage in various industries where privacy is of paramount importance, such as finance, health, and many others.

When we think about confidentiality we can divide it into two further optional requirements, **conditional privacy** and **selective disclosure.** Conditional privacy means that a system should have the ability to conditionally make data visible to a third party such as auditors, but keep the data hidden from all other parties except those who are privy to the transactions. Note that **unconditional privacy** is also not good in practice, because if everything is hidden from everyone it can permit criminal activities such as money laundering and terrorist financing, without anyone knowing about it. So, there is a need, especially in financial applications of blockchain, to ensure that in some cases a third party such as a regulator or an auditor is able to access and view data to ensure compliance with regulatory requirements.

The other confidentiality type is **selective disclosure**, where a system should have the ability to selectivity share some part of the data, such as age only, from a larger dataset containing personal information. Another variation of this is called **range proof**, which is the ability to prove that some data is in a range, for example, my ability to prove that I am older than 18 years of age or my salary range is between $10,000 and $20,000.

We will now look at some techniques that we can use to achieve privacy.

Techniques to achieve privacy

As the blockchain is a public ledger of all transactions and is openly available, it becomes easy to analyze it. When combined with traffic analyses, transactions can be linked back to their source IP addresses, thus possibly revealing a transaction's originator. This is a big concern from a privacy point of view.

In the Bitcoin domain it is a recommended and common practice to generate a new address for every transaction, which allows some level of unlinkability. However, this is not enough, and various techniques were developed and successfully used to trace the flow of transactions throughout the network and link them back to their originator. These techniques analyze blockchains by using transaction graphs, address graphs, and entity graphs, which facilitate linking users back to the transactions, thus raising privacy concerns.

These techniques are based on the idea that every transaction on the Bitcoin network (or any public blockchain) is publicly recorded and can be linked to other transactions, addresses, and entities. For example, transaction graph analysis involves creating a graph of all transactions and identifying patterns of transactions between addresses. Address graph analysis involves creating a graph of all addresses and their associated transactions. This type of analysis can help identify the relationships between different addresses and the flow of funds between them.

The techniques mentioned earlier can be further enriched by using publicly available information (for example, public internet forum users' Bitcoin addresses) about transactions and linking them to the actual users. There are open source block parsers available that can be used to extract transaction information, balances, and scripts from the blockchain database.

In this section, we'll describe different techniques to achieve privacy, including anonymity and confidentiality.

Similar to scalability solutions, we can also divide privacy solutions into three categories, based on the layer within the blockchain architecture stack in which they operate:

- **Layer 0** methods, or network layer methods, where the mechanism to achieve privacy operates at a network level.
- **Layer 1** methods, also called on-chain methods, where the blockchain protocol itself is enhanced to achieve privacy.
- **Layer 2** methods, or off-chain methods, where mechanisms that exist outside of the main blockchain are used to achieve privacy on the blockchain.

Layer 0

These solutions are network layer-level methods that provide anonymity by using TOR and I2P. These methods allow us to hide the identities of the parties involved.

Tor

Tor, **The Onion Router**, is a software that enables anonymous communications. More information on Tor is available here: `https://www.torproject.org`. We can use Tor to enable anonymous communication in cryptocurrency blockchain networks. Tor is a common choice to enable anonymous communication and has been used in Monero, Verge, and in **Bitcoin** to provide anonymity.

For Bitcoin, see `https://en.bitcoin.it/wiki/Tor`. Monero can also be run with Tor (`https://web.getmonero.org/`). Verge is another example of cryptocurrency making use of Tor for IP obfuscation.

I2P

I2P, the **Invisible Internet Project**, is an anonymous network built on the internet. It enables censorship-resistant peer-to-peer communication. It is used in Monero to provide anonymity services. More information on I2P is available here: `https://geti2p.net/en/`. Monero and Zcash support anonymous transactions. Monero makes use of ring signatures, stealth addresses, and confidential transactions, whereas Zcash uses zk-SNARKs.

As many of the techniques described in the following sections can be used at Layers 1 and 2—for example, ZKPs—we are not distinguishing between these layers. Instead, we'll only list these solutions and discuss what they can be used for. Furthermore, solutions for anonymity and confidentiality also overlap and techniques used to achieve confidentiality are also more or less applicable to achieving anonymity. We do, however, segregate Layer 0, the *network layer*, and we will describe some of the network-level approaches to achieve anonymity, before moving on to introduce a range of privacy mechanisms that are applicable to Layers 1 and 2.

Indistinguishability obfuscation

This cryptographic technique may serve as a silver bullet to all privacy and confidentiality issues in blockchains, but the technology is not yet ready for production deployments. **Indistinguishability obfuscation** (**IO**) allows for code obfuscation, which is a very ripe research topic in cryptography. If applied to blockchains, IO can serve as an unbreakable obfuscation mechanism that will turn smart contracts into a black box where the behavior of the obfuscated code is indistinguishable. In simpler words, the inner functionality of the smart contract is totally hidden.

The key idea behind IO is what's called by researchers a *multilinear jigsaw puzzle*, which basically obfuscates program code by mixing it with random elements, and if the program is run as intended, it will produce the expected output. However, any other way of executing it would make the program output random garbage data.

In other words, IO makes it computationally infeasible for an attacker to distinguish between two different program executions, even if the attacker has complete access to the program's code and input/output behavior.

In other words, IO provides a way to protect the privacy of a program's implementation, making it impossible for an attacker to determine the exact details of how the program executes or what inputs it was given.

This research paper is available at `https://eprint.iacr.org/2013/451.pdf`.

Homomorphic encryption

This type of encryption allows operations to be performed on encrypted data. Imagine a scenario where the data is sent to a cloud server for processing. The server processes it and returns the output without knowing anything about the data that it has processed. This is also an area ripe for research and fully homomorphic encryption, which allows all operations on encrypted data, is still not fully deployable in production; however, major progress in this field has already been made. Once implemented on blockchains, it can allow processing encrypted transactions without requiring to decrypt them, which will inherently allow the privacy and confidentiality of transactions.

For example, the data stored on the blockchain can be encrypted using homomorphic encryption, and computations can be performed on that data without the need for decryption, thus providing privacy service on blockchains. This concept has also been implemented in a project named *Enigma*, which is available online at `https://www.media.mit.edu/projects/enigma/overview/`, by MIT's Media Lab. Enigma is a peer-to-peer network that allows multiple parties to perform computations on encrypted data without revealing anything about the data. The research is available at `https://crypto.stanford.edu/craig/`.

Secure multiparty computation

The concept of secure multiparty computation is not new and is based on the notion that data is split into multiple partitions between participating parties, under a secret sharing mechanism, which then does the actual processing on the data without the need to reconstruct data on a single machine.

The output produced after processing is also shared between the parties. Multiple parties carry out the computation mutually without revealing their inputs. Only the output of the computation is revealed.

A secure multiparty computation platform called Enigma was proposed in 2015. The paper is available here: `https://web.media.mit.edu/~guyzys/data/enigma_full.pdf`.

Trusted hardware-assisted confidentiality

Trusted computing platforms can be used to provide a mechanism by which confidentiality of a transaction can be achieved on a blockchain, for example, by using Intel **Software Guard Extensions** (**SGX**), which allows code to be run in a hardware-protected environment called an **enclave**. Once the code runs successfully in the isolated enclave, it can produce a proof, called a **quote**, which is attestable by Intel's cloud servers.

It is a concern that trusting Intel will result in some level of centralization and is not in line with the true spirit of blockchain technology. Nevertheless, this solution has its merits and, in reality, many platforms already use Intel chips anyway, so trusting Intel may be acceptable by some in some cases.

If this technology is ap1plied to smart contracts, then once a node has executed the smart contract, it can produce a proof of correctness, thus proving successful execution, and other nodes will only have to verify it. This idea can be further extended by using any **Trusted Execution Environment** (**TEE**), which can provide the same functionality as an enclave and is available even on mobile devices with **Near Field Communication** (**NFC**) and a secure element. For example, Ekiden is a platform that makes use of Intel SGX's TEE to run smart contracts while preserving confidentiality. More information on Ekiden is available here: `https://arxiv.org/pdf/1804.05141`.

Mixing protocols

A **mixing protocol**, or mixer, is a service that allows users to preserve their privacy by mixing their coins with other users.

These schemes are used to provide anonymity to cryptocurrency transactions. In this model, a mixing service provider (an intermediary or a shared wallet) is used. Users send coins to this shared wallet as a deposit, and then the shared wallet can send some other coins (of the same value deposited by some other users) to the destination. Users can also receive coins that were sent by others via this intermediary. This way, the link between outputs and inputs is no longer there, and transaction graph analysis will not be able to reveal the actual relationship between senders and receivers:

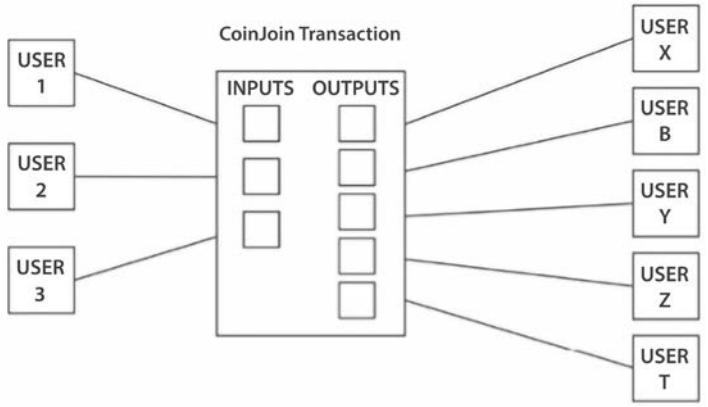


Figure 18.1: Three users joining their transaction into a single larger Coinjoin transaction

Coinjoin is one example of mixing protocols, where two transactions are joined together to form a single transaction while keeping the inputs and outputs unchanged. The core idea behind Coinjoin is to build a shared transaction that is signed by all participants. This technique improves privacy for all participants involved in the transactions.

Coinjoin is a technique that is used to anonymize **Bitcoin** transactions by mixing them interactively. The idea is based on forming a single transaction from multiple entities without causing any change in inputs and outputs. It removes the direct link between senders and receivers, which means that a single address can no longer be associated with transactions, which could lead to the identification of users.

Coinjoin needs to cooperate between multiple parties that are willing to create a single transaction by mixing payments. Therefore, it should be noted that, if any single participant in the Coinjoin scheme does not keep up with the commitment made to cooperate in creating a single transaction, by not signing the transactions as required, then it can result in a **DoS** (**Denial of Service**) attack.

In this protocol, there is no need for a single trusted third party. This concept is different from mixing a service, which acts as a trusted third party or intermediary between the Bitcoin users and allows the shuffling of transactions. This shuffling of transactions results in the prevention of tracing and linking payments to a particular user.

Various third-party mixing services are also available, but if a service is centralized, then it poses the threat of tracing the mapping between the senders and receivers. This is because the mixing service knows about all inputs and outputs. In addition to this, fully centralized miners pose the risk of the administrators of the service stealing the coins.

Various services, with varying degrees of complexity, such as CoinShuffle, Coinmux, and Darksend in Dashcoin, are available, which are based on the idea of Coinjoin (mixing) transactions. CoinShuffle is a decentralized alternative to traditional mixing services as it does not require a trusted third party.

Coinjoin-based schemes, however, have some weaknesses, most prominently the possibility of launching a **DoS** attack by users who committed to signing the transactions initially but now are not providing their signature, thus delaying or stopping joint transactions altogether.

CoinSwap

CoinSwap is a privacy mechanism that is based on the idea of **atomic swaps**. Atomic swaps allow two parties to exchange coins without requiring a trusted third party. CoinSwap can also be used for cross-chain swaps. CoinSwap works by utilizing a third party in the transaction flow and also requires private communication channels between all parties. This way, the addresses of the sender and the receiver cannot be linked. This third party receives funds from the sender and sends them to the receiver.

The technique here is that the third party pays funds to the receiver by using a totally different source of funds, thus disconnecting the link between the sender and the receiver. This disconnection between the sender and the receiver results in providing an unlinkable transaction and thus privacy. The sender uses multi-signature transactions (usually 2 of 2—sender and third party) to allow transactions to be spent, while the receiver also requires multi-signature transactions (usually 2 of 2—receiver and third party) for transactions to be spent. In other words, the sender and the third party will sign the transaction output to send the Bitcoin to the third party, while the receiver and the third party will sign the transaction output to send the Bitcoin to the receiver.

CoinSwap uses hash-locked transactions where a pre-image of the hash is required to unlock the transaction. Using hash locked transactions prevents the third party from stealing the Bitcoin. More information on CoinSwap can be found at the following link, where it was originally proposed by Gregory Maxwell: `https://bitcointalk.org/index.php?topic=321228`.

TumbleBit

The TumbleBit protocol was introduced in 2016. TumbleBit is fully compatible with the Bitcoin protocol. It is an anonymous, fast, and off-chain payments (unlinkability) protocol that allows parties to transfer funds via an untrusted third party or intermediary called a **tumbler**. In this protocol, even the tumbler is unable to deanonymize the payers and payees involved in a payment. It involves using two fair exchange protocols that prevent any malicious activity, such as cheating participants or the tumbler. TumbleBit relies on a protocol called **RSA puzzle solver**, which allows a payer to make payments to the tumbler. Unless tumbler solves this RSA puzzle, it cannot claim any Bitcoin paid by the payer. Another fair exchange protocol, called the **puzzle-promise** protocol, is used between the tumbler and the payee to claim the payment.

TumbleBit consists of three phases, as listed here:

1. Escrow phase, where all payment channels are set up.
2. Payments phase, where payers transfer funds.
3. Cash-out phase, where payers and payees close the payment channels.

More information on TumbleBit can be found here: `https://eprint.iacr.org/2016/575.pdf`. A proof of concept implementation of TumbleBit is available here: `https://github.com/BUSEC/TumbleBit`.

A recent innovation is the introduction of TumbleBit++, which is an improved mixing protocol based on TumbleBit. It provides anonymity and confidentiality of transaction amounts by combining confidential transactions and a centralized untrusted anonymous payment hub. More information regarding TumbleBit++ is available here: `https://link.springer.com/chapter/10.1007/978-3-030-31919-9_21`.

We will now discuss **Dandelion**, which provides inherent anonymity and is implemented by redesigning the network layer of the Bitcoin protocol.

Dandelion

In addition to the approaches mentioned previously, a recent proposal called **Dandelion** has also been made. Dandelion (the improved version is called Dandelion++) is a proposal that aims to make transactions on a Bitcoin network untraceable. This protocol will allow anonymous transactions to occur on the Bitcoin network as opposed to pseudonymous transactions, where an adversary, by using network analysis methods, can trace the transaction back to its source node. Consequently, the adversary can then discover the original IP address of the transaction sender.



Deanonymization of Bitcoin users is a known problem, and a number of research papers are available on this topic. For example, the paper *Deanonymization of clients in Bitcoin P2P network* is available at `https://arxiv.org/pdf/1405.7418.pdf`.

We can say that Dandelion is a mechanism to provide inherent anonymity for Bitcoin transactions because, once implemented, the P2P layer of the network Bitcoin protocol is modified in such a way that tracing transactions back to their source node and IP would become extremely difficult.

A Bitcoin improvement proposal is available at `https://github.com/bitcoin/bips/blob/master/bip-0156.mediawiki`. The original Dandelion proposal paper is available at `https://arxiv.org/pdf/1701.04439.pdf`. The Dandelion++ research paper is available at `https://arxiv.org/pdf/1805.11060.pdf`.

In the Bitcoin network, the **epidemic flooding** mechanism (Gossip protocol) is used for transaction propagation. On top of this mechanism, there is a somewhat effective method called **diffusion**, which is used to provide some level of anonymity. In this protocol, each node introduces independent and exponential delays for spreading transactions to its neighbors. This scheme results in reducing the symmetry of the epidemic protocol, which makes network analysis difficult and unreliable. However, this scheme is predictable and hence does not provide sufficient anonymity guarantees.

Dandelion proposes a new but backward-compatible routing mechanism. In this protocol:

- First, a privacy graph (anonymity set) is constructed. This phase is called the **private graph construction** phase. It is a sub-graph of the existing Bitcoin P2P network, and each node selects a subset of its outbound peers in this phase. This is where the **random line of nodes** is selected.
- The messages (transactions) are then routed through this privacy graph during the **Stem** phase. This is where the message is propagated on a random line of nodes.
- Finally, there is the **Fluff** phase, where the messages are routed (broadcast) to the entire network by diffusion.

In summary, the dandelion protocol is composed of an **anonymity** phase and a **spreading** phase. In the anonymity phase, this protocol spreads a message over a random line for a number of random hops. After this step, the message is broadcast over the whole network using diffusion. With this combination of random path selection and diffusion, the **Dandelion** protocol provides near-optimal anonymity guarantees.

This protocol can be visualized using the following diagram:

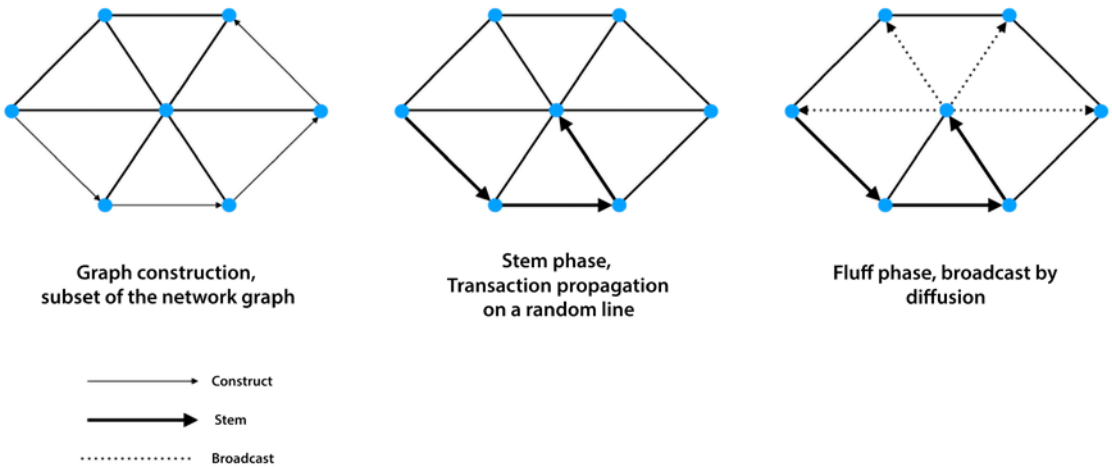


Figure 18.2: Dandelion protocol routing

In the preceding diagram, first, graph construction is performed, which is shown on the left-hand side. Then, in the middle, we have the transaction propagation along a random line of nodes. Finally, the broadcast, on the right-hand side, is shown. More information on Dandelion is available in the following paper: `https://arxiv.org/pdf/1701.04439.pdf`.

Confidential transactions

Confidential transactions make use of **Pedersen commitments** in order to provide confidentiality. Commitment schemes allow a user to commit to some value while keeping it secret with the capability of revealing it later. Commitment schemes can be constructed simply by using cryptographic hash functions. Two properties that need to be satisfied in order to design a commitment scheme are binding and hiding.

Binding makes sure that the committer is unable to change the chosen value once committed, whereas the **hiding** property ensures that any adversary is unable to find the original value to which the committer made a commitment.

A Pedersen commitment is a type of information-hiding, computationally binding commitment scheme, under a discrete logarithm assumption. Pedersen commitments allow for additional operations and preserve commutative property on the commitments, which makes them particularly useful for providing confidentiality in **Bitcoin** transactions. In other words, they support partial homomorphic encryption of values, meaning that using commitment schemes allows us to hide payment values in a Bitcoin transaction. This concept is already implemented in the *Elements Project*: `https://elementsproject.org/features/confidential-transactions`.

MimbleWimble

The **MimbleWimble** scheme was proposed anonymously on the Bitcoin IRC channel and since then has gained a lot of popularity. MimbleWimble extends the idea of confidential transactions and Coinjoin, which allows the aggregation of transactions without requiring any interactivity. However, it does not support the use of the Bitcoin scripting language, along with various other features of the standard Bitcoin protocol. This makes it incompatible with the existing Bitcoin protocol. Therefore, it can either be implemented as a sidechain to Bitcoin or on its own as an alternative cryptocurrency. It was however implemented in Litecoin in May 2022.

This scheme can address both privacy and scalability issues at once. The blocks created using the MimbleWimble technique do not contain transactions as in traditional **Bitcoin** blockchains; instead, these blocks are composed of three lists: an input list, an output list, and something called *excesses*, which are lists of signatures and differences between outputs and inputs. The input list basically references the old outputs, and the output list contains confidential transaction outputs.

The blocks created using the MimbleWimble scheme are verifiable by nodes by using signatures, inputs, and outputs to ensure the legitimacy of the block. In contrast to Bitcoin, MimbleWimble transaction outputs only contain `pubkeys`, and the difference between old and new outputs is signed by all participants involved in the transactions.

Zkledger

This distributed ledger provides transaction privacy, including confidentiality of transaction amount and sender and receiver addresses. It utilizes NIZKs and Sigma protocols to achieve this. However, this protocol is not scalable because the size of transactions grows linearly with the number of parties involved.

Attribute-based encryption

The **attribute-based encryption** (**ABE**) is a type of public key cryptography that provides confidentiality and access control simultaneously. The key idea behind this scheme is that the private key of a user and its ciphertext are dependent on the user's attributes such as their location, time zone, or their role in the organization. This means that decryption is only possible when not only the private key is available but also the matching attributes are present.

Anonymous signatures

Anonymous signatures are types of digital signatures where the signatures do not reveal the identity of the signer. There are primarily two schemes available for anonymous signatures: **group signatures** and **ring signatures**.

Group signatures allow a set of signers to form a group managed by a group manager. Each member of the group is issued with a group signing key by the group manager. This group signing key allows each member of the group to anonymously sign messages on behalf of the group. The group manager is able to figure out who is the signer of the message, whereas external entities cannot. While group signatures can work well, one of the limitations of this scheme is that the group manager is able to identify the users. This issue was addressed with ring signatures.

Ring signatures allow a set of signers to form a group (a ring) of members. Each member in this group is able to sign messages on behalf of the ring. Unlike group signatures, there is no group manager in ring signatures, so no one is able to identify the signers.

While all these techniques are useful and provide many desirable properties of anonymity, there is also a problem with using this technology maliciously. Imagine if anonymity is misused by criminals involved in money laundering or selling illegal drugs by using anonymous cryptocurrency. It would be almost impossible to trace illegal activities back to the originator if absolute anonymity is achieved. Imagine a **Silk Road marketplace** (`https://en.wikipedia.org/wiki/Silk_Road_(marketplace)`) variant using a fully anonymous and confidential cryptocurrency. How could that be traced and stopped?

So far we have discussed generic cryptographic techniques and some examples thereof to enable privacy in blockchain. However, we haven't touched on the important concept of private smart contracts. Programmable blockchains, starting from Ethereum and now chains like Solana and Polkadot, all support smart contracts. It is desirable to achieve smart contract-level privacy too, instead of only standard transaction-level privacy. Think of an ERC-20 contract where all operations like transfers, balance checking, and others are performed with full privacy. There are several proposals to do that as well, some of which we'll discuss next.

Zether

Zether allows a private transaction mechanism that supports confidentiality and anonymity. It is implemented on the Ethereum blockchain but, in theory, can be implemented on any programmable chain. Zether is implemented as a smart contract where ether is deposited and turned into a Zether token.

Other proposals include Zexe (`https://eprint.iacr.org/2018/962.pdf`), PPChain (`https://ieeexplore.ieee.org/document/9199417`), Hawk (`https://ieeexplore.ieee.org/document/7546538`), and ZKledger (`https://dci.mit.edu/zkledger`).

Privacy using Layer 2 protocols

Privacy using Layer 2 and state channels is also possible, simply because all transactions are run off-chain, and the main blockchain does not see the transactions at all except for the final state output and minimal transaction data (representing transactions), which ensures anonymity and confidentiality. However, note that usually privacy is not the main purpose of Layer 2 protocols, instead they are more concerned with provided scalability. Nevertheless, Layer 2 can be used to provide anonymity and confidentiality if zero-knowledge is used not only as proof of correct computation (integrity) but also as a confidentiality measure. This is achieved in Aztec (`https://aztec.network`).

Privacy managers

Privacy managers provide a mechanism that provides confidentiality of transactions. These are off-chain components that substitute the transaction payload with a hash index, in such a way that only those participants who are party to the transaction are able to find the corresponding encrypted payload represented by the hash index. Other parties who are not privy to the transaction will simply ignore the hash. This concept was discussed in *Chapter 16*, *Enterprise Blockchain*.

Other than scalability and privacy, there are several other general security aspects that need to be addressed in blockchain. We'll describe these general security topics in the next chapter, *Chapter 19*, *Blockchain Security*.

There are two approaches when it comes to scalability and privacy solutions. First is the development of domain-specific languages, which can be used to write programs that can be verified by a verifier on Layer 1. Secondly are ZK virtual machines, which allow computations (programs) to be verified on Layer 2. DSLs require developers to learn a new language whereas ZK VMs are compatible with mainstream tools and languages already in use. A key advantage of DSLs is that the generated ZK proof or circuit is agnostic to the underlying chain, which makes it easy to use it on any chain, as long as we write a verifier for that chain. ZK VMs however have some risk of vendor lock-in.

Privacy using zero-knowledge

When we think about privacy, the first thing that comes to mind is cryptography and we can think of several ways to achieve privacy, which we describe next:

1. We can possibly encrypt transactions so that data is confidential, but this approach is impractical for two reasons. First, public verification of transactions is not possible as the values are hidden; secondly, even if some encryption scheme is utilized, key management becomes a big issue due to the sheer number of keys required.

2. The second approach is commitment schemes, which is quite a promising solution. We can replace the amount/value of the transaction with a commitment to the value, then we commit to the value and post it on the blockchain. Again, the issue is that as the value is hidden, the transaction is not publicly verifiable; also, commitment to a negative value is possible.
3. The solution is to attach ZFPs with the commitment, which proves two things: range commitment, which ensures that the number is positive and is in a valid range, and value commitment, which hides the actual transaction value. Range commitments are harder to do whereas value commitment is comparatively straightforward, as it translates as a simple linear equation, which is easier to handle.

Let's now explore cryptographic commitment, an essential cryptographic primitive to build zero-knowledge protocols on blockchain.

Cryptographic Commitments

A cryptographic commitment scheme is a two-phased protocol composed of two algorithms, a `commit` algorithm, and a `verify` algorithm:

- `Commit(m, r) -> c`
- `Verify(m, c, r) -> accept or reject`

In the first phase, called the commit phase, the sender sends the receiver a commitment string `c`. Alice runs the `commit` algorithm to commit to a message `m` using secret randomness `r` and produces a commitment string `c`. This commitment string `c` is sent to Bob.

In the second phase, called the open phase, the receiver opens the commitment to verify that the sender indeed committed to the message `m` and did not cheat.

This is checked using the verify algorithm. This algorithm takes three values, message `m`, commitment `c`, and randomness `r`, and based on the output the verifier either accepts or rejects the commitment. Once the commitment is open anyone can publicly verify that the commitment is opened correctly.

We can think of cryptographic commitments as a digital analog or cryptographic equivalent of sealed envelopes. There are two security properties of cryptographic commitments, namely hiding and binding:

- The hiding property ensures that the committed message is not revealed before the open phase. In other words, commitment `c` reveals nothing about the committed message.
- The binding property ensures that once the value is committed, it cannot be changed later on. In other words, once the committer has committed to a value, the committer must not be able to open the commitment with a different message `m`, which the verify algorithm also accepts.

There are two types of commitment schemes based on the strength of the security properties:

- Standard commitment schemes, which protect against a PPT receiver and an unbounded powerful sender. Such schemes are computationally hiding and information-theoretically binding.
- Perfect commitment schemes, which protect against and PPT sender and an unbounded powerful receiver. Such schemes are information theoretically hiding and computationally binding.

Commitment schemes that are both information theoretically hiding and binding do not exist.



A **Probabilistic Polynomial Time** (**PPT**) adversary has access to a randomized polynomial time algorithm to break the security of the system. We can think of it as a computationally bound entity that can only break the security of the system if it has access to a PPT algorithm. Polynomial time algorithms are those algorithms that run in a "reasonable" amount of time or take a reasonable time to compute. More formally, a polynomial time algorithm is an algorithm that gets an input of size n; it would be considered polynomial time if it runs in $O(n^c)$ time where c is a constant. A PPT algorithm is a polynomial time algorithm that is randomized, meaning that it has access to a source of randomness, i.e., allowed to flip coins.

There are several models of commitment schemes including **hash-based commitment schemes**, **Pedersen commitments**, and **polynomial commitment schemes.** A type of commitment scheme called Pedersen commitment is very useful due to its homomorphic properties. Let's first see how the Pedersen commitment is built and then we'll see how its homomorphic property works.

The scheme works in three phases, public setup, commit, and open.

Setup phase:

1. Let $\mathbb{G}$ be a finite cyclic group of prime order
2. Get two elements $g, h \in \mathbb{G}$, where g is a known fixed group element called *generator* and h is a uniformly random group element from $\mathbb{G}$
3. Let q be the order of $\mathbb{G}$ which is exponentialy large

Commit phase:

1. The committer commits to a number $m \in \{0, \ldots . q - 1\}$
2. The committer picks a random integer $z \in \{0, \ldots q - 1\}$
3. Sends the commitment c to the receiver, which is $c \leftarrow g^m . h^z$

Open phase:

1. To open the commitment c, the committer sends (m, z)
2. The receiver verifies that $c = g^m . h^z$

The Pedersen commitment is perfectly hiding and computationally binding. Pedersen commitments are additively homomorphic. Suppose we have two commitment strings, c_1 and c_2, which are commitments to messages m_1 and m_2 respectively. If we multiply these two commitments the result is a new commitment c_3 to the sum m_3 of m_1 and m_2. This means that without knowing the original m_1 and m_2 (the originally committed messages) the receiver can construct a commitment to the sum of the committed values, i.e., a commitment c_3 to m_3, where $m_3 = m_1+m_2$ and c_3 can be opened to m_3 by using the opening information for c_3, which is (m_1+m_2).

Pedersen commitments are commonly used with some modifications in several privacy-preserving blockchains, including Monero and Zcash. Note that Pedersen commitments alone cannot create a complete privacy-preserving solution, but they serve as a component in a larger solution that includes ZKPs, which we discuss next.

Zero-knowledge proofs

The first ZK proof was invented by Goldwasser, Micali, and Rackoff in 1985. Succinct transparent arguments from PCP were introduced by Kilian and Micali in 1992 and 1994 respectively. These proofs are the first SNARKs, but they were impractical due to impractical prover time; nevertheless the proofs were short in size and fast to verify. For a long time, ZKPs were considered impractical but in 2013 a breakthrough was made that introduced linear prover time with a constant size proof. This was later improved in 2016. Both papers are available here (GGPR13 – `https://eprint.iacr.org/2012/215` and Groth16 – `https://eprint.iacr.org/2016/260`).

In 2013, Pinocchio, the first practical ZK SNARK, was introduced. Also, tinyRAM was introduced in 2013, which is the first ZK VM. In 2018, the first ZK rollup from Barry Whitehat, the Circom language form `iden3`, and the first ZKSTARK from Starkware were introduced. In 2019, PLONK, a major milestone, was achieved, which improved the usability of SNARKs. Its setup is universal and updateable, which means that the setup needs to run only once and then can be used for any program instead of running a separate setup for each new program, as was the case in the earlier schemes, e.g., Zcash. Secondly PLONK uses polynomial commitment, which is swappable with any other commitment scheme to achieve different improvements. This means that the proving scheme and the commitment scheme are separate, and a different more efficient commitment scheme can be used. Plonk uses KGZ10 (Kate) but it can be swapped with FRI or some other scheme.

The acronym **FRI** stands for **Fast Reed-Solomon IOP of Proximity**, and **IOP** stands for **Interactive Oracle Proof**. The FRI protocol ensures that a committed polynomial has a bounded degree.

FRI would allow it to become transparent and post-quantum secure (resistant), while keeping the proving scheme the same, i.e., PLONK. Earlier proving schemes like Marlin and Sonic were more complex and costly. Then in 2020, Cairo ZKVM and Halo2 were developed. 2021 saw Circom 2.0 and Mina development. In 2022, a lot of projects were introduced, including Plonky2, which is the fastest ZKSNARK so far, along with many other projects, such as ZKEVMs, including Scroll and zkSync. There is no sign of this development stopping or even reaching a plateau anytime soon; more and more research and development are being carried out by enthusiastic teams, and this trend is only expected to grow further with more innovation on the horizon. As it is now an established belief that ZK very nicely complements blockchains, the future of blockchain is going to be heavily oriented around zero-knowledge.



Note that *proof* and *argument* are two terms that are used sometimes interchangeably. In S*ARKs, **AR** stands for **arguments**, indicating that these are not proofs. Zero-knowledge proofs are however proofs, and not arguments. It's important to understand this difference. The difference between an argument and a proof is to do with how strong the soundness property in the protocol is. In the case of arguments, the soundness property is secure against a polynomially bounded adversary (prover) whereas proofs are secure against a computationally unbounded adversary (prover). In other words, proofs are information theoretically secure whereas arguments are computationally secure.

The first cryptocurrency to successfully implement ZKPs (specifically ZK-SNARKs - **Succinct Non-Interactive Argument of Knowledge**) to ensure privacy on the blockchain is Zcash. The same idea can be implemented in **Ethereum** using smart contracts and other blockchains also. The original research paper is available at `https://eprint.iacr.org/2013/879.pdf`. Another excellent paper is available here: `http://chriseth.github.io/notes/articles/zksnarks/zksnarks.pdf`.

There is another type of ZKP called **zero-knowledge Succinct Transparent Argument of Knowledge** (**zk-STARK**), which is an improvement on zk-SNARKs in the sense that zk-STARKs consume a lot less bandwidth and storage. Also, they do not require the initial, somewhat controversial, trusted setup that is required for zk-SNARKs. Fundamentally they are SNARKs but without a trusted setup. Moreover, zk-STARKs are much quicker as they do not make use of elliptic curves or rely on hashes. The original research paper for zk-STARKs is available here: `https://eprint.iacr.org/2018/046.pdf`.

Bulletproofs are non-interactive zero-knowledge short proofs. They require no trusted setup. This scheme allows a prover to prove that an encrypted number is within a range of numbers without revealing any other information about the number. More information on Bulletproofs is available here: `https://electroneropulse.org/public/doc/Bulletproof%20RingCT.pdf`.

A quick comparison of the techniques mentioned above is shown below:

Type/Property	Proof size	Prover time	Verification time	Trusted setup required
SNARKs	Small 288 bytes	Medium ~2.3 seconds	Small ~10 milliseconds	Yes
STARKs	Large ~40 – 50 KB	Small ~1.6 seconds	Medium ~16 seconds	No
Bulletproof	Medium <1 KB	Large ~30 seconds	Large ~1100 seconds	No

We covered ZKPs in *Chapter 21* and *Chapter 4*. Here we'll focus more on non-interactive ZKPs and cover SNARKs in more detail.

Generally, there are two classes of zero-knowledge arguments, sigma protocols and SNARKs.

Sigma protocols are an efficient way to create ZKPs. These are three-round protocols with a proof, challenge, and response phase with a verifier, who is able to throw a public coin (randomness), as shown in *Figure 18.3* below:



Figure 18.3: Sigma protocols

As SNARKs are the most used and promising technology to implement privacy in blockchains, we'll focus on exploring SNARKs more.

As we saw in *Chapter 4*, ZKPs traditionally have large data structures and require multiple challenge-response interactions between a prover and verifier for the prover to prove some assertion. Interactive protocols can be useful in limited scenarios; however, NIZKs are more suitable for blockchain-based scenarios where the prover can post the proof online/on-chain and verifier(s) can verify independently (asynchronously). This is shown in *Figure 18.4* below:

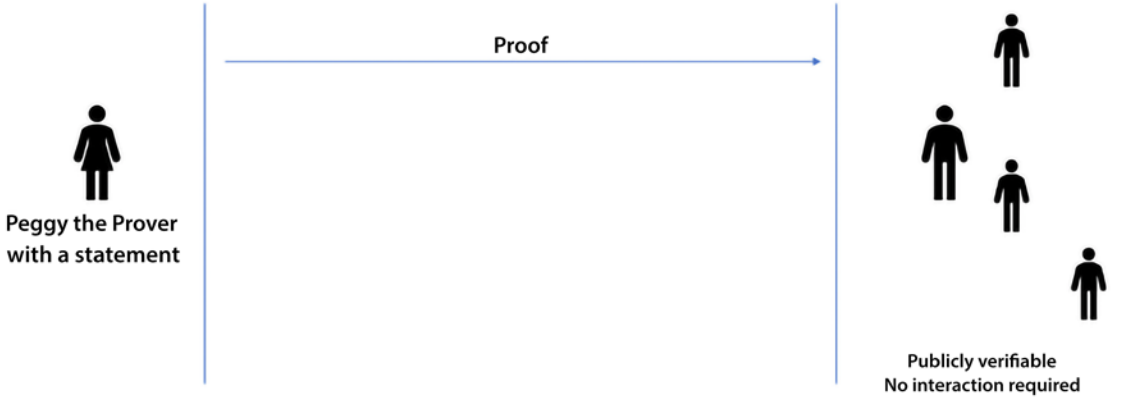


Figure 18.4: NIZK

For achieving non-interactivity several techniques can be employed, including a **common reference string model** where participants share a random string. It is crucial for NIZK proofs; without CRS, NIZK cannot exist. Practically speaking CRS is a set of elliptic curve points of a specific form.

A typical visual representation of a CRS scheme is shown in *Figure 18.5*:

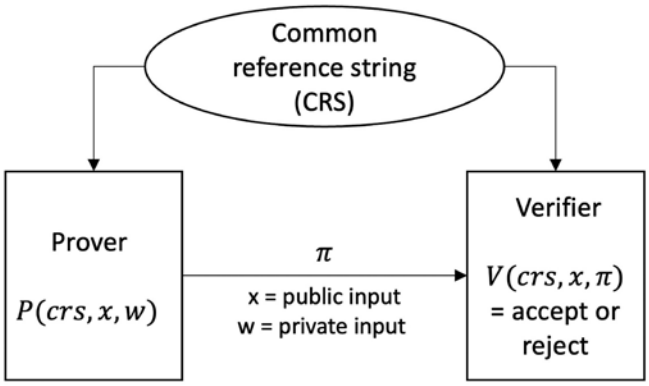


Figure 18.5: CRS-based proving scheme

Trusted setups are usually called setup ceremonies, where the proving key for the prover and the verification key for the verifier are generated. Here some "toxic waste" is also produced, which needs to be destroyed. Here *toxic waste* simply means some secret randomness that needs to be destroyed privately; otherwise, the security of the system is compromised.

There are four methods that can be used to generate reference strings, namely trusted, subverted, transparent, and MPC:

- A **trusted model** simply means that there is just a single trusted party generating the reference string, which clearly is not acceptable in practice because if that single party turns malicious, there is no way to stop it.
- **The subverted approach** says that even if the prover has generated the SRS (i.e., knows the trapdoor, or in other words, fully subverted the setup process) even then it won't be able to prove a false proof. This of course again is not practical, as it has been proven as an impossibility result that ZK cannot be achieved using this method.
- In **transparent setups**, instead of SRS, a **uniform reference string** (**URS**) is randomly generated and used, which means that no toxic waste (a trap door, i.e., keys and randomness, etc.) is produced. Note that the polynomial commitment scheme used in the protocol entirely dictates the transparency of the SNARK; if the PCS uses URS, then the SNARK protocol also uses URS. This option enables trustless setup.
- **The MPC-based approach** allows multiple parties to jointly generate a structured reference string in a decentralized manner without compromising security. As long as even a single participant has deleted their part of the secret, the entire scheme can be considered secure. **Multiparty computation** (**MPC**) is a method of allowing multiple parties to cooperatively compute a function over their private inputs, without revealing those inputs to each other. MPC enables multiple parties to collaborate and execute a computation while keeping their inputs and intermediate results private.

Trusted setup has been performed in Zcash, Aztec, and several other protocols. The caveat here is that if the SRS is not universal then this setup ceremony needs to be run every time the circuit requirements change, e.g., a new type of transaction or changes in the existing transaction structure. This is why Zcash had to run the ceremony again in 2018 (the Sapling ceremony), after the first time in 2016 (the Sprout ceremony).



Zcash has released Orchard shielded payment protocol, which uses the Halo 2 proving system and removes the need for a structured reference string and thus the trusted setup ceremonies. This means that future circuit upgrades can be done without a trusted setup. More information on this is available here: `https://zips.z.cash/zip-0224`.

We can also divide SNARKs into three types based on the setup requirements. There are three main types of setups:

- **Trusted non-universal setup** (specific purpose – trusted setup per circuit)-based SNARKs where they are built for a single circuit. They have a large overhead of a large common reference string. Here the setup procedure generates some random data that must be kept secret; otherwise, the prover will be able to prove false statements. Once the setup is done, this data is destroyed. They are used first in Zcash; it's OK even with the limitations because the setup doesn't need updating, as there is only a single type of circuit, i.e., a transaction transfer. The setup must be done again if the circuit type changes.
- **A trusted universal setup** where the SNARK only requires a trusted setup once. They have a **structured reference string** (**SRS**), which is updatable and smaller than CRS. Here the reference string has some structure based on secret randomness. This is also a trusted setup but is universal and updatable because the secret material (randomness, trapdoor, etc.) in this setup is independent of the circuit. This can be thought of as a two-step process where an initial one-time procedure runs secretly to generate an updateable setup string and any secrets are destroyed. However, now this setup string is updatable for any new circuit and can preprocess any new circuits without relying on any secret data.
- The third type is **transparent SNARKs** or STARKs, which require no trusted setup as they don't use any secret data.

As SNARKs produce small proofs (succinct) that are quick to verify, these proofs can be used for not only privacy but also for scalability, as we saw in *Chapter 17*, *Scalability*, because they are short in size, and verifiers only have to process and verify a small piece of data instead of all transactions. These proofs, while efficient, have two key issues. First it takes a long time to generate proofs and secondly SNARKs require SRSs, which means trusting third parties.

Let's now see how ZK-SNARKs are built.

Building ZK-SNARKs

Generally speaking, generating a SNARK is a two-step process. The first step, which involves the "**frontend**" of the system, is to convert a program into an equivalent model that can be checked probabilistically.

This means transforming the program to be proven into an arithmetic circuit, i.e., generally into a circuit satisfiability problem. The arithmetic circuit is further represented by an arithmetic constraint system, which can be R1CS – the **Rank 1 constraint system** or **Hadamard Product Relation** (**HPR**). At a high level the constraint system encodes the circuits as matrices. The first step can be seen as a process where the program is transformed into an equivalent circuit, which basically allows us to make checking for the proof by a verifier non-interactive. The program code is transformed into an arithmetic circuit, which is an algebraic circuit that represents the program code. Practically the arithmetic circuit is described as matrices and relevant operations in linear algebra. The prover and verifier run the SNARK for circuit satisfiability. Here usually DSLs like Circom and Zokrates are used to write the circuit. Circom is a lower-level language where the entire program is written almost gate by gate, Zokrates is a bit higher level than that and allows us to create programs in a more familiar Python-style code. Cairo and Noir are two other languages that can be used to write circuits to represent the program. This approach of using a DSL to write programs is much easier than writing a compiler to take a high-level program written in Rust or C and convert that directly into an equivalent circuit. Such an approach is not much practical; therefore, specific DSLs are used for writing circuits.

Any program that the prover wants to prove is first converted into an arithmetic circuit, which is also called *flattening*. An arithmetic circuit is a construct that represents a mathematical function as a **directed acyclic graph** (**DAG**) consisting of gates, inputs, wires, and outputs. Gates are vertices and wires are edges. Each gate in the circuit performs a specific arithmetic operation (such as addition or multiplication) on its inputs and produces an output, and the outputs of the gates are used as inputs to other gates until the final output is produced. For carrying values, wires (edges) are used. The inputs' nodes are labeled with variables and constants.

To build ZK-SNARKs, arithmetic circuits are used to represent the computation that the prover wants to prove. More precisely the prover tries to convince the verifier that it has a solution to the arithmetic circuit.

Such a circuit is shown in *Figure 18.6* below, which computes the expression (*a*+*b*) x *b* x *c*:

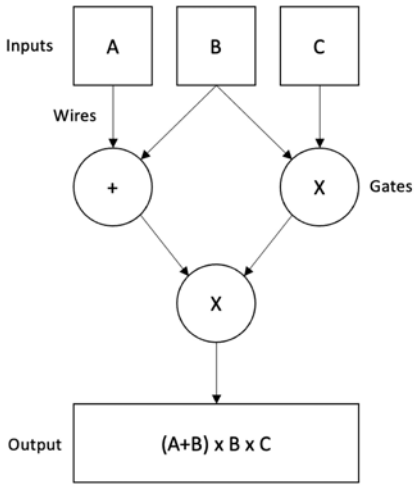


Figure 18.6: An arithmetic circuit

Polynomials are extensively used to represent many problems in cryptography. Circuits are also defined over a prime finite field as multivariate polynomials, and the circuit structure provides a computational model to evaluate the polynomial.

The key feature of ZKPs based on arithmetic circuits is that they enable a prover to produce a proof that shows that the prover knows an input value for the circuit, which produces a predefined output while keeping all or some inputs secret. These proof systems ensure that a prover is not able to generate a valid proof without the knowledge of correct inputs for the circuit and that the private inputs are obfuscated, so that no information is leaked about private inputs.

We saw earlier, what an argument and a proof are. Let's now define the ZK argument system in light of what we have learned about circuits. In an argument system we have two parties, a verifier and a prover. Imagine we have a circuit that takes two inputs, `x` and `w`. `x` is the public input and `w` is the secret witness that the prover is trying to prove knowledge of. The prover's aim is to convince the verifier that there is a witness `w`, which when processed by the circuit along with the public input `x` produces the desired output. The verifier doesn't know what `w` is and only knows `x`, the public input. The verifier and prover engage in several rounds of exchanging messages, and at the end of the protocol the prover either rejects or accepts the proof. In other words, the verifier is either convinced that the verifier indeed knows x, or not. A non-interactive zero-knowledge argument system has four properties: completeness, the argument of knowledge, zero-knowledge, and succinctness:

- **Completeness** means that an honest prover is always able to convince an honest verifier that the proof is correct.
- An **argument of knowledge** means that if the verifier has accepted the proof, then the prover undeniably knows a secret piece of information, called a **witness**, that satisfies the circuit. In other words, the valid proof can only be generated by the prover who indeed knows the private input. More precisely this means that the prover knows the witness if it can be extracted from the prover. The extraction means that an efficient procedure exists that can extract the secrets from a prover. An argument system is computationally sound, instead of information theoretically sound, where the assumption is that it is impossible to prove a false statement to the prover. In an argument system we say it is infeasible to fool the prover instead of impossible. If we assume impossible, then it's a proof instead of argument.
- **Zero-knowledge** means that the proof produced by the provers does not reveal anything at all about the witness. In other words, it doesn't reveal anything apart from whether the statement asserted is true or false. Formally this property means that if the verifier is able to regenerate the proof on its own it would learn nothing new; the proof would still not reveal anything about the witness. We can say that a ZK argument system is zero-knowledge for a circuit if there is an efficient procedure, called a **simulator**, that can generate public parameters and the proof, which are indistinguishable from the real public parameters generated by the setup procedure and the proof generated by the real prover by only using public input x, i.e., without knowing the witness.
- **Succinctness** means that the proof produced is small in size and is quickly verifiable by the verifier. The speedup in verifier time is in fact achieved because of the setup phase, where public parameters are generated.

> We can think of the setup phase as the circuit being "preprocessed" in a way that the verifier doesn't have to read the entire circuit. The prover of course has to read it fully, but the verifier can avoid it by relying on essentially a "short" version of the circuit produced, due to the pre-processing performed in the setup phase.

After the circuit is generated, the public parameters can be generated by the setup procedure, which takes the arithmetic circuit and produces the required parameters. These public parameters are used by the prover along with the public input x and secret witness w to produce the proof (usually denoted by pi π). The verifier also uses these public parameters along with the public input x to verify the proof sent by the prover.

SNARKs rely on a trusted setup where a common reference string is generated to make the proof sizes smaller. We have several types of reference strings, including CRS, SRS, and URS. The SRS is essentially a proving key and is usually as big as the circuit. There is a trapdoor function in this SRS, and if the prover discovers it, they can easily create false statements that they can prove to the verifier as true. This trapdoor is in fact known to the entity that generates this SRS, and this entity needs to be trusted by the system. The expectation is that this party will destroy this "toxic waste" after the SRS is set up. This is of course not ideal, and a lot of research has been done to minimize the trust requirements. One key example is creating SRS through the MPC protocol. There is however no requirement for an SRS setup in STARKs. SRS for simpler circuits is manageable, but complex circuits with possibly billions and trillions of gates can become a problem to manage in terms of storage and distribution to provers.

In the second step, involving the **backend** of the system, a proving scheme is applied for circuit satisfiability. Some examples of the proving schemes include Groth16, GM17, Marlin, PLONK, Libra, Supersonic, Hyrax, and many others. The key requirement behind the development of the backend is to create a SNARK that both the prover and verifier can run to satisfy the circuit. The proving scheme is expected to have fast verifier verification times, be succinct (small in size – a few bytes or KBs), and be at least linear (in terms of algorithmic complexity) to the program size for prover times.

We can visualize the frontend, setup, and backend as a ZK-SNARK system in *Figure 18.7* below:

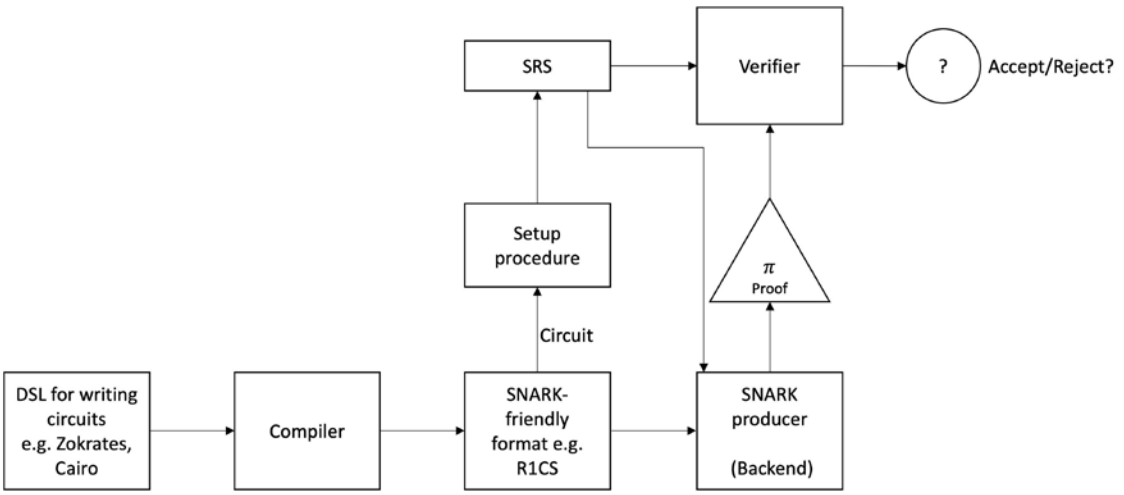


Figure 18.7: A typical ZK-SNARK system

The design of the second step, i.e., the backend or proving scheme, generally includes three steps:

- Create (use) a **Polynomial Interactive Oracle Proof** (**PIOP**) for circuit satisfiability. In other words, create a proof scheme that proves that the statement (program) for which the circuit has been built is true.
- Create (use) a polynomial commitment scheme to build a succinct interactive argument.
- Use the Fiat-Shamir heuristic to transform this interactive argument into a non-interactive protocol. This is achieved by substituting the verifier's randomness with a random oracle, which is instantiated by a cryptographic hash function. The prover is unable to predict the outcome of the hash function, thereby providing the randomness necessary to achieve the soundness property of the ZKP.

Combining the first two steps in the "backend" produces a SNARK, i.e., PIOP + PCS = SNARK.

We have used the term *IOP* quite a bit so far. Let's see what this is. **IOP** stands for **Interactive Oracle Proof**. It is an interactive proof in the sense that the verifier doesn't have to read the entire message from the prover; instead, the verifier has access to an oracle for the prover's message and can query it probabilistically. Practically this means that the verifier queries some evaluation points instead of reading the entire message. Here the prover has a public input and a witness, whereas the verifier has only the public input. Prover messages are some arbitrary bit strings (d i.e. polynomial encodings), whereas verifier messages are some randomness (i.e. queries on evaluation points). The prover and verifier engage in several rounds of these messages and eventually, the verifier generates queries to query the oracle at particular locations (evaluation points).

Once the result of these queries is generated the verifier decides whether to accept or reject the proof. To ensure that the messages are indeed polynomials a test called a "low-degree test" is performed, which checks the degree of the polynomial. However, this produces large-size proofs and requires a number of rounds to operate, which is quite costly. However, if we can assume that messages sent from the prover to the verifier are polynomials over a finite field, then there is no need for a low-degree test, and this will result in a low cost. This is exactly what polynomial IOPs do when prover messages are polynomials and because of this assumption, there is no low-degree test, which reduces cost and improves efficiency. IOPs can be categorized into three classes based on how they've evolved over time:

- **Interactive proofs**, which are a standard category where a single prover and a verifier engage in a protocol to prove the assertion of a statement by the prover. Some examples of SNARKs in this category are Libra and Hyrax. The prover time is lowest in this category, but the proof size is big and the verifier time is high.
- **Multi-prover interactive** proofs, where instead of a single prover, multiple provers exist. Some key examples in the category are Spartan and Brakedown.
- **Constant round PIOPs** are IOPs with constant round complexity. Key examples are Marlin and PLONK. The prover time is high in this category; however, the proof size and verifier time are fast.

After a PIOP is created, it's time to use an existing or create a new **polynomial commitment scheme** (**PCS**) to build a succinct interactive argument. Polynomial commitment schemes allow a committer (prover) to commit to a polynomial by sending a short string to the verifier.

Later the verifier can ask the prover to evaluate a single point in the polynomial. The prover responds with the result and a proof, which the verifier evaluates. The aim here is that the prover must not be able to produce a convincing proof of a wrong evaluation of the point on the polynomial. Secondly commitment and the proof are easy to generate and small, and the proof is easy and quick to verify. Polynomial commitment schemes can be classified into four categories according to the cryptographic assumptions they are based on:

- **IOP** with a **collision-resistant cryptographic hash function**, which doesn't require a setup (i.e., is transparent) and is post-quantum resistant. Most common examples of this scheme include FRI (`https://drops.dagstuhl.de/opus/volltexte/2018/9018/pdf/LIPIcs-ICALP-2018-14.pdf`) and Ligero commitments.
- **ECDLP-based**, which uses elliptic curves and is transparent but not post-quantum resistant. Bulletproofs (`https://eprint.iacr.org/2017/1066.pdf`) are a common example of such a type of commitment scheme.
- **Pairing group-based**, which requires a trusted setup and is also not post-quantum resistant. A common example is the KZG10 (`https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf`) commitment scheme. Another is Dory.
- **Groups of unknown order-based**, which are transparent if using class groups but not post-quantum resistant. One major drawback is very slow prover times, which is due to the use of class groups. A common example of such schemes is DARK (`https://eprint.iacr.org/2019/1229.pdf`).

We can visualize a generic construction pipeline of SNARKs based on what we have discussed so far in *Figure 18.8*. We can also view this as a process of how we go from a program to SNARK:

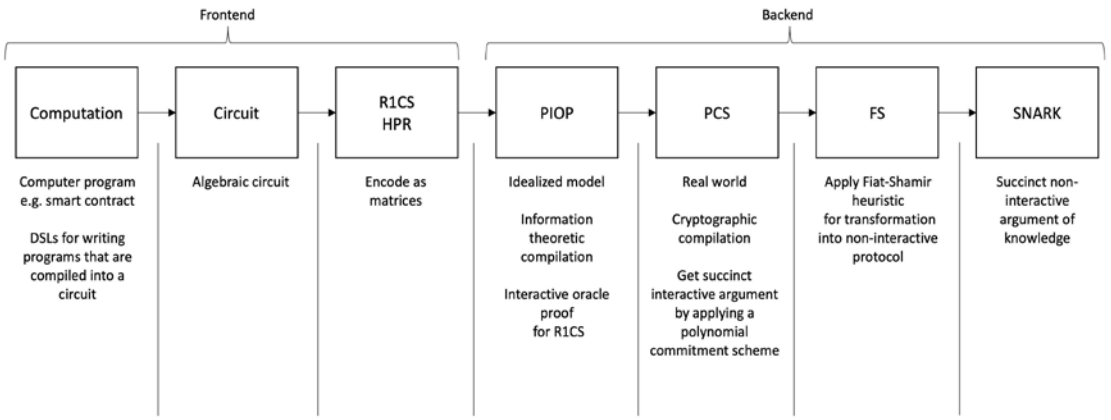


Figure 18.8: Generic construction pipeline of a SNARK

In summary, the construction pipeline starts with the arithmetization of a computation where techniques like R1CS and HPR are used. The next step is the compilation of this arithmetic representation into an information theoretical protocol, where idealized models such as **polynomial IOPs** (**PIOPs**), PCPs, and linear PCPs based on oracles are used.

Once we are happy with the information theoretic construction, to make it practical and real, we replace the information theoretic model with the cryptographic constructs called polynomial commitments. Here we essentially compile IOP to SNARK. A usual candidate for PCS is the Kate commitment scheme (`https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf`). This is where finally the SNARK is produced. We can think of it as combining PIOP with PCS to yield a SNARK scheme.

A polynomial commitment scheme allows a committer to commit to a polynomial. The hiding and binding properties apply to polynomial commitment schemes just as other commitment schemes. They are also composed of commit and open phases like other commitment schemes. However, there is an additional property that we can call "selective opening." This property means that the committer is able to open certain evaluations of the commitment without revealing the entire polynomial. In other words, the PCS allows commitment to a polynomial with a short string, which can be used by the receiver to verify the claimed evaluations of the committed polynomial. This means that the prover can prove that it possesses a polynomial, which satisfies some required properties without disclosing the polynomial.

Now let's see what the Kate PCS is and how it works. The scheme can be divided into three phases, setup, commit, and open and verify, just like other commitment schemes.

Setup

This is the one-time trusted setup to generate the SRS:

- Let e be a bilinear map pairing groups $\mathbb{G}, \mathbb{G}_t$ of prime order p.
- Let $g \in \mathbb{G}$ be a generator. $\mathbb{G}$ is a pairing-friendly elliptic curve group.
- Let l be the maximum degree, i.e., the upper bound on the degree of the polynomials we want to be able to commit to.
- Pick a random integer τ from the prime field, i.e., $\tau \in \mathbb{F}_p$, i.e., $\{1, \ldots, p-1\}$.
- Compute SRS as $(g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^l})$. This SRS consists of encodings in $\mathbb{G}$ of all powers of τ.
- Note that τ is the "toxic waste," i.e., the secret parameter (or trapdoor) of the setup ceremony, which must be destroyed once the SRS is generated.

Commit

In this phase, commitment to a polynomial ϕ over $\mathbb{F}_p$ is made:

- A polynomial $\phi(x) = \Sigma_{i=0}^{l} \phi_i x^i$.
- The compute commitment $c = g^{\phi(\tau)}$. Here even if the τ is destroyed the committer can still compute this using the SRS and additive homomorphism. Recall that our SRS is $(g, g^{\tau}, g^{\tau^2}, \ldots, g^{\tau^l})$. If $\phi(x) = \Sigma_{i=0}^{l} \phi_i x^i$ then $g^{\phi(\tau)} = \Pi_{i=0}^{l} \left(g^{\tau^i}\right)^{\phi_i}$, meaning that given the values g^{τ^i} for all $i = 0, \ldots, l$ without knowledge of τ, commitment c can be computed.

In simpler terms this means that we evaluate the polynomial that we want to commit to, at point τ, and multiply that by g, which will produce a single element from $\mathbb{G}$, and that is our commitment c. An amazing property to note here is that the polynomial can be of an arbitrarily large degree, but the commitment produced is still just a single element, i.e., a very short-sized piece of data, usually ~64 bytes.

Open

This is where we prove an evaluation. To open the commitment at input $a \in \{0,\ldots,p-1\}$ to some value b, i.e.:

- Given an evaluation $\phi(a) = b$.
- the committer computes and outputs the proof $\pi = g^{q(\tau)}$. This is basically a witness polynomial calculation.
- $q(x) = \frac{(\phi(x)-b)}{x-a}$ is called the quotient polynomial. Such a $q(x)$ can exist only if $\phi(a) = b$. If this quotient polynomial exists, then it is a proof of evaluation. Mathematically it checks if the polynomial remainder theorem holds.

Verify

This is where the evaluation proof is verified. A key property here is that verification can be done in constant time:

- Given a commitment $c = g^{\phi(\tau)}$, an evaluation $\phi(a) = b$ and the proof $\pi = g^{q(\tau)}$.
- The verifier checks that $e(c.g^{-b}, g) = e(\pi, g^{\tau}.g^{-a})$ where e is a non-trivial bilinear mapping.

Note that the verifier knows the commitment c, evaluation b , proof π, input a and g^{τ}. Values c, b, and π are provided by the prover. Input a is the opening query, which is provided by the verifier. g^{τ} is an entry in the SRS. For verification, only g^{τ} and g are needed. This is why SRS is sometimes called the proving key and (g, g^{τ}) are called the verification key. The verifier only needs the verification key to verify the proofs, and the prover uses the entire SRS as a proving key to generate the proofs.

We can visualize the Kate PCS in *Figure 18.9*:

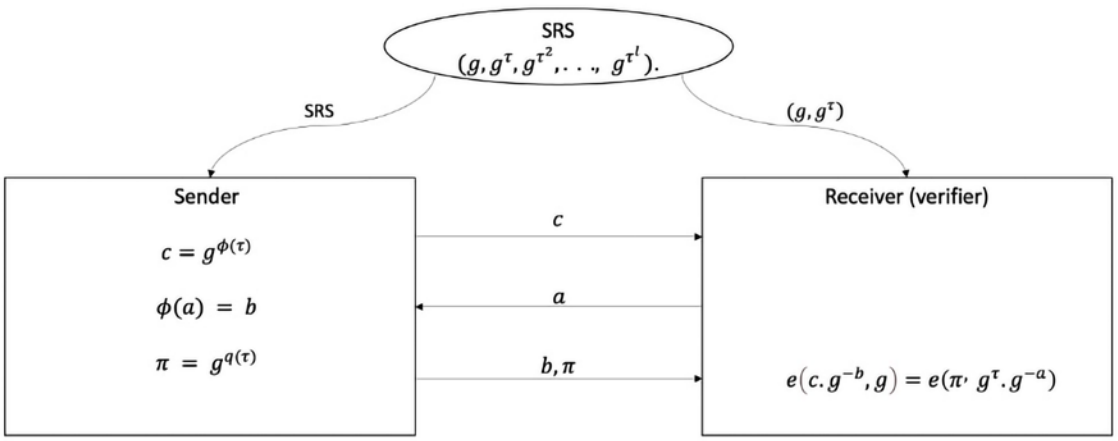


Figure 18.9: A high level view of KGZ PCS

If we have large amounts of data, how can we commit to it in such a way that a short message represents that entire dataset? In other words, we should be able to commit succinctly to that data. Secondly, we then should be able to prove values at specific locations.

This is where KZG is useful; we can represent large data as a polynomial and then evaluate it at secret points. Now you can see how PCS can be useful in blockchain. As the commitment of an arbitrarily large degree polynomial can be represented by just one group element, we can post just that commitment on the chain without needing to post everything to the blockchain, thus saving storage, bandwidth, and gas costs. In a rollup system a computation (e.g., state transition) can be represented as polynomials which the prover commits to and post on chain for verification. The verifier can ask the committer (e.g., the rollup provider) for evaluation on some random points; if the evaluations turn out to be correct, then it means that the entire computation is correct.

Ok, so now, after applying the commitment scheme we have achieved a protocol that allows us to commit to a polynomial with a short string. This means that it is a succinct protocol, but it is still interactive.

In order to make it non-interactive, we can use the **Fiat-Shamir** (**FS**) heuristic to transform this interactive protocol into a non-interactive one. FS transformation is a technique that is commonly used to transform an interactive protocol into a non-interactive one. The idea behind FS transformation is that the prover generates random bits on its own. Interactive protocols have a verifier that generates random bits and sends them to the prover, but non-interactive proof schemes transformed via FS enable the prover to generate random bits on their own. The requirement here is to provide enough uniform randomness to satisfy the soundness property of ZKPs. The interaction is replaced with access to a non-interactive random oracle. In practice, this is instantiated using a cryptographic hash function with carefully chosen inputs. After applying FS transformation we finally get a succinct non-interactive protocol, i.e., a SNARK.

There are different techniques to build the backend of a SNARK, with different pros and cons. For example, linear PCP-based SNARKs have the benefits of shortest proof and fastest verifier times. However, they need a circuit-specific setup, have slow prover times, and are not post-quantum resistant. A key example of this type is Groth16. Polynomial IOPs combined with KZG PCS produce SNARKs, which have the advantage of a universally trusted setup; however, the proofs are larger, the prover time is also slower, and they are also not post-quantum resistant. Some examples in this category are PLONK and Marlin. Linear PCP and constant-round PIOPs produce SNARKs that require a trusted setup. Combining a PIOP with FRI PCS produces proofs that are the shortest and post-quantum resistant; however, the prover time is slow and the proof size is still quite large.

This is how a SNARK is created at a high level. ZKPs is a huge subject and not everything can be covered in a single chapter. For further exploration, an excellent online resource is `https://zkp.science`, which can provide more insight into this fascinating subject.

ZK-SNARKs have many applications in the blockchain world. They can be used to improve the scalability of blockchain by creating a SNARK-based rollup. They can also be used to provide transaction privacy by obfuscating the transaction amounts and sender/receiver information.

Now let's see an example where we see these concepts in practice.

Example

For this example, we'll use Zokrates (`https://zokrates.github.io/introduction.html`), which is an open source toolkit that allows users to build and deploy ZKP systems. It includes a high-level programming language called Zokrates DSL, which is designed to write programs that can be verified with zero-knowledge. To use Zokrates, programmers first write the ZKP program using the Zokrates DSL. It is then compiled into a low-level arithmetic circuit using the Zokrates compiler.

In this example the prover will prove to a verifier that they know a value `x`, which, when used to solve a simple equation, results in an answer that the prover already knows. Let's assume the equation is:

$$y = x^2 + 2$$

The condition here is that if y is greater than or equal to 11, then it's a proof that the prover knows the value of x, without revealing the value of x.

So, let's write a simple program in Zokrates to do that:

```
def main(private field x) -> bool {
    bool y = if x*x + 2 >= 11 { true } else { false };
    return y;
}
```

Here we are simply defining a main function with a private field x. We declare it as private because we do not want to reveal this input but still prove that we know this value x.

Next we have the check that if the square of value of x and adding 2 to it, is greater than or equal to 11 then variable y being a bool, will be assigned with true; otherwise, it's false. In the next statement, we return y, which will be either true or false depending on the value of x. If the prover knows a legitimate value of x, that results in a value that is greater than or equal to 11 then they can prove to the verifier that they indeed know such a value of x that satisfies the equation; otherwise, the verifier will be convinced that the prover doesn't know the value of x, which if plugged into the equation results in a value greater than or equal to 11.

We will use Remix IDE and Zokrates plugin to demonstrate this. Let's start:

1. First open Remix IDE.
2. On the home page, there is a feature plugin view; in that, find Zokrates and install it in Remix. Once it's installed you will notice that it's available in the left-hand side column on Remix IDE, as shown below in *Figure 18.10*:

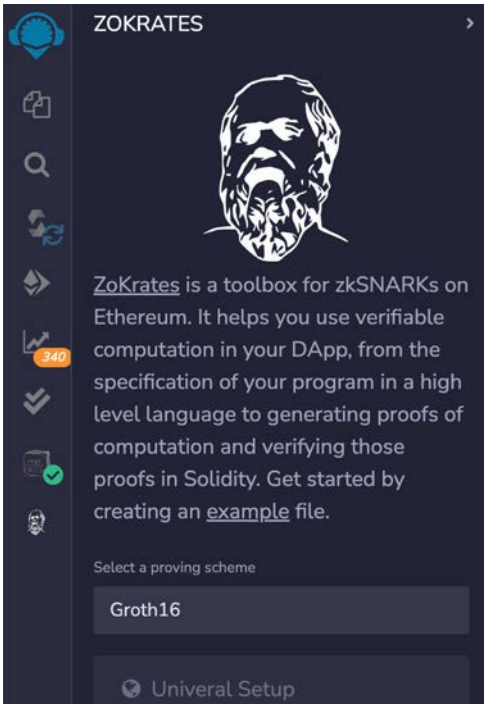


Figure 18.10: The Zokrates toolkit in Remix IDE

3. Once Zokrates is available in the Remix IDE, create a new file in it, and name it `main.zok`, as shown in *Figure 18.11*:



Figure 18.11: The main.zok file

4. Once the file is there, click on the Zokrates icon on the left-hand side, and click on **Compile** as shown below. If all is correct, it will compile the file. This compilation is equivalent to the creation of the circuit as discussed earlier.

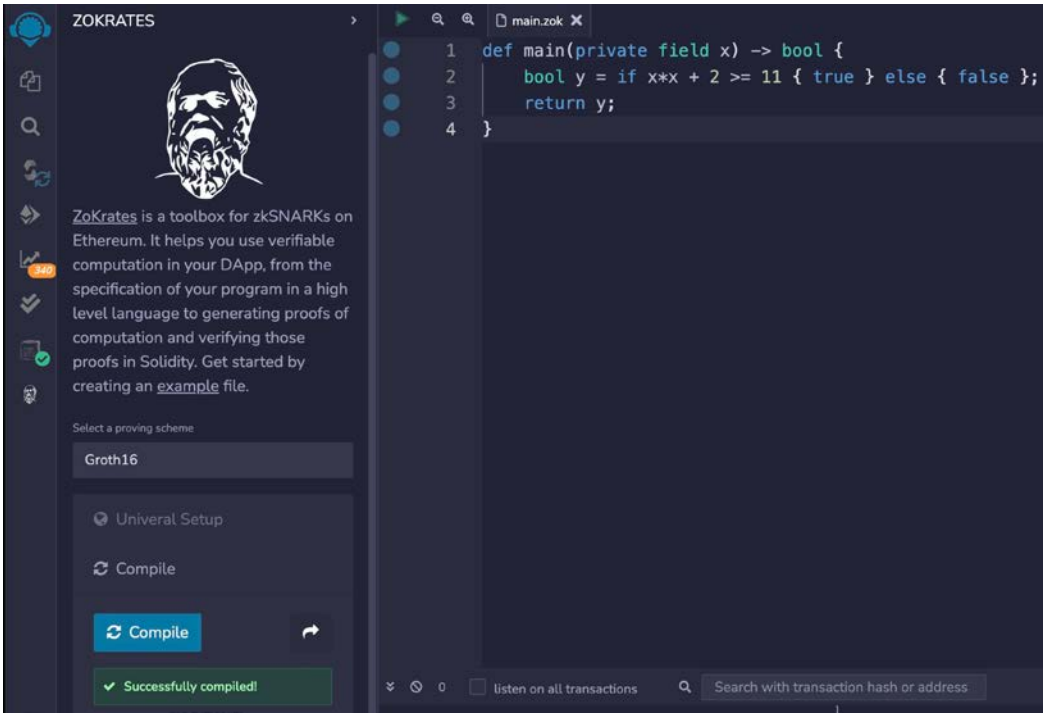


Figure 18.12: Zokrates compilation

5. The next step is to compute the witness, which can be achieved by inputting the value of x and clicking on **Compute**. All these options are available under the Zokrates plugin, as shown below.

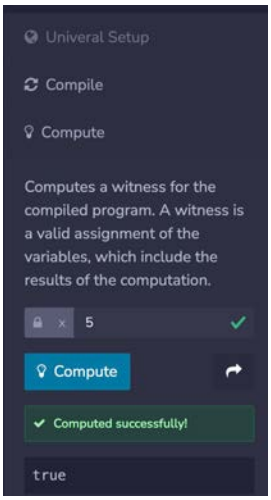


Figure 18.13: Compute witness

6. Now run the setup, which creates a proving key and a verification key as shown in *Figure 18.14*:

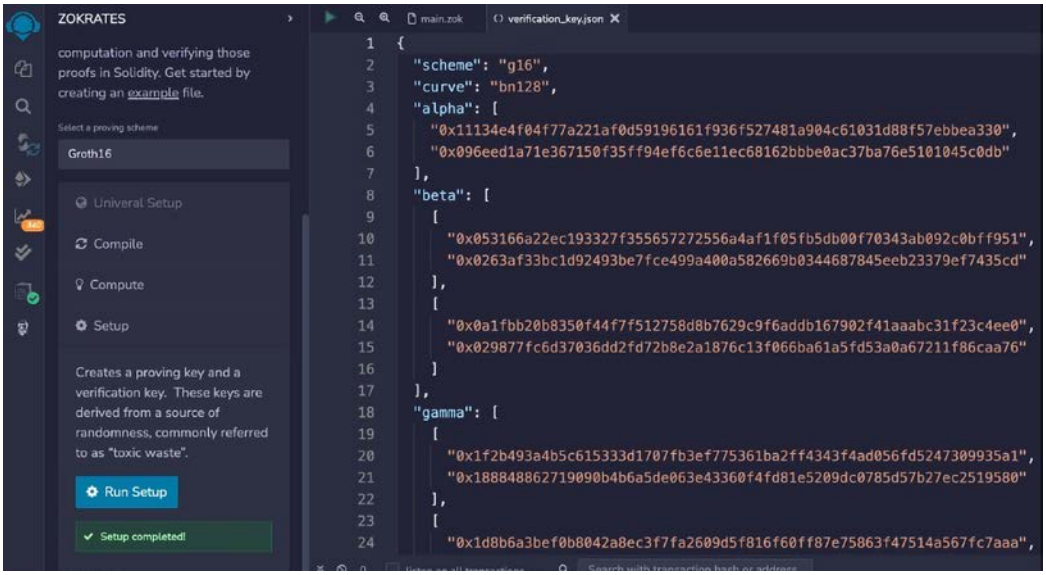


Figure 18.14: Run Setup

7. The next step is to generate the proof by using the roving key and witness generated in the previous steps. For this, click on **Generate** under the **Generate Proof** option in the Zokrates plugin, as shown below in *Figure 18.15*:

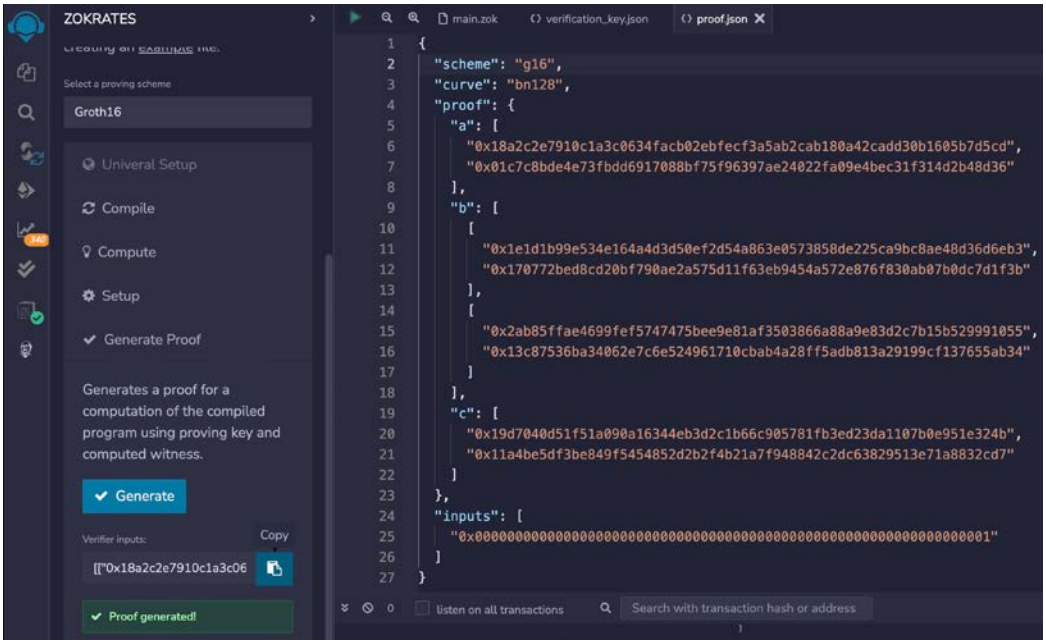


Figure 18.15: Generate proof

8. Once the proof is generated, it will generate a `proof.json` file containing the proof. Notice that on the left-hand column there is a **Verifier inputs** option; click on the **Copy** button to keep this proof in the clipboard or save it somewhere, because this is the input to the verifier smart contract that will be generated in the next step. We'll use this proof later on to pass it to the verifier contract for verification.
9. The next step is to generate the verifier; click on the **Export** button under the **Export Verifier** option to generate a solidity smart contract that contains a public function to verify the proof, i.e., the solution to the compiled Zokrates program. Note that this smart contract also contains the verification key. This is shown in *Figure 18.16* below:

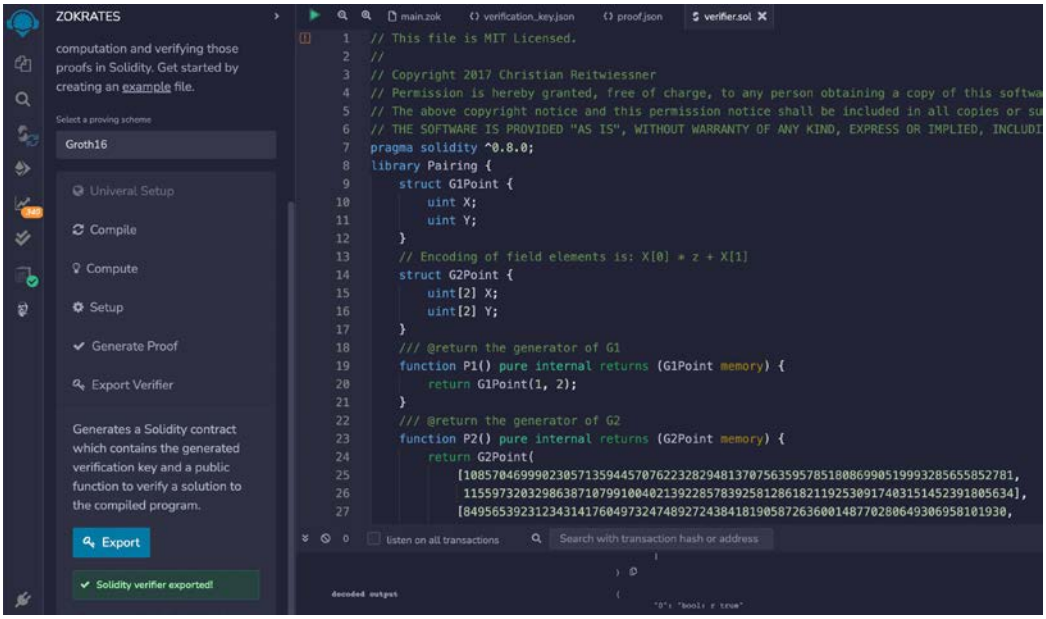


Figure 18.16: The verification smart contract in Solidity

This contract file name is `verifier.sol`. Now we can compile it and deploy it on the blockchain, either a *testnet* or *mainnet* Ethereum, as we learned in the previous chapters. If you want to deploy on the *mainchain* you can use MetaMask and select the **Injected web3** option in Remix IDE, as you learned before in the previous chapter.

Here we'll just use the Remix VM and run this example:

1. Now go to the **Solidity Compiler** option, and compile `verifier.sol`.
2. Under **DEPLOY & RUN TRANSACTIONS**, select `verifier.sol` under **CONTRACT** and click on **Deploy**.
3. Once deployed, simply invoke the `verifyTx` public function, by passing on the two parameters and clicking on the **call** button, as shown in the following screenshot in *Figure 18.17*:

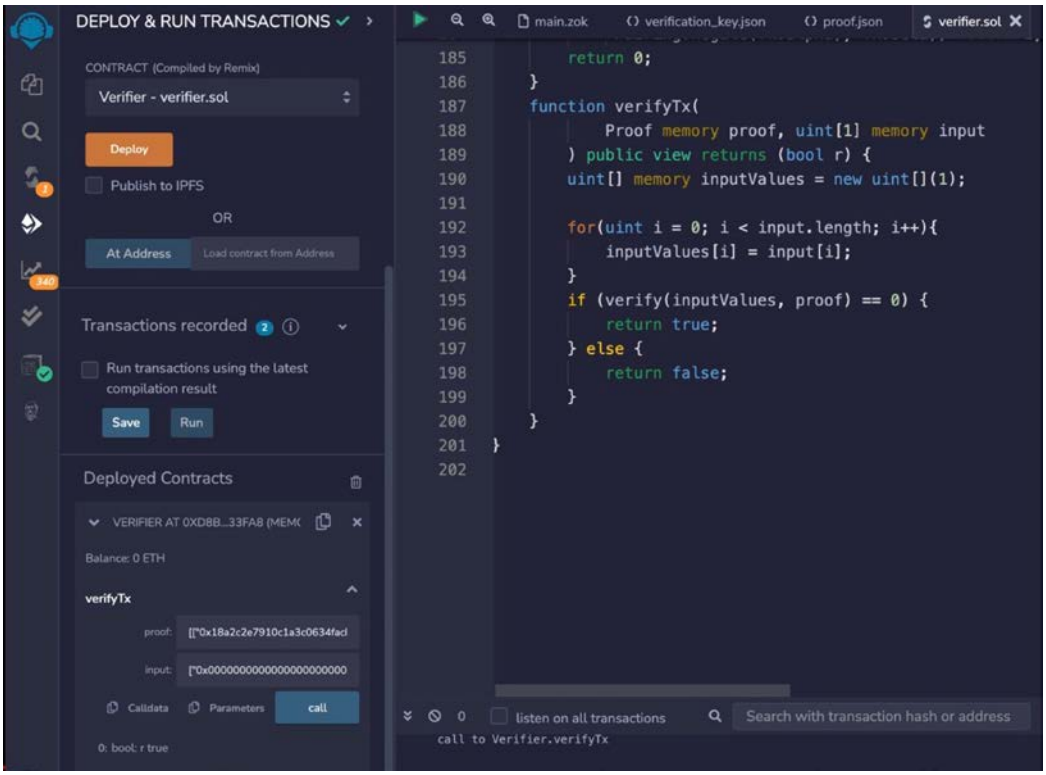


Figure 18.17: Invoking the verifier smart contract's verityTx public function

Notice that in the screenshot above, the `bool: true` value is returned, indicating that the verifier has been able to verify the proof. Remember that we did not reveal the value of x at all to the verifier contract but simply passed two parameters, the proof tuple and the input value, as shown below:

```
[["0x18a2c2e7910c1a3c0634facb02ebfecf3a5ab2cab180a42cadd30b1605b7d5cd",
"0x01c7c8bde4e73fbdd6917088bf75f96397ae24022fa09e4bec31f314d2b48d36"],
[["0x1e1d1b99e534e164a4d3d50ef2d54a863e0573858de225ca9bc8ae48d36d6eb3",
"0x170772bed8cd20bf790ae2a575d11f63eb9454a572e876f830ab07b0dc7d1f3b"],
["0x2ab85ffae4699fef5747475bee9e81af3503866a88a9e83d2c7b15b529991055",
"0x13c87536ba34062e7c6e524961710cbab4a28ff5adb813a29199cf137655ab34"]],
["0x19d7040d51f51a090a16344eb3d2c1b66c905781fb3ed23da1107b0e951e324b",
"0x11a4be5df3be849f5454852d2b2f4b21a7f948842c2dc63829513e71a8832cd7"]],
["0x0000000000000000000000000000000000000000000000000000000000000001"]
```

With this we complete our introduction to Zokrates, and how programs can be built and proven with zero knowledge. Note that we can write the same program in another way, as shown below, where we use the `assert` statement instead of the `if else` structure:

```
def main(private field x) -> bool {
    assert (x*x + 2 >= 11)
    return;
}
```

I'll leave this as an exercise for the reader, where you can compile and run this program using the Zokrates plugin and Remix IDE.

Now that we know how Zokrates works and how we can build zero-knowledge solutions, can you build a program that proves to a verifier with zero knowledge that you are over 18 years of age without revealing exactly how old you are? It's a simple example but can be used in various scenarios, for example, proving that someone is eligible to drive a car, without revealing their exact age or date of birth.

Overall great progress has been made in the research and development of zero-knowledge proofs, but there are still some challenges. For example, the key theme that's emerging is that while succinctness and universality are increasing, the prover speed is not improving much. High-end hardware is required to generate proofs in a reasonable time; this is the reason why rollup providers tend to be high-end machines on the cloud. Some have even suggested ASICs and FPGAs for this purpose.

If in the future somehow, prover times can be decreased while keeping or making the verifier times and proof sizes smaller, it would further improve user adoption and enable use cases where comparatively low-end consumer hardware, like mobile phones, would be able to generate proofs without affecting user experience. Quantum resistance is another challenge; while STARKs being quantum-resistant, this is another interesting area of research to develop more techniques. Some research can be conducted into building new polynomial commitment schemes that are even more efficient. Formal verification and ensuring that the security properties of these protocols are correctly designed and implemented is another subject of keen interest. Imagine a DeFi protocol handling billions of dollars' worth of digital assets; if I have formal proof that the zero-knowledge system implemented in DeFi will work correctly in all cases, it will increase consumer confidence. This is also applicable generally to not only any ZK protocols but also protocols in blockchain and software, which we'll discuss in more detail in the next chapter, *Chapter 19*, *Blockchain Security*.

Summary

This chapter covered blockchain privacy – a subject of profound importance. We covered what privacy is, its different facets, and how can we solve the privacy problem on blockchain. We also saw major technologies, such as ZKPs and particularly SNARKs, which are the most advanced and used form of ZKPs to solve privacy problems on the blockchain.

Moreover, we explored how Zokrates can be used to create zero-knowledge programs to complement the theoretical concepts, covered earlier in this chapter. Regarding privacy, we have many solutions, including but not limited to Zcash, Manta, Aleo, ZKOPRU, Aztec, and Espresso. To solve scalability, some key solutions are zkSync, Scroll, and Polygon Hermez.

In the next chapter we will introduce security, which is another challenge that blockchain technology faces.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

19

Blockchain Security

Other than scalability and privacy, there is another challenge that needs to be addressed in order for blockchains to become even more trustworthy. The issue is that of security. The security of blockchain is crucial because it is the backbone of many use cases, including sensitive ones like healthcare, finance, and supply chain management. With the advent of DeFi, it has become even more important to ensure the correct functioning of blockchain so that there are no vulnerabilities left in the blockchain that could be exploited, resulting in a loss of funds or other unintended consequences.

As the blockchain ecosystem is composed of several layers, there are security concerns at every layer, and we'll discuss them in this chapter. There are also layer 2 (side chains, etc.) related security concerns, which we'll also shed some light on in this chapter.

We'll cover the following topics in this chapter:

- Security in blockchain
- Background and historic attacks
- Blockchain layered model
- Threats and vulnerabilities at each layer of blockchain, including smart contract security, blockchain layer security, and security at other layers, how to address them, and best practices
- Layer 2 security concerns
- Tools and techniques to find vulnerabilities
- Models to perform threat analysis

So, let's begin by first looking at the subject of security in the context of blockchains.

Security

Blockchains are generally secure and make use of asymmetric and symmetric cryptography, as required throughout the blockchain network, to ensure that the core layer of the blockchain is secure. However, there are still a few caveats that can result in the security of the blockchain being compromised.

There have been many high-profile successful attacks on the blockchain ecosystem over the years. Some were outright malicious, and some were accidental; however, they all resulted in significant losses. Some examples include:

- The attack on Binance Smart Chain, which occurred on October 6, 2022. The hackers infiltrated the BNB cross-chain bridge Token Hub where they were able to mint fraudulent BNB tokens, through fake withdrawal proofs. Initially, the estimated damage was 566 million USD.
- In August 2021, the Poly Network theft, which resulted in 610 million USD being stolen. This attack exploited a vulnerability that occurred due to mismanaged access rights between Poly smart contracts.
- In February 2022, the Wormhole bridge exploit, which resulted in 321 million USD being lost. It occurred in February 2022, when a hacker found a vulnerability in the smart contracts that allowed him to mint 120,000 WETH on Solana not backed by collateral and was then able to swap this for ETH.
- In March 2022, Ronin Bridge, which was exploited for around 612 million USD. The reason for this exploit was unauthorized access to private keys, which resulted in compromised validator nodes, which consequently approved fraudulent transactions, draining funds from the bridge.
- The Luna collapse in May 2022, which wiped out an estimated 60 billion USD from the cryptocurrency world. This mainly occurred due to an unfitting algorithm behind the stable coin Luna where UST was backed by Luna, instead of a fiat currency.

Other hacks include MtGox, KuCoin, Pancake Bunny, Parity wallet hack, and many more. There have also been losses incurred due to poor management and controls, such as:

- **FTX collapse**: FTX was one of the biggest cryptocurrency exchanges in the world. It collapsed in November 2022 after a surge of customer withdrawals that FTX failed to fulfill due to not having enough assets in reserve to meet customer demands. Moreover, the situation was exacerbated due to FTX wallets being targeted and hacked by attackers, which resulted in almost 640 million USD being stolen. This was allegedly an insider attack by employees or the result of a malware attack. Eventually, FTX filed for bankruptcy due to liquidity crunches. The collapse of FTX left more than 1 million customers unable to withdraw assets worth an estimated 8 billion USD.
- **Binance client money issue**: In January 2023, Binance mistakenly mixed Crypto.com Exchange's customer funds with B-Token collateral. Thankfully, this mistake was realized before any material damage was done, but it highlights the importance of controls and compliance requirements. This issue, if not spotted earlier, could have led to owners not being able to withdraw money due to a lack of liquidity/funds by the exchange, as there is no segregation of assets between clients' money and any collateral used. This situation would have meant substantial reputational and financial loss due to a lack of liquidity if left unnoticed.

Now, before discussing security further in blockchains, let's have a look at a layered view of blockchain, which helps us to categorize different attacks efficiently. Generally, a blockchain ecosystem is composed of several layers and each layer presents its own security challenges. We can visualize a layered model in *Figure 19.1*:

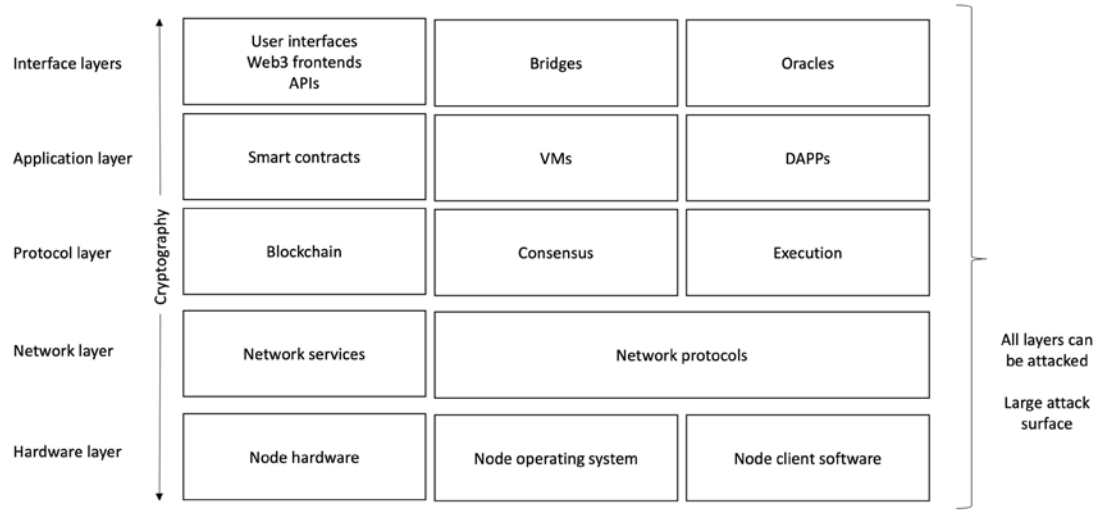


Figure 19.1: Blockchain layered view

In the preceding diagram, note that all layers can be attacked by an adversary and as such, the attack surface is quite large. In other words, it means that the attack surface spans across multiple layers, and an attacker can try to attack all layers, including the hardware, network, protocol, applications, and interface layers. Let's now see what each layer is composed of. Note that an attacker can try to find vulnerabilities in each layer and every component thereof. Also, as the financial gain from these attacks can be significant, hackers could be motivated to attack and try different ways to exploit any possible vulnerability.

Blockchain layers and attacks

In this section, we'll look at how we can see blockchain as a layered architecture to study security issues, attacks, and mitigation.

- **Hardware layer**: This is the core hardware on which a blockchain network node runs.
- **Network layer**: This is the core network layer on which blockchain protocols run. This layer includes protocols like TCP/IP, P2P, UDP, and other relevant services and protocols.
- **Blockchain protocol layer**: This is the blockchain layer where consensus, transaction execution, incentive management, and mining processes run.
- **Blockchain application layer**: This is the layer that consists of smart contracts and virtual machines, e.g., EVMs for executing smart contracts. DAPPs also exist at this layer, and so on.
- **Interface layer**: This layer includes bridges, oracles, governance, user interfaces for Web3 DAPPs, and APIs. While these components could possibly exist in the blockchain application layer, I think differentiating between these components here as a separate layer helps to define and address specific challenges better.
- **Cryptography layer**: Cryptography plays a vital role at each layer. Here, we will treat this as a separate layer, just for the purpose of discussion; otherwise, it exists in some shape or form at every layer in the blockchain ecosystem.

Each layer is open to an adversary to try and exploit any weaknesses or limitations. Blockchain is particularly interesting because while conventional software, hardware, and general information security concerns apply, because it uses conventional hardware and software to run, it also has its own challenges, such as smart contract security. The challenge is protecting the blockchain ecosystem at all layers, especially at the blockchain protocol and blockchain application layers.

In summary, we can say that there is a lot that needs to be secured in a blockchain ecosystem, including layer 1, layer 2, smart contracts, virtual machines, wallets, and even people. Why people? Because they can be victims of social engineering attacks. Moreover, people need to be educated about these threats so that they can identify social engineering patterns and protect themselves.

Cryptography plays an important role at every layer; therefore, we'll discuss cryptography as a separate layer/topic.

Now let's discuss attack vectors and mitigations at each layer.

Hardware layer

As this layer is composed of hardware that runs the operating system and node software, this layer is impacted by the usual threats, including viruses, malware, and unauthorized access. Blockchain-specific malware also exists, which specifically targets cryptocurrency blockchain network nodes. Such crypto malware is especially targeted to perform cryptojacking. Cryptojacking can be defined as the unauthorized use of computing resources that belong to someone to mine cryptocurrency.

Cryptojacking is a method of taking over a computer or web browser to mine for cryptocurrency without the user's permission. A successful cryptojacking attack doesn't necessarily result in software being installed on the victim's computer, but this can work within web browsers directly – so-called "in-browser mining." This occurs when a user browses to a website that executes a mining script on their browser. Even when the user leaves the website, the mining script can remain executing.

The malware used for cryptojacking is called **crypto malware**. It can infect operating systems including Windows, Linux, macOS, iOS, and Android. Browser extensions sometimes also include JavaScript to mine cryptocurrency, e.g., Archive Poster and Iridium. Due to cryptojacking, the machine performance is significantly reduced as CPUs and GPUs are running the mining process instead of performing the usual expected operations on the system. As we know, mining is a resource-intensive process, especially PoW, so it can be beneficial for hackers to hack into someone else's computer and run mining from their computer.

There can also be a **Denial of Service** (**DoS**) attack on a blockchain node, which can cause the resources on the computer to struggle to keep up with the incoming requests. In a DoS attack, the attacker floods the node with a large number of requests, overwhelming it and preventing it from processing legitimate transactions.

There can also be remote execution attacks via open ports, not only on the operating system but also in the node software, e.g., Bitcoin nodes' RPC port `8332`, which hackers can try to exploit. A list of vulnerabilities is available here for Bitcoin and Ethereum:

- Bitcoin: `https://www.cvedetails.com/vulnerability-list/vendor_id-12094/Bitcoin.html`
- Ethereum: `https://www.cvedetails.com/vulnerability-list/vendor_id-17524/Ethereum.html`

There may be other vulnerabilities that have not been discovered yet or could be introduced with future upgrades.

There is also a possibility that the operating system is outdated or even that the node software is outdated, which could open doors for exploitation by hackers. Also, misconfigured operating system or client node software could lead to a weakness that hackers can exploit. The hardware on which the node is running could also be vulnerable due to flaws such as Spectre, Meltdown, and Thunderclap. The firmware can also be exploited due to weaknesses such as Thunderstrike, ROCA, etc. It is important that standard precautionary and security measures such as system hardening, operating system upgrades, node software upgrades, and antivirus software are applied to the hardware node hosting the blockchain client software.

Blockchain node software can be attacked by traditional and blockchain-specific malware, which can result in the theft of private keys, the creation of rogue transactions, the censoring of some traffic, and resource starvation, resulting in a DoS.

Network layer

There can be several attacks that can be carried out on the network layer. We will discuss them next:

1. **Sybil attack**: First, we introduce the Sybil attack. A Sybil attack is a type of network attack in which an attacker creates and uses multiple fake identities, or "Sybil nodes," to gain an unfair advantage or perform malicious actions within a network. This can be accomplished by creating multiple fake identities and using them to gain a disproportionate level of influence within the network, such as by participating in a consensus process or voting system. The goal of a Sybil attack is to manipulate the network or disrupt its normal operation, often for personal or financial gain. If enough fake identities are created, this attack can be used to outvote the honest nodes on the network. These nodes can then take over the network and refuse to receive or transmit a block, effectively rendering the network useless. If attackers manage to control more than 51% of the hash rate, then they can reorder the transactions, censor transactions, and effectively take over the entire network. Consensus protocols are designed in such a way that Sybil attacks are not possible, though in some cases, due to weaknesses and bad design, Sybil attacks can still work.
2. **DoS attack**: The network can also be impacted by DoS, which could result in overwhelming the network with unnecessary traffic to make it inaccessible to users. While blockchain is a decentralized peer-to-peer network with SPOF, these networks are generally immune to network DoS attacks. However, some points of centralization exist in most blockchains, which can be targeted in a DoS attack; for example, boot nodes can be attacked, which are known and hardcoded in the blockchain client node software. Of course, here traditional network protection mechanisms to protect against DoS can be used, such as firewalls, intrusion detection and prevention systems, load balancers, rate limiters, and traffic shaping techniques.
3. **Eclipse attack**: An eclipse attack is a type of attack on a blockchain network in which an attacker tries to isolate a specific node or some nodes from the rest of the network, effectively "eclipsing" them and making them unable to communicate with other nodes, stopping them from receiving new information.

In this attack, a malicious node can control routes of communication between two segments of the blockchain network, which allows this node to control the information flow between these two network segments. This node can stop forwarding transactions and blocks to a victim node, resulting in it being "eclipsed." An eclipse attack can result in DoS because the victim node cannot receive and process any new transactions. It can also result in double-spending because if the targeted node is not able to communicate with the rest of the network, it is not aware of transactions that have been broadcasted and confirmed by other nodes. This can allow the attacker to double-spend their digital assets because the targeted node will not have the most up-to-date information about the blockchain.

4. **Network spoofing**: Another attack could be network spoofing. In blockchain networks like Ethereum and Bitcoin, boot nodes are hardcoded in the node client software and are used to provide a starting point for new nodes joining the network to start searching for neighbor nodes. It is possible for an attacker to launch a "spoofing" attack against a boot node by impersonating the boot node. New nodes will connect normally to this false boot node, which could result in a denial of service attack and other security implications.

Blockchain layer

The blockchain layer represents the layer 1 or layer 2 blockchain that is the core of the protocol. There are several attacks that can be carried out here.

Attacks on transactions

There are several attacks that can be carried out on transactions. The transactions are constructed by the users, and they can be malformed or invalid. As transactions are propagated to all nodes and all nodes process all transactions (except light nodes), rogue transactions or transactions especially crafted to cause harm can become an attractive attack vector for hackers. The transaction can contain a peculiar data structure, which can cause nodes to malfunction. Usually, this happens due to a bug in the node software, but theoretically, it is possible for a virus or malicious code to be spread through transactions in a blockchain network.

While invalid and incorrectly signed transactions with invalid signatures will be rejected by all nodes, it is possible that the transaction is properly signed and passes all checks before executing transactions and ending up executing some rogue code that results in an undesirable effect. One example is from 2010 when over 184 billion Bitcoins were created out of thin air due to an integer overflow vulnerability in the Bitcoin core node software. Some relevant attacks are transaction malleability, which allows a hacker to modify the unlock script of a pending transaction and still pass the transaction validation checks.

This attack would allow changing the transaction ID of the transaction, which could lead to a situation where the recipient of the transaction could claim that they never received the funds, tricking the sender to send the payment again. The Bitcoin core software is now quite stable and has gone through several bug fixes, upgrades, and rigorous testing, such as a segregated witness upgrade, which fixed the transaction malleability issue. Therefore, such attacks are quite difficult to achieve on a Bitcoin network. However, other cryptocurrency blockchains could have some zero-day vulnerabilities that can be exploited through malformed transactions. Even Bitcoin could still have some unknown vulnerabilities.

The serialization and deserialization of data for transactions and blocks is a standard operation that blockchain nodes do. It is possible that if some data structure contains malicious code, when it is deserialized by a node, it could result in integer overflow, buffer overflow issues, or even DoS. Nodes could end up executing malicious code if they are running old software that contains a specific bug related to this vulnerability where after being deserialized, the code doesn't check for all conditions.

It is also possible that due to the ability to encode some arbitrary data in a transaction, this can be achieved. For example, through a Bitcoin transaction's Coinbase field, the `OP_RETURN` opcode, or some other relevant technique. With this ability to embed arbitrary data in a transaction, it is possible for a special transaction to be crafted that, once executed, could result in the malfunction of not only nodes but also other software that reads transactions, such as Blockchain Explorer, third-party monitoring tools, and some enterprise backend software reading transactions from a private blockchain or even a public blockchain.

As blockchains have other surrounding software, including blockchain explorers, frontends for DAPPs, and other user frontends, it is possible, by using injection attacks, for the blockchain to be attacked. For example, inadequate input validation, cross-site scripting attacks, or inappropriate inputs, e.g., asking the user to enter their private keys in a web frontend, could lead to private key exposure and other unintended consequences, such as loss of funds, unintended execution of smart contracts, and rogue transaction execution or a DoS attack. Similar attacks can be performed at the block level too.

Transaction replay attacks

A replay attack is a type of vulnerability that arises when two separate cryptocurrencies based on a fork of the same original chain allow transactions to be recognized as valid on both chains.

In a transaction replay attack, an attacker captures a valid transaction from a network and tries to reuse it by "replaying" it on the same or a different network. This can be done by an attacker who has gained access to the transaction data, for example, by intercepting it over the network or by gaining access to a device that was used to initiate the transaction. If the transaction is replayed on a different network, it can potentially be used to transfer funds or other assets to the attacker's own account.

Replay attacks can be a problem in networks that use the same keys for signing transactions on different networks, or that do not have a mechanism in place to prevent the reuse of transactions. To protect against replay attacks, it is important to use unique keys to sign transactions on different networks and to implement a mechanism for detecting and preventing the reuse of transactions.

In 2016, Ethereum suffered from a transaction replay attack when the Ethereum network underwent a hard fork to resolve the "DAO hack" issue. A hard fork is a change to the protocol of a blockchain network that is not backward compatible, meaning that all nodes on the network must upgrade to the new version of the protocol or else they will be unable to participate in the network.

During the hard fork, the Ethereum network split into two separate networks: **Ethereum** (**ETH**) and **Ethereum Classic** (**ETC**). This meant that transactions on one network would not be recognized on the other network, and vice versa. However, because both networks were using the same keys to sign transactions, an attacker was able to capture a valid transaction on one network and replay it on the other network.

To fix this issue, Ethereum implemented a new transaction signing algorithm implemented under **EIP-155** that included a chain ID in the signed data. This chain ID is unique to each network and allows the network to verify that a transaction was intended for it and not another network. As a result, transaction replay attacks are no longer possible on Ethereum.

Recently, replay attacks have come into the limelight again after the merge (the Ethereum PoW to **Proof of Stake** (**PoS**) switch) in September 2022. In order to prevent replay attacks, the chain ID of the old forked-off Ethereum PoW chain was changed from 1 to 10,001. While this change protects against core chain replay attacks, it doesn't protect against some loopholes that emerged as a result of not updating smart contracts, DAPPs, and other peripheral components around the core chain. A lot of contracts remain vulnerable to replay attacks.

One key example is an attack on Omni Bridge smart contracts. This replay attack launched against the Ethereum old PoW chain and Omni Bridge resulted in the loss of 200 WETH from the Ethereum PoW chain. The attacker first transferred 200 WETH from the Ethereum new PoS chain (after the merge) and was replayed on the Ethereum PoW chain. Omni Bridge did not validate the chain ID before approving the transaction, which resulted in the PoW chain losing 200 WETH and attackers earning an extra 200 WETH. One thing to note is that this wasn't a true replay attack but a *calldata* replay; however, it highlights the importance of ensuring that whenever there are upgrades, and other changes in the core protocol, protection against replay attacks is baked into the new protocols. The transaction wasn't a replay attack on the chain, but a *calldata* replay attack, which occurred due to a flaw in the specific contract on the Gnosis chain. This raises a general concern about peripheral systems, especially bridges, which must verify the correct chain ID of the cross-chain messages. This can be seen as an attack at the interface layer too, which we'll discuss shortly.

Blockchain bridges have become a target for hackers. More on this shortly, when we discuss the interface layer.

Attacks on consensus protocols

There are several attacks that can be carried out against consensus protocols in blockchain, including:

1. **A 51% attack**: This occurs when a miner or group of miners controls more than 50% of the mining power on the network, allowing them to control the confirmation of transactions and potentially double-spend coins.
2. **Selfish mining**: This occurs when a miner withholds the blocks they mine from the network in order to increase their chances of finding the next block and earning the block reward.
3. **An eclipse attack**: This occurs when an attacker is able to isolate a node from the rest of the network, allowing them to control the information the node receives and potentially manipulate the consensus process.
4. **A Sybil attack**: This occurs when an attacker creates multiple identities or "sybils" in order to control a significant portion of the network and influence the consensus process.
5. **A nothing-at-stake attack**: This occurs in **PoS-b** based blockchain where validators are not required to put up any collateral, allowing them to vote on multiple chains and potentially undermine the integrity of the network.

There are several attacks against the PoS mechanism:

1. **Nothing-at-stake problem**: This attack occurs when there are only rewards in a PoS mechanism, and no penalties. This lax situation can result in a validator attesting any blocks and/or multiple forks/blocks of a blockchain to increase rewards for them.
2. **Stake grinding attack**: In this attack, an attacker tries to bias the validator selection algorithm to favor the selection of their own validators. This would result in the attacker's validator being selected more than their fair share and would result in minting more unjustified rewards.

Double-spending

In a double-spending attack, a malicious user sends the same cryptocurrency or token to two or more different recipients, effectively spending the same funds twice. There are different types of attacks that can result in double-spending. These attacks include a race attack, a Finney attack, and a 51% attack.

A race attack is a type of double-spending attack that occurs when a fast transaction is broadcasted before a slow transaction that spends the same coins.

In a Finney attack, a malicious actor first sends a transaction to a merchant for a certain amount of cryptocurrency and then quickly mines a new block that doesn't include this transaction. The attacker then sends a second transaction spending the same coins to a different recipient before the first transaction can be confirmed by the network. Since the attacker can mine the new block and validate the second transaction before the first transaction can be confirmed, the merchant is left with an invalid transaction and no payment. Finney attacks are low risk since they require the attacker to have significant computational power to mine a new block before the first transaction can be confirmed. However, they are still a potential threat and should be taken into consideration when designing and implementing cryptocurrency systems.

Selfish mining

A selfish mining attack can occur when an attacker manages to solve the PoW and mine a block but instead of announcing it to the network, they withhold it from the public chain. This would create a fork, and the attacker can then keep mining on the new forked chain, albeit only known to them, to build an alternative chain and get ahead of the public blockchain. When this malicious chain is ahead of the honest public chain, the latest block can be released to the network. As the network is programmed to accept the most recent block, the malicious fork will overwrite the honest/original chain. This attack can result in achieving virtually any type of malicious goal, including double-spending, stealing cryptocurrency, censoring transactions, and similar goals.

Forking and chain reorganization

Forking in blockchain refers to the creation of two or more parallel chains from a single blockchain. This can occur when multiple validators on the network generate blocks simultaneously, leading to two or more separate chains of blocks that are different from each other. Chain reorganization refers to the process of changing the current longest chain in a blockchain network. This can occur if a longer chain becomes available after a network split or if a malicious miner creates an alternative, longer chain.

An attacker can induce these attacks by using various methods, such as the nothing-at-stake attack, the long-range attack, or the selfish mining attack. In such attacks, a malicious validator may attempt to manipulate the blockchain by creating multiple chains or altering the longest chain, with the goal of compromising the integrity and security of the network.

Blockchain application layer

There are several vulnerabilities that can exist at this layer. The most prominent are smart contract-related vulnerabilities. What makes smart contract security particularly interesting is that there is no patching after deployment, which is different from traditional software. A proxy pattern can help here, but remember it is best to avoid bugs.

Smart contract vulnerabilities

Several security bugs in smart contracts have been discovered and analyzed in the wild. These include transaction ordering dependence, timestamp dependence, mishandled exceptions such as call stack depth limit exploitation, and reentrance vulnerability. Attacks can be classified into malicious actions, improper design (protocol or app), bugs, and social engineering. Now we present a list of smart contract vulnerabilities:

1. **The transaction ordering dependency bug** basically exploits scenarios where the perceived state of a contract might not be what the state of the contract changes to after execution. This weakness is a type of race condition. It is also called front-running and is possible due to the fact that the order of transactions within a block can be manipulated. As all transactions first appear in the memory pool, the transactions there can be monitored before they are included in the block. This allows a transaction to be submitted before another transaction, thus leading to controlling the behavior of a smart contract.
2. **Timestamp dependency bugs** are possible in scenarios where the timestamp of the block is used as a source of some decision-making within the contract, but timestamps can be manipulated by the miners (block producers). The *call stack depth limit* is another bug that can be exploited due to the fact that the maximum call stack depth of EVM is 1,024 frames. If the stack depth is reached while the contract is executing, then in certain scenarios, the send or call instruction can fail, resulting in the non-payment of funds. The call stack depth bug was addressed in the EIP-50 hard fork at `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md`.
3. **Send fail issue**: When sending funds to another contract, sending can fail, and even if `throw` is used as a *catch-all* mechanism, it will not work.
4. **Timestamp dependence** is another vulnerability that is quite common. Usually, the timestamp of a block is accessed via `now` or `block.timestamp`, but this timestamp can be manipulated by miners, leading to influencing the outcome of a function that relies on timestamps. Often, this is used as a source of randomness in lottery games to select the next winner. Thus, it might be possible for a miner to modify the timestamp in such a way that the chances of becoming the next winner increase.
5. Other usual software bugs, such as **integer overflow and underflow**, are also quite significant, and any use of integer variables should be carefully implemented in **Solidity**.

For example, a simple program where `uint8` is used to parse through elements of an array with more than 255 elements can result in an endless loop. This occurs because `uint8` is limited to 256 numbers.

6. **Reentrancy** is a common bug in smart contracts. The reentrancy bug was exploited in the DAO attack to siphon out millions of dollars into a child DAO. The reentrancy bug essentially means that a function can be called repeatedly before the previous (first) invocation of the function has completed. In other words, reentrancy attack occurs when an attacker controlled contract manages to call back into a smart contract before the first function call is complete.

   This is particularly unsafe in Ether withdrawal functions in Solidity smart contracts.

7. **Unguarded selfdestruct**: If there are no proper access controls implemented, anyone can destroy the contract by calling self-destruct. This famously occurred in Parity wallet self-destruction where a user accidentally triggered the self-destruct function in Parity smart contract code, which resulted in a loss of 513,774.16 Ether. Even though it was based on multi-signature and seemed to be a secure contract, the self-destruct function was not secured, so when called, it resulted in the deletion of the code.
8. **Unprivileged write to storage**: This attack occurs when a smart contract does not properly validate and restrict access to its state variables.
9. **Integer underflow and overflow** occur when an operation is performed on an integer that results in a value outside the range of the integer data type. For example, in a 256-bit unsigned integer, the minimum value is 0, and the maximum value is *2^256-1*. If an operation subtracts 1 from 0, it results in an underflow, and if an operation adds 1 to *2^256-1*, it results in an overflow. An attacker can use integer underflow and overflow to manipulate a contract's state or cause it to behave unexpectedly. For example, they could use an underflow to change the balance of an account to a large positive value or use an overflow to make a value appear to be 0. To prevent integer underflow and overflow in Solidity, developers can use libraries such as SafeMath, which automatically checks for underflow and overflow and reverts the transaction if it occurs. Additionally, developers can use specific data types, such as `uint256`, to limit the range of values an integer can take and write tests to validate the contract's behavior under different inputs.
10. **Oracle manipulation**: An oracle manipulation attack in blockchain occurs when an attacker modifies the data being provided to a smart contract by an oracle. This can be done by either compromising the oracle itself or by exploiting vulnerabilities in the smart contract that integrates with the oracle. These attacks can occur due to various vulnerabilities, such as:

    1. **Lack of data validation**: If the smart contract does not validate the data received from the oracle, an attacker can provide false or malicious data to the contract.
    2. **Unrestricted access**: If the oracle is accessible by anyone, an attacker can modify the data being sent to the contract.
    3. **Poorly designed access control**: If the smart contract does not have proper access controls in place, an attacker can modify the oracle to provide malicious data to the contract.

To prevent oracle manipulation attacks in blockchain, data validation, appropriate access control, and digital signatures can be used. In addition, a multi-oracle approach can be used where multiple oracles are used to provide the same data to the contract, which can reduce the risk of manipulation. If one oracle is compromised, the contract can still receive accurate data from the other sources:

1. **tx.origin**: In Solidity, `tx.origin` is a global variable that returns the address of the account that sent the transaction. Using this variable for authorization could render a smart contract vulnerable if an authorized account calls into a malicious contract. A call made to the vulnerable contract would pass the authorization check because `tx.origin` returns the original sender of the transaction, which in this case is the authorized account. In order to remediate this issue, do not use `tx.origin` for authorization; instead, use `msg.sender`.
2. **Transaction order dependence**: A transaction order dependence vulnerability can occur when the execution of a smart contract depends on the order in which transactions are processed on the blockchain. This can create a race condition where the outcome of a contract can be altered based on the order in which transactions are processed, potentially allowing attackers to manipulate the intended results. For example, a smart contract might have a function that rewards the first person to submit a solution to a problem. If two people submit solutions, one with a standard gas fee and another (attacker) with a higher gas fee, then the miner who confirms the block will choose the transaction with the highest gas price to be included first, resulting in awarding the prize to the person who paid a higher gas fee, instead of the person who solved the problem first. This vulnerability can also occur in ERC-20 contracts where the approve function allows one address to allow another address to spend tokens on its behalf. An attacker who is aware of a change in approval can submit a transfer request with a higher gas price, allowing them to receive more tokens than intended.
3. **Bad randomness**: Randomness is useful in many applications, for example, gaming apps or lottery apps, where a random winner is chosen based on the generated random number. Strong randomness is, however, not easy to generate. Usual sources like *block.timestamp* are insecure as a miner can choose to provide any timestamp within a few seconds and still have their block accepted by others. *blockhash* and *block.difficulty* are also insecure as miners have control over them. In order to remediate this issue, it's recommended to use an externally trusted source of randomness through multiple oracles or use a RANDAO.
4. **Using an outdated compiler**: Using an outdated Solidity compiler version could lead to issues, especially if there are publicly disclosed bugs that adversely affect the current compiler version. It is recommended to use a recent version of the Solidity compiler to avoid such issues.
5. **Unchecked call return value**: The message call return value is not checked. Execution will continue even if the called smart contract throws an error. If the message call fails accidentally or a hacker somehow manages to induce a failure on the call, such a situation can result in unexpected behavior in the program. It is recommended to check the return value of the call to handle the call failure. Also, sometimes it is recommended to avoid making external calls if possible. It might be possible for your contract to be manipulated by a third party/external malicious contract. This occurs because the execution control is transferred to an external contract during an external call. Attacks such as reentrancy and race conditions could occur due to this.

DeFi attacks

1. **Flash loan attacks**: In a flash loan attack, an attacker can borrow large amounts of funds with no collateral. This way, the attacker manages to move the market in their favor; e.g., they manipulate the price of a cryptocurrency asset on one exchange, gaining funds and then immediately selling it on another.
2. **Sandwich attacks**: A "sandwich attack" is a type of cryptocurrency front-running scheme that involves the attacker scanning for high-value transactions on the blockchain (in mempools), then placing a transaction ahead of the victim's with a higher gas fee to ensure it is processed first. This causes the price of the coin to increase, making the victim pay more. The attacker then sells their own coins, profiting from the difference in price. This tactic is referred to as a "sandwich attack" because it sandwiches the victim's transaction between two transactions submitted by the attacker. Over the period of May 2020 to April 2022, it was estimated that over 60,000 Ether, worth over 72 million USD, was lost due to these types of attacks.
3. **MEV/BEV**: MEV refers to the highest amount of value that can be gained beyond the regular block reward and gas fees through the manipulation of transactions within a block, such as by including, omitting, or altering the order of transactions.
4. **Stablecoins stability/security risks**: There can be several risks for stable coins, including liquidity risk, solvency risk, and regulatory risk. Stable coins are usually backed by reserve assets but if the demand for the stable coin increases, there may not be enough reserve assets to meet the demand, leading to a drop in its value.
5. **Identity spoofing**: In the DeFi ecosystem, identity and access control is of the utmost importance. It must be ensured that only authorized entities can operate on the network. This can be achieved by verifiable credentials and DIDs. We will discuss this in more detail in *Chapter 20, Decentralized Identity*. The key problem here is how to verify that an identity is true.
6. **Forged NFTs**: This is where a scammer creates art or even generates art online and claims it is by a famous artist. Also, in some cases, the scammer may take original art from an artist, copy it, create an NFT out of it, and sell it with their name. In both these situations, the original artist does not get their fair payment.
7. **NFT DoS**: This can occur due to the off-chain storage of actual NFT art where an attacker takes over or takes down the server where the actual art is hosted.
8. **Unlimited (scarcity-free) token generation**: There is a possibility that tokens are generated without any economic scarcity, which results in the devaluation of the asset.
9. **Airdrop hunting**: Usually, airdrops are limited in supply and a fixed number is intended for a single user; however, it might be possible for a scammer to create multiple identities (usually just multiple emails) and claim more airdrops than intended by the airdrop provider.

Next, we will discuss the interface layer.

Interface layer

Several attacks exist at the interface layer, as described below.

Oracle attacks/oracle manipulation attacks

The utilization of blockchain oracles poses a potential security risk due to the dependence on external information they provide. These oracles, typically implemented as smart contracts, have the capability to supply incorrect data, which can result in detrimental consequences for processes linked to the data feed. Such manipulation can have far-reaching effects, including unwarranted asset liquidations and malicious trading practices. In this context, it is crucial to examine and understand the common vulnerabilities and failures associated with blockchain oracles. Some oracle attacks include:

- **Tampering with data sources**: Attackers can manipulate the data sources that the oracles rely on to feed false information into the blockchain, potentially causing the smart contract to make incorrect decisions.
- **Sybil attacks**: Attackers can create multiple fake identities to control a large portion of the oracles, allowing them to manipulate the data that is fed into the blockchain.
- **Bribing oracles**: Attackers can bribe oracles to feed false information into the blockchain, potentially causing the smart contract to make incorrect decisions.
- **Oracle censorship**: Attackers can prevent oracles from accessing certain data sources, leading to a lack of information for the smart contract to make decisions.
- **DoS**: Attackers can overload oracles with too many requests, making them unavailable to the smart contract and potentially causing it to fail.
- **Freeloading attacks**: In this type of attack, a node can utilize another oracle or off-chain component, such as an API, and simply replicate the values without any validation. For instance, an oracle responsible for providing weather data may expect data providers to measure temperature at a designated location. However, nodes may be incentivized to use a publicly accessible weather data API and simply present their data to the system. This can lead to centralization issues with the data source and, if carried out on a large scale, can significantly affect the accuracy of the data. This is particularly noticeable when sampling rates vary, for instance, when the on-chain oracle expects a 10-minute sampling rate while nodes that are freeloading provide data from an API that is updated only once per hour. The prevalence of freeloading in decentralized oracle data marketplaces can exacerbate a downward price spiral, as freeloading requires only a simple data retrieval.

Attacks on wallets

Cryptocurrency wallets can be subject to several attacks, listed below:

- **Phishing attacks**: Where an attacker creates a fake wallet website or app that looks similar to a real one, tricking users into revealing their seed phrase or private keys.
- **Malware attacks**: Where malware infects the computer or mobile device used to access the cryptocurrency wallet and steals seed phrases or private keys.
- **Man-in-the-Middle (MITM) attacks**: Where an attacker intercepts and manipulates transactions by posing as a trusted entity during the communication between the wallet and the Ethereum network.

- **Security vulnerabilities in the cryptocurrency wallet code**: Where a hacker exploits a weakness in the wallet code to gain access to private keys and consequently the funds.

Hardware wallets can be subject to some attacks, listed below:

- **Supply chain attacks**: Where an attacker modifies the hardware wallet during production or distribution to gain access to private keys or seed phrases.
- **Physical tampering**: Where an attacker physically opens the hardware wallet and accesses the private keys or seed phrases stored within.
- **Malware attacks**: Where malware infects the computer used to connect to the hardware wallet and steals private keys or seed phrases.
- **MITM attacks**: Where an attacker intercepts and manipulates transactions by posing as a trusted entity during the communication between the hardware wallet and the computer.
- **User-level attacks**: If users do not follow instructions correctly or adopt good practices to secure their keys, then this could simply lead to a loss of funds. There are many examples from the wild ranging from lost hard disks containing Bitcoin wallets to forgetting passwords to wallets and even forgetting pin codes to hardware wallets.

Frontend attacks can include several types:

- **Malicious scripts**: Usual scripting attacks on a web page to steal keys or account details or create rogue transactions
- **Misaligned frontend**: WFE functionality/UI not matching the backend contract
- **Account hacking**: Account details phished/lost for the WFE
- **DoS attacks**: WFE SPOF

At the interface layer, it is possible for cryptojacking attacks to be carried out against browsers and the operating system.



Cryptojacking is the unauthorized use of someone else's computer to mine cryptocurrency. It is usually done without the knowledge or consent of the computer owner and can slow down the affected machine as it uses its resources to mine the cryptocurrency. The mining process involves solving complex mathematical puzzles to validate transactions on the cryptocurrency's network, and it requires a lot of computing power. Cryptojackers can use malware to infect a computer and then use its resources to mine cryptocurrency, or they can use browser-based scripts that run on a website and mine cryptocurrency using the resources of the visitor's computer. There are several ways to remediate cryptojacking. First, ensure up-to-date antivirus software is running on the host. Chrome extensions such as no coin or miner block can defend against cryptojacking attacks. Another technique is to null route the mining URIs by updating the host's file or DNS settings. Also, use adblocker extensions, which can stop the cryptojacking script from running.

So far, we've discussed attacks on layer 1 blockchains. However, with the advent of layer 2 blockchains, new classes of attacks have emerged, which we discuss next.

Attacks on layer 2 blockchains

Some of the attacks are similar to layer 1 attacks; however, some are novel and specific to layer 2:

- **Transaction censoring**: In layer 2 systems, if a rollup provider is centralized, that could result in transaction censoring.
- **Attacks on a rollup provider**: There can be several attacks against a rollup provider, including DoS, and any other usual attacks that could be carried out against a node, including viruses/malware and network attacks.
- **Data availability attacks**: There are two options available for data availability. One is to store the data on a chain to ensure its availability and the other is to store it off-chain with third-party entities, which will make it available when required. Usually, economic incentives are applied to reward off-chain entities to store data. However, badly designed incentive mechanisms or centralized service providers could censor the data, which can lead to unintended consequences. Also, in the case of storing data on a layer 1 chain, the efficiency and benefits of rollups would be limited.



Data availability is required to make the committed data available to any party outside the rollup provider (layer 2) to ensure liveness if the rollup provider fails or disappears for whatever reason (even malicious). Data availability is also required if some user wants to withdraw their funds and need to bypass the layer 2 entirely, either due to unavailability of layer 2, or malicious actors on layer 2, or simply "want their money back.

- **Blockchain bridge-related vulnerabilities**: This is a very common problem (as of early 2023). Several high-profile attacks like the Ronin Bridge attack, Wormhole attack, BNB smart chain attack, and Nomad Bridge have resulted in the loss of over a billion US dollars. Cross-chain bridges are very desirable features in the blockchain ecosystem as they enable cross-chain communication, which enables interoperability, results in better liquidity and provides services like cross-chain token transfers. However, there are several issues that can result in vulnerabilities. For example:
  - A badly designed bridge, if exploited, can lead to the loss of funds.
  - Also, the design of the bridges is usually quite complex, which can also lead to vulnerabilities that can be exploited by hackers.
  - There are several exploits and vulnerabilities, such as false deposits, validator design flaws, and validator takeover by malicious actors. Usually, these bridges work on the basis of events generated from one chain and read by the other chain to generate tokens or initiate a transfer on the other chain. If an attack somehow can produce a valid event without actually depositing funds on one chain (a false deposit), then it can result in generating new coins on the other chain, without really spending any coins on the first chain.
  - If the validators misbehave, collude, or are taken over by a malicious entity (adversary), then this can also lead to unintended consequences.

- **State channels and side chain-related attacks:** Attacks against state channels include channel jamming and replay attacks. Channel jamming is particularly problematic on a lightning network. It occurs when a DoS attack prevents nodes from routing while their liquidity is locked up and they are unable to earn fees because they cannot forward payments. An attacker can route the payment via other nodes under their control and refuse to finalize the payment. This situation will result in effectively "jamming the channel" until the HTLC time lock expires, and the payment is refunded.

  Side chain attacks can include:

  - Centralized block producers, which can lead to transaction censorship.
  - No inherited security from layer 1 (independent layer 2 security) can lead to stations where a side chain tries to run its own validator set and security mechanism, which could be at a lot lower level than layer 1. This can result in security issues and exploitation vulnerabilities on side chains.
  - No atomicity between two chains.
  - Vulnerable bridge.

- **DSL bugs**: As more and more **Domain-Specific Languages** (**DSLs**) like ZoKrates, Circom, and Cairo are being written to facilitate the creation of zero-knowledge proofs to be used in a layer 2 system, if they are not designed and/or implemented correctly, it could lead to language-level bugs, which can make the system vulnerable to certain attacks.

Cryptography layer

Several vulnerabilities and attacks can exist in the cryptography used at each layer:

Attacking public key cryptography

Public key cryptography is a crucial aspect of blockchain security and, generally, information security, but it is not 100% immune to attacks. Some common attacks against public key cryptography are listed below:

- **Brute-force attack**: A brute-force attack involves trying every possible combination of characters to find the private key. This attack can be prevented by using longer keys, but it becomes infeasible as the length of the key increases.
- **MITM attack**: In a man-in-the-middle attack, the attacker intercepts communication between two parties and replaces the public key of one party with their own public key. The attacker can then decrypt and read all messages sent between the two parties. A common example of a man-in-the-middle attack is *SSL stripping*, where the attacker intercepts communication between a client and a server and downgrades the connection to an insecure version, such as HTTP instead of HTTPS.
- **Impersonation attack**: In an impersonation attack, an attacker creates a fake certificate that appears to be from a trusted source, in order to trick a user into accepting the attacker's public key. Phishing attacks can be used to impersonate trusted websites and steal sensitive information.

This is particularly a problem in the blockchain ecosystem where users could be victims of phishing attacks or poor security of the web frontends for DAPPs make them vulnerable.

- **Key reuse attack**: A key reuse attack occurs when the same key is used for more than one purpose, making it easier for an attacker to gain access to the private key. For example, the **Wired Equivalent Privacy** (**WEP**) encryption protocol used in early Wi-Fi networks was vulnerable to key reuse attacks, as the same key was used to encrypt all packets. In blockchain networks, it's advisable to use a key only for the purpose for which it was generated.
- **Same key used for a long time**: As a good practice, if a private key has served its purpose, e.g., for signing a blockchain transaction, then it's advisable to use a new key for a new transaction, instead of reusing the same key again and again.
- **Side-channel attack**: A side-channel attack exploits information leaked from the implementation of a cryptographic algorithm, such as power consumption, electromagnetic radiation, or timing information, to obtain information about the secret key. For example, **Differential Power Analysis** (**DPA**) is a type of side-channel attack that analyzes the power consumption of a device to extract the secret key used in encryption.

Such attacks can be mitigated by using strong encryption algorithms and proper key management practices and verifying the authenticity of the public keys used in communication.

Attacking hash functions

Several common attacks that can be carried out against hash functions are as follows:

- **Collision attack**: A collision attack is an attempt to find two inputs that produce the same hash output, also known as a hash collision. A successful collision attack can compromise the integrity of digital signatures and other applications that rely on hash functions.
- **Preimage attack**: A preimage attack is an attempt to find an input that hashes to a specific output value. A successful preimage attack can compromise the security of password storage systems and other applications that use hash functions to store secrets.
- **Birthday attack**: A birthday attack is a type of collision attack that exploits the birthday paradox to find collisions more efficiently than a brute-force search. This type of attack is especially relevant for hash functions with smaller outputs, such as MD5 and SHA-1.
- **Length extension attack**: A length extension attack is a type of attack that allows an attacker to append data to a message after it has been hashed, without knowing the original message or hash value. This type of attack is possible when using hash functions with certain properties, such as the Merkle-Damgård construction.

In blockchain systems, usually SHA-256 and SHA-3 and Keccak are used, which are considered secure against these threats, but with a quantum computer, it might be possible to find some collisions.

Key management-related vulnerabilities and attacks

Key management-related vulnerabilities and attacks can compromise the confidentiality, integrity, and availability of encrypted data and result in significant financial and reputational damage. Some key management-related vulnerabilities and attacks are listed below:

- **Insecure key storage**: Insecure key storage can result in the exposure of private keys, making it easier for an attacker to sign illegitimate transactions. To mitigate this vulnerability, private keys should be stored in a secure location, such as a **Hardware Security Module** (**HSM**) or encrypted file, and access should be restricted to authorized personnel. It is also advisable to use a hardware wallet such as Trezor, SafePal, etc. to secure the private keys. User crypto wallets must be sure to use good cryptography and security practices to mitigate this threat.
- **Unauthorized key sharing**: Unauthorized key sharing can result in the exposure of private keys, making it easier for an attacker to gain access to encrypted data. To mitigate this vulnerability, access to private keys should be restricted to authorized personnel and policies should be in place to prevent unauthorized sharing.
- **Key loss or theft**: Key loss or theft can result in the permanent loss of access to encrypted data. To mitigate this vulnerability, keys should be stored in multiple locations and backups should be made to ensure that keys can be recovered in the event of loss or theft.
- **Key escrow attack**: Key escrow is a process in which a third party holds the private keys for encrypted data. Key escrow attacks can occur when the third party holding the keys is compromised, making it easier for an attacker to gain access to encrypted data. To mitigate this vulnerability, keys should be stored in a secure location and access should be restricted to authorized personnel.

ZKP-related attacks

There can be several attacks that could target zero-knowledge proofs:

- **Inadequate bit security**: If an inadequate bit length is chosen for security, that could result in a security compromise; e.g., if instead of choosing 4,096-bit security in RSA, only 512 bits are used, which would result in it being exploited, as 512-bit is easy to break. Similarly, in the context of layer 2, some claims have been made, for example, by StarkWare, that SHARP prover runs at 80 bits of security. However, the FRI commitment scheme only runs at 48 bits of security with very lax assumptions about its soundness. The formally proven security turns out to be at most 22 bits. The mechanism also use a 32-bit PoW puzzle to mitigate this limitation and achieve a higher security level, but this limitation could still be exploited. This means that in practice, the probability of forging a proof could be higher than expected.
- **Attacks on privacy**: There could be several attacks against privacy, including deanonymization attacks, confidentiality breaches, and identity disclosure attacks.
- **Digital signature vulnerabilities**: Digital signature malleability refers to the ability to modify a digital signature in a way that does not invalidate it but still modifies the underlying message. This vulnerability can have serious consequences for the security of digital signatures, as it can lead to undetected tampering with signed messages and the circumvention of cryptographic protocols.
- **Quantum threats**: It is possible, with the availability of quantum computers, for some schemes like ECC and RSA to be broken. Quantum computers pose a threat to the security of blockchain technology because they have the ability to break the cryptographic algorithms that are currently used to secure transactions on most blockchain networks.

For example, the popular public-key cryptography system RSA and **Elliptic Curve Digital Signature Algorithm** (**ECDSA**) can be broken by a large enough quantum computer, making it possible to compromise the security of private keys and steal cryptocurrencies. To mitigate the quantum threats against cryptography, several solutions are being developed, including:

- **Post-quantum cryptography**: This involves the development of new cryptographic algorithms that are quantum-resistant, meaning they cannot be broken by quantum computers.
- **Quantum-safe signature schemes**: This involves modifying existing signature schemes to make them quantum-resistant, such as the development of hash-based signature schemes, such as Lamport signatures and Winternitz signatures.
- **Quantum Key Distribution (QKD)**: This is a secure communication method that allows two parties to establish a shared secret key by transmitting quantum information, which is immune to eavesdropping.
- **Hybrid approaches**: This involves combining classical cryptographic algorithms with quantum-resistant algorithms to enhance security against quantum attacks.

- **Proof malleability**: Getting the cryptography right is quite tricky, which is especially true with zk-SNARKs, which is a somewhat new subject. Problems with the underlying mathematics and then implementation can lead to unintended consequences. One of the issues is proof malleability where an attacker can alter a proof without invalidating it. In other words the attacker can manipulate the proof in a way that it still appears to be valid and authentic, but its content has been altered. This can lead to various security issues, such as replay attacks, where same proof can be used multiple times for different messages.
- **Setup vulnerabilities**: If the trusted setup is not conducted properly or is compromised and secret values leak, then it can allow a prover to create false statements. It is important to delete the secret values in the trusted setup, i.e., so-called toxic waste, forever.

Security analysis tools and mechanism

There are several techniques to check the correctness of programs. First, we have unit testing, which is a common method. Second, we have property-based testing, i.e., fuzzing. At the next level, we have model checking, and finally, formal proofs are the most advanced technique to ensure the correctness of programs, in this case, smart contracts. Testing includes unit tests, integration tests, full end-to-end tests, and property-based testing, also called fuzzers.

Static analysis allows us to check the code against a set of coding rules to find code defects. The code does not execute; instead, it is statically checked. On the other hand, there is a dynamic analysis technique where the code is executed to find bugs and is tested against test criteria. Dynamic analysis usually constitutes unit tests and is not considered a formal verification technique. Static analysis using formal techniques is used to formally verify the correctness of the program.

There are three types of proofs. We are familiar with pen-and-paper proofs, which are written by hand and verified manually. This can be prone to errors. Secondly, we have proof assistants, which provide support while writing proofs.

Proofs are still written manually by a human, but the proof can be checked for correctness automatically. Also, the tool provides support in writing the correct proof. Finally, we have automated proofs, but this suffers from an undecidability problem and exponential search space. It's worth noting that it takes a lot of effort to build formal proofs, so it only makes sense to use a formal verification approach for code and algorithms that are complex and where correctness is hard to establish. Usually distributed systems, concurrent systems, and safety-critical systems require formal verification. For straightforward code, it is not worth spending time and effort on building formal proofs.

For smart contract formal verification, some common tools include VERX and KEVM. The formal verification of smart contracts can allow the detection of complex bugs, which are hard to detect manually, or using simple automated tools or unit testing. The formal verification of smart contracts proves that the smart contract satisfies a formal specification of its functionality. Manual analysis, while a cumbersome process, has its place too to ensure the correctness of smart contract code. Auditors can manually go through code to try and find bugs and vulnerabilities. However, this method is not very effective and is very time-consuming.

While the usual testing of code with automated tools, where unit tests are written, is usually an acceptable technique, code coverage is not a hundred percent guaranteed in some cases, leading to bugs in the software. Manual analyses and, in fact, most methods suffer from false negatives and false positives. False negatives can be defined as the inability to find vulnerabilities that actually existed in the code. False positives can be defined as the report of the existence of vulnerabilities that are actually not vulnerabilities. Symbolic execution is a technique for checking program correctness. It uses symbolic inputs to represent a set of states and transitions formulated in a precise mathematical language.

Formal verification has become a very interesting topic in the blockchain space, due to its strong technical merits as a technique to ascertain the correctness of code, algorithms, and protocols related to blockchain. Note that formal verification is not a new technique, but due to the strict security requirements of blockchain systems, it has found suitable applications in blockchain technology.

Formal verification

Before diving into different formal verification techniques that are available in the blockchain space, first, let's develop some understanding of what **formal verification** is, what its types are, and why it is desirable.

Formal methods are the set of techniques used to model systems as mathematical objects. These methods include writing specifications in formal logic, model checking, verification, and formal proofs. Generally, formal methods can be divided into two broad disciplines called **formal specifications** and **formal verification**. The first discipline is concerned with writing precise and concrete specifications, whereas the latter encompasses the development of proofs to prove the correctness of a specification.

In essence, formal verification is comprised of three steps:

1. First, create a formal model of the system to be checked.
2. Second, write a formal specification of the properties that are expected to be satisfied by our model.
3. Finally, the model is checked to ensure that the model satisfies the specification.

For checking, there are two broad categories of techniques that can be used, namely *state exploration-based* approaches and *proof-based* approaches. There are pros and cons to both:

- State exploration-based approaches, where all possible states are checked, are automatic but are inefficient and difficult to scale. A usual problem is state explosion, where the number of states to check grows so exponentially big that the model does not fit in a computer's memory.
- On the other hand, proof-based approaches (theorem proving) are more precise but are somewhat challenging to implement because they require manual proof writing by a human. Even though these proofs are assisted by the proof assistant, they require more in-depth knowledge of the proofs and more time and effort to implement.



Proof-based verification is performed using proof assistants such as Coq (`https://coq.inria.fr/`) and Isabelle (`https://isabelle.in.tum.de/`).

Formal verification of smart contracts

A lot of research on the formal verification of smart contracts has also been conducted. The proposed techniques include, but are not limited to, model checking, static analysis, dynamic analysis, and verification using proof assistants.

Smart contract security is a very exciting and vast area of research. It is almost impossible to cover all the aspects in this short chapter. You are encouraged to read the introduction to formal verification earlier in this chapter and then go through some of the preceding papers for further research.

After this discussion on smart contract security, we should now understand its significance. While all the issues and techniques discussed here to mitigate the security issues in smart contracts are valuable, a question still arises about what else we can do, given that smart contract security is a sensitive topic that can result in serious financial losses.

The answer could be that we treat smart contracts and the underlying blockchain as safety-critical systems, and then apply all attributes of safety-critical systems to smart contracts. From an engineering perspective, it will become a subject of safety-critical systems and will be treated as such. This idea might sound a bit over the top, but imagine if, one day, a smart contract is responsible for generating an event that would trigger a shutdown as a result of overheating in a nuclear reactor. It's only logical that every safety precaution is taken. Again, a little exaggerated perhaps, but entirely possible. With IoT and blockchain convergence, such scenarios might become a reality soon, where nuclear reactors are running using a blockchain as a mechanism to control and track associated operations. Similar critical scenarios can be found in healthcare, aviation, and defense.

In Solidity, formal verification is achieved through the use of **Satisfiability Modulo Theories** (**SMT**) and **Horn solving**, where the `SMTChecker` module analyzes the code to ensure that it meets the conditions outlined in the require and assert statements. If an error is found, the `SMTChecker` provides a counterexample to the user. If no errors are found, it is assumed that the code is safe. The `SMTChecker` also checks for potential issues such as arithmetic overflow and underflow, division by zero, unreachable code, access to an out-of-bounds index, insufficient funds for a transfer, and more.

We just introduced two new terms, SMT and Horn solving. Let's see what they are.

SMT is a well-established decision procedure that assesses the satisfiability of logical formulas in the presence of predefined theories. This technique is widely used in the field of formal verification and automated theorem-proving for evaluating the correctness of various systems. The SMT solver processes a logical formula as input and returns a binary outcome of either `"SAT"` or `"UNSAT"` to indicate whether the formula can be satisfied given the predefined theories. The application of SMT has been shown to provide significant benefits in the development of high-quality systems, as it enables the automatic proof of desired properties and the detection of potential bugs and errors before they occur.

Horn solving is a technique used in formal verification to automatically prove the correctness of a program or system. It is based on the notion of Horn clauses, which are a type of logical statement made up of one or more literals and at most one positive literal. In formal verification, the Horn solver is used to check if the program satisfies a set of Horn clauses, which represent the desired properties of the system. Horn solving works by reducing the proof obligation to a set of Horn clauses and then using a theorem prover to check if there is a solution that satisfies all the clauses. The theorem prover applies a combination of decision procedures and search algorithms to find a solution. If a solution is found, it is proof that the program meets the desired properties. If no solution is found, the Horn solver produces a counterexample, which is a trace of execution that violates one of the properties.

In the next section, we'll introduce a formal verification technique called model checking, which is used to check a system model for its correctness.

Model checking

Model checking can be defined as a technique used to verify finite state systems automatically. The underlying process is based on running an exhaustive search of the state space of the system. With this brute-force approach, it can be determined whether a specification is true or false. This option is becoming more popular because it does not require time-consuming proof writing.

Instead, this paradigm allows a designer to write the specification and its properties formally and the model checker will automatically explore the entire state space and evaluate the model against the specified properties. For example, if the specification says that a particular state should never be reached, and during state exploration, the model checker finds that there is an execution (trace) that does enter that very state that is undesirable, then it will report that. The designer can then address the problem accordingly.

A visual representation of the model checking process is shown in the following diagram:

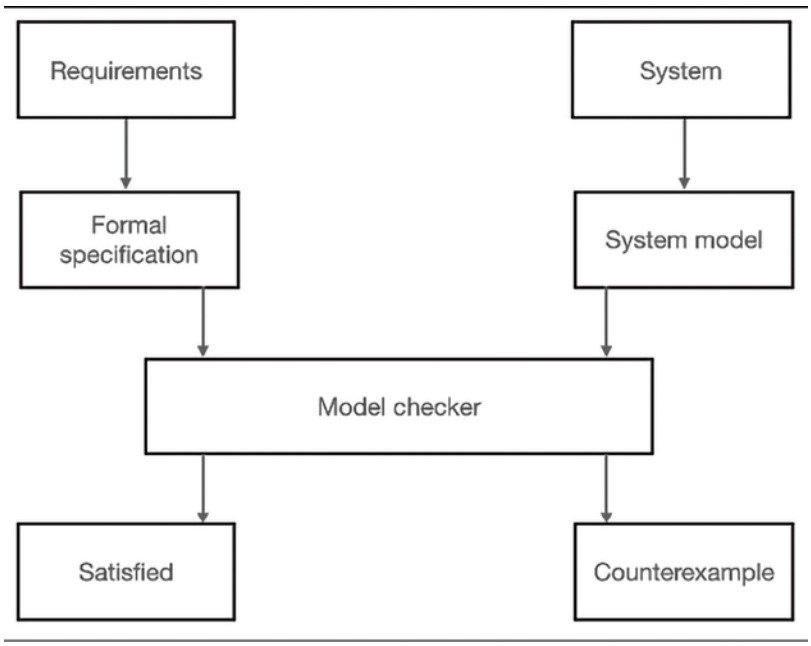


Figure 19.2: Model checking

The process of model checking starts with modeling, where a formal specification is created from an informal design of the model. The specification step comes next, where relevant properties of the design are specified using some logical formalism. Finally, verification of the model is carried out, which checks all the properties defined in the specification and provides results. In the case of false or negative outcomes, error traces with counterexamples (roughly defined as exceptions) are provided to help the designer trace the error.

Mathematical assertion languages called temporal logics are used to describe the states of a system over time. Temporal logic is used at the specification step of the model checking process, which represents the evolution of the behavior of a system over time, i.e., the ordering of events of a system over time. In this paradigm, a specification of the behaviors of the system over time is built using various fundamental properties of individual states. For example, in a state, it can be said that "something is on" or "something is active." There are also propositional connectors used in the specification design, such as *AND* $\wedge$, *OR* $\vee$, *NOT* $\neg$, and *IMPLY* $\Rightarrow$.

Finally, there are also temporal connectives, which state something like the system always only has one process active, or the system can never be in a state where both read and write operations are performed simultaneously.

There are two types of temporal logic, namely **Linear Temporal Logic** (**LTL**) and **Computation Tree Logic** (**CTL**). LTL is concerned with the model properties on a single computation path or execution, whereas CTL is used to express properties over a tree of all possible paths.

There are a few fundamental temporal operators used in LTL to describe a system's behavior. These properties can be used to specify the safety and liveness properties of a system. The formulas are $\Box$P (always), $\Diamond$P (eventually), *U* (until), and $\bigcirc$ (next). The always and eventually properties are most used.

For example, $\Box$ *((message)*$\neg$*signed* $\vee$ $\neg$*sealed)* $\Rightarrow$ $\bigcirc$ $\neg$*valid)* means that it is always true that if a message is not signed or sealed, then the message is not valid. Another example could be $\Box$ *((broadcast)* $\Rightarrow$ $\Diamond$ *received)* with broadcast implying that eventually (sometimes referred to as possible or in some future time) the message will be received. Another general example could be that of a microwave oven ensuring that heating must not start until the door is closed, which can be represented in temporal logic as $\neg$*Startheating U Door(closed)*.

Formal methods are needed to ensure the safety of a system and have been in use in the avionics, electronics, hardware, and embedded systems industries for quite some time, as well as most recently in the software industry. Due to the renewed interest in distributed systems research with the advent of blockchain, and several high-profile incidents resulting in financial loss, such as the infamous **DAO hack**, the need to use formal methods in the blockchain space is becoming increasingly prominent. Some initiatives have already been implemented; however, there is still a long way to go.

Blockchain needs to leverage the research that has already been done in the distributed systems and formal verification space, so that not only can existing blockchain systems be made safer but new platforms can also be developed based on a formal specification. It should be noted that not all aspects of a blockchain need to, or should, be formally verified, due to the time-consuming and highly involved nature of the verification process. Therefore, at times, it is sufficient to model and test the most critical parts of a blockchain, such as consensus and security protocols.

There are many dimensions in a blockchain where model checking and specification using temporal logic can be quite useful. It can be used to describe an aspect of the blockchain platform formally, which can then be verified formally to ensure that all required properties are met. For example, in smart contract development, regardless of the language used, a model could be developed and checked before the actual coding of the smart contract. This approach will ensure that the program code, if programmed correctly according to the verified specification, will behave as it is intended. As smart contracts are used in all kinds of critical use cases, including financial transactions, it is advisable to apply model checking in this domain to ensure the fairness and security of the system.

Now, we'll briefly look at how consensus mechanisms can be verified.

Verifying consensus mechanisms

From another angle, a **consensus mechanism**, such as **PBFT**, exists in a blockchain, which ensures that all the nodes in a blockchain network reach an agreement on the proposed values, even in the presence of faulty nodes. Consensus mechanisms such as PBFT and Raft were described in *Chapter 5*, *Consensus Algorithms*.

This is an area of prime importance and due to its critical nature is regarded as the most vital core mechanism of a blockchain. Model checking can play a crucial role in the formal description and verification of consensus algorithms, to ensure that the protocols meet the required security properties.

A general approach to verifying the properties of a consensus algorithm in blockchain starts with specifying the requirements, required properties, and specification in a formal language.

There are several options available for modeling and verifying programs, but tools such as TLA+ and the TLC model checker are making the processes more accessible and are becoming more prominent in this space. Another popular tool is known as SPIN, which uses PROMELA to write specifications. However, note that there is no single right answer: the choice of formal verification tools mostly depends on the judgment and experience of the designer, the type and depth of verification required, and, to a certain degree, the usability of the verification tools.

A distributed consensus algorithm is evaluated against two categories of correctness properties, namely *safety* and *liveness*. Safety generally means that nothing bad will happen, whereas liveness implies that something good will eventually occur. Both of these properties, then, have some sub-properties, depending on the requirements. Usually, for consensus mechanisms, we have *agreement*, *integrity*, and *validity* attributes under safety properties and for liveness, often, *termination* is a desired property.

We can say that a program is correct if, in all possible executions, the program behaves correctly according to the specification. The specification is formally defined, which is then verified using a model checker or theorem provers.

With this, we've completed our basic introduction to formal verification. Now, let's have a look at smart contract security and see what formal tools are available for smart contract security verification.

Smart contract security

Recently, a lot of work has been started relating to smart contract security and, especially, the formal verification of smart contracts is being discussed and researched. This was all triggered especially by the infamous DAO hack and other attacks.

Formal verification is a process of verifying a computer program to ensure that it satisfies certain formal statements. This is not a new concept and there are a number of tools available for other languages that achieve this; for example, Frama-C (`https://frama-c.com`) is available for analyzing C programs.

The key idea behind formal verification is to convert the source program into a set of mathematical statements that is understandable by the automated provers. For this purpose, **Why3**, which is a platform for program verification, is commonly used.



Note that an experimental but operational Why3 verifier was available in Remix IDE initially but was later removed. However, now, in Remix IDE, static analysis options are available. Details can be found here: `https://remix-ide.readthedocs.io/en/latest/static_analysis.html`.

Smart contract security is of paramount importance now, and many other initiatives have also been taken in order to devise methods that can analyze Solidity programs and find bugs. Some examples include **Oyente**, Manticore, and Slither, which are the tools built by researchers to analyze smart contracts.

Smart contracts and, generally, all aspects of blockchain can be formally verified. Static analysis of Solidity code is also now available as a feature in the Solidity online Remix IDE.

The code is analyzed for vulnerabilities and reported in the **Analysis** tab of Remix IDE. Static analysis in Remix IDE analyzes several categories of vulnerabilities, including **Security** and **Gas & Economy**. Other than static analysis capabilities within Remix IDE, other tools are also available, which we'll discuss next:

- Slither is a static analyzer framework for smart contracts.
- Manticore is a **symbolic execution** framework that can be used for binaries and smart contracts. More information on Manticore can be found here: `https://arxiv.org/pdf/1907.03890`.

Symbolic execution is a method used to analyze computer programs to determine which part of the computer program executes as a result of what input. In other words, it establishes which input executes which part of the program. In symbolic execution, instead of actual input data, symbolic values are used, which are then used to evaluate the program. Also, an automated theorem prover is used to check whether there are values that result in incorrect behavior of the program or cause it to fail. It is used for debugging, software testing, and security.

Oyente

Currently, **Oyente** is available as a Docker image for easy testing and installation. It is available at `https://github.com/melonproject/oyente` and can be downloaded and tested. In the following example, a simple contract taken from the **Solidity** documentation that contains a reentrancy bug has been tested. First, we will show the code with a reentrancy bug:

```
1  pragma solidity ^0.4.0;
2  contract Fund {
3      mapping(address => uint) shares;
4      function withdraw() public {
5          if (msg.sender.call.value(shares[msg.sender])())
6          shares[msg.sender] = 0;
7
8      }
9  }
```

Figure 19.3: Contract with a reentrancy bug, sourced from the Solidity documentation

This sample code contains a reentrancy bug, which basically means that if a contract is interacting with another contract or transferring ether, it is effectively handing over the control to that other contract. This allows the called contract to call back into the function of the contract from which it has been called, without waiting for completion. For example, this bug can allow calling back into the `withdraw()` function shown in the preceding example again and again, resulting in obtaining ETH multiple times. This is possible because the share value is not set to `0` until the end of the function, which means that any later invocations will be successful, resulting in withdrawing again and again.

An example is shown of Oyente running to analyze the contract shown here, and as can be seen in the following output, the analysis has successfully found the reentrancy bug.

The bug is proposed to be handled by the `Checks-Effects-Interactions` pattern described in the Solidity documentation:

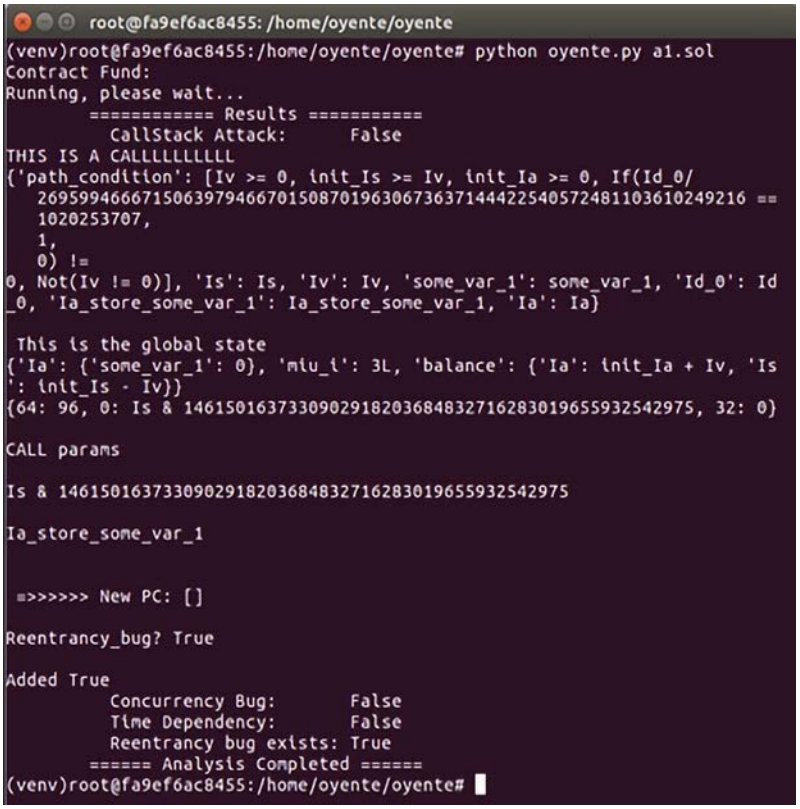


Figure 19.4: Oyente tool detecting Solidity bugs

With this example, we conclude our introduction to security and analysis tools for Solidity. This is a very rich area of research, and more and more tools are expected to be available with time.

Solgraph

Visualizing the function control flow in a program makes it easy to understand the flow and analysis better. A tool called Solgraph can be used to generate a graph of the function control flow of a Solidity contract and finds possible security vulnerabilities. More information on this tool is available here: `https://github.com/raineorshine/solgraph`.

Threat modeling

Threat modeling is the process of identifying, analyzing, and documenting potential threats to a system or application. The goal of threat modeling is to identify vulnerabilities and potential attack vectors so that appropriate security controls can be put in place to mitigate the risk. Threat modeling is a standard practice carried out in traditional information security scenarios; however, the same techniques can be applied to blockchain as well.

There are many ways threat modeling can be performed for blockchain by using standard threat modeling techniques. There are several different standard models that can be used for threat modeling, including:

- **STRIDE:** This model, developed by Microsoft, helps identify and categorize threats based on the type of threat they pose. STRIDE stands for Spoofing, Tampering, Repudiation, Information Disclosure, DoS, and Elevation of Privilege.
- **DREAD:** This model, developed by the **Open Web Application Security Project** (**OWASP**), helps assess the risk of a threat based on its likelihood of occurrence and the potential impact it could have. DREAD stands for Damage, Reproducibility, Exploitability, Affected Users, and Discoverability.
- **CVSS:** The **Common Vulnerability Scoring System** (**CVSS**) is a standardized method for rating the severity of vulnerabilities based on their impact and likelihood of exploitation.
- **Attack trees:** This model represents threats as a tree structure, with the root node representing the attack goal and the child nodes representing the various steps or actions required to achieve the goal.
- **Threat matrix:** This model is a visual representation of the likelihood and impact of potential threats. It allows analysts to prioritize threats based on their potential risk.

We'll describe STRIDE in a bit more detail now, and how it can be applied to blockchain. Note that other models can also be used and adapted for use in the blockchain ecosystem.

STRIDE is a security model that helps identify potential threats and vulnerabilities in a system or network. It stands for:

- **S – Spoofing:** This refers to the act of impersonating someone or something to gain access to a system or information.
- **T – Tampering:** This refers to the act of altering or manipulating data or information in a system.
- **R – Repudiation:** This refers to the ability to deny that an action or event took place.
- **I – Information Disclosure:** This refers to the act of disclosing sensitive or confidential information to unauthorized parties.
- **D – DoS:** This refers to an attack that is designed to make a system or network unavailable to legitimate users.
- **E – Elevation of Privilege:** This refers to the act of gaining unauthorized access to a system or information with higher privileges than one's own.

STRIDE helps identify and prevent these types of threats by examining the system or network and looking for ways to reduce the risk of these types of attacks. This can be done through a variety of methods, such as implementing strong authentication and access controls, encrypting sensitive data, and regularly updating and patching systems to fix vulnerabilities.

Benefits of using the STRIDE model include:

- It helps identify potential threats and vulnerabilities in a system or network.

- It provides a systematic approach to evaluating and addressing security risks.
- It can be used to develop a comprehensive security plan for a system or network.

To implement STRIDE, we would first need to identify the assets and resources in the blockchain ecosystem that need to be protected. We would then need to evaluate the potential threats and vulnerabilities to those assets and determine the appropriate measures to reduce the risk of those threats. This could include implementing good practices and implementing controls and technical practices to prevent and mitigate **s**poofing, **t**ampering, **r**epudiation, **i**nformation disclosure, **D**oS, and **e**levation of privilege attacks.

For example, if we apply STRIDE to do threat modeling on the consensus layer, we can see how for each dimension of STRIDE, we can model the impact of the attacks:

- **S –** Spoofing: In a consensus protocol, it might be possible to introduce a byzantine node that appears to be a legitimate node (validator). The malicious validator node can take over and impersonate an honest node.
- **T –** Tampering: Collusion or sybil attacks in a consensus protocol could result in the blockchain being written by rogue blocks with transactions that are created by the attacker.
- **R –** Repudiation: It is possible, due to tampering, for the attacker to repudiate past transactions or, due to control over the network (by collusion), simply repudiate any previous transaction, resulting in the loss of funds.
- **I –** Information disclosure: As the attacker has influence over the network due to tampering and spoofing, it could be possible for the attacker to extract sensitive information from the blockchain.
- **D –** DoS: It could be possible, e.g., in a BFT protocol, for a "follow the leader" type of attack to be launched. In this case, the attacker predicts or finds out due to a simple leader election algorithm which node is likely to be the next proposer (leader) node. The attacker can "follow the leader" and keep launching DoS attacks or malware attacks against that single node that is expected to be the next leader or is already a leader. If the DoS attacks are successful, then there won't be any new block proposals made to the network because every time a leader is elected, the attacker attacks it with DoS or even launches DoS against the next expected leader. This can consequently stop block generation on the network, resulting in DoS.
- **E –** Elevation of Privilege: It is possible, with a byzantine node introduced in the system, for the attacker to gain elevated control over the network and thus the blockchain state.



Note that all STRIDE dimensions aren't always applicable to every attack or layer. Sometimes only, for example, S or R is applicable, but not the other dimensions.

With this, we complete our introduction to threat modeling. STRIDE and other threat modeling techniques can be used in many other scenarios on the blockchain ecosystem, at every layer.

Regulation and compliance

Regulatory compliance is another dimension of blockchain security. How do we ensure that the blockchain platform is GDPR-compliant? How do we ensure that the blockchain platform is compliant with the enterprise security policy? Such questions fall into the category of regulatory compliance-related security requirements. There are certain requirements around data localization, e.g., customer data may not leave a specific geographic location. In such scenarios, a globally replicated blockchain may not be the best solution. In those cases, restricted private transactions (introduced in *Chapter 16*, *Enterprise Blockchain*) could be a useful construct to follow. In other situations, the use of off-chain transaction managers located in specific geographic locations could be helpful. In certain cases, specific cryptographic protocols are mandated by organizations such as NIST. For example, specific curves are mandated in SP 800-186 by NIST for federal agencies, which means that for a client to be used in government settings, it must use the same cryptography as mandated by NIST for US government use. Adhering to standards also facilitates interoperability, better security, and faster development. The document can be found here: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf`.

Blockchain can itself be used for malicious purposes. Blackhat hackers can use it as a vehicle to get paid for ransomware, for example, Petya, WannaCryptor, or Locky. There have been many ICO scams and Ponzi schemes. The point to note here is that blockchain is a tool that can be used for both good and bad. For example, Tornado Cash could be used for charitable purposes or money laundering. It is possible to follow some regulations to ensure the appropriate use of such platforms.

Summary

This chapter introduced blockchain security. We started by discussing a layered model of the blockchain ecosystem and its components. After this, we introduced attacks at different layers and how they can be mitigated. Moreover, we introduced the tools that can be used to analyze smart contracts and identify vulnerabilities. We also introduced STRIDE, which can be used to model threats in the blockchain ecosystem. In the next chapter, we will introduce identity, which is a major area of interest in the blockchain world. Note that blockchain security is a vast and deep subject and we cannot cover everything in a single chapter. However, this should provide a solid foundation to research further.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

20

Decentralized Identity

In this chapter, we cover decentralized digital identity. Especially with the advent of blockchain, digital identity has taken the limelight as blockchain is seen as an enabler to transform the current digital identity paradigm, which suffers from various issues.

Along the way, we will cover the following topics:

- Identity
- Digital identity
- Identity in Ethereum
- Identity in the world of Web3, DeFi, and Metaverse
- SSI-specific blockchain projects
- Challenges

So, let's dive in.

Identity

Identity refers to the distinguishing features, personality attributes, physical attributes, and expressions that define an individual, entity, or group (someone or something). It is the unique set of characteristics that make an entity recognizable and distinct.

Identity can encompass different aspects, such as personal identity, social identity, cultural identity, and digital identity. It is an important aspect of our day-to-day lives and plays a fundamental role in shaping how individuals or entities interact with others around them.

An important aspect of identity is credentials. Credentials can be defined as an attestation by a relevant authority of an attribute associated with an entity. The attribute can be a personal attribute like age, name, or gender or could also be something that you have earned, such as a qualification. Credentials issued to individuals include a passport, driving license, license to kill, diploma/degree, and many others. An organization can be issued with credentials such as authority for them to operate in a certain jurisdiction, company registrations, building permits, and many more. Traditionally, all these credentials are paper based.

For example, a driving license is a card (paper) issued to you by the concerned authority once you have passed the driving exam. You present it as proof of your ability to drive to a police officer when necessary. Such a model is called a paper-based credential model, which is founded on paper documents issued by a concerned authority. The individual who is issued the paper document is the holder, the entity who issues the paper document is the issuer, and a verifier is an entity that verifies this credential to ensure its legitimacy and allow access to some resources. For example, to open a bank account, I may need to present my driving license as proof of my address and identity. These paper credentials contain some claims, e.g. your qualifications. These credentials are also expected to prove who issued the credential, who the holder of the credential is, and most importantly, that what is claimed is correct and accurate and that the claims have not been altered in any way. In short, the claim must be verifiable. However, in the real world, the forging of documents is quite common. Because this model is based on assumed trust between holders, issuers, and verifiers, it poses some challenges.

When an identity is represented electronically, it can be defined as a digital identity. We will discuss this next.

Digital identity

Digital identity can be defined as an electronic representation of a real-life entity, such as a human, machine, device, or organization. A digital identity can be represented by a citizen identification number, employee ID, host name, and in many other forms.

In other words, a digital identity is information that identifies an individual or entity in the digital world, e.g., on the internet.

Identity management can be defined as the process of creating, updating, deleting, and storing digital identity accounts and managing access to resources through the process of authentication and authorization.

There are several models of digital identity, including centralized, federated, and decentralized identity models. We'll explore these models next.

Centralized identity model

The centralized identity model is what we are most familiar with. It is the most widely used model with identifiers and credentials issued by centralized service providers and governments, for example, Facebook logins, Google logins, passports, citizen cards, and driving licenses. We can think of two variations of this model:

- **Independent centralized identity model**: This model represents paradigms where a single user holds a credential from a single service provider. There is a direct one-to-one relationship, and the issued credentials such as login and password can only work with a single entity from which they're issued. The user has to maintain individual credentials issued by each of the entities.
- **Shared centralized model**: This model is based on a paradigm where a central service provider maintains the credentials and they are shared across different service providers.

This model is the original internet identity model, also called the "account-based identity model." Here, a user establishes their identity by providing registration details to a service provider and creating an account with the service provider, e.g. a website.

There are, however, several problems with this model:

- It's centralized, so all your data resides on the service provider's side.
- You don't have any control over your data, how it's used, where it's used, when it's used, and for what purpose – even if there is regulation around it, it's not really under your control. What if the service provider is trying its best to serve its customers but a hacker manages to steal personal data from the service provider?
- The onus lies with the user to maintain their usernames and passwords for each service provider.
- Being centralized, they are subject to hacking and data breaches. Just do an internet search of "data breaches in 2023" or "recent data breaches" to gauge the magnitude of this issue.
- Every service provider has their own security policies around the management of accounts and can impose those policies on customers, including around the choice of password, its length, its usage, etc. While the intention is good here, in practice it becomes quite cumbersome for customers to manage their accounts.
- Basic account model – a username and password.

Federated identity model

The fundamental of the federated identity model idea is to enable a user to use the same account (username and password) and other credentials to sign on with multiple service providers. Members in a federation (hence the name federated identity) rely on each other or a central provider to authenticate their respective users and vouch for their right to use services. For example, once a user is authenticated using one member, the user doesn't have to log in again to another service provider with a separate username and password. The user can log in using the same credentials that were used to log in to the identity provider on other services in the network without having to enter their username and password again. Once the user is logged in to an identity provider, the user can use all services that are provided by service providers that accept credentials from the identity provider they are using.

The model operates with a central entity called an identity provider, which sits between the organization and the user. Moreover, once a user has logged in to a service provider using their credentials, if accepted by another service provider, the same credentials can be used to authenticate to other service providers. For example, logging in to another website using Google or another third-party identity provider is an example of a federated identity model. In this case, users authenticate themselves to the identity provider (Google) using their login credentials, and the identity provider issues a security token to the user. The user can then use this security token to access other websites or applications without creating a new login or sharing their personal information with the other websites or applications.

This approach allows for a seamless user authentication process while allowing the website or application to maintain its security domain. It also reduces the need for users to remember multiple passwords and login credentials. In addition, it helps ensure that user information is not shared between applications or systems. Federated identity is commonly used on the internet and is supported by protocols such as OAuth and OpenID Connect.

Figure 20.1 below depicts the key architectural difference between these two models:

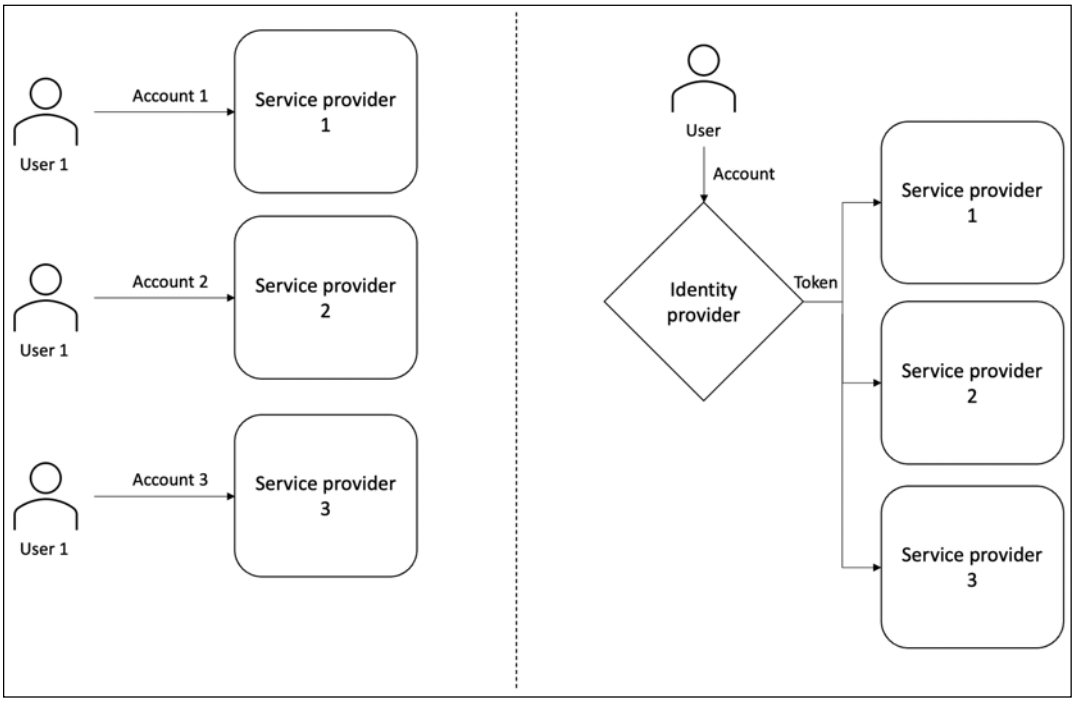


Figure 20.1: Centralized model vs federated model

The preceding diagram shows, on the left-hand side, the relationship between a user and service provider in a traditional centralized identity model where a user has an account with the service provider. On the right-hand side, the identity provider entity below the user provides the identity services in a federated identity model.

The idea is to use a central identity provider where you just have one identity account. However, now, through the IDP, you can log in to multiple service providers, e.g., websites or apps. This collection of all the service providers that use the same identity provider is called a federation, hence the name of the model. The parties in a federation rely on each other for authentication, hence they are called relying parties. Several federated identity protocols to enable **Single Sign-On** (**SSO**) have been developed and work quite well, including:

- **Security Assertion Markup Language** (**SAML**) enables the exchange of authentication and authorization data between entities, allowing for federated single sign-on and access to resources across different domains. It uses XML-based messages to securely transmit identity and authorization information between an identity provider and a service provider.
- **OAuth: Open Authorization** is an open standard protocol for delegated access authorization that enables users to grant third-party applications or services access to specific resources on their behalf without the need to share their login credentials. It provides a token-based authentication mechanism that allows resource owners to grant and revoke access to their resources anytime, providing a secure and user-friendly way to share information across different platforms and services.

The OAuth protocol enables authorization between different systems in a flexible and standardized way, making it a widely adopted framework for securing user data in the digital environment. It is commonly used by service providers like Amazon, Google, Facebook, and Twitter to enable users to authorize access to services provided by third-party services. More information on the standard is available here: `https://oauth.net/2/`.

- **OpenID:** The OpenID protocol is an open and decentralized authentication mechanism that the non-profit OpenID Foundation promotes. It allows user authentication across multiple cooperating websites, known as **Relying Parties** (**RPs**), using a third-party **Identity Provider** (**IDP**) service, eliminating the need for website owners to create their own login mechanism. This means users can log in to different websites without having to remember separate usernames and passwords for each one. To create an account, users select an OpenID identity provider, which they can use to sign on to any website that accepts OpenID authentication. It is a commonly used standard by many services on the internet. Some providers of OpenID include Ubuntu One and Microsoft.

This model is also used on the internet quite commonly, e.g. where you use your Twitter account to log in to some other website or use your Amazon account to check out other online sellers' websites.

This model can alleviate some of the problems in the centralized model, but it still suffers from some key problems, which are described below:

- No single service; there are many identity providers, leading to users having to maintain credentials from different identity providers.
- Cross-identity provider security, contractual obligation, data sharing, and management become difficult and also raise security and privacy concerns.
- Identity providers are still centralized.
- High reliability on a single identity provider, which is subject to the same hacking and data breach problems that the centralized identity model suffers from.
- Moreover, if an identity provider is compromised, it means compromising all service providers linked with the same account.

In short, the federated identity model works well but does have limitations mainly due to its centralized architecture. Generally, both of these models suffer from trust, access management, resource allocation, shared identity, privacy, and difficult-to-use problems.

To manage users' identities and their access to the system, **Identity and Access Management Systems** (**IAMS**) are used. These systems mainly perform two functions, authentication and authorization. Authentication provides assurance about the identity of an entity and authorization determines the access rights, i.e., what an authenticated user is allowed to do on the system.

There are several actors that make up an identity and access management system. These entities include users, service providers, identity providers, and attribute authorities, such as directory services, DNS, and certificate authorities. These systems are prevalent on the internet and in enterprise settings.

IAMS suffer from several challenges, including:

- Privacy and security
- Identity theft possibility
- Forged credentials
- Still using paper credentials for onboarding or, at best, scanned copies of the credential documents!
- How to update personally identifiable information after changes in personal circumstances
- Takes a long time for onboarding, requires lengthy attestation and verification of documents, etc.
- Centralization, subject to cyber-attacks, jurisdictional segmentation, anticompetitive behavior, censorship, exclusion, and inclusion

So, what changed with blockchain? Can we use blockchain to improve or build an altogether new identity system that doesn't suffer from the existing problems that we mentioned earlier? A new model inspired by blockchain has emerged that can alleviate most of these problems, called the decentralized identity model.



Note that it is not necessarily the case that blockchain technology is essential to provide decentralized identity, but the solid infrastructure, security, integrity, and decentralization benefits that blockchain provides makes it a first choice as a platform on which robust decentralized identity solutions can be built. Blockchain provides a strong foundational layer to build a decentralized identity system utilizing the properties of decentralization, integrity, and security that would have otherwise been difficult and in some cases simply impossible (e.g., decentralization) to achieve on their own by other means.

Some benefits that can be realized by using blockchain technology for IAM are listed below:

- Information about identity is auditable, traceable, and verifiable
- Openly accessible by anyone
- Censorship resistant due to blockchain security guarantees
- Thwarts fraud due to transparency
- Promotes better trust due to transparency
- Efficient
- Can lower fraud ratio
- Reduces verification costs
- Can facilitate the provision of **Self-Sovereign Identity** (**SSI**) and DIDs. It can serve as the layer on which SSI and DIDs can be provided with relative ease as compared to other approaches

However, some challenges are the privacy and protection of personally identifiable information, and whether we store personal data on a chain or not. These can be addressed in different ways. See *Chapter 18*, *Blockchain Privacy*, for more details on privacy.

Blockchain can enable the decentralized identity model, facilitating the creation of a decentralized identity, blockchain-based decentralized identity and access management systems, and self-sovereign identity.

Now let's discuss the decentralized identity model.

Decentralized identity model

This model does not rely on either a centralized or a federated identity model but is decentralized at a fundamental level. It is not account based and works with real identities in a peer-to-peer manner where a direct relationship between entities (peers) is established instead of going through a central provider. There is no "account" that exists in this world; instead, a direct relationship between peers based on credentials is established. Moreover, there is no single party controlling this mechanism; everyone is in control. As long as they want to keep a relationship with an entity, the "connection" is there. If any one party (peer) doesn't want it anymore, they just drop off the channel without leaving any personal information behind.

With this property of peer-to-peer connectivity, decentralization can be achieved because any peer can connect to any other peer directly. The original intent of the internet was to be decentralized and P2P, but over many decades it became centralized due to centralized service providers. What's changed is whether I can now create a truly decentralized identity mechanism that relies on a peer-to-peer network but is truly decentralized, and no single authority can control it. The answer to this is – you guessed it – blockchain!

Blockchain provides a peer-to-peer network and a **Decentralized Public Key Infrastructure** (**DPKI**), which provides the necessary ingredients for developing a decentralized identity model.

Current **Public Key Infrastructure** (**PKI**) systems are based on a trusted third party called a **Certificate Authority** (**CA**), which verifies the identity of users and issues digital certificates that bind public keys to specific identities. They are based on a hierarchical trust model that extends from the user's certificate to a trusted root CA. While this system works reasonably well, ensures the integrity of digital identities, and secures online communication, it is fundamentally centralized. As a result, it suffers from issues such as impersonation and *man-in-the-middle* attacks to get an undeserving valid certificate from the CA, which implicitly trusts a third party, making CAs a single centralized point of failure.

As blockchain is a decentralized and distributed key-value store. It allows for a better approach to managing digital identities securely and transparently without relying on a centralized CA. We can build a more efficient certificate issuance and revocation system using blockchain. Traditional PKI systems have a slow and less secure certificate revocation process. However, in DPKI, revocation and issuance can be done quickly with all nodes on the network updated in real time. By eliminating the dependence on a centralized CA, DPKI makes PKI systems more secure, transparent, efficient, and decentralized, giving users more control over their digital identity.

Fundamentally, it provides a secure way of exchanging public keys to establish a secure P2P connection between two parties and also provides a secure storage layer to store these public keys to allow the verification of digital signatures on digital identity credentials (i.e., **verifiable credentials** (**VCs**)) that peers can exchange to prove their real-world identity.

As interest in the decentralized identity paradigm grew, a new term emerged: SSI.

Self-sovereign identity

Self-sovereign identity (**SSI**) can be defined as an identity of a person that is fully owned and controlled by the person. It is not dependent on or subject to any other authority or trusted third party.

Note that SSI doesn't mean that anyone can assert anything about their identity and get away with it. Instead, there is a secure and stable issuance model behind these identities and credentials that, with the help of various other actors, components, and technologies (especially cryptography), ensures the integrity of the system. Moreover, SSI is not just for individuals; it applies to any entity, machine, IoT device, and virtually anything that needs an identity in the digital world.

The control in the decentralized SSI model remains with the user, whereas in the traditional centralized and federated models, the control is in the hands of service providers, issuers, and verifiers.

We can visualize the difference between the centralized and **decentralized SSI** models in the following image:

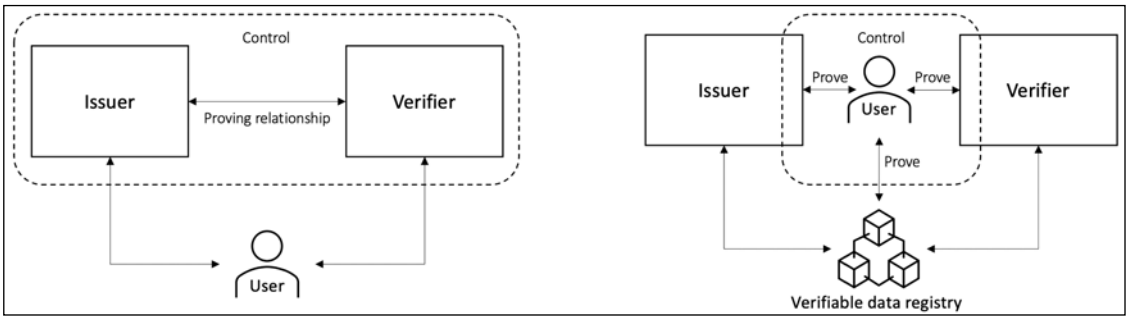


Figure 20.2: Centralized model vs SSI

The preceding diagram shows how the decentralized SSI model, shown on the right-hand side, gives control back to the user, instead of the centralized model shown on the left-hand side, where the user is sitting outside the control mechanism controlled by issuers and verifiers. Note that in the SSI model, the user is in the center of the ecosystem and in control.

This fundamental shift in the control from centralized entities to the user makes the SSI paradigm so impactful and profoundly elegant, which results in enabling tremendously powerful use cases. It can help improve business processes, establish trust, improve the customer experience, address regulatory requirements in innovative ways, resist surveillance and data piracy, and improve government, finance, health, and virtually all other industries.

There are two key constructs that enable this model: VCs and **decentralized identifiers** (**DIDs**). Let's now have a look at the composition of SSI.

Components of SSI

In this section, we'll explore the components of SSI, including VCs, which are tamper-evident and cryptographically VCs, and **verifiable presentations** (**VPs**), which are data formats used for sharing one or more VCs.

Verifiable credentials

We saw what a paper credential is earlier. VCs are the digital counterparts of paper credentials. Even though we expect paper credentials to be verifiable too, due to the issues covered earlier, the current paper credential paradigm is not secure and foolproof.

Formally, we can define *credentials* as a set of information about an entity (subject) that an authority (issuer) claims to be true, and using these claims, the subject can convince other parties (verifiers) that these claims are true. Here, a trust relationship between the verifier and issuer is implied. Moreover, the subject trusts that the issuer has issued convincing claims that are verifiable. Credentials can belong to human subjects, machines, organizations, and any other entity that needs to convince others about the truth of their claims. The claims can be divided into three broad categories:

- Personal, such as age and ethnicity
- Relationships and associations, such as country of residence, spouse, member of some institute, etc.
- Entitlements, such as legal rights, pension, tax relief, state benefits, etc.

These credentials must be verifiable, i.e., they must be able to convince the verifier that the claims being made are true. To achieve this, the verifier must be able to establish:

- Who the issuer of the credential is, e.g., a university or a government.
- That the credential has not been tampered with.
- That the credential is valid, i.e., has not expired or been revoked. There might be some other validity conditions depending on the use case, e.g., a credential that allows someone to have tax relief is not transferrable. So, one validity condition could be that it has not been transferred or perhaps reused if it is just a one-time offer.

Traditionally, with paper credentials, these guarantees are provided by security features such as guilloche patterns, holograms, or some other feature that cannot be modified or copied. In the digital world, however, we use cryptography to achieve these guarantees. For example, the issuer, validity, and non-tampering (authenticity) can all be verified by using digital signatures.

As we said earlier, VCs are the digital equivalent of physical paper credentials. In essence, VCs are digital versions of the paper credentials that users can carry in their mobile devices just like they were carried in physical wallets before. The difference is that they are digitized (electronically verifiable, cryptographically secure) and have several benefits over their physical counterparts. A comparison between physical paper credentials and digital VCs is shown in the table below, which highlights the benefits that **VCs** have over physical credentials.

Attribute/type	Physical paper credentials	Digital verifiable credentials
Cloning	Possible, even though in some cases it is extremely hard, e.g., national ID cards and passports. This refers to paper passports, not the new digital machine-readable ones; however, there is the possibility of cloning and alteration with them too.	Not possible.
Hacking	Not too difficult, relatively.	Extremely hard, unless the device on which the VCs are stored is fully accessible to the attacker, which is quite difficult with current security schemes such as 2FA, biometrics, etc.
Privacy preservation	No or little privacy. A border control officer sees everything on your passport.	More privacy is possible as selective disclosure is possible using VCs.
Access and permissions	Full access to anyone; a verifier can see all claims and attributes related to an entity.	Supports the principle of least authority/selective permissions (access control), which allows for more security.
Cost	High, especially when producing travel documents and other sensitive documents due to the high cost of special security features that go into every document.	Very low cost, only initial setup/ infrastructure cost; once it has been established you can virtually issue and verify as many credentials as required.
Physical limitations	As physical objects, they can be difficult to carry, prone to stealing and loss, and protection from physical wear and tear is difficult.	Virtually impossible to steal; they are usually stored in mobile devices. Even if the mobile device is lost, replacement VCs can be issued relatively quickly. They are not subject to any physical wear and tear. Easy to carry and protect.
Delegation	Not possible.	Possible, if permitted as per the issuer's policy.

Architecture

The VCs ecosystem consists of various actors and components, which are described below:

- **Issuer:** The entity that issues VCs to the subjects.
- **Subject:** An entity whose claims (attributes) are stored in the VC.

- **Holder**: An entity that is in possession of the VC and presents it to the verifier for verification as and when required.
- **Verifier**: An entity that ensures that the claims made in the VC are true and correct. The verifier receives VCs from the holder and verifies them.
- **Digital wallet**: An entity that stores the VCs for the holder.
- **Digital agent**: Software that acts as an interface between the VC ecosystem and the holder, e.g., an app on a mobile phone.
- **Verifiable data registry**: This is an entity that is the foundation of the VC ecosystem and decentralized identity ecosystem. It exists on the internet and is accessible to nearly all the actors in the VC ecosystem. It stores the data necessary to successfully operate a VC ecosystem, such as issuer public keys, the schema of all VC properties (attributes), a revocation list, an expiry list, and metadata describing VCs and other metadata. **Verifiable Data Registries** (**VDRs**) can be blockchains or decentralized and centralized databases. However, blockchains, due to their inherent security features, can be more suitable in the decentralized identity ecosystem.

The following image shows the high-level architecture of the VCs ecosystem:

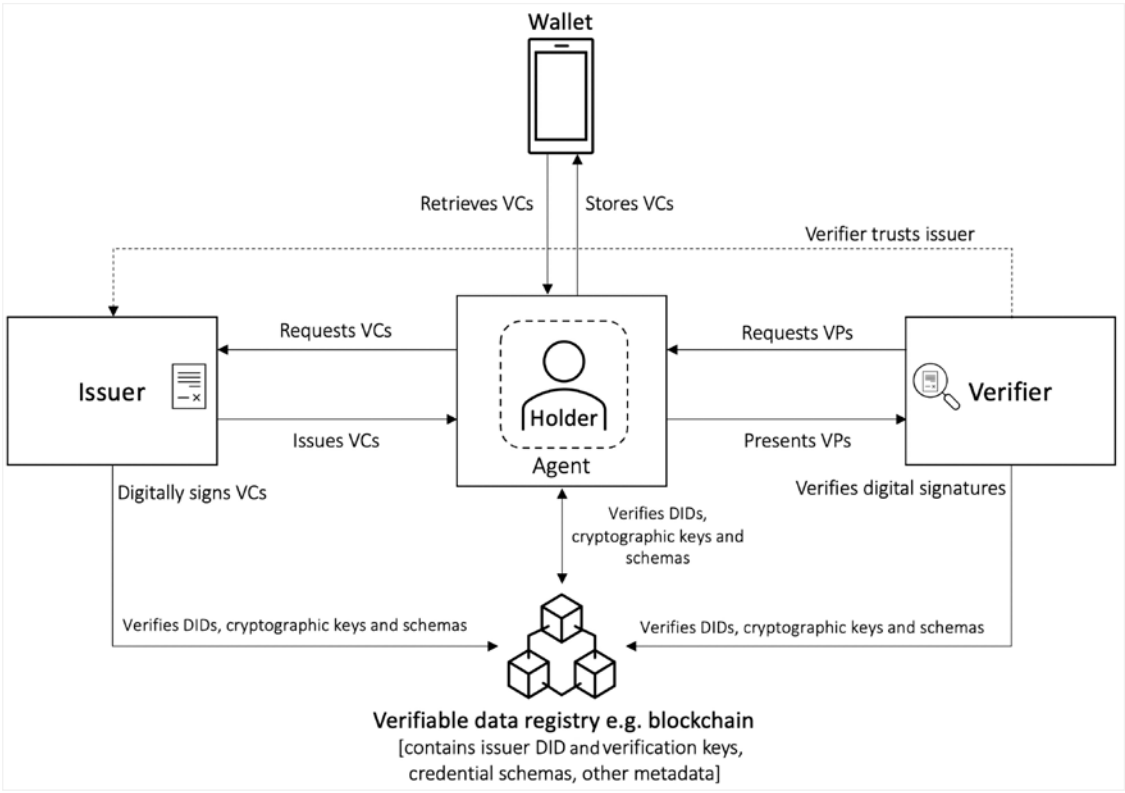


Figure 20.3: VCs ecosystem

Note that in the diagram, the holder of the VC is in charge, while all other actors around it facilitate the credential issuance and verification process. In the VC ecosystem, there is a need for the verifier to trust issuers as authorities who are authorized to issue VCs; however, the verifiable data registry is trusted by all actors. Verifiers are allowed to establish their own trust relationship and verification rules for the verification of VCs. For example, the VC issued by a national health authority for a Covid vaccination is trustworthy and nearly all verifiers accept it without hesitation. However, one issued by a private entity (even though it is cryptographically sound and correct) may not bear the same level of authenticity and trustworthiness as the one issued by a national health authority. So, verifiers have the choice as to which VCs they trust and are willing to accept. The subject also trusts the wallet in which the VCs are stored to store them securely. Moreover, the verifier and subject of the VC trust the issuer to issue legitimate and correct credentials and revoke them if they expire, are no longer valid, or are compromised. Trust involves some type of public and private key/certificate utilization and management, and blockchain delivers this mechanism.

Remember that the subject and holder are usually the same entity, but sometimes they can be different, e.g., an access pass to a nursery issued to a child but managed by parents or a vaccine pass for a cat issued to the pet owner. However, it can really be the cat who owns the VC in its chip (tag).

Now we'll describe the structure of a VC.

Structure of a VC

A VC is usually composed of eight elements:

- **Context**: Contains one or more **Uniform Resource Identifiers** (**URIs**), which provide the information needed to properly interpret and verify a credential, including the standards, data models, and vocabularies used to construct the VC.



A URI is a string of characters identifying a name or resource on the internet or any other network. It provides a standard way to identify resources such as web pages, images, videos, and other files. A URI locates a resource on a network and specifies the protocol to retrieve it, such as HTTP, FTP, or file.

- **Type**: Contains URIs that assert the type of the VC. The verifier reads the type and uses this information to decide whether or not they can interpret and handle the credential correctly. If the verifier encounters a type that is unfamiliar or unrecognized, they simply reject the credential.
- **ID**: This is the unique ID created by the issuer for this VC and contains a single URI.
- **Issuer**: This describes the entity that issued the VC. It contains a URI that points to a document that describes the issuer.
- **Credential subject**: This property conveys information about the subject of the credential. It includes a pseudonymous identifier of the subject in the form of a URI, along with a set of properties (claims) that the issuer is asserting about the subject. The pseudonym preserves the privacy of the subject.
- **Proof**: Contains proof – usually a digital signature to verify the VC.
- **Issuance date**: The date of issuance of the VC, in RFC3339 format.

- **Status:** Allows discovery via a URI of information about the current status of the VC, such as whether it is suspended, revoked, etc.

The structure of the VC is shown in *Figure 20.4* below:

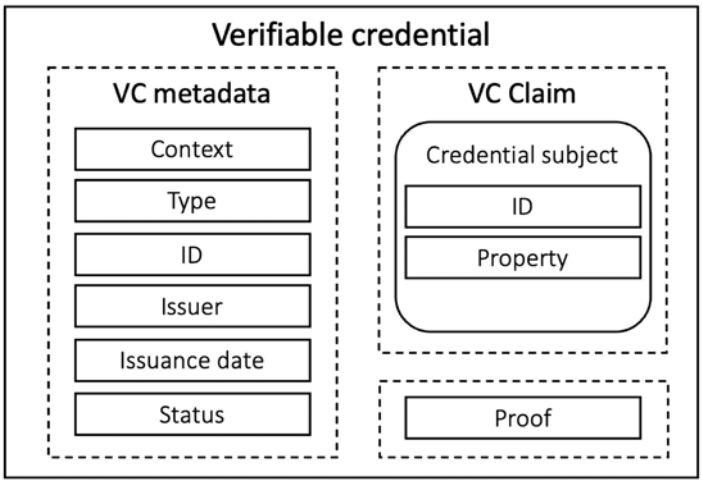


Figure 20.4: Structure of a VC

Figure 20.4 shows the structure of a VC, including metadata, claim, and proof components.

Other properties may include issuance date, expiry date, credential status, and other fields depending on the use case. Moreover, a VC depending on the use case can contain multiple claims and multiple proofs, and can be optionally presented in what's called a VP.

An example VC is shown below, which shows some of the properties discussed above. It's encoded in JSON-LD:

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1",
    "https://www.w3.org/2018/credentials/examples/v1"
  ],
  "id": "https://example.com/credentials/123",
  "type": ["VerifiableCredential", "DegreeCredential"],
  "issuer": {
    "id": "https://example.com/issuers/14",
    "name": "Example University"
  },
  "issuanceDate": "2023-02-16T22:12:03Z",
  "credentialSubject": {
    "id": "did:example:123",
```

```
    "degree": {
      "type": "BachelorDegree",
      "name": "Bachelor of Science in Computer Science",
      "college": "College of Engineering and Applied Science",
      "university": "Example University",
      "degreeDate": "2022-05-15",
      "degreeStatus": "awarded"
    }
  },
  "proof": {
    "type": "Ed25519Signature2018",
    "created": "2023-02-16T22:12:03Z",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "https://example.com/issuers/14#key-1",
    "jws": "eyJhbGciOiJFZERTQSIsImI2NCI6ZmFsc2UsImNyaXQiOlsiYjY0Il19..0pw88foZz
KHJmP4x4m0F5m5ZOyP5WAGGbsS0L9CKiKjxI5_ZfE5i5y5q5vqBQ9XivnC5pwcbZi75jmZ1xou4Dw"
  }
}
```


The example above shows a VC for a person's degree from `"Example University"` with all the necessary details and cryptographic proof. Let's see what each element means:

- `"@context"`: Specifies the JSON-LD context for the document
- `"id"`: A unique identifier for the credential
- `"type"`: The type of the credential, which in this case is `VerifiableCredential` and `DegreeCredential`
- `"issuer"`: The entity that issued the credential, which includes an identifier and name, i.e., `Example University`
- `"issuanceDate"`: The date and time when the credential was issued by the example university
- `"credentialSubject"`: The subject of the credential, which in this case is a person with a Bachelor of Science in Computer Science degree, including the degree details such as the type, name, college, university, degree date, and degree status
- `"proof"`: Cryptographic proof of the credential, including the type of signature, time of creation, purpose of the proof, verification method, and **JSON Web Signature** (**JWS**) of the credential



JWS is an IETF proposed standard (described in RFC 7515) for signing arbitrary data.

It is possible to combine multiple VCs for which VPs are used.

Verifiable presentation

A VP is used to present a single or multiple VCs to a verifier for verification, as shown in *Figure 20.5* below:

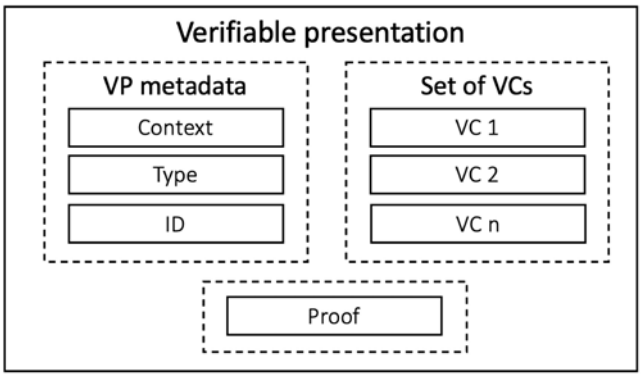


Figure 20.5: Structure of a VP

The preceding diagram shows the structure of a VP, containing metadata about the VP, proof signed by the subject (holder), and a set of VCs.

A VP also allows a holder of a VC to only disclose a subset of information from the VC in order to preserve privacy. A VP can be created that only contains the minimum necessary information required for verification and no more than the holder is willing to share. For example, only reveal that the holder is above 18 years of age and nothing else, so the holder will only send that information as a VP to the verifier for verification. This is called selective disclosure or minimum disclosure.

Decentralized identifiers

A DID is a self-sovereign identity that is permanent, portable, and verifiable and does not depend on any centralized authority. In practical terms, we can think of a DID as a new type of globally unique identifier or address that is cryptographically augmented to support secure verification and decentralization.

Formally, we can define a DID as a globally unique, permanent, usually cryptographically generated and cryptographically registered identifier that doesn't rely on any centralized authority.

A DID is a new type of URI. *Figure 20.6* below shows what a DID looks like:

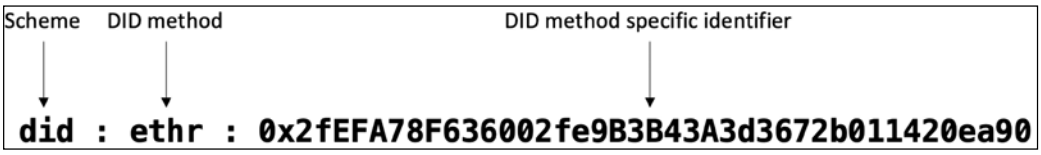


Figure 20.6: DID

The preceding diagram shows the format of a DID, which is composed of three elements separated by a colon. It's a simple structure, consisting of the scheme name, the DID method, and a DID method-specific string (also called a method-specific identifier). Nevertheless, it carries far-reaching implications in terms of enabling a self-sovereign identity ecosystem:

- **Scheme**: This is a fixed string, `did`, indicating that this is a DID.
- **Method**: This is the method that has been used to generate the DID, e.g., `ethr`, `btcr`, `sov`, `web`, etc. It basically describes where the DID is located and which protocol it is on. This element helps the DID resolve to a corresponding DID document. It can also be defined as a definition of how a specific syntax of an identifier is implemented. The DID method specification specifies the operation using which DIDs and DID documents are created, resolved, updated, and deactivated. A DID document can be defined as a set of elements that describe the DID subject and cryptographic material that the DID subject can use to authenticate and prove its association with the DID. Each DID method specification defines a scheme that works with a specific DID method.
- **DID method-specific string**: Also called a method-specific identifier. The syntax of the method-specific identifier is defined by the DID method. It is usually a long string generated using cryptographic methods and must be unique. For example, the btcr method-based DID is built on the Bitcoin blockchain. The method-specific identifier (string) is generated from the position of the Bitcoin transaction in the blockchain. Another method, web, is simply a DNS name secured with a TLS/SSL certificate with a path to the DID document. Take the example `did:web:masteringblockchain.com`; here, the resolution to the DID document is simply through the domain name I already own.
    - `did` is the schema, `web` is the World Wide Web, and `masteringblockchain.com` is the method-specific identifier, i.e., the fully qualified domain name.
    - Another example could be `did:ethr:0x2fEFA78F636002fe9B3B43A3d3672b011420ea90`, where `did` is the schema, `ethr` is the blockchain, i.e., VDR, and the hexadecimal string is the smart contract address.

There are many already-established DID methods and the specifications are available here: `https://www.w3.org/TR/did-spec-registries/#did-methods`.

Several requirements of DIDs are:

- **Decentralized**: It does not rely on any centralized entity, identity provider, or authority.
- **Cryptographically verifiable**: The ability to cryptographically prove identity and ownership, which is associated with a private/public key pair.
- **Persistent**: It is permanently bound to a subject and doesn't change.
- **Resolvable**: It is possible to discover more information about the subject (for metadata discovery).
- **Ease to create**: Creating DIDs should be quick, easy, and very low cost.

DIDs are the counterparts of VCs. They are cryptographically generated and verifiable. The format of a DID is "scheme: method, did specific string," as shown in *Figure 20.6*.

The DID has several relationships with various entities in the ecosystem. *Figure 20.8* below shows the high-level DID architecture and relationships between the DID, DID document, and DID subject or controller and VDR:

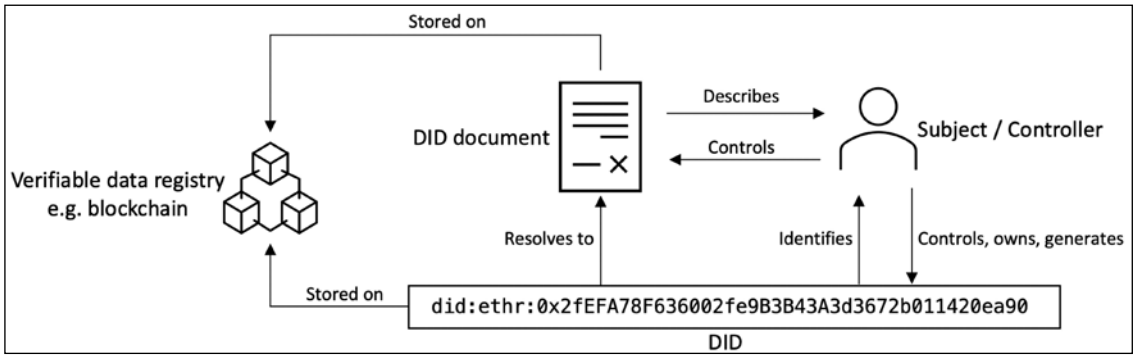


Figure 20.7: DID architecture and relationships

As shown in the diagram, a DID is a unique string following a specific format, described earlier. This DID is resolvable to a DID document. This DID document can exist on a blockchain or some other storage, i.e., a global decentralized key-value store. It can also exist on IPFS, STORJ, or even a web server. The DID document contains controller identification information about the subject, cryptographic material for verification, and some other metadata. The DID and the DID document make cryptographic verifiability possible by a verifier. How it works is that the DID and the DID document are tightly linked together cryptographically. The DID document can be serialized for storage and transmission using JSON, JSON-LD, CBOR, or any other serialization mechanism fit for the use case. DIDs primarily work in conjunction with VCs to form a cryptographically verifiable decentralized identity mechanism.

There are many types of DIDs, and while they all support a basic set of functions, each can be generated using a different mechanism and set of rules, i.e., they differ in how the functionality is implemented. These different implementation mechanisms are called DID methods. There are many DID methods.

Several operations that can be performed on DIDs can be defined as CRUD:

C – how to generate a DID and the associated document – Create

R – the DID document retrieval mechanism – Read

U – the DID document update mechanism – Update

D – the DID deactivation mechanism – Deactivate

For example, on a blockchain, this may translate to executing a smart contract that manages these operations.

A DID method defines how to read or write a DID and a DID document on a specific verifiable data registry, e.g., a blockchain. A DID always identifies a DID subject and resolves to a DID document. A DID document contains information about a specific DID. It contains information such as public keys, authentication methods used for authentication, service end points used for interaction, time stamps, which can be used for audit logs, and digital signatures, which are used to provide data integrity.

So, let's now think about what the relationship between a DID and a VC is. Due to public and private key relationships (i.e., digital signatures), a verifier can verify that the credential did come from an authentic issuer. How does a verifier know that the public key provided is the right one? In a decentralized world, it seems impossible to solve this issue. In traditional PKI, we have high-level CAs that everyone trusts; they sign on behalf of entities who trust them, so in a way a trusted third party is certifying that the public key used by a company is indeed correct, and due to the trust relationship with the root-level CA, everyone is satisfied that the public key is authentic. This is clearly a centralized model.

How does this work in an SSI ecosystem? How is the verifier assured, in a decentralized system, that the public key from the issuer is correct and is indeed from the authentic issuer? This is where DIDs come in. DIDs are identifiers that get tied to the public key, which verifiers can resolve ("lookup") and hence be confident that the public key is indeed authentic.

Now, how can we prove that the DID is bound to the public key and the DID controller (holder, subject) that is in control of the corresponding private key?



Essentially, the controller is an entity that can make changes to the DID document. Also note that the controller is not necessarily the subject of the identification.

The solution might be to mathematically generate the DID from the public key instead of arbitrarily creating DIDs. Now, if a mathematical relationship exists between the DID and the public key, the controller can prove that the DID indeed belongs to it, because it has the private key and can respond to any challenges posed by the verifier, e.g., in a challenge/response mechanism. Seems reasonable so far, but what if the controller has to rotate the keys? The DID will have to change, but that's not in line with the goals of DIDs that we saw earlier; DIDs must be persistent. So, this is where DID documents help. The DID document contains the public key, and as a DID is always resolvable to a DID document, there is room to update the DID document and hence the public key and other metadata as required. So, the DID can remain persistent, but the DID document can change, which allows for key rotation and other updates.

The mechanism works as described below:

1. The controller (private key holder) generates the DID based on the genesis public/private key pair.
2. The controller publishes the DID document, which contains the DID and the public key.
3. Now any entity using the DID document can cryptographically verify that the DID and the associated public key are linked (bound) together.
4. If the controller, for some reason, changes (rotates) the key pair, the controller creates an updated DID document and signs it with the previous private key. This can be an update on the blockchain to register the updated DID document. In essence, the controller publishes an updated DID document that contains the original DID and the new public key, but signs with the original private key, which creates a chain of trust between the DID document, which is trackable back through to any number of updates to the original DID document, and the original DID. Each DID document serves as a new digital certificate for the new public key without the need for a certificate authority or any other trusted third party.

So, in summary, we can say that a DID document is a data structure that is associated with a DID and contains information about the entity it represents. It provides a way to publish and retrieve information about a DID, such as its public key, authentication mechanisms, and other relevant metadata. The DID document is typically hosted at a URL, known as the DID endpoint, which can be resolved by a DID resolver. The DID resolver is responsible for looking up the DID document associated with a given DID and returning the associated data to the requester. A generic DID document example is shown below.

```
{
  "@context": [
    "https://www.w3.org/ns/did/v1",
    "https://w3id.org/ethr/credential/v1"
  ],
  "id": "did:ethr:0x123456789abcdefghijklmnopqrstuvwxyz",
  "authentication": [
    {
      "id": "did:ethr:0x123456789abcdefghijklmnopqrstuvwxyz#controller",
      "type": "Secp256k1SignatureAuthentication2018",
      "publicKey": "0x123456789abcdefghijklmnopqrstuvwxyz#controller",
      "controller": "did:ethr:0x123456789abcdefghijklmnopqrstuvwxyz"
    }
  ]
}
```

Some key elements in this document are:

- `context`: Denotes the type of the document, e.g., a JSON-LD document
- `id`: The DID
- `authentication`: Describes cryptographic schemes for verification

There are many others; for a detailed reference, refer to the `https://www.w3.org/TR/did-core/` document by W3C.

A DID is a unique identifier that is used to represent an individual or entity on a decentralized network. It is a persistent identifier that is independent of any centralized authority or registry and can be used to authenticate and authorize the holder of the identifier to access resources or participate in transactions. A VC is a digital document that contains verifiable claims about an individual or entity, such as their identity, credentials, or attributes. VCs are designed to be portable so that they can be shared with others and are used to establish trust between parties. They are typically issued by trusted entities, such as universities, governments, or financial institutions, and can be verified by reliable parties using cryptographic proofs. The combination of an identifier (DID) and VC (age, citizenship, or qualification) is what makes a digital identity. These VCs live in your digital wallet on your mobile phone and you can present them at will to any party who needs verification. Again, it's in your full control whether you decide to present it or not. The relationship between DIDs and VCs is that DIDs are used as the subject identifier in VCs.

In other words, a VC can contain a verifiable claim about an individual or entity, and that claim can be associated with a DID that uniquely identifies the subject of the claim. By using DIDs in this way, VCs can be tied to specific entities in a decentralized manner, without relying on centralized authorities or registries.

Digital wallets

A digital wallet stores VCs securely, protecting them from theft and corruption and making them available when required. It is like a cryptocurrency wallet, which we discussed in *Chapter 6*, *Bitcoin Architecture*, but there are some differences. By definition, however, it is the same as a cryptocurrency wallet. There is another type of wallet that is common, the one available in iOS and Android to store payment cards and other credentials.

A digital wallet is a piece of software or hardware that allows its owner to create, store, organize, and secure cryptographic keys, secrets, and other types of sensitive, private information. Sensitive information can include VCs, decentralized identifiers, personal data, accounts, and any digital object that comes under the definition of sensitive or private information.

An SSI wallet is a digital wallet that is primarily used in the decentralized identity ecosystem (self-sovereign identity ecosystem) to store and manage VCs. An SSI wallet may store DIDs, VCs, cards, personal data, access credentials, digital identity documents, travel documents, and many others. It is based on the design principles of portability, privacy first, and security first.

A high-level architecture of an SSI wallet is shown in *Figure 20.8* below:

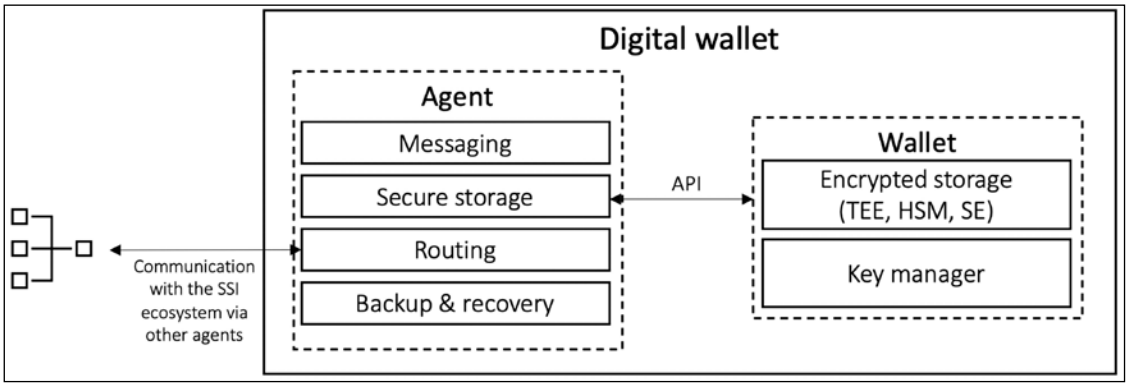


Figure 20.8: SSI wallet high-level architecture

Figure 20.8 shows a component called **Agent**, i.e., a digital agent. This agent mediates communications between wallets, users, and other agents in the SSI ecosystem. The wallet itself is the core protected element where, for example, keys and other sensitive data are stored. One key advantage of this design is the separation of concerns. Even though sometimes there is no distinction made between these two elements, and a wallet is just called a wallet, without any distinction made between the agent and the wallet, internally, the interface component (agent) and wallet are usually two different components.

The agent consists of secure storage, a backup and recovery mechanism, a messaging interface to handle communications, and a routing functionality. Secure storage communicates with the wallet services via a secure API. The wallet consists of two subcomponents, including encrypted storage (trusted execution environments, secure elements, and secure enclaves) and a key management system.

Identity wallets can create DIDs. Remember, in the SSI model, you are creating your own identities and are not dependent on some third party to create one for you. These wallets can facilitate storing these DIDs on a blockchain through blockchain transactions. They can store, sign, and present a VC to a verifier.

Verifiable data registries

Fundamentally, VDRs are databases that serve as a canonical source of truth and trust for DIDs and public keys. VDRs are global, distributed key-value databases. We can use centralized databases for achieving SSI but that might only be acceptable within a small group or consortium. For a global-scale SSI, blockchains are expected to become the VDR of choice. The VDRs can be a public blockchain, a consortium chain, or a private permissioned chain. They can also be application-specific blockchains, purposefully built for SSI, for example, Evernym, Sovrin public ledger, Hyperledger Indy, Hyperledger Ursa, Hyperledger Aries, and Veres One. Such ASBCs for SSI support transactions and record types that make managing DIDs easy.

Governance frameworks

Fundamentally, the trust is established in SSI through cryptography, i.e., cryptographic trust. However, this on its own is not necessarily enough because the element of human trust is also required to establish another layer of trusted relationships between issuers, verifiers, and subjects. This is important because VCs issued by some unknown agency may not be considered valuable unless there is a human level of trust and acknowledgment of their authority is established in the real world. This trust can allow verifiers to determine the legitimacy of issuers under a governance framework that the verifier trusts. Governance frameworks also specify the policies, standards, and procedures from a business, legal, and technical perspective that issuers must follow to issue, or subjects must adhere to to obtain, the VC.

With all the components discussed so far in the SSI ecosystem, we can think of the SSI stack as a four-layer model, which is shown in *Figure 20.9*:

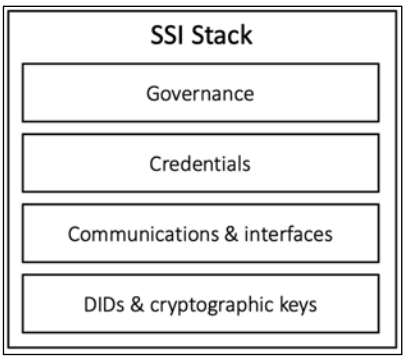


Figure 20.9: Four-layer SSI stack

In the preceding diagram, the governance layer is where a governance authority or a professional body publishes a governance framework that is expected to be adhered to by all actors in the ecosystem. The credentials layer is where the VCs ecosystem exists with issuers, verifiers, and holders. The communication and interfaces layer includes components such as digital agents, wallets, and other communication interfaces. Finally, the identities and keys layer includes DIDs, DID registries, and VDRs. The bottom two layers are concerned with achieving trust at a technical (cryptographic) level, whereas the top two layers achieve trust at a human level.

Identity in Ethereum

Accounts on the Ethereum blockchain can be seen as DIDs. This is because any number of accounts can be generated by anyone without requiring any permission and storing them on a centralized server. While they are not a DID and don't have any VCs associated with it, due to their permissionless nature they can be considered DIDs. As an analogy to DIDs and the SSI ecosystem we discussed earlier, an Ethereum account has a private key and a public key. The public key can be seen as the identity of the controller, whereas the private key is used to sign messages. The Ethereum blockchain can serve as a VDR that can store DIDs. Any VCs issued to the DID can be verified on a chain by validating the issuer's DID that is stored on the Ethereum blockchain. It can be a smart contract where all DIDs are stored.

DIDs are issued, held, and controlled by individuals. An Ethereum account is an example of a DIDs. You can create as many accounts as you want without permission from anyone and without the need to store them in a central registry.

DIDs are stored on distributed ledgers (blockchains) or peer-to-peer networks. This makes DIDs globally unique, resolvable with high availability, and cryptographically verifiable. A DID can be associated with different entities, including people, organizations, or government institutions.

Identity in the world of Web3, DeFi, and Metaverse

Recall that in the times of web 1.0, the simple account-based model of a username and password pair was prevalent. This is still the case with a huge number of websites. However, with web 2.0, a new paradigm started to appear where one identity could be used to authenticate to other service providers too, instead of only the one on which it was originally created. We see this in use on many online services, e.g., "Sign in with Apple" or "Sign in with Google" buttons. Web 2.0 operates in a top-down trust model where centralized platforms are in charge. In the Web3 world, the user is empowered, and we can simply connect to a service provider based on our decentralized identity and the credentials that we possess. No more centralized account models and relying on trusted identity providers; all you need is a wallet and appropriate credentials to connect to a service provider. We saw a glimpse of this when we used MetaMask to connect to Ethereum; however, there are a lot richer applications with this ability.

We can visualize this evolution in a simple diagram; see *Figure 20.10* below.

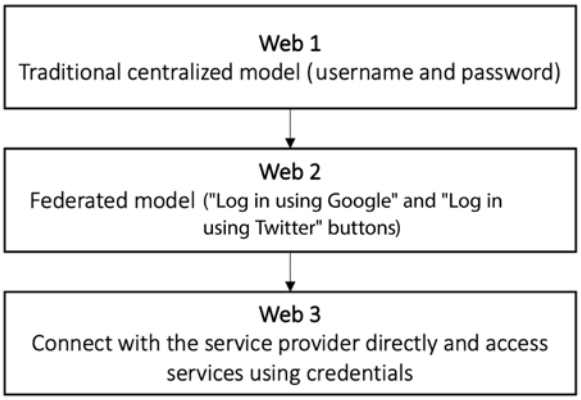


Figure 20.10: Identity paradigms evolution

Note that we covered this evolution earlier in the chapter, from a different angle; we called them centralized, federated, and decentralized models. However, in the Web3 world, it's helpful to explain these concepts from a web-oriented perspective for better understanding.

A Web3 identity can have applications in many various sectors, including but not limited to finance, gaming, health, and government. Let's discuss some relevant use cases now:

- Imagine a use case on a public blockchain where a financial institution offers a trading facility. It is critical that the identity of the traders is appropriately scrutinized. For this purpose, each trader can be issued a VC from the financial institution that is stored in their digital wallets. At the time of executing the trade, the trading mechanism first checks whether the identity and VC presented by the trader allows them to trade or not. If yes, then the trade can be allowed and executed. This enables use cases that were not possible before. Imagine a fully public blockchain that is running a trading system but is given permission in the sense that only holders of appropriate VCs can use the system. SSI enables such use cases.
- Imagine another use case, where a customer has properly gone through the **Know Your Customer** (**KYC**) process and has been onboarded by a financial institution after all the checks have passed. Now, this person can be issued with a VC, which the customer receives and is stored in their digital wallet. Now, this VC empowers this customer to go to any other institution, present the VC, and use the financial services provided by that institution. The customer is now accepted by other institutions because of the VC they hold, which tells others that they are already correctly verified, so they don't have to go through the cumbersome and traditionally repetitive process of KYC again. This can immediately and significantly reduce costs and onboarding times, improve user experience, and increase revenue for all because now the customer can easily and conveniently access more services quicker than ever before.

- Another use case could be where a credit scoring company (issuer) checks the credit score of an individual. If the credit score is satisfactory, it issues a VC to the individual (holder), who presents it to a bank (verifier) to get the loan. This VC can then be used with any financial institution without going through credit score checks every time the holder applies for a loan. There will be an expiry associated with the VC, though, to ensure that a new VC is issued based on the new credit score accordingly.

The concepts of full decentralization and data privacy are the foundational stones of Web3 and SSI plays a critical role in making Web3 a reality. The utilization of SSI in DeFi, Metaverse, gaming, and NFTs is going to shape the future of Web3.

Imagine a governance token stored in your digital cryptocurrency wallet. It could allow you to do something on the chain, e.g., vote on a protocol improvement decision. This very ability to use a token to give you the right to do something on a blockchain is the foundation on which many novel Web3 use cases can be built.



Remember the key point here is that if you log in with a username and password, you are in the Web 1.0/Web 2.0 world, whereas if you use your wallet to access a service, you are in the world of Web3. Note that I haven't used the words "log in" and instead used "access a service." This means that you don't really have to log in to the system; your VC enables you to do something by just presenting it to a verifier. Do note that the words log in can still be used synonymously, but remember Web3 works differently.

Imagine that a non-transferrable token is actually a VC. Think of how this could empower anyone, on a global scale, to own a token that works as an access right to do some operation on a chain or even do something in real life. For example, an **Non-Fungible Token** (**NFT**), which is actually a VC, that I own might be my credential to buy groceries at a discount at a certain department store.

SSI is also being considered in many government initiatives for issuing citizen IDs. Such initiatives include Canadian, Australian, and European commissions.

In a metaverse or an online game, a VC could be a token of accomplishments made in an online gaming competition.

Currently, in the DeFi space, there is no way to verify the true identity of payees receiving payments except their wallet addresses. While this allows anonymity, in some cases, it is legally required to identify the payees. Under SSI, utilizing VCs can solve this problem by issuing users VCs that allow them to make and receive payments. Also, there is no need to reveal all attributes; the VC can contain only limited information that is necessary to ensure that the payments are not being made for illegitimate purposes like financing terrorism and money laundering. This can just be an email address or a citizen ID number that only makes sense to a regulatory authority or issuer but not everyone else on the network. We'll discuss liquidity pools in *Chapter 21*, *Decentralized Finance.* Imagine how **Decentralized Finance** (**DeFi**) pools can benefit from VC operating under an SSI model. A DeFi user can be issued with a VCs that exists in a wallet owned by the user. This VC is used when the wallet connects to the service provider, i.e., blockchain in this case where the DeFi protocol is running.

This VC allows the user to do several operations on the DeFi protocol. For example, they can trade, provide liquidity, or act as a liquidator. All of this can be achieved by using VCs issued to them by the DeFi protocol. The user will not have to log in using account names and passwords; all they need is a VC with defined rights as to what they can do on the DeFi protocol, and they simply connect their wallet to the service provider. The service provider (the DeFi protocol) in this case reads the VC and allows users to perform activities in the protocol according to the role defined in the VC.

A key advantage of decentralized identity is that it can replace account-based login mechanisms. There is no need to remember and maintain usernames and passwords; the users keep VCs issued by service providers in their cryptocurrency/identity wallet and can access the services provided by service providers by simply presenting the VC to them. The service providers can fetch the associated VC associated with the blockchain address (identifier) from the blockchain.

Remember that Web3 is certain, and the journey toward achieving it has already begun, with many important milestones already achieved. The infrastructure, standards, ideas and use cases in the form of blockchains, SSI, and various standardization specifications, e.g., W3C specifications, already exist (albeit improvements are required); but it's time to build Web3 DAPPs so that the transition from web 2.0 to Web3 becomes a reality. It's going to be a lengthy process, but as the foundation is already laid, some applications are already built and in production; it's time to accelerate and do more.

SSI-specific blockchain projects

In this section, we describe ASBCs that are developed specifically for facilitating an SSI ecosystem.

Hyperledger Indy, Aries, Ursa, and AnonCreds

The Hyperledger identity stack consists of Indy, Aries, Ursa, and AnonCreds, which are flexible and can interoperate with other layers in the SSI stack:

- **Hyperledger Indy** was the Hyperledger foundation's first blockchain framework that targeted identity use cases. It is a public permissioned blockchain specifically built for decentralized identity use cases. It can build and publish DIDs, VCs, and other similar elements of the blockchain.
- **Hyperledger Ursa** grew out of Hyperledger Indy as a separate project, which consists of cryptography components of Hyperledger Indy. Now it's a separate project, used by not only Indy but also other projects that need cryptography components. Ursa contains the cryptographic components to be used by Indy, Aries, and any other project that needs a vetted cryptographic layer.
- **AnonCreds** is another project that stems out of Indy. It is the ZKP-based VCs mechanism that has been extracted from Indy as a standalone project. It works with not only Indy but also other VDRs, and can be seen as a VDR-agnostic layer.
- **Aries** is another project that spun out of Indy. It can be seen as the digital agent mechanism in the Hyperledger identity stack. Aries is an attempt to bridge DIDs and VCs from multiple SSI ecosystems. Aries enables DID-oriented messaging between agents and uses some protocols to enable the issuance, presentation, and verification of VCs.

Other projects

- **KILT**: KILT is a decentralized blockchain protocol for issuing VCs and DIDs for Web3.
- **Proof of Humanity**: This project aims to thwart sybil attacks by utilizing social verification with video submission to allow users to prove that they are real humans.
- **Sovrin**: This is a public service utility that enables SSI on the internet. More details here: `https://sovrin.org`.
- **uPort**: This platform allows users to create a portable, reusable, and decentralized digital identity that can be used across different services. The identity is a smart contract on the Ethereum blockchain as a digital representation of the user. More information here: `https://www.uport.me`.

Some other initiatives

- **Decentralized Identity Foundation**: `https://identity.foundation`
- **Standardization efforts by W3C**:
    - The VC data model specification can be found here: `https://www.w3.org/TR/vc-data-model/`
    - DID specifications can be found here: `https://www.w3.org/TR/did-core/`

There are many other projects; as the ecosystem is rapidly evolving and thriving, it is expected that more and more projects will emerge.

While using VCs and DIDs allows building a robust and secure decentralized identity system or SSI, it does have its own challenges, which need to be addressed to make it even more efficient and secure. We discuss these challenges next.

Challenges

With all this innovation and evolution, it is clear that the future is Web3 and identity will play a vital role in making the ecosystem a reality and will enable novel use cases that were simply not possible in the web 2.0 world. However, some challenges need to be addressed to further improve the SSI ecosystem:

- Lost device – What if a device (mobile phone) on which all of a person's VCs are stored is stolen or left behind on a train? It is actually the same as a physical wallet with paper credentials such as a passport, ID card, etc. being lost. Some effort has been made to alleviate this issue, but a lot needs to be done. How can lost credentials be recovered instead of requesting them again from the issuer? It's probably quicker, due to their digital nature, to get them issued to you again, but still, if there is a possibility of recovery, it would be a much more user-friendly solution.
- The transition from web 2.0 to Web3 poses some challenges, in terms of migration of data, accounts, etc. from web 2.0 to Web3.

- Digital wallet vulnerabilities – As the device on which the wallet is hosted and the wallet itself are subject to malware and hacking attacks, it's important to ensure that the wallets are tested thoroughly before public release. Nevertheless, many information security risks apply to digital wallets, as we have in the usual IT world, including viruses, hacking, storage corruption, substandard software, and many others.
- Other challenges include limitations on the VDR/blockchain layer, such as privacy, scalability, and interoperability.

Summary

In this chapter, we covered the important subject of decentralized digital identity and how blockchain can enable a decentralized identity and self-sovereign identity ecosystem. We covered various concepts including the models of identity, VCs, DIDs, and SSI wallets, and learned how the VCs ecosystem works. We also explored some examples of projects in this space, what role identity plays in Web3, and how Web3 enables decentralized self-sovereign digital identities. We then covered some projects like Hyperledger Indy and finally looked at some challenges.

In the next chapter, we'll cover **Decentralized Finance**, which is another exciting topic.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

21

Decentralized Finance

Decentralized finance is a new financial system built using smart contracts and the blockchain technology that operates in a decentralized manner without relying on traditional financial intermediaries.

In this chapter, we'll cover the following topics:

- Introduction
- Financial markets and trading
- Application of blockchain in finance
- Decentralized finance
- DeFi primitives
- DeFi services
- DeFi benefits
- Using Uniswap

Now let's have a quick introduction to finance and financial markets. This will provide a foundation for the material presented next, such as DeFi, as many of the terms and ideas are the same, albeit in a different context.

Introduction

Finance is the study and management of money, investments, the creation of money, and other financial instruments to maximize wealth and minimize risk. The current global financial system operates through a complex network of financial markets, institutions, and intermediaries, with the goal of facilitating the efficient allocation of financial resources. Financial markets, such as stock exchanges and bond markets, serve as venues for buying and selling financial assets. Financial institutions, such as banks and investment firms, play a crucial role in intermediating between borrowers and lenders and providing financial services to their clients. Regulators, such as government agencies, are responsible for overseeing and regulating financial institutions and markets to ensure their safe and sound operation and to protect consumers. Investors, such as individuals, institutions, and organizations, invest in financial assets to generate returns and manage financial risk.

All these entities and components of the current global financial system work together to promote financial stability, growth, and prosperity and to serve the financial needs of individuals, businesses, and society.

The financial services industry offers a diverse range of financial assets and instruments to meet the needs of its clients. For example, stocks, bonds, mutual funds, and **Exchange-Traded Funds** (**ETFs**); money market instruments, such as certificates of deposit; and derivatives, such as options and futures contracts. These are just a few of the many financial assets and instruments offered by the financial services industry.

A financial market is a platform where financial instruments are traded between buyers and sellers. In a financial market, the prices of financial assets are determined by supply and demand and reflect the expectations of market participants regarding future events and conditions.

The current financial system is centralized and, traditionally, all financial institutions hold custody of customer funds and assets. They also serve as intermediaries for transactions. This centralization means that customer accounts can be frozen, transaction censoring can happen, and institutions can charge unfairly for the services provided. All these financial institutions are very highly regulated and must adhere to strict compliance rules such as **Know Your Customer** (**KYC**), **Anti Money Laundering** (**AML**), and **Combating the Financing of Terrorism** (**CFT**). Such regulations provide confidence to investors and keep the financial system working efficiently; however, note that because all these systems are inherently centralized, 100% fairness, transparency, competitive fees, and reliability cannot be guaranteed. Moreover, the customer must share personal and financial details with these service providers and as a result, the institutions know the real identity and complete account and transaction history of the customer. This is OK and works reasonably well in the current financial system; however, this lack of privacy might not be acceptable to some customers. Moreover, the customer is totally oblivious to what is going on behind closed doors; there is limited transparency when it comes to transaction processing. Customers don't know how their data is handled or how transactions are performed; all of this is opaque and means that the system is not as transparent as it should be. Databases are opaque and siloed, and data sharing between institutions becomes a problem. Customers are mostly totally unaware of how their data and transactions are handled and processed. This means that customers must fully trust these centralized entities.

Note that it is not the case that the current financial system is totally broken and nothing works. The current financial system works quite well and indeed upholds the highest standards of quality, adherence to regulation, and compliance requirements, but it can be improved in terms of privacy, financial inclusion, transparency, trust, and efficiency.

In 2008, a breakthrough invention, known as Bitcoin, provided a platform that could provide solutions to these problems. Bitcoin (the first electronic cash system, which is fully peer-to-peer and needs no trusted third party to operate) introduced blockchain, which is the fundamental layer that can provide solutions to most, if not all, of these problems.

Now let's dig deeper and understand some of the traditional finance concepts. First, let's have a look at financial markets.

So far, we learned what finance is. Finance is the broader concept that includes the management of money and assets, while financial markets refers to the platforms, venues, or systems where financial activities such as trading are performed.

Financial markets

Financial markets enable the trading of financial securities such as bonds, equities, derivatives, and currencies. There are broadly three types of markets: money markets, credit markets, and capital markets:

- Money markets are short-term markets where money is lent to companies or banks for interbank lending. Foreign exchange, or forex, is another category of money markets where currencies are traded.
- Credit markets consist mostly of retail banks that borrow money from central banks and loan it to companies or households in the form of mortgages or loans.



Retail banks are commercial banks that offer financial products and services to individuals and businesses, whereas central banks oversee the monetary system and regulation of the financial system of a country.

- Capital markets facilitate the buying and selling of financial instruments, mainly stocks and bonds. There are many types of financial instruments, such as cash instruments, derivative instruments, loans, securities, and many more. Securitization is the process of creating new security by transforming illiquid assets into tradeable financial instruments. Capital markets can be divided into two types: primary and secondary markets. Stocks are issued directly by the companies to investors in primary markets, whereas in secondary markets, investors resell their securities to other investors via stock exchanges. Various electronic trading systems are used by exchanges today to facilitate the trading of financial instruments.

A major activity performed in financial markets is trading, which we discuss next.

Trading

A market is a place where parties engage in exchange. It can be either a physical location or an electronic or virtual location. Various financial instruments, including equities, stocks, foreign exchanges, commodities, and various types of derivatives are traded at these marketplaces.

Derivatives are financial contracts whose value is derived from an underlying asset, such as stocks, bonds, commodities, currencies, or indices. They are used to hedge risk, speculate, and manage exposure to price movements of the underlying asset. Several classes of derivatives include futures, options, swaps, and forwards. They are traded in financial markets on organized exchanges, **Over-the-Counter** (**OTC**) markets, and various electronic trading platforms. Almost all financial institutions have introduced electronic trading software platforms to trade various types of instruments from different asset classes.

Trading can be defined as an activity in which traders buy or sell various financial instruments to generate profit and hedge risk. Investors, borrowers, hedgers, asset exchangers, and gamblers are a few types of traders. Traders have a short position when they owe something; in other words, if they have sold a contract, they have a short position. When traders buy a contract, they have a long position. There are various ways to transact trades, such as through brokers or directly on an exchange or **OTC** where buyers and sellers trade directly with each other instead of using an exchange. Brokers are agents who arrange trades for their customers and act on a client's behalf to deal at a given price or the best possible price.

Traders use exchanges to perform trading functions like buying and selling securities, which we introduce next.

Exchanges

An exchange is a centralized platform where securities, commodities, derivatives, and other financial instruments are bought and sold. Exchanges serve as intermediaries between buyers and sellers of financial assets, providing a standardized and regulated marketplace for trading. They ensure that transactions are executed fairly and transparently, provide market data, and facilitate the settlement of trades. Some well-known examples of exchanges include the **New York Stock Exchange** (**NYSE**), the Nasdaq Stock Market, the Tokyo Stock Exchange, the London Stock Exchange, and Euronext. Each exchange specializes in trading specific types of securities, such as stocks, bonds, futures, options, or currencies, and operates under its own set of rules and regulations.

Exchanges are usually considered to be very safe, regulated, and reliable places for trading. During the last few decades, electronic trading has gained popularity over traditional floor-based trading. Now, traders send orders to a central electronic order book from which the orders, prices, and related attributes are published to all associated systems using communications networks, thus, in essence, creating a virtual marketplace. Exchange trades can be performed only by members of the exchange. To trade without these limitations, the counterparties can participate in OTC trading directly.

Exchanges deal with orders; let's now explore what an order is and look at its various properties.

Orders and order properties

Orders are instructions to trade, and they are the main building blocks of a trading system. They have the following general attributes:

- The instrument's name
- The quantity to be traded
- Direction (buy or sell)
- The type of order that represents various conditions, for example, limit orders and stop orders



In finance, a limit order is a type of order that allows the selling or buying of an asset at a specific price or better. A stop order is similar, but the key difference is that a limit order is visible to the market, whereas a stop order only becomes active (as a market order) when the specified stop price is met.

Orders are traded by bid prices and offer prices. Traders show their intention to buy or sell by attaching bid and offer prices to their orders. The price at which a trader will buy is known as the *bid price*. The price at which a trader is willing to sell is known as the *offer price*.

In order to facilitate the correct handling of orders, order management and routing systems are used, which we introduce next.

Order management and routing systems

Order routing systems route and deliver orders to various destinations depending on the business logic. Customers use them to send orders to their brokers, who then send these orders to dealers, clearing houses, and exchanges.

There are different types of orders. The two most common ones are *market orders* and *limit orders*. A market order is an instruction to trade at the best price currently available in the market. These orders get filled immediately at spot prices.

In finance, a *spot price* is the current price of an asset in a marketplace at which it can be bought or sold for immediate delivery.

On the other hand, a limit order is an instruction to trade at the best price available, but only if it is not lower than the limit price set by the trader. This can also be higher depending on the direction of the order: either to sell or buy. All of these orders are managed in an *order book*, which is a list of orders maintained by the exchange, and it records the intention of buying or selling by the traders.

A position is a commitment to sell or buy a number of financial instruments, including securities, currencies, and commodities for a given price. The contracts, securities, commodities, and currencies that traders buy or sell are commonly known as **trading instruments**, and they come under the broad umbrella of **asset classes**. The most common classes are real assets, financial assets, derivative contracts, and insurance contracts.

A trade is composed of several elements, which we discuss next.

Components of a trade

A trade ticket is the combination of all of the details related to a trade. However, there is some variation depending on the type of the instrument and the asset class. These elements are described here.

First, we have the underlying instrument that is the basis of the trade. It can be a currency, a bond, an interest rate, a commodity, or an equity.

The attributes of financial instruments include:

- **General attributes**: This includes the general identification information and essential features associated with every trade. Typical attributes include a unique ID, an instrument name, a type, a status, a trade date, and a time.
- **Economics**: Economics are features related to the value of the trade; for example, the buy or sell value, ticker, exchange, price, and quantity.

- **Sales:** Sales refers to the sales characteristic-related details, such as the name of the salesperson. It is just an informational field, usually without any impact on the trade lifecycle.
- **Counterparty:** The counterparty is an essential component of a trade as it shows the other side (the other party involved in the trade) of the trade, and it is required to settle the trade successfully. The normal attributes include the counterparty name, address, payment type, reference IDs, settlement date, and delivery type.

Trade lifecycle

A general trade lifecycle includes various stages from order placement to execution and settlement. This lifecycle is described step-by-step as follows:

- **Pre-execution**: An order is placed at this stage.
- **Execution and booking:** When the order is matched and executed, it is converted into a trade. At this stage, the contract between counterparties is matured.
- **Confirmation:** This is where both counterparties agree to the particulars of the trade.
- **Post-booking:** This stage is concerned with various scrutiny and verification processes required to ascertain the correctness of the trade.
- **Settlement:** This is the most vital part of the trade lifecycle. At this stage, the trade is final.
- **End-of-day processing**: End-of-day processes include report generation, profit and loss calculations, and various risk calculations.

This lifecycle is also shown in the following image:



Figure 21.1: Trade lifecycle

In all the aforementioned processes, many people and business functions are involved. Most commonly, these are divided into functions such as front office, middle office, and back office.

While the trading industry is very secure and works well, there are some problems that can occur. One of them is order anticipation, where order anticipators try to make a profit before other traders can carry out trading. This is based on the anticipation of a trader who knows how the activities of other trades will affect prices. Frontrunners, sentiment-oriented technical traders, and squeezers are some examples of order anticipators.

Also, there is a possibility of market manipulation. It is strictly illegal in all countries. Fraudulent traders can spread false information in the market, which can then result in price movements, enabling illegal profiteering. Usually, manipulative market conduct is trade-based, and it includes generalized and time-specific manipulations. Actions that can create an artificial shortage of stock, an impression of false activity, and price manipulation to gain criminal benefits are included in this category.

Both of these concepts are relevant to financial crime. However, it is possible to develop blockchain-based systems that can thwart market abuse due to its inherent transparency and security properties.

It was quickly realized after the invention of Bitcoin that blockchain can enable many use cases in the financial services industry that can bring about efficiency, transparency, and security. We present some of these use cases next and explore what impact blockchain has on traditional finance and how it improves existing services. After this section, we'll dive into decentralized finance, also known as DeFi.

Applications of blockchain in finance

Blockchain has many potential applications in the finance industry. Blockchain in finance is currently the hottest topic in the industry, and major banks and financial organizations are researching to find ways to adopt blockchain technology, primarily due to its highly desired potential to save costs. These applications include but are not limited to payments, cross-border payments, remittance, trade finance, supply chain finance, security trading, clearing and settlement, accounting, identity, KYC and AML, insurance, post-trade settlements, financial crime prevention, lending, and borrowing. We discuss some of these next.

Insurance

In the insurance industry, blockchain technology can help to stop fraudulent claims, increase the speed of claim processing, and enable transparency. Imagine a shared ledger between all insurers that can provide a quick and efficient mechanism for handling intercompany claims. Also, with the convergence of IoT and blockchain, an ecosystem of smart devices can be imagined, where all these things can negotiate and manage their insurance policies, which are controlled by smart contracts on the blockchain.

Blockchain can reduce the overall cost and effort required to process claims. Claims can be automatically verified and paid via smart contracts and the associated identity of the insurance policyholder. For example, a smart contract, with the help of an **oracle** and possibly IoT, can make sure that when the accident occurred, it can record related telemetry data and, based on this information, release payment. It can also withhold payment if the smart contract, after evaluating conditions of payment, concludes that payment should not be released; for example, in a scenario where an authorized workshop did not repair the vehicle or was used outside a designated area, and so on and so forth. There can be many conditions that a smart contract can evaluate to process claims and the choice of these rules depends on the insurer, but the general idea is that smart contracts, in combination with IoT and oracles, can automate the entire vehicle insurance industry.

Post-trade settlement

This is the most sought-after application of blockchain technology. Currently, many financial institutions are exploring the possibility of using blockchain technology to simplify, automate, and speed up the costly and time-consuming post-trade settlement process.

To understand the problem better, the trade lifecycle will be described briefly. A trade lifecycle contains three steps: **execution**, **clearing**, and **settlement**. Execution is concerned with the commitment of trading between two parties and can be entered into the system via front-office order management terminals or exchanges. Clearing is the next step, whereby the trade is matched between the seller and buyer based on certain attributes, such as price and quantity.

At this stage, accounts that are involved in payment are also identified. Finally, the settlement is where, eventually, security is exchanged for payment between the buyer and seller.

In the traditional trade lifecycle model, a central clearing house is required to facilitate trading between parties, which bears the credit risk of both parties. The current scheme is somewhat complicated, whereby a seller and buyer have to take a complicated route to trade with each other. This comprises various firms, brokers, clearing houses, and custodians, but with blockchain, a single distributed ledger with appropriate smart contracts can simplify this whole process and can enable buyers and sellers to talk directly to each other.

Notably, the post-trade settlement process usually takes two to three days and has a dependency on central clearing houses and reconciliation systems. With the shared ledger approach, all participants on the blockchain can immediately see a single version of truth regarding the state of the trade. Moreover, P2P settlement is possible, which results in the reduction of complexity, cost, risk, and the time it takes to settle the trade. Finally, intermediaries can be eliminated by making use of the appropriate smart contracts on the blockchain. Also, regulators can view the blockchain for auditing and regulatory requirements.



This can be very useful in implementing MIFID-II regulation requirements (`https://www.fca.org.uk/markets/mifid-ii`).

Financial crime prevention

KYC and AML are the key enablers for the prevention of financial crime. In the case of KYC, currently, each institution maintains its own copy of customer data and performs verification via centralized data providers. This can be a time-consuming process and can result in delays in onboarding a new client.

Blockchain can provide a solution to this problem by securely sharing a distributed ledger between all financial institutions that contain verified and true identities of customers. This distributed ledger can only be updated by consensus between the participants, thus providing transparency and auditability. This can not only reduce costs but also enable regulatory and compliance requirements to be satisfied in a better and more consistent manner.

In the case of AML, due to the immutable, shared, and transparent nature of blockchain, regulators can easily be granted access to a private blockchain where they can fetch data for relevant regulatory reporting. This will also result in reducing complexity and costs related to the current regulatory reporting paradigm. This is where data is fetched from various legacy and disparate systems, and then aggregated and formatted together for reporting purposes. Blockchain can provide a single shared view of all financial transactions in the system that are cryptographically secure, authentic, and auditable, thus reducing the costs and complexity associated with the currently employed regulatory reporting methods. A simple solution is shown in *Figure 21.2*:

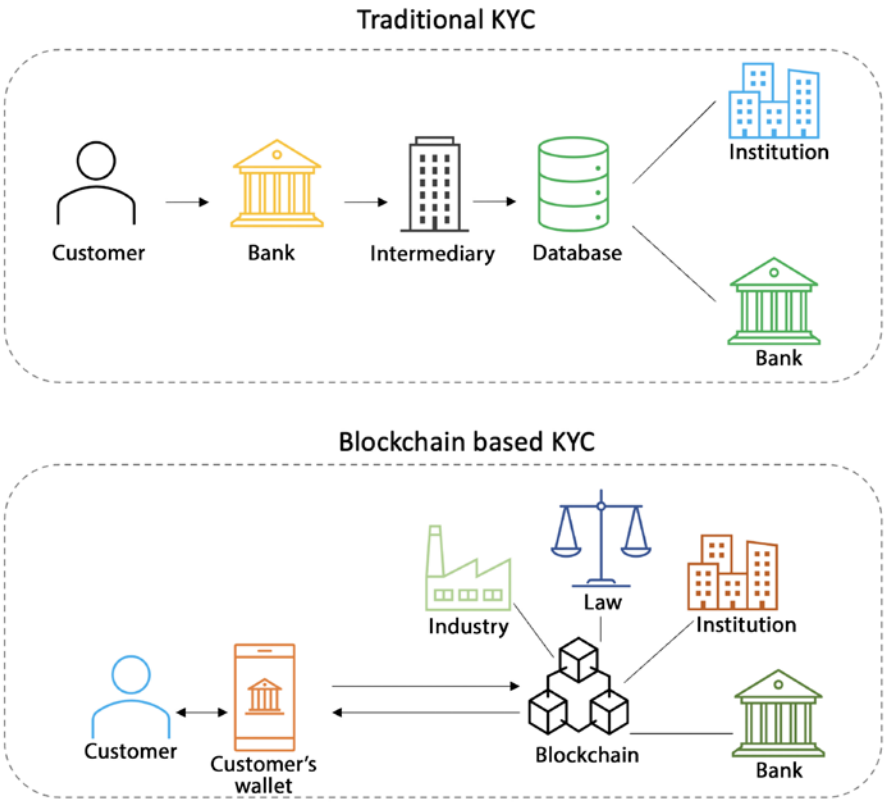


Figure 21.2: Traditional vs blockchain-based KYC

Blockchain-based KYC is much more efficient than traditional KYC. In a traditional system, a customer submits KYC documents/data to a bank. The bank verifies and stores the data and also sends it to a third-party centralized intermediary for storage. Other banks perform the same process and/or request data from the centralized intermediary, which is not efficient. Mostly, current KYC processes are siloed, and banks perform this process individually. Usually, all institutions run their own databases with not much sharing. There is also a centralized registry in some cases, but as it's centralized, it doesn't have the benefits that come with decentralization. As a result of this complex state of affairs, client onboarding takes a long time – it can even be two months in some cases. Also, it results in duplication of effort as each bank has to do its own KYC. In blockchain-based KYC, a customer submits KYC data to the blockchain. An institution verifies KYC data and posts the KYC status on the blockchain for that customer. The customer receives the KYC-compliant status (in the form of a verifiable credential or some other means).

Banks and other institutions access KYC data from the chain without needing to request it from other institutions and the customer can present the issued VC to any other institution as proof that the customer is properly KYC-cleared already and doesn't need KYC done again. This means that other banks can reuse KYC data, without duplication, and can even use verifiable credentials to verify (without even looking at the customer's data) that the customer is already KYC cleared and can access financial services. This is all made possible by blockchain.

Payments

A payment is a transfer of money or its equivalent from one party (the payer) to another (the payee) in exchange for services, goods, or for fulfilling a contract. Payments are usually made in the form of cash, bank transfers, credit card payments, and cheques. There are various electronic payment systems in use, such as **Bankers' Automated Clearing System** (**BACS**) and the **Clearing House Automated Payment System** (**CHAPS**). All these systems are, however, centralized and governed by traditional financial service industry codes and practices. These systems work adequately, but with the advent of blockchain, the potential of technology has arisen to address some of the limitations that exist in currently used systems.

The key advantages that blockchain technology can bring to payments are decentralization, faster settlement times, better resilience, and high availability. With all these advantages, it is easy to see how the payments industry can benefit from blockchain technology. There is also another branch of payments that deals with international or cross-border payments and comes with its own challenges. In traditional finance, cross-border payment is a complex process that can take days to process and involves multiple intermediaries. Current mechanisms suffer from delays incurred by multiple intermediaries, enforcement of regulations, differences in terms of regulations between different jurisdictions...the list goes on. All of these issues can be addressed by utilizing blockchain technology. The most significant advantage is decentralization, where, due to the lack of the requirement of intermediaries, payments can be made directly between businesses or individuals. Also, due to P2P connectivity, the whole process becomes a lot faster—almost immediate, in fact—which results in more productivity and business agility. We can see, in *Figure 21.3*, how a complex system can be transformed into a simpler system using blockchain.

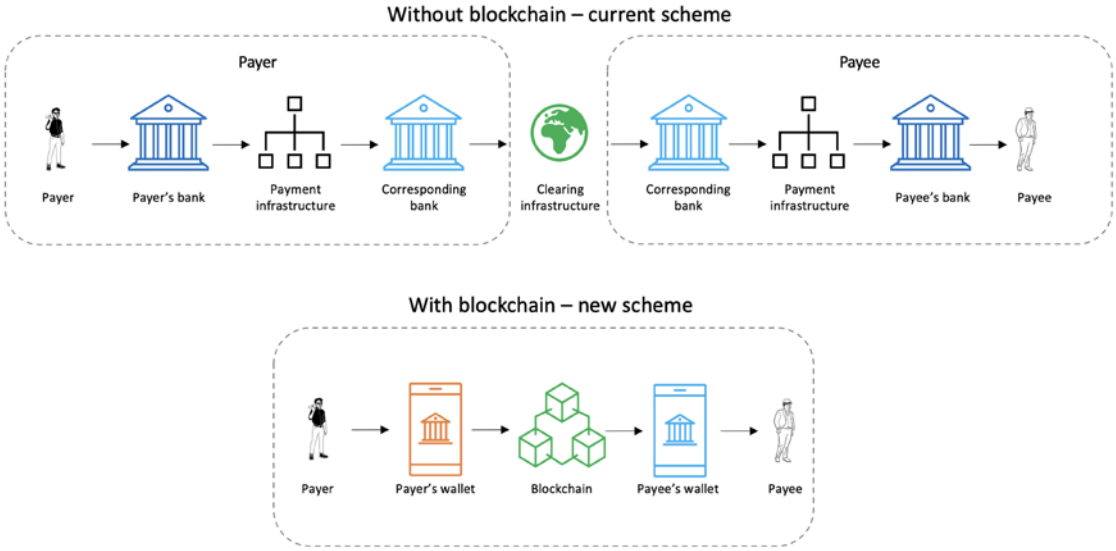


Figure 21.3: Cross-border payment system – from complex to simple using blockchain

The use of blockchain in the cross-border payment system removes settlement risks. The payee receives funds in seconds/minutes instead of days, even for international payments. Moreover, it's low cost as there are no intermediary banks that can charge fees. There is also no need for reconciliation as all data is there on the chain, already verified, and all participants can see and use it. There are no cutoff times or delays due to holidays etc. as the system is running 24/7 as long as the blockchain is running. It also enables **Peer-to-Peer** (**P2P**) payments, where payers and payees can deal with each other directly through wallets connecting to the blockchain, without any intermediaries at all. In some cases, due to regulatory requirements, institutions might be needed, but still, the overall experience for a customer is much easier, quicker, and seamless.

More and more use cases are emerging: clearing and settlements, identity, primary and secondary markets, trade finance, supply chain, and many others. It is now clear that blockchain indeed enables use cases that were not possible before.

While most of the innovations discussed so far are largely based on permissioned chains and aim to improve existing financial processes by implementing them on blockchains, another phenomenon called decentralized finance on public chains has emerged.

Let's now explore what DeFi is.

Decentralized finance

We can define **DeFi** as a financial mechanism built using smart contracts on public blockchains aiming to provide traditional and novel financial services in a decentralized, trustless, interoperable, and permissionless manner.

While **Traditional Finance** (**TradFi**) – or **Centralized Finance** (**CeFi**) – has strived for decades to achieve efficiency and improve its services and has achieved quite a lot, DeFi however, due to the advantages of blockchain, could result in a total transformation of financial services in the future and has enabled (and is enabling and will enable) use cases that were never possible before the invention of blockchain. It is an ecosystem that has emerged as a result of the development of many different types of financial applications built on top of blockchains. With blockchains being decentralized and the applications being related to finance, the term decentralized finance emerged, or DeFi for short.

DeFi can be defined as an umbrella term used to describe a financial services ecosystem that is built on top of blockchains. **Centralized Finance** (**CeFi**), is a term now used to refer to the traditional financial services industry, which is centralized in nature, while DeFi is a movement to decentralize the traditional centralized financial services industry.

Tokenization plays a vital role in the DeFi ecosystem. DeFi is based on asset tokenization. We discussed tokenization in detail in *Chapter 15*, *Tokenization*. DeFi is a vast subject with many different applications, protocols, assets, tokens, blockchains, and smart contracts.

The DeFi ecosystem is the fastest-growing infrastructure running on different blockchains. Originally, most DeFi applications ran on Ethereum, but with the advent of new chains like Solana, Cosmos, Polkadot, Avalanche, EOS, Hedera, and many others, we see an exponential expansion of the DeFi ecosystem. However, Ethereum remains the most preferred. DeFi is enabling use cases that are novel and were simply not possible before.

The range of DeFi DApps includes, but is not limited to, lending and borrowing, trading, asset management, insurance, tokenization, and prediction markets. All these decentralized applications, along with their smart contracts and infrastructure, make up the DeFi ecosystem.

In order to determine if an application or protocol is truly a DeFi protocol, we can ask three questions:

1. Is the user in full control of the financial asset?
2. Can a single entity censor transactions?
3. Can a single entity censor the protocol execution?

If the answer to all these questions is no, then the DApp is indeed a true DeFi DApp. If the answer to any of the questions is yes, then it's not a true DeFi application or protocol.

Let's now compare centralized finance with decentralized finance, which helps to understand how DeFi differs from CeFi.

	CeFi	**DeFi**
Admission	Permissioned	Permissionless
Base layer	Centralized databases	Decentralized blockchains
Authorization	Required	Usually not required
Asset holding	Custodial – trusted third party	Non-custodial – trustless, no single party in charge
Trust	Centralized	Decentralized
Governance	Centralized – under the control of a single party	Decentralized – no single party in control
Privacy	Limited, real identity must be revealed through KYC/AML processes	Mostly pseudonymous, or anonymous; however, regulatory pressure has introduced KYC/AML even in this space, but with advances in zero-knowledge technologies, true privacy with reasonable regulatory compliance is expected to be achieved.
Transparency	Opaque	Transparent/open
Risk	Less risky (due to stable and established platforms, control, and regulatory requirements)	More risky (due to lack of regulation and nascent technology)
Accessibility	Restricted due to regulation, controls, policies, etc.	Open to anyone with the internet

Properties of DeFi

DeFi should demonstrate five key properties, including:

- **Non-custodial**: users have full control over their assets.
- **Permissionless**: the system is fully inclusive, and anyone can access financial services.
- **Decentralized**: there is no single authority in control of the platform, protocol, or transaction execution. In other words, there are no trusted third-party requirements.
- **Transparent**: the system is open to audit and inspection by anyone to ensure its integrity.
- **Composable**: the system allows the creation of new financial products from a few basic building blocks.

DeFi layers

The DeFi ecosystem can be described in terms of a layered architecture, where each layer represents a class of operations and technology.

Let's look at all these layers one by one:

1. **Settlement layer**: This is the base layer where blockchain platforms like Ethereum, Solana, and Polkadot exist.
2. **Asset layer**: This layer is composed of tokens and cryptocurrencies, native tokens, stablecoins, NFTs, and other tokens – for example, ERC20 and ERC721.
3. **Protocol layer**: This layer consists of DeFi protocols including exchanges, loan platforms, DeFi insurance platforms, and many other protocols. This is implemented using smart contracts. More specifically, **Decentralized Autonomous Organizations** (**DAOs**) exist at this layer.
4. **Application layer**: This layer is composed of decentralized applications that run on top of the protocols in the protocol layer and include applications and interfaces.
5. **Interoperability layer**: This layer is responsible for providing cross-chain interoperability, which includes bridges, hubs, relay chains, and various cross-chain messaging protocols.
6. **Aggregation layer**: This layer introduces the ability to aggregate multiple applications into an easy-to-use single platform for end users.

We can visualize this layered architecture with different actors and entities in *Figure 21.4* below:

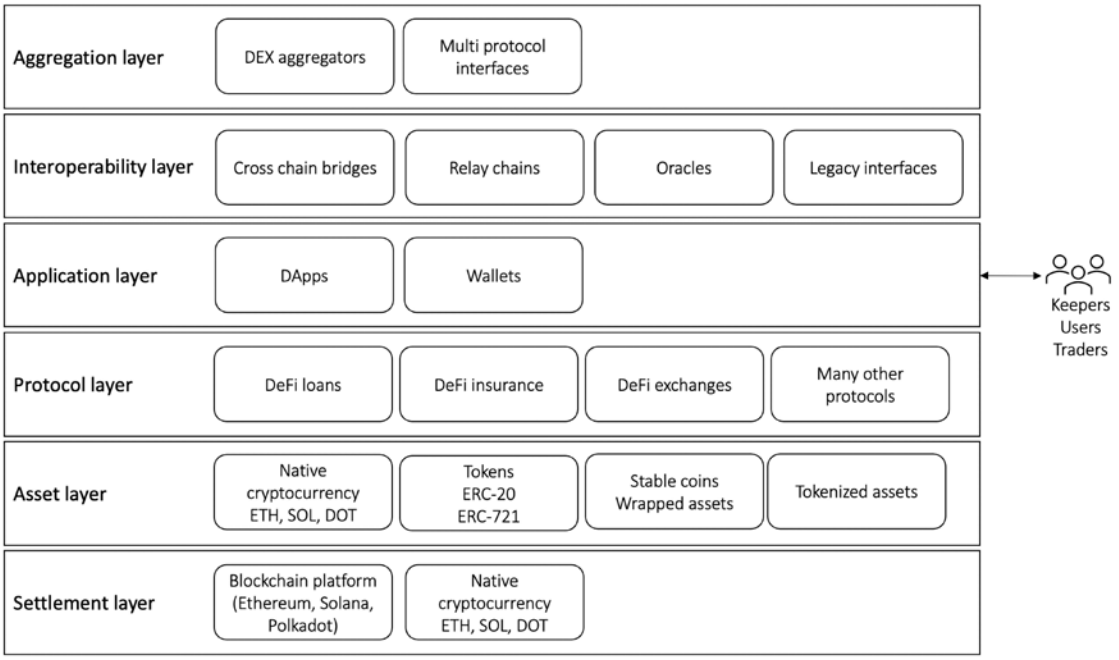


Figure 21.4: DeFi layered architecture

Each layer in the DeFi ecosystem plays a crucial role in enabling the creation and growth of decentralized financial applications and services. The DeFi ecosystem has the potential to provide greater financial access and inclusion, as well as new financial opportunities and innovations. However, it is important to note that DeFi is rapidly evolving, and there are some risks and uncertainties associated with participating in the DeFi ecosystem. We'll cover these risks later in this book, where we discuss blockchain security and other challenges.

DeFi primitives

The DeFi ecosystem comprises many protocols, services, actors, and entities. These elements are listed below:

- **Transactions and smart contracts**: Transactions and smart contracts are the basic building block on which DeFi runs. Transactions and smart contract functionality are provided by the underlying blockchain, such as Ethereum.
- **Keepers**: Keepers can be defined as a class of **External Owned Accounts** (**EOAs**) that have the incentive to perform an action in a DeFi protocol. A keeper can also be an autonomous agent or a bot that monitors and triggers certain actions based on specific conditions within a smart contract and earns rewards to do so. Keepers act as the "maintenance crew" of the DeFi ecosystem by continuously monitoring the state of smart contracts and executing pre-defined transactions or activities when certain conditions are met. For example, a keeper may monitor a lending protocol and automatically repay a loan on behalf of the borrower if it falls below a certain collateralization ratio. Keepers can help maintain the health and stability of DeFi protocols and are usually incentivized with rewards for their services.
- **Token**: A token is a digital representation of an asset, such as a cryptocurrency, a commodity, a stock, or a fiat currency. These tokens are created and managed on a blockchain and are used as a means of exchange, store of value, governance, or for other functions within DeFi protocols. Tokens can be used in a variety of DeFi protocols such as decentralized exchanges and lending platforms. They can also be used to incentivize users to participate in these protocols or as a right to vote on governance decisions within the protocol. Some of the popular tokens used in DeFi protocols include stablecoins like USDC, DAI, and USDT, and governance tokens like UNI, AAVE, and DOT. For more details on tokenization, refer to *Chapter 15*, *Tokenization*, which covered this subject in detail.
- **Oracle**: An oracle can be defined as a mechanism that feeds external information from the outside world into the blockchain. It plays a vital role in the DeFi ecosystem. For example, it enables the creation of applications that require real-time data, such as decentralized exchanges, lending platforms, and insurance protocols.
- **Governance**: Governance refers to the decision-making process for managing and evolving a decentralized protocol. It is typically achieved through consensus among stakeholders, such as token holders, who express their preferences through voting mechanisms. **DAOs** are a common mechanism used in DeFi protocols for governance. Governance can include decisions about technical changes, economic incentives, and the addition or removal of features and functionalities in the protocol. Decentralized governance enables the creation of more open, transparent, and democratic DeFi protocols controlled by users rather than centralized intermediaries or trusted third parties.

- **Custody**: Custody is a fundamental primitive in DeFi that allows users to escrow or keep funds in a smart contract. Such custody allows the creation of different solutions, especially lending protocols, insurance funds, market making, and automated disbursement of incentives.
- **Incentive**: Incentives in the DeFi ecosystem play an important role in keeping the DeFi ecosystem profitable for users. A common type is stake incentives, where a user stakes some assets to secure or participate in the governance of a DeFi protocol and earn rewards as a return. It is worth noting that incentives can be reduced or increased in a protocol based on the behavior of the user, market conditions, and protocol rules. Incentives can be negative or positive – for example, a staking reward is a positive incentive earned by contributing assets in a protocol to secure it. Negative incentives can occur due to slashing rules coded in a protocol and are used to discourage undesirable behavior of the stakers.
- **Bridge**: A bridge is a mechanism that connects two separate blockchain networks, allowing assets and data to be transferred between them. Bridges are important for enabling interoperability between different DeFi ecosystems, as they allow users to move assets and data between different networks. Bridges can be implemented as cross-chain atomic swaps, token wrapping, or by using bridge protocols that use a set of smart contracts to facilitate the transfer or exchange of assets between blockchain networks. Some common bridges are the Ethereum-Binance Smart Chain bridge, the Ethereum-Polygon bridge, and the Ethereum-Polkadot bridge. Bridges are important for expanding the functionality and reach of DeFi, as they enable assets and liquidity to be moved between different blockchain networks, making it easier for users to access a wider range of DeFi applications and services.

Let's now discuss what services make up the DeFi ecosystem.

DeFi services

DeFi is composed of many different protocols, components, and services. We discuss these services next.

Asset tokenization

This is the foundation on which DeFi is built. It is the process of adding new assets to a blockchain platform. We can think of a token as a digital representation of a real-world asset. Tokenization makes assets more accessible, programmable, flexible, and easy to transfer. There are many types of tokens, including security tokens, governance tokens, NFTs, stablecoins, and quite a few others. For more details on tokenization, refer to *Chapter 15*, *Tokenization*, where these concepts are explained in detail.



Tokenization can make illiquid assets liquid, which is not possible to do by traditional means. e.g., tokenize a piece of famous art.

Decentralized exchanges

We can divide exchanges into two types: centralized exchanges and decentralized exchanges.

An exchange is traditionally centralized, which means that some of them might not be entirely trustworthy. They are not transparent, and custody is also centralized. We have seen how centralized cryptocurrency exchanges have been subject to hacking and malpractices and, as a result, billions of funds are lost. For example, Mt. Gox, Coincheck, and Binance suffered losses due to successful hacking attacks in the past.

Decentralized Exchange (**DEX**) alleviates the problems that a **Centralized Exchange** (**CEX**) faces. We can define a DEX as a type of cryptocurrency (token) exchange that operates on a blockchain. DEXs allow for the direct exchange of cryptocurrencies between users (peers), without the need for a central authority or intermediaries. This is in contrast with centralized exchanges, which act as intermediaries and act as custodians of assets of their users.

In DeFi and blockchain-based trading systems in general, the trading of tokens is the prime activity. Tokens can be traded on exchanges. With the advent of DeFi, decentralized exchanges have emerged. DEXs are decentralized and therefore require no central authority or intermediary to facilitate trading. DEXs are decentralized, transparent, and non-custodial.

While the entry barrier to DEXs is low due to a lack of traditional regulatory requirements such as KYC requirements, this also makes them risky for investors because if somehow the DEX is hacked, then there is no protection against loss of funds. KYC is a standard due diligence practice in the financial services industry that ensures that the customer is legitimate and genuine. However, some centralized crypto exchanges have now started to do KYC and AML checks.

Some examples of DEXs are Uniswap, Bancor, WavesDEX, 0x, and IDEX. This ecosystem is growing at a very fast pace and is only expected to grow further.

The basic structure of a DEX is composed of a smart contract that facilitates trade between two parties. This smart contract consists of three functions: price discovery, trade matching, and trade clearing. We can visualize this in *Figure 21.5* below:

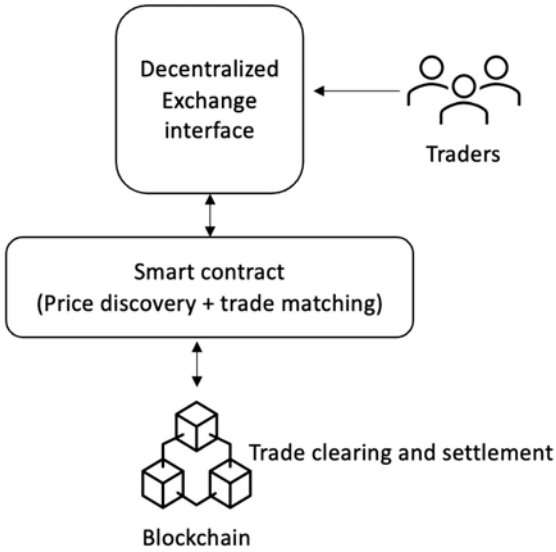


Figure 21.5: High-level architecture of a DEX

Price discovery is the process of determining the fair market value of a financial asset through the forces of supply and demand in the marketplace. It helps to establish the price at which a buyer and a seller are willing to trade the asset and ensures that assets are priced fairly and accurately. Price discovery occurs through various mechanisms, such as trading on exchanges, OTC transactions, and the use of derivatives, and is influenced by economic data, geopolitical events, and market sentiment. It is essential for the efficient allocation of capital and risk in financial markets.

There are several types of DEXs including AMM (also called liquidity pool-based DEX), order book-based DEX, and DEX aggregator. Classification is shown below:

1. Automated Market Maker:
    a. Order book DEX
    b. On-chain order book
2. Off-chain order book
3. DEX aggregators

Automated Market Maker (**AMM**) is a type of DEX where a liquidity pool provides a basis for automated market making. In a liquidity pool, the market-making is automated using a smart contract. A user with tokens (assets) adds liquidity to the liquidity pool, which can consist of several assets.

Any trader using the liquidity pool for swapping assets pays a fee to the user, who provided liquidity by adding their tokens into the liquidity pool. The core idea behind AMM is to let smart contracts do market-making instead of users placing orders manually. In short, liquidity pools can be thought of as shared pots of funds used by DEXs and deposited by users of the DEX, known as liquidity providers. Liquidity pools are used by DEXs to buy, sell, and fulfill orders. Liquidity providers earn pool fees in return for locking funds in the pool through a process called liquidity mining.

There are various AMM models. AMMs use math formulae to adjust prices in the liquidity pool. This math formula is programmed in the smart contract. There are several constant function market maker models including CPMM, CSMM, and CMMM.

CPMM

The constant product market maker formula is simply a product of two assets, as shown in the equation below:

$$x \times y = k$$

Where x represents the quantity of some asset x, y represents the quantity of some asset y, and k is a constant. The idea is that the product k of assets x and y must remain constant according to a defined constant. If a greater quantity of assets is added to the liquidity pool, then the constant will change; however, as long as they are traded, the net effect would keep k constant. In other words, the constant product formula ensures that the price of assets is adjusted according to supply and demand. For example, the purchase of asset x increases the price of x and decreases the price of y, and vice versa. The ratio of the number of assets of x and y sets the price accordingly. This constant product formula ensures that liquidity is always available. Functions like price discovery, trade matching, and market making are all served by this simple formula.

One of the key benefits of AMMs is that they provide instant liquidity instead of waiting for someone to place a qualifying/appropriate order to fulfill our trade exchange. In other words, it ensures liquidity even in the absence of other traders, by allowing buyers and sellers to exchange assets with the liquidity pool instead of each other. Another advantage of AMM is that it decreases the likelihood of price manipulation tactics, such as wash trading and front running, as the formula coded in the smart contract keeps the value of token pairs constant.

There is, however, the risk of slippage. Slippage can be defined as the difference between the current market price of an asset and the price at which the order is filled. However, if the liquidity pool has deep liquidity, and the trade sizes are also smaller, then slippage can be minimized; however, it is not 100% avoidable in CPMMs. The slippage can be expected or unexpected and needs to be managed accordingly.

Various DEXs use constant product formulas including SushiSwap, Bancor, and quite a few others.

CSMM

Constant sum market makers use a simple formula as shown below:

$$x + y = k$$

The formula simply means that the market maker would maintain a fixed sum of the two assets in the pool. While this is a simple function and provides protection against slippage, it doesn't provide unlimited liquidity.

CMMM

Constant mean market makers facilitate the formation of AMMs with more than two tokens and weightings that differ from the conventional 50/50 distribution. In this approach, the weighted geometric average of each reserve is unchanged. For instance, in a pool with three assets, x, y, and z, the formula would be:

$$(x \times y \times z)^{\left(\frac{1}{3}\right)} = k$$

This attribute permits variable exposure to different assets in the pool and allows swaps between any of the pool's tokens.

A comparison in visual form between different AMM models is shown in *Figure 21.6*, below:

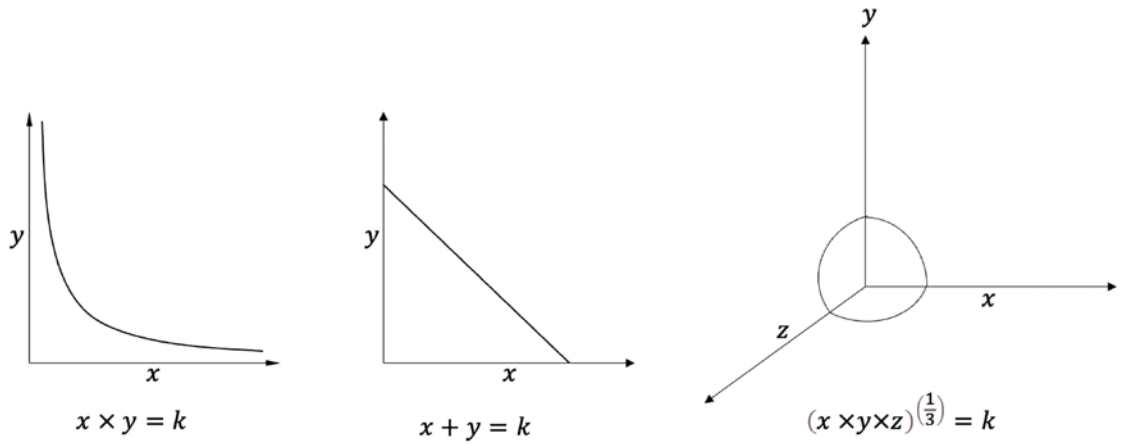


Figure 21.6: CPMM vs CSMM vs CMMM

In the preceding diagram, x, y, and z are the asset quantities, with different mathematical relationships between them as per the formula.

Order book-based DEX

Order book-based DEXs use the traditional trading model, which was in existence for a long time before the emergence of DeFi. The order book model, usually known as **Central Limit Order Book** (**CLOB**), matches orders from buyers and sellers according to a set of predetermined rules. In order book-based DEXs, traders can set buy and sell orders for an asset, and the order book will organize them based on their prices. If there is supply and demand for an asset, it can be traded in these exchanges. Order book exchanges are suitable for markets with high liquidity since they can accurately determine market prices and handle large orders without significant slippage. They are preferred by institutional and retail traders alike due to their capital-efficient and transparent trade execution model. There are two types of order book-based DEXs: on-chain order book and off-chain order book.

DEX aggregator

While there are many DEX platforms, each has different levels of liquidity and prices. It could become difficult for users to figure out which DEX offers the best price and/or liquidity. This is where DEX aggregators can help. DEX aggregators, as the name suggests, are platforms where liquidity from different exchanges is aggregated. Instead of going to individual DEXs, users can use the aggregator to trade their assets. The job of an aggregator is to find the deepest liquidity, best prices, lowest fees, lowest slippage, and other most suitable attributes for the user according to their requirements.

There are two types of such aggregators commonly in use: off-chain aggregators and on-chain aggregators. Off-chain aggregators are usually implemented as websites providing the aggregation service. The key advantage of such an aggregator is that they are usually linked with many blockchains, are very flexible and efficient, and can find the optimal strategy for executing the trade for users. The downside, however, is that these aggregators are effectively trusted third parties, and these are centralized services, which can lead to transaction front running, suboptimal strategy selection, and even imposed biased strategy selection that might favor a particular exchange.

On-chain aggregators, as the name suggests, run on a chain, and provide aggregation services using a smart contract. This smart contract finds a provably optimal strategy for execution with the best routing, best profits, and arbitrage possibility. There is also an extremely low likelihood of front running or any biasedness due to all aggregation logic coded in the smart contracts running on a decentralized platform, i.e., on-chain. While this is an advantage, a key disadvantage is that these aggregators don't scale well and can usually cover roughly three exchanges.

Some limitations of DEX include lower liquidity, comparatively limited features to centralized exchanges, where features like limit orders, stop orders, etc. are available but such features are not available on DEXs, and very limited cross-chain interoperability, where most DEXs really operate just on one chain and seldom talk to other chains. However, the DEX ecosystem is thriving, and these limitations are not a great hindrance to the adoption and use of DEXs. Specifically, in AMM models, impermanent loss and low capital efficiency are two main limitations.

Impermanent loss occurs when the value of the assets you have deposited into a liquidity pool fluctuates, i.e., either increases or decreases compared to the value when you deposited them. This means that the value of the deposited assets could be different when they are withdrawn compared to when they were initially deposited into the pool. It's worth noting that the term "impermanent" can be misleading, as a decrease in the price of a token may only be temporary and could go up again due to market conditions or other factors. In this case, the loss would be considered temporary, or impermanent, because the price eventually went back up. However, if the dollar value (value generated after conversion of the crypto asset into USD) of the token at the time of withdrawal is less than the value when it was deposited, then the loss becomes permanent.

AMMs also need a large amount of liquidity to match the pricing impact of an order book-based exchange, which results in low capital efficiency.

To address these limitations, several innovations have been proposed, including dynamic automated market makers, hybrid constant function market makers, proactive market makers, and virtual automated market makers.

A DAMM model can use price feeds through oracles and implied volatility to distribute liquidity more effectively along the price curve. The model can create a more resilient market maker that can adapt to changes in market conditions by integrating various dynamic variables into the algorithm. When volatility is low, the model can focus liquidity near the market price to improve capital efficiency, which can be extended during periods of high volatility to protect traders from impermanant loss. In simple terms, the mathematical relationship (e.g., $x \times y = k$) between assets is adjusted dynamically to ensure that the pool price continually and automatically aligns with the market price. This technique is intended to eliminate arbitrage opportunities.

Hybrid **Constant Function Market Makers** (**CFMMs**) combine several properties of AMMs, functions, and parameters to achieve a more stable, efficient, and profitable mechanism for traders.

Virtual AMMs aim to minimize price impact and mitigate impermanent loss. They use the same constant product formula as traditional AMMs – i.e., $x.y = k$, where . is some mathematical relationship – but instead of relying on a liquidity pool, traders deposit collateral to a smart contract. This allows trading synthetic assets instead of the underlying asset, which enables users to gain exposure to the price movement of many crypto assets efficiently.

The **Proactive Market Maker** (**PMM**) model has been developed to increase liquidity in protocols. This model mimics the behavior of a human market maker in a traditional central limit order book by using accurate market prices from an oracle. In response to market changes, the PMM protocol proactively moves the price curve of an asset, increasing the liquidity near the current market price. This facilitates efficient trading and reduces the impermanent loss for liquidity providers.

Another issue that AMMs suffer from is front running. This occurs when another user places a trade similar to a prospective buyer's but quickly sells it back, causing the price to rise, and then profiting from the price increase. This is possible because transactions in AMMs are public. Also known as a "sandwich attack," this is often automated using bots, which exacerbates the situation. In many cases, miners are the ones responsible for front running, and this has led to the term **Miner Extractable Value** (**MEV**), referring to the unfair profits that a third party can make from the original transaction.

Pros and cons of AMMs

One of the key advantages of AMMs is that there is no order book required to be maintained. The formulae related to CP are simply implementable in smart contracts. However, some disadvantages include the risk of impermanent loss, high slippage, and other technology security risks, which we'll cover in *Chapter 19*, *Blockchain Security*.

CEX vs DEX

A comparison of CEX and DEX is shown below and highlights the key differences between the two paradigms.

Attribute/exchange type	**CEX**	**DEX**
Asset custody	Third party	User
Entry barrier	High	Very low

Regulation	High	None, or very low
Infrastructure availability	High (but could be compromised due to centralization)	Very high (due to blockchain security guarantees)
User experience	Easy to use; supported	Could be difficult to use for new users; less support
Liquidity	Much deeper and professionally provided	Not very deep
Impermanent loss	Not applicable as highly liquid	Applicable and highly likely in case of fluctuations
Fees	Higher due to intermediaries	Much lower, as no intermediaries exist
Identity verification	Required	Not needed

As DeFi is a thriving ecosystem with massive profiting opportunities, a question arises: is it possible to automatically find the most profitable trading strategy and create DeFi trades accordingly to maximize profits? There are two methods employed to do so – the Bellman-Ford algorithm and SMT solvers.

Flash loans

A flash loan is a service that only exists in the DeFi world. In CeFi, a loan is an instrument that refers to an agreement between a lender and a borrower, in which the borrower receives a certain amount of funds (the loan) from the lender, with the obligation to repay the loan amount plus interest over a specified period. Loan instruments typically involve the payment of interest and principal on a regular basis and are secured or unsecured. Secured loans are backed by some type of collateral, such as a property or a vehicle, while unsecured loans are not backed by any collateral and are usually offered at a higher interest rate. The lender bears the risk of default and is compensated by the interest amount charged over the loan period. If the loan is for a long duration, then the interest rate is also higher because there is a greater exposure to risk for the lender that the borrower may default due to a longer time period. This means that if the lending is for a shorter period, then the risk is less and therefore requires less compensation for the lender.

A flash loan enables such short terms loans, which are instantaneous and are paid back in the same transaction. In other words, repayment of the loan occurs in the same transaction in which the lending occurred. Flash loans are atomic in nature due to the underlying blockchain, meaning that if the loan (principal) is not repaid with the required interest within the same transaction, the whole process rolls back to the previous state as if no money ever left the lender's account. This means that there is no counterparty or duration risk in flash loans, which is totally different from loans in the traditional finance world.

This safety condition is enforced via blockchain-enabled atomic transactions. The operations within a transaction are executed completely in sequential order or fail altogether. A blockchain transaction can fail due to three reasons. Transaction fees are not sufficient, the transaction (function call) fails to meet the conditions in the smart contract, or the transaction is illicit (e.g., attempting to double spend). In all these cases, the state will be reverted automatically to the previous state as if no transaction was ever executed. This inherent safety mechanism enables flash loans.

If a user borrows a million dollars worth of tokens and fails to meet the loan conditions set forth in the smart contract (e.g., pay it back by the end of the transaction with interest), the transaction will be rolled back due to the inherent safety of the blockchain transaction execution mechanism. Imagine within a single transaction, the loan is issued from the liquidity pool, the loan is used, and the principal amount with interest is paid back to the liquidity pool. Imagine if the last condition is not met, i.e., the loan and interest are not paid back; the entire flash loan transaction fails and no state change occurs, as if no loan was ever issued. This safety guarantees the lender that the loan will always be paid back, otherwise, no loan will ever be issued. Secondly, there is no limit on the amount the borrower can request given that enough funds are available in the pool. This is very empowering, and no such construct exists (or can exist) in traditional finance.

Flash loans are uncollateralized, which means that they allow a user to take advantage of arbitrage opportunities (price difference) or to refinance without having to pledge any asset as collateral. Flash loans are accessible to anyone; all that's required is a wallet and the required amount of funds, regardless of their credit-worthiness or financial status. This again is in stark contrast to loan instruments in traditional finance and allows anyone to access prospects that were not possible before in the CeFi world. Flash loans offer risk-free arbitrage. In short, there is no analog of flash loans that exists in traditional finance.

Some examples of flash loan providers include Equalizer (`https://equalizer.finance`) and Aave (`https://aave.com`).

Derivatives

Derivatives in the DeFi world serve the same purpose as derivatives in traditional finance, i.e., hedge price risk and interact with a specific asset without buying it. The value of DeFi derivatives is usually derived from cryptocurrency markets, but they can also be linked to other traditional assets like commodities and fiat currencies. Protocols for DeFi derivatives allow users to create synthetic assets that are tied to some underlying real-world assets. Some examples of DeFi derivative protocols are Synthetix (`https://synthetix.io`), Hegic (`https://www.hegic.co`), Opyn (`https://www.opyn.co`), and many others. These protocols enable investors to trade derivatives that are linked to various underlying assets such as commodities, cryptocurrencies, and indexes.

There are two types of decentralized derivatives: asset-based derivatives and event-based derivatives. Asset-based derivatives are offered by services like Synthetix and Mirror, whereas event-based derivatives are offered by services such as Augur.

Asset-based derivatives have their value tied to the value of an underlying asset, which can be a major cryptocurrency like Bitcoin, Ethereum, or any other financial asset. While the underlying asset in asset-backed derivatives is always financial, in the case of event-based derivatives, the underlying asset is some event, i.e., an observable variable. In these markets, individuals place bets on the outcome of events such as games and elections, among others.

Money streaming

Decentralized payments are already a norm in the DeFi world; however, what if we can make them even cheaper, faster, programmable, conditional, time-based, real-time, and more flexible?

The concept of streaming payments has emerged in DeFi, which means that instead of waiting for a traditional fixed period (e.g., weekly or monthly) for payment, the payers can "stream" payments in real time, in small increments, just like we stream videos online. Funds can be streamed in agreed-upon intervals between involved parties. One such platform is Sablier, which allows streaming tokens at regular intervals. One of the key advantages of streaming payments is that payees can verify that they are being paid now, instead of waiting and trusting the payer for a specific date and time to be paid in the future. Moreover, by earning a salary in real time, the concept of paydays is eliminated, which leads to a significant reduction in the need for payday loans. This method allows individuals to verify that they are being paid immediately and regularly at the time intervals that are suitable for them, rather than relying on the promise of future payment. Companies can also avoid wasting a considerable amount of money on accounting, invoicing, and timestamping because the money-streaming process can be more efficient and eliminate traditional accounting needs. Another idea could be stream billing, where the bills are paid in more frequent, shorter real-time intervals, which can alleviate the problem of paying the bills every month in one go. Similar terms such as real-time finance, token streaming, and social money have emerged in literature, which fundamentally mean the same thing – the ability to make payments in real time and flexibly.

Yield farming

Yield farming is a popular method for generating passive income from traders. These yield farming protocols are programmed using smart contracts. Smart contracts secure users'/traders' tokens and offer interest on the locked assets. If these locked-in tokens (funds) are used to provide liquidity, traders can earn interest based on transaction fees. Liquidity is usually required for DEXs to facilitate trading. As these locked tokens can be used in DEXs to execute buy and sell orders, yield farmers can earn income through transaction fees. Moreover, if the locked tokens are used for loans, the investors can receive loan interest.

The interest paid to traders can balance out the potential risks of locking their tokens, such as *impermanent loss*, *token volatility* and the risk of *rug pulls*. The annual percentage rate can either be decided by the creator of the pool or automatically determined by the yield farming protocol, the logic of which is coded within the smart contracts that implement the protocol on the blockchain.

Insurance

We discussed some insurance use cases before in this chapter, in the context of blockchain use cases in finance. Those use cases are concerned with providing and improving insurance products that exist in the traditional financial world. For example, a use case where weather monitored by an IoT sensor on agricultural land automatically triggers an insurance payment through a smart contract if it detects conditions that are likely to adversely affect the yield of the crop. Perhaps another scenario could be to monitor the weather using IoT devices, and in case of hurricanes, house repairs are automatically triggered and paid off by the smart contract. So, this is the type that we can call the replacement and/or improvement of traditional insurance products using blockchain. However, there is another branch of insurance products that is looking to mitigate the risks associated with DeFi activity. For example, protection against risks posed by malfunctioning DApps, smart contract bugs, exchange hacks, or coin price crashes. If a significant amount is held by a user in a cryptocurrency exchange, it could be beneficial to cover the assets against the risk of cryptocurrency exchange hacks or even bankruptcy.

There are certain benefits of DeFi insurance, including increased speed of claim processing and action, automated payment of claims, reduction of false claims, quicker onboarding of customers, and automated quick risk analysis using smart contracts to algorithmically determine and set policy parameters.

There are many DeFi projects that offer decentralized insurance including but not limited to Opium Finance insurance (`https://opium.finance`) and Nexus Mutual (`https://www.nexusmutual.io`).

Decentralized lending – lending and borrowing

DeFi lending and borrowing are quite like the lending services that exist in traditional finance except that they are managed and offered by decentralized applications (DApps) running on the blockchain.

The loan mechanism consists of five main actors:

1. **A vault**: a smart contract that encapsulates the logic to manage lending and borrowing functions, relevant actors, and financial assets (i.e., tokens).
2. **Lender**: this is the entity that lends the funds by depositing the principal to the vault in the hope of redeeming the principal and earning interest as profit.
3. **Borrower**: this is the entity that offers collateral (security deposit) and borrows the assets:
    a. Collateral is a crypto asset that a borrower deposits as security to secure the debt. The collateral guarantees that the borrower can pay back the loan.
    b. There are two methods that exist in DeFi protocols for collateralization:
        i. Over-collateralization means that the value of the collateral provided by the borrower is more than the loan value. Once the loan is issued, the borrower can use the loan freely due to the security provided by the over-collateralization.
        ii. Under-collateralization means that the value of the offered collateral is less than the loan value. Once the loan is issued, the borrower is still not free to use the funds as they please, because the risk is higher due to under-collateralization; therefore, the smart contract keeps control of all the assets and releases funds cautiously.

4. **Liquidator**: this is the entity that proposes liquidation to the vault in case the borrower's collateral falls below a certain value, and it earns an incentive to do so. Recall that in traditional finance, liquidation occurs when an entity sells off some of its assets at a reduced value to settle a debt. In DeFi loans, liquidation means the same thing, where individuals borrow from a DeFi lending protocol and secure it by offering crypto assets as collateral. The liquidation occurs in the DeFi loan protocol when the vault (lending protocol smart contract) automatically sells deposited collateral to pay off the debt. Liquidators are incentivized to buy the discounted collateral and cover the debt. Essentially, the liquidator repays the debt and gets the collateralized asset at a discounted rate. This discount is called the *liquidation spread*. Liquidators can also compete in an auction mechanism facilitated by a smart contract to win the bid to liquidate. There is also a limit sometimes imposed on the maximum fraction of the loan that can be liquidated in a single liquidation transaction. This limit is called the *close factor*. Usually, this is set to 0.5, meaning that liquidators can pay off up to half of a borrower's loan in one go.

   The safety of the deposited assets against the borrowed assets and their underlying value is represented by a numeric metric called the "health factor."

   The **Health Factor** (**HF**) is represented by a simple formula:

$$HF = \frac{\sum(Collateral_i\ value) \times Liquidation\ threshold_i}{Total\ value\ of\ borrowing}$$

   Where the liquidation threshold is a value between 0 and 1 multiplied by the sum of the collateral value and divided by the total value of the assets borrowed. Essentially, the HF serves as an indicator of the stability of the funds in relation to the potential risk of liquidation. The HF increases or decreases in relation to value fluctuations of the assets. If it increases, it improves the borrow position and makes liquidation less likely. If the value of the collateral decreases, the HF also reduces, increasing the liquidation risk. In practice, if the HF value falls below 1, the debt position becomes likely to be liquidated.

   If the collateral value declines so low that paying back the principal amount becomes more expensive, then it means that the borrower has defaulted. Liquidation serves as a security mechanism that triggers under the rules defined by the smart contract to sell the deposited collateral from the borrower to limit the losses faced by the lender.

5. **Price oracle**: this entity feeds prices (market data) into the vault to ensure that (among other things) the price of collateral is up to date so that if it falls below a threshold, the liquidation can be triggered, which involves the liquidator voting for liquidation of the borrower.

We can visualize this high-level architecture of the DeFi lending mechanism in *Figure 21.7*:

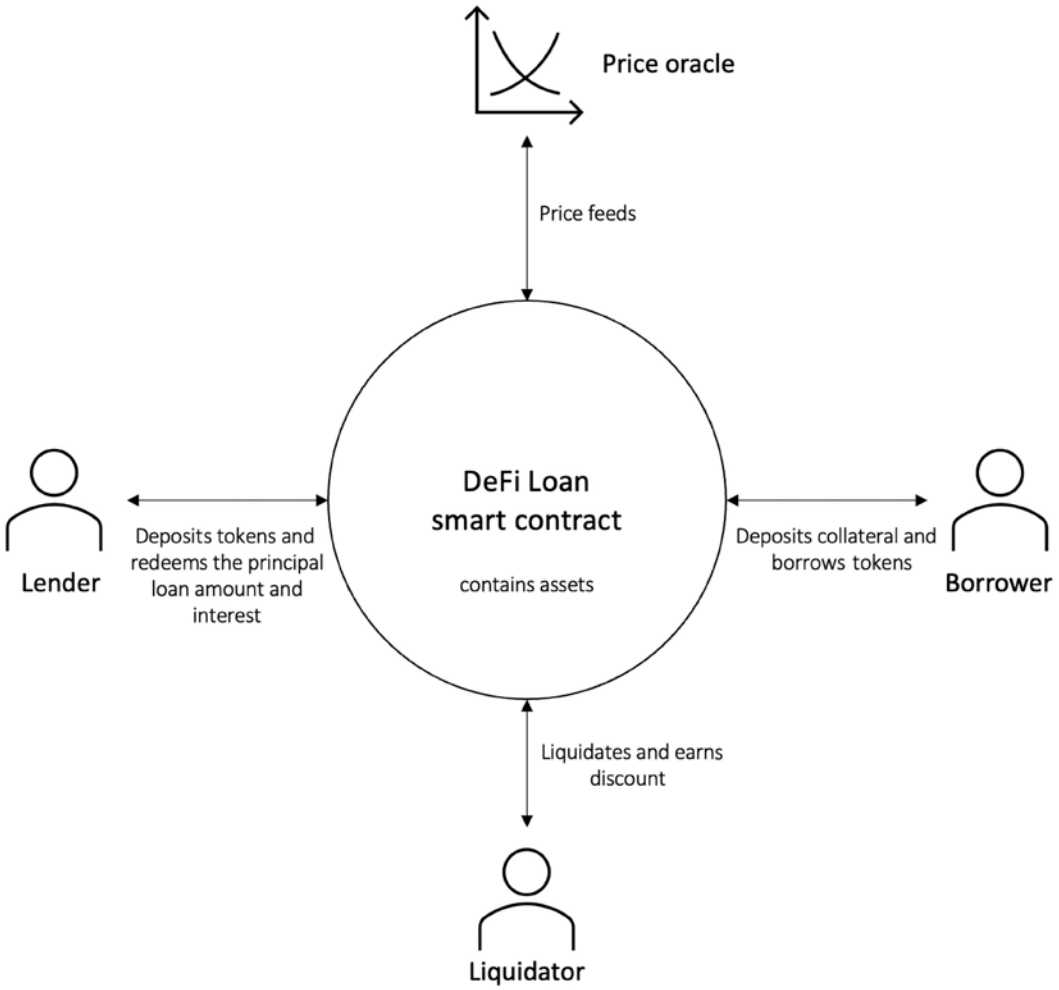


Figure 21.7: DeFi and lending/borrowing

There are many DeFi projects that offer lending and borrowing services including Aave (`https://app.aave.com`), Euler (`https://www.euler.finance`), and Compound (`https://compound.finance`). There are many others, and an internet search can reveal many results.

Non-Fungible Tokens (**NFTs**) discussed in *Chapter 15*, *Tokenization*, have several applications in DeFi, where DeFi applications leverage the properties of NFTs to create new financial instruments and services. Some of the most common applications of NFTs in DeFi include:

- **Collateralized loans**: In DeFi lending platforms, users can borrow funds by depositing collateral. In traditional lending, the collateral is typically a physical asset such as real estate or stocks. In DeFi, however, users can use NFTs as collateral for loans. This allows users to leverage their digital assets to access funds without selling them.

- **Tokenized assets**: NFTs can be used to represent unique and illiquid assets, such as real estate, artwork, or collectibles. These assets can be tokenized into NFTs, which can then be traded on DeFi platforms, providing liquidity to previously illiquid assets.
- **Fractional ownership**: NFTs can be divided into smaller units, allowing for fractional ownership. This means that several investors can own a portion of an NFT, which can be beneficial for high-value assets such as real estate or art. Fractional ownership allows investors to share the risk and cost of acquiring and maintaining the asset.
- **Yield farming**: Yield farming is a DeFi practice where users provide liquidity to a pool of funds and receive rewards in return. NFTs can be used to incentivize liquidity provision by providing exclusive rewards to users who stake specific NFTs.
- **Gamification**: NFTs can also be used to gamify DeFi applications, incentivizing users to participate and compete for rewards. For example, users can earn NFTs as rewards for completing certain tasks or achieving specific milestones.

These are just a few examples of how NFTs are being used in DeFi. As the DeFi space continues to grow, we can expect to see new and innovative applications of NFTs in financial services.

With all these services discussed so far, we can see that DeFi has many benefits. We summarize them below.

Benefits of DeFi

Key benefits of DeFi include:

- **Reduction in risk**: DeFi offers atomic settlement and transparency, which results in the reduction of risk.
- **Less centralized or fully decentralized**: DeFi operates without traditional intermediaries (middlemen), which results in low-cost and frictionless financial services.
- **Open and inclusive**: DeFi has no traditional barriers that we see in traditional finance with capital requirements, cumbersome onboarding, and identification requirements.
- **Transparent**: As the entire ecosystem is open and auditable and publicly verifiable, it enables trust among consumers.
- **Interoperable**: The DeFi ecosystem not only exists on individual chains such as Ethereum and Solana but also is a multiprotocol ecosystem with cross-chain bridges, layer 2 systems, and sidechains. This very property aggregates and improves liquidity across networks. Moreover, in some cases, there is also connectivity with the traditional financial services industry, e.g., to get market data and interoperate with existing legacy and traditional financial infrastructure, which gives rise to an even more efficient and rich ecosystem.
- **Self-custodial**: Usually in DeFi platforms, owners hold digital assets as they hold their private keys.
- **Low cost**: DeFi enables value movement and settlement without intermediaries, thus reducing costs.
- **Programmability**: As DeFi is fundamentally based on smart contracts, it allows automation and programmability of workflows.

- **More efficient**: DeFi is more efficient, due to faster executions, flexibility, and programmability.
- **Better financial inclusion**: Anyone with an internet connection, a mobile device, and entry-level hardware should be able to use the system.
- **Censorship resistance**: DeFi is censorship-resistant due to the underlying blockchain technology.

Of course, this doesn't mean that CeFi is totally going to be replaced with DeFi and CeFi will cease to exist altogether. The future of finance or DeFi is to coexist with CeFi, where the traditional financial system connects with DeFi, and vice versa. For example, a pricing data feed coming from traditional market data to the DeFi protocol via oracles. Stablecoins in the DeFi world are based on/pegged to the traditional financial world's assets and fiat currency. Reserves of stablecoins could be kept in the CeFi traditional financial world. Only time will tell what the eventual shape of the financial service industry will be in a few years; however, a collaborative future where connectivity between CeFi and DeFi exists is likely and is already the case in many use cases.

Like any technology, DeFi also suffers from its own challenges, which are mostly based on blockchain limitations. Some of the challenges include regulation, interoperability, scalability, and privacy. We'll cover these challenges and relevant solutions in detail later in this book.

In the next section, we'll see how the Uniswap DEX works and will use a token pair to provide liquidity to the Uniswap protocol.

Uniswap

Uniswap is a DEX that operates on EVM chains, mainly on the Ethereum blockchain. It allows users to trade cryptocurrencies without relying on a central authority or middleman. The platform uses an AMM system to determine the price of assets, which allows for instant trading without the need for an order book. Users can also provide liquidity to Uniswap by depositing their cryptocurrency holdings into liquidity pools. In exchange for providing liquidity, users receive a portion of the trading fees. Uniswap is non-custodial, meaning users are in control of their own funds and do not have to trust a third party with their assets. The platform is open source, allowing anyone to contribute to the development of the platform. Uniswap also has a decentralized governance system that allows UNI token holders to vote on proposals and changes to the platform.

Swap the token

Follow the steps described below to swap the token:

1. Browse to the Uniswap website (`https://app.uniswap.org/`) and connect to the Goerli test network through MetaMask. Once connected, you should be able to see the details shown in the screenshot below:

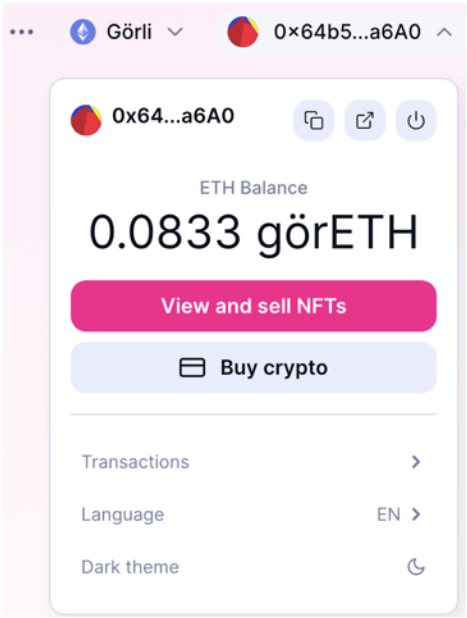


Figure 21.8: Uniswap connected to Goerli

2. Click on **Swap**, select tokens, and confirm the swap, as shown below:

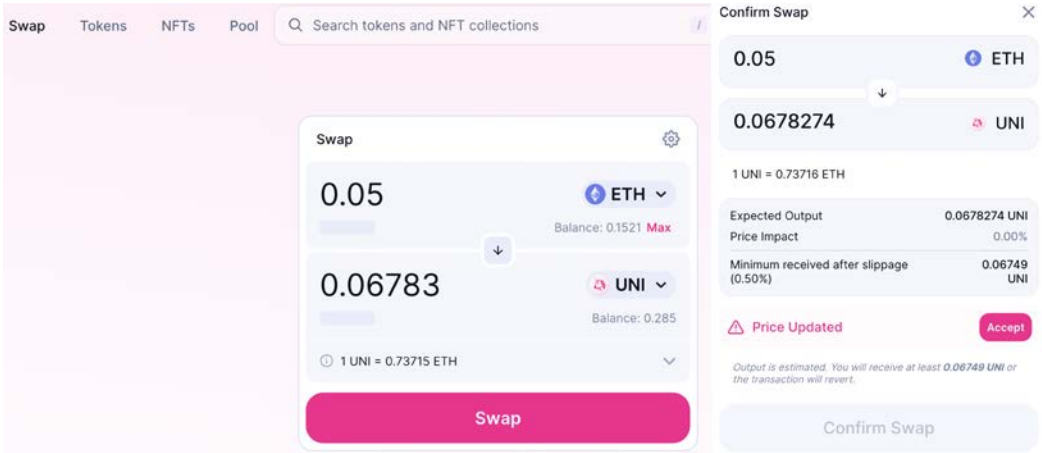


Figure 21.9: Swap tokens

3. MetaMask will open, confirm the transaction there, and wait for it to be processed (mined). It will also ask you to add it to your MetaMask wallet; click **Yes** if you want to and it will be added as an asset to MetaMask.
4. Now we have swapped some ETH for UNI.

Let's now see how we can create a liquidity pool.

Uniswap liquidity pool

Follow the steps described below to create a new liquidity pool in Uniswap:

1. Click on **Pool**, then **More**, and select the option **Create a pool**:

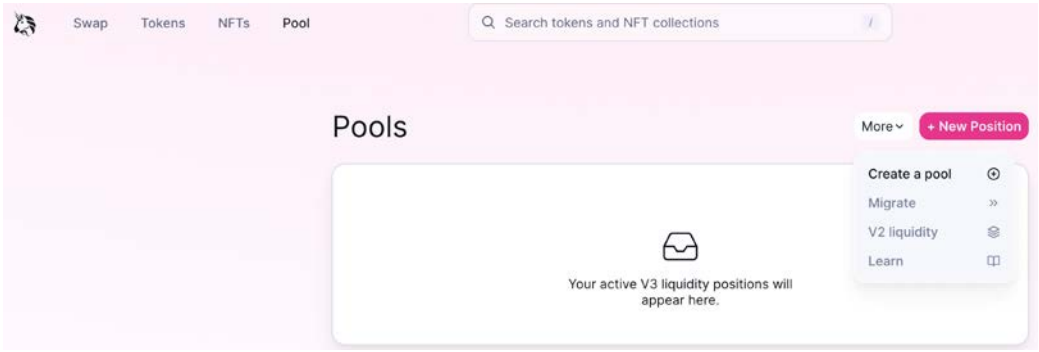


Figure 21.10: Create a liquidity pool

2. It will open the **Add Liquidity** user interface.
3. Under the **Select Pair** option, click on the dropdown and enter the address of the MET token contract. Import the token, and ignore any warnings suggesting that the token is not listed.
4. Enter the rest of the details as shown below in the screenshot:
    a. Select **Pair: ETH** and **UNI**
    b. Select fee tier: **0.05**
    c. Enter the price range between the minimum price and maximum price.

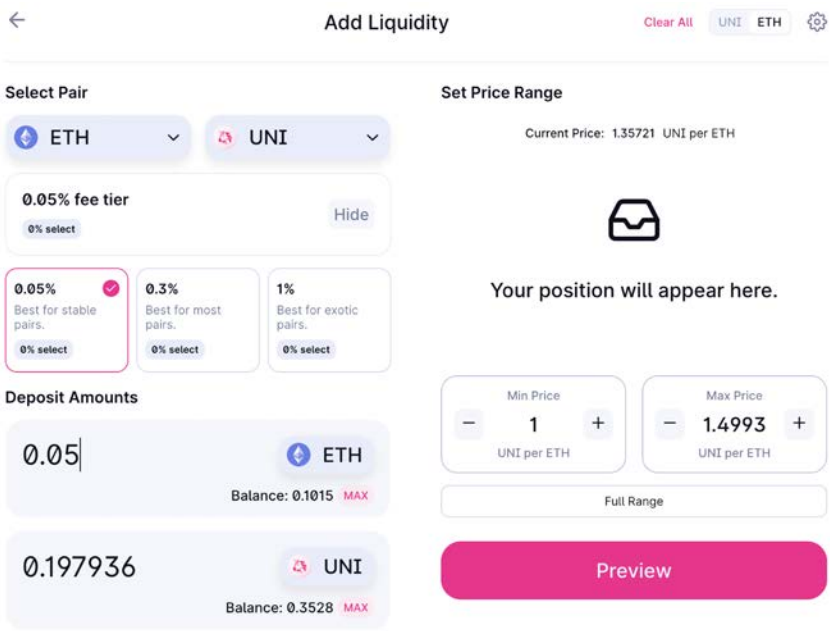


Figure 21.11: Add liquidity parameter

5. Click on the **Preview** button and confirm in MetaMask.
6. In the **Add Liquidity** popup shown below, click **Add:**

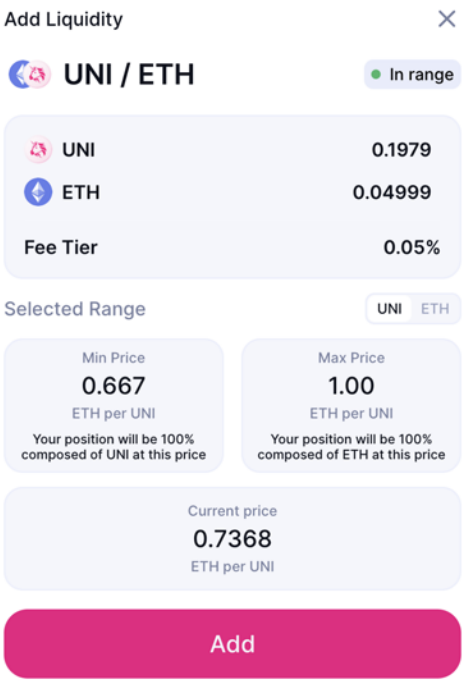


Figure 21.12: Add Liquidity

7. Confirm in MetaMask, and wait for it to be mined.
8. Once mined, it will be added to the pools that you have created. As shown below, notice the UNI/ETH pair at the second-last position in the list:

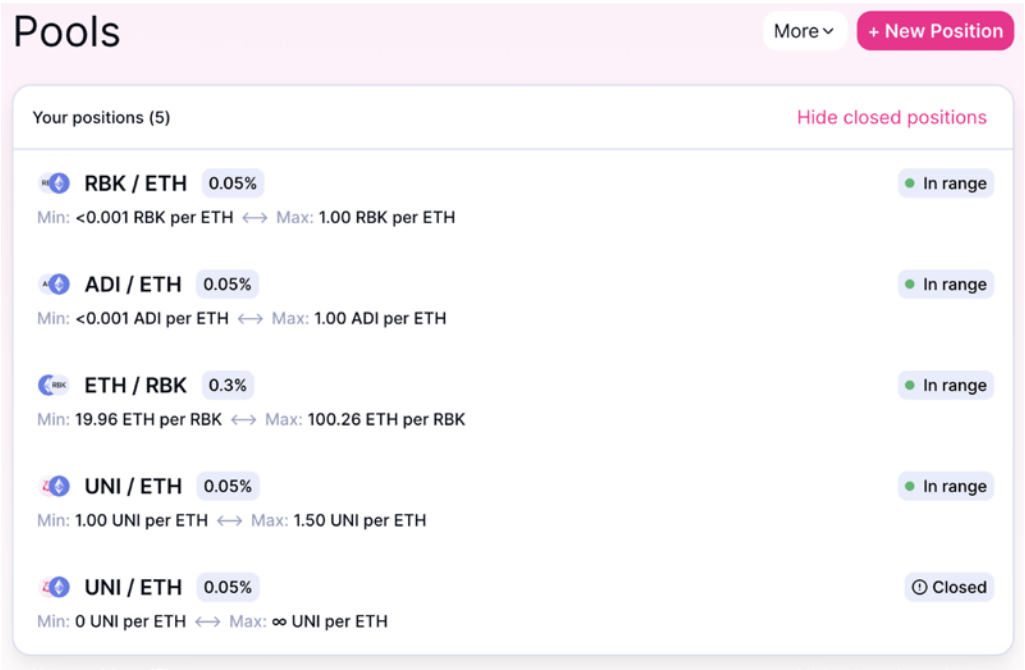


Figure 21.13: Liquidity pools

9. Click on the pool to see more details, such as **Liquidity**, **Unclaimed fees**, and **Price range**:

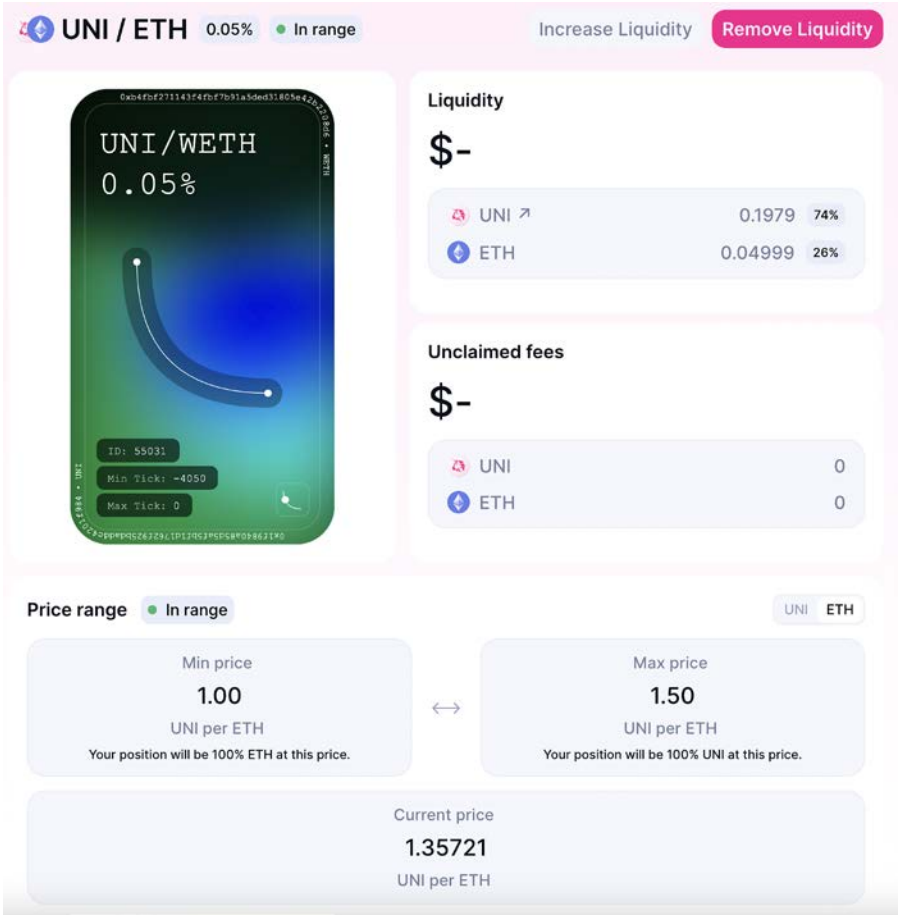


Figure 21.14: Liquidity pool

10. You can also add more liquidity, i.e., tokens, by clicking the option **Increase Liquidity**.
11. You can also remove liquidity and close your position, and claim fees if any are earned, by clicking on **Remove Liquidity**.

In this example, we used already available tokens on the Uniswap platform, swapped ETH that we had on the Goerli network with UNI, and then created a pool using the pair. However, as the Uniswap platform is a DEX, any ERC-20 by any user can be used to create a liquidity pool.

As an exercise, can you figure out how to do this? Recall that we created the MET ERC-20 token in *Chapter 15*, *Tokenization*. Either use that same contract token or create a new one and deploy it on the Goerli test network. We already learned how to do this in *Chapter 15*, *Tokenization*. Better yet, you can also use the OpenZeppelin library to create ERC-20 tokens quite easily.

Follow the example here: `https://docs.openzeppelin.com/contracts/4.x/erc20`. Once deployed, you can explore and confirm the deployment of the contract at Etherscan too.

Once the token contract that you created is deployed successfully, you can optionally import the token into MetaMask. After this, you can create your own liquidity pool with your own new ERC-20 token on the Uniswap platform by following the steps described earlier in this example.

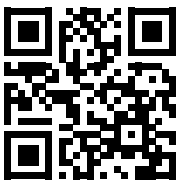
With this, we complete our introduction to DeFi.

Summary

It is clear that DeFi is rapidly evolving and will continue to make a difference in our lives by providing financial services that were simply not possible before the advent of DeFi. Currently, we are witnessing the historical transformation from the internet of information into the internet of value, and as such, blockchain and DeFi will continue to make this possible. Moreover, concepts such as institutional DeFi, new types of tokens, and innovative DeFi protocols and services are going to make financial services more accessible, inclusive, tremendously efficient, and cost-effective. Use cases for private permissioned blockchains in finance and public blockchains will continue to evolve and are likely to converge at some point in the future to enable even more innovative use cases and efficiency.

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

22

Blockchain Applications and What's Next

Blockchain technology is evolving rapidly, and it will continue to affect the way we conduct our day-to-day business as new developments emerge. It has challenged existing business models and promised great benefits, such as cost-saving, efficiency, and transparency. So far in this book, we've explored the technical underpinnings of blockchain technology, such as cryptography, consensus mechanisms, and distributed systems concepts.

This chapter will explore some use cases and applications of blockchain technology in the **Internet of Things** (**IoT**) and the finance, government, and media sectors, and we will learn how blockchain technology can address challenges in these industries and others. Moreover, we'll learn about the latest developments, emerging trends, issues, and future predictions related to blockchain technology. We'll also present some topics related to open research problems and improvements to blockchain technology. Along the way, we'll cover the following main topics:

- Use cases, including IoT, government, health, and **Artificial Intelligence** (**AI**)
- Some emerging trends
- Some challenges

Use cases

Digital currencies were the first-ever application of blockchain technology, which arguably didn't realize the technology's full potential. Although **Bitcoin**, the first major blockchain conceptualization, was introduced in 2008, it was not until years later that blockchain technology's possible applications beyond cryptocurrencies were realized. In 2010, the discussion started regarding BitDNS, a decentralized naming system for domains on the internet.

Then, **Namecoin** (`https://en.bitcoinwiki.org/wiki/Namecoin`) was started in April 2011 with a different vision to Bitcoin, the sole purpose of which was to provision electronic cash. This can be considered the first example of blockchain usage other than purely as a cryptocurrency.

Next, in 2013, the first **Initial Coin Offering** (**ICO**), **MasterCoin**, emerged, which paved the way for more ICOs. After that, Ethereum's ICO was hugely successful. With the availability of Ethereum in 2015, a general-purpose smart contract platform, more ideas started to emerge around various applications of blockchain technology. Since then, many use cases of blockchain technology in various industries have been proposed. In this chapter, five main industries have been selected due to their popularity and promising blockchain use cases for discussion:

- IoT
- Government
- Health
- Media

Let's begin with one of the most exciting use cases of blockchain technology: IoT.

IoT

IoT has recently gained a lot of traction due to its potential for transforming business applications and everyday life. IoT can be defined as a network of computationally intelligent physical objects (any objects, such as cars, fridges, and industrial sensors) that are capable of connecting to the internet, sensing real-world events or environments, reacting to those events, collecting relevant data, and communicating this over the internet.

This simple definition has enormous implications and has led to exciting concepts, such as wearables (such as health trackers or watches), smart homes, smart grids, smart connected cars, and smart cities, which are all based on this basic concept of an IoT device. One thing to note is that all these components are accessible and controllable over the internet, via the IoT.

After dissecting the definition of IoT, four functions come to light as being performed by an IoT device: **sensing**, **reacting**, **collecting**, and **communicating**. All these functions are performed by using various components on the IoT device. Sensing is performed by **sensors**. Reacting or controlling is performed by **actuators**; collecting is the function of various sensors, and communication is performed by chips, which provide network connectivity.

In the next section, we'll introduce the typical architecture of an IoT-based ecosystem.

IoT architecture

A typical IoT architecture can consist of many physical objects connected to each other, and to a centralized cloud server. This is shown in the following diagram:

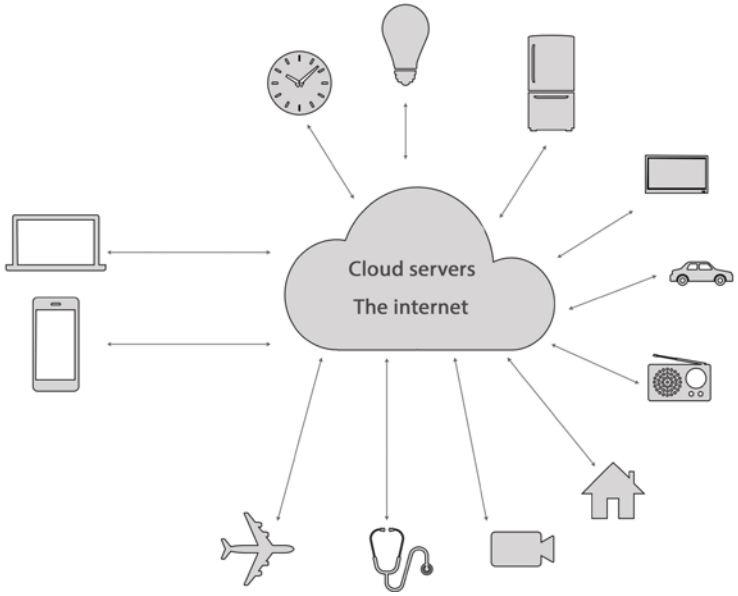


Figure 22.1: A typical IoT network

Elements of the IoT are spread across multiple layers, and various existing reference architectures can be used to develop IoT systems. A five-layer model can be used to describe IoT, which contains a **physical object** layer, a **device** layer, a **network** layer, a **services** layer, and an **application** layer. Each layer or level is responsible for various functions and includes multiple components. These are shown in the following figure:

Application layer Transportation, financial, insurance, and many others
Management layer Data processing, analytics, security management
Network layer LAN, WAN, PAN, routers
Device layer Sensors, actuators, smart devices
Physical objects People, cars, homes

Figure 22.2: IoT layered model

Now, let's examine each layer in detail.

The physical object layer

This layer includes any real-world physical objects. It includes objects like cars, fridges, trains, and homes. In fact, anything that is required to be monitored and controlled can be connected to the IoT.

The device layer

This layer contains things that make up the IoT, such as sensors, transducers, actuators, smartphones, smart devices, and **Radio-Frequency Identification** (**RFID**) tags. There can be many categories of sensors, such as body sensors, home sensors, and environmental sensors, based on the type of work they perform. This layer is the core of the IoT ecosystem where various sensors are used to sense real-world environments. This layer includes sensors that can monitor temperature, humidity, liquid flow, chemicals, air, pressure, and much more. Usually, an **Analog-to-Digital Converter** (**ADC**) is required for a device to turn the real-world analog signal into a digital signal that a microprocessor can understand.

Actuators in this layer provide the means to enable the control of external environments; for example, starting a motor or opening a door. These components also require digital-to-analog converters to convert a digital signal into analog. This method is especially relevant when control of a mechanical component is required by the IoT device.

The network layer

This layer is composed of various network devices that are used to provide internet connectivity between devices and the cloud, or servers that are part of the IoT ecosystem. These devices can include gateways, routers, hubs, and switches. This layer can include two types of communication.

First, there are the horizontal means of communication, which include radio, Bluetooth, Wi-Fi, Ethernet, LANs, Zigbee, and PANs, and can be used to provide communication between IoT devices. This layer can optionally be included in the device layer as it physically resides on the device layer where devices can communicate with each other.

Second, we have communication to the next layer, which is usually done through the internet and provides communication between machines and people or other upper layers.

The management layer

This layer provides the management layer for the IoT ecosystem. This includes platforms that process data gathered from IoT devices and turning it into meaningful insights. Device management, security management, and data flow management are included in this layer. It also manages communication between the device and application layers.

The application layer

This layer includes the applications running on top of the IoT network. This layer can consist of many applications, depending on the requirements, such as transportation, healthcare, finance, insurance, or supply chain management tasks. This list, of course, is not exhaustive by any stretch of the imagination; there is a myriad of IoT applications that can fall into this layer.

With the availability of cheap sensors, hardware, and bandwidth, IoT has gained popularity in recent years and currently has applications in many different areas, including healthcare, insurance, supply chain management, home automation, industrial automation, and infrastructure management. Moreover, advancements in technology such as the availability of IPv6, smaller and more powerful processors, and better internet access have also played a vital role in the popularity of IoT.



IPv6 is the latest version of the internet protocol, which, with a size of 128 bits, offers more address space compared to the 32-bit IPv4.

In the next section, we'll discuss some advantages of IoT and blockchain convergence.

Benefits of IoT and blockchain convergence

The benefits of IoT with blockchain technology range from saving costs to enabling businesses to make vital decisions, and thus improve performance based on the data provided by the IoT devices. Even in domestic usage, IoT-equipped home appliances can provide valuable data for cost savings. For example, smart meters for energy monitoring can provide valuable information on how energy is being used and can convey that back to the service provider. Raw data from millions of IoT devices is analyzed and provides meaningful insights that help in making timely and efficient business decisions.

The usual IoT model is based on a centralized paradigm, where IoT devices usually connect to a cloud infrastructure or central servers to report and process the relevant data. This centralization is somewhat vulnerable to exploitation, including hacking and data theft. Moreover, not having control of personal data on a single, centralized service provider also increases the possibility of security and privacy issues. While there are methods and techniques for building a highly secure IoT ecosystem based on the normal IoT model, there are much more specific and desirable benefits that a blockchain-based model can bring to the IoT.

Blockchain technology for IoT can help to build trust, reduce costs, and accelerate transactions. Additionally, decentralization, which is at the very core of blockchain technology, can eliminate single points of failure in an IoT network. For example, a central server may be unable to cope with the amount of data that billions of IoT devices (things) produce at high frequency, whereas a decentralized blockchain serving as a communication layer between these IoT devices enables all the IoT devices in the blockchain to maintain a copy of the data store of their own, which means that this architecture is inherently highly available and resilient.

Some IoT devices losing their database does not result in the entire network coming to a standstill, as might be the case with a centralized server, even with a disaster recovery solution. Also, the **Peer-to-Peer** (**P2P**) communication model provided by blockchain technology can help reduce costs because there is no need to build high-cost centralized data centers or implement complex public key infrastructure for security. Devices can communicate with each other directly or via routers.

As an estimate made by various researchers and companies, by 2025, there will be roughly 41 billion devices connected to the internet. With this explosion of billions of devices connecting to the internet, it is hard to imagine that centralized infrastructures will be able to cope with the high demands of bandwidth, services, and availability without incurring excessive expenditure. A blockchain-based IoT will be able to solve scalability, availability, privacy, and reliability issues in the current IoT model.

Blockchain technology enables things to communicate and transact with each other directly, and with the availability of smart contracts, negotiation and financial transactions can also occur directly between devices instead of requiring an intermediary, an authority, or human intervention. For example, if a room in a hotel is vacant, it can rent itself out, negotiate the rent, and can open the door lock for a human who has paid the required fee. Another example could be that if a washing machine runs out of detergent, it could order it online after finding the best price and value based on the logic programmed into its smart contract.

The aforementioned five-layer IoT model can be adapted to a blockchain-based model by adding a blockchain layer on top of the network layer. This layer will run smart contracts and provide security, privacy, integrity, autonomy, scalability, and decentralization services to the IoT ecosystem. The management layer, in this case, will just consist of software related to analytics and processing, and security and control can be moved to the blockchain layer.

This new model with six layers is visualized in the following table:

Application layer Transportation, finance, insurance, and many other domains
Management layer Data processing, analytics, security management
Blockchain layer Security, consensus, P2P (M2M) autonomous transactions, decentralization, smart contracts
Network layer LAN, WAN, PAN, routers
Device layer Sensors, actuators, smart devices
Physical objects People, cars, homes

Figure 22.3: Layered blockchain-based IoT model

In this model, the other layers are expected to remain the same, but an additional blockchain layer will be introduced as middleware between all participants of the IoT network.

It can also be visualized as a P2P IoT network after abstracting away all the layers mentioned earlier. This model is shown in the following diagram, where all the devices are communicating and negotiating with each other without a central command and control entity:

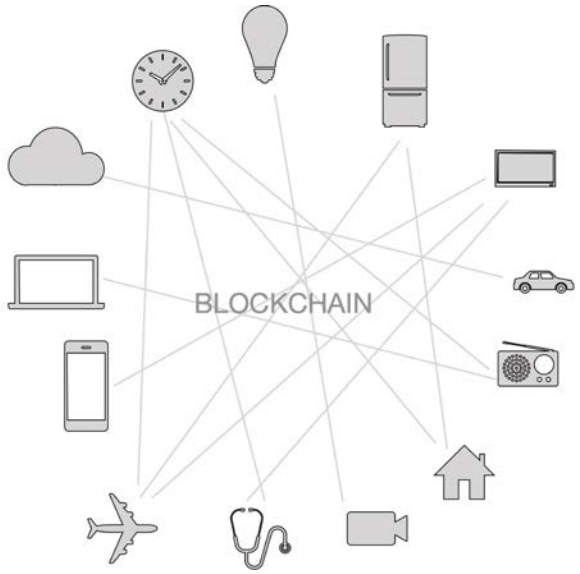


Figure 22.4: Blockchain-based direct (P2P or machine-to-machine (M2M)) communication model

An IoT-based blockchain can also result in cost savings due to the easier device management offered by a blockchain-based, decentralized approach. The IoT network can also be optimized for performance by using blockchain technology. In this case, there will be no need to store IoT data centrally for millions of devices. This is because storage and processing requirements can be distributed to all IoT devices on the blockchain. This can result in completely removing the need for large data centers for processing and storing IoT data.

A blockchain-based IoT can also thwart **denial-of-service** (**DoS**) attacks: hackers can target a centralized server or data center more efficiently, but with blockchain's distributed and decentralized nature, such attacks are no longer possible. Additionally, if, as estimated, there will soon be billions of devices connected to the internet, it will become almost impossible to manage security and updates for all those devices from traditional, centrally owned servers. Blockchain technology can provide a solution to this problem by allowing devices to communicate with each other directly in a secure manner, and even request firmware and security updates from each other. On a blockchain network, these communications can be recorded immutably and securely, which will provide auditability, integrity, and transparency to the system. This mechanism is not possible in traditional P2P systems.

In summary, there are clear benefits that can be reaped due to the convergence of IoT and blockchain, and a lot of research and work in academia and across industries are already in progress. Practical use cases and platforms have emerged in the form of **Platform as a Service** (**PaaS**) for blockchain-based IoT, such as the IBM Watson IoT blockchain. There are various projects that have already been proposed that provide blockchain-based IoT solutions, such as IBM Blue Horizon and IBM Bluemix, two examples of IoT platforms that support blockchain IoT. Various start-ups, such as **Filament**, have already proposed novel ideas about how to build a decentralized network that enables devices on the IoT to transact with each other directly and autonomously, driven by smart contracts. In the next section, we discuss the implementation of a blockchain-based IoT system.

Implementing blockchain-based IoT in practice

In the following section, a practical example is provided of how to build a simple IoT device and connect it to the Ethereum blockchain. The IoT device in this example will be connected to the Ethereum blockchain and used to open a door (in this case, the door lock is represented by a **Light Emitting Diode** (**LED**) when the appropriate amount of funds is sent by a user. Note that this is a simple example and requires a more rigorously tested version to be implemented in production.

The prerequisite hardware components are listed as follows:

- A **Raspberry Pi** device, which is a **Single Board Computer** (**SBC**). The Raspberry Pi is an SBC developed as a low-cost computer to promote computer education, but it has also gained much more popularity as the tool of choice for building IoT platforms. You may be able to use earlier models too, but those have not been tested. You can explore more about Raspberry Pi at `https://www.raspberrypi.com/products/`.
- **LED**: An LED can be used as a visual indication for an event.
- **Resistor**: A 330-ohm component is required, which provides resistance to the passing current based on its rating. It is not necessary to understand the theory behind this for this experiment; any standard electronics engineering text covers all these topics in detail.
- **Breadboard**: This provides a means of building an electronic circuit without requiring soldering.
- **T-shaped cobbler**: This is inserted on the breadboard, as shown in the following figure, and provides a labeled view of all **General-Purpose I/O** (**GPIO**) pins for the Raspberry Pi.
- **Ribbon cable connector**: This is simply used to provide connectivity between the Raspberry Pi and the breadboard via GPIO.

All these components are shown in the following figure:

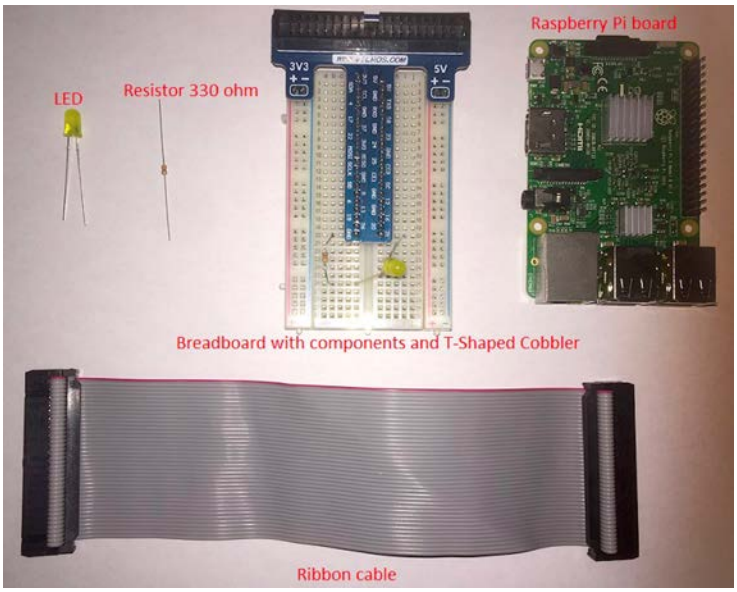


Figure 22.5: Required components

Setting up Raspberry Pi

First, the Raspberry Pi needs to be set up. This can be done by using **New Out Of Box Software** (**NOOBS**), which provides an easy method of installing Raspbian or any other OS.

NOOBS is a user-friendly and easy-to-use OS installation manager for the Raspberry Pi.



NOOBS can be downloaded and installed from `https://www.raspberrypi.org/downloads/noobs/`.

Alternatively, you can just install Raspbian from `https://www.raspberrypi.org/downloads/raspbian/`.

Another alternative, available at `https://github.com/debian-pi/raspbian-ua-netinst`, can also be used to install a minimal non-GUI version of the Raspbian OS.

For this example, NOOBS has been used to install Raspbian. As such, the rest of the exercise assumes Raspbian is installed on the SD memory card of the Raspberry Pi. The command output in the following screenshot shows which architecture the OS is running on.

In this case, it is `armv71`; therefore, the ARM-compatible binary for Geth will be downloaded.

The platform can be confirmed by running the command `uname -a` in a terminal window in the Raspberry Pi Raspbian OS:

```
$ uname -a
Linux raspberrypi 4.4.34-v7+ #930 SMP Wed Nov 23 15:20:41 GMT 2016 armv71 GNU/
Linux
```

Once the Raspbian OS has been installed, the next step is to download the appropriate Geth binary for the Raspberry Pi ARM platform.

The download and installation steps are described in detail as follows:

1. First, download the Geth binary on Raspberry Pi. We use `wget` to download the `geth` client images:

```
$ wget https://gethstore.blob.core.windows.net/builds/geth-linux- arm7-
1.5.6-2a609af5.tar.gz
```



Note that, in this example, a specific version of Geth is being downloaded. Other versions are available, which can be downloaded from `https://geth.ethereum.org/downloads/`. However, it's recommended that you download the version that has been used in the examples in this chapter.

2. Unzip and extract this into a directory. The directory named `geth-linux-arm7-1.5.6-2a609af5` will be created automatically with the `tar` command, as shown here:

```
$ tar -zxvf geth-linux-arm7-1.5.6-2a609af5.tar
```

3. This command will create a directory named `geth-linux-arm7-1.5.6-2a609af5` and will extract the Geth binary and related files into that directory. The Geth binary can be copied into `/usr/bin` or the appropriate path on Raspbian to make it available from anywhere in the OS. When the download is finished, the next step is to create the genesis block.
4. The same genesis block should be used that we created previously in *Chapter 10*, *Ethereum in Practice*. The genesis file can be copied from the other node on the network. This is shown in the following code segment:

```
{
     "nonce": "0x0000000000000042",
     "timestamp": "0x00",
     "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
     "extraData": "0x00",
     "gasLimit": "0x8000000",
     "difficulty": "0x0400",
          "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
     "coinbase": "0x3333333333333333333333333333333333333333",
     "alloc": {
     },
     "config": {
         "chainId": 786,
         "homesteadBlock": 0,
         "eip155Block": 0,
         "eip158Block": 0
     }
}
```



Alternatively, an entirely new genesis block can be generated. Elements of the genesis file were explained in the *Private nets* section of *Chapter 9*, *Ethereum Architecture*, which you can refer to as a refresher.

5. Once the `genesis.json` file has been copied onto the Raspberry Pi, the following command can be run to generate the genesis block. It is important that the same genesis block is used that was generated previously; otherwise, the nodes will effectively be running on separate networks:

```
$ ./geth init genesis.json
```

6. This will create the genesis block and initialize the chain.

7. After genesis block creation, we add peers to the network. This can be achieved by creating a list of nodes on our private network. In order to define the list, we create a file named `static-nodes.json` using any text editor and add the `enode` of the peer that `geth`, on the Raspberry Pi, will connect to for synchronization.
8. The `enode` URL can be obtained from the Geth JavaScript console by running the following command, which should be run on the peer to which Raspberry Pi is going to connect:

```
> admin.nodeInfo
```

9. This will show an output similar to the one shown in the following screenshot:

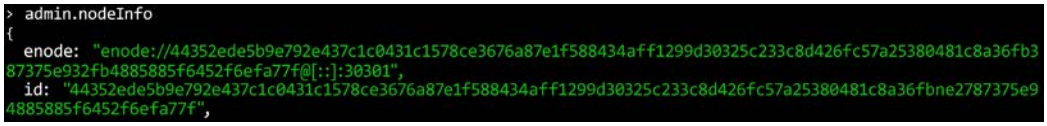


Figure 22.6: Geth nodeInfo

10. The `static-nodes.json` file should contain the `enode` value, as shown in the following screenshot. We can view the contents using the `cat` command:

```
$ cat static-nodes.json
```

11. This command will produce the output shown here:

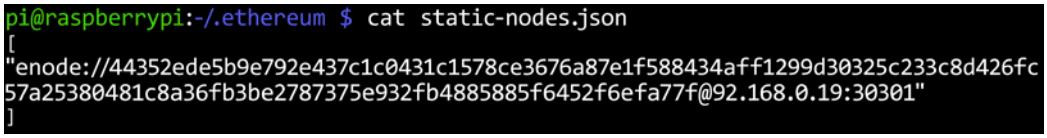


Figure 22.7: Static node configuration

After this step, further instructions presented in the following sections can be followed to connect Raspberry Pi to the other node on the private network. In this example, the Raspberry Pi will be connected to the network with the ID 786 that we created in *Chapter 10*, *Ethereum in Practice*. The key is to use the same genesis file created previously and different port numbers. Using the same genesis file will ensure that clients connect to the same network in which the genesis file originated. Different ports are not a strict requirement; however, if the two nodes are running in a private network and access from an environment external to the network is required, then a combination of a DMZ, a router, and port forwarding will be used. Therefore, it is recommended to use different TCP ports to allow port forwarding to work correctly.

Setting up the first node

First, the `geth` client needs to be started on the first node using the following command. The `--identity` switch allows an identifying name to be specified for the node:

```
$ geth --datadir .ethereum/privatenet/ --networkid 786 --maxpeers 5 --http
--http.api web3,eth,debug,personal,net --http.port 9001 --http.corsdomain "*"
--port 30301 --identity "drequinox"
```

This will start up the `geth` node. Once the node has been started up, it should be kept running, and another `geth` instance should be started from the Raspberry Pi node.

Setting up the Raspberry Pi node

On Raspberry Pi, the following command is required to be run to start `geth` and to sync it with other nodes (in this case, only one node). The following is the command:

```
$ ./geth --networkid 786 --maxpeers 5 --http --http.api
web3,eth,debug,personal,net --http.port 9002 --http.corsdomain "*" --port 30302
--identity "raspberry"
```

This should start up the `geth` node on the Raspberry Pi. Note that when the output log contains a row displaying `Block synchronisation started`, this means that the node has connected successfully to its peer that we started up earlier.

This connectivity can be further verified by running commands in the `geth` console on both nodes, as shown in the following screenshot. The `geth` client can be attached by simply running the following command on the Raspberry Pi:

```
$ geth attach
```

This will open the JavaScript `geth` console for interacting with the `geth` node. We can use the `admin.peers` command to see the connected peers:

```
> admin.peers
[{
    caps: ["eth/62", "eth/63"],
    id: "44352ede5b9e792e437c1c0431c1578ce3676a87e1f588434aff1299d30325c233c8d426fc57a25380481c8a36fb3be2787375e932f
b4885885f6452f6efa77f",
    name: "Geth/drequtnox/v1.5.2-stable-c8695209/ltnux/go1.7.3",
    network: {
      localAddress: "192.168.0.21:56550",
      remoteAddress: "192.168.0.19:30301"
    },
    protocols: {
      eth: {
        difficulty: 117194155397,
        head: "0x2d32c90b4c9dacea9a109b0ae52c1ebf511915bb618a2d3c55a80a63852e89f6",
        version: 63
      }
    }
}]
```

Figure 22.8: Geth console admin peers command running on the Raspberry Pi

Similarly, we can attach to the `geth` instance by running the following command on the first node:

```
$ geth attach ipc:.ethereum/privatenet/geth.ipc
```

Once the console is available, `admin.peers` can be run to reveal the details about other connected nodes, as shown in the following screenshot:



Figure 22.9: Geth console admin peers command running on the other peer

Once both nodes are up and running, further prerequisites can be installed to set up the experiment. The installation of Node.js and the relevant JavaScript libraries is required.

Installing Node.js

The required libraries and dependencies are listed here. First, Node.js and `npm` need to be updated on the Raspberry Pi Raspbian OS. For this, the following steps can be followed:

1. Install Node.js on the Raspberry Pi using the following command:

```
$ curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
```



Note that we are using Node version 7.x for this example, simply for demonstration. You can use a later version if desired.

This should install Node.js.

2. Run the update via `apt-get`:

```
$ sudo apt-get install nodejs
```

3. Verification can be performed by running the following command to ensure that the correct versions of Node.js and `npm` have been installed, as shown in the following screenshot:

```
$ npm -v
4.0.5

$ node -v
v7.4.0
```



It should be noted that these versions are not a necessity; any of the latest versions of `npm` and Node.js should work. However, the examples in this chapter make use of `npm 4.0.5` and `node v7.4.0` simply for demonstration, so it is recommended that you use the same versions to avoid any compatibility issues.

4. Install Ethereum web3 using the following command:

```
$ npm install web3@0.18.0
```

5. This should install Web3. Web3 is required to enable JavaScript code to access the Ethereum blockchain.



Make sure that the specific version of Web3 shown in the snippet, or a version similar to this (for example, 0.20.2), is installed instead of the default, version 1.2.11 (at the time of writing). As this example was originally developed and tested using version 0.18.0, it is recommended that a Web3 0.20.2 or 0.18.0 stable version should be used for this example. The aim of this example, which is to show how Ethereum-based IoT networks can be created, doesn't change regardless of whether a newer or older version of Web3 is used. You can install the recommended version by using:

```
$ npm install web3@0.20.2
```

6. Similarly, `onoff` can be installed, which is required to communicate with the Raspberry Pi and control GPIO:

```
$ npm install onoff --save
```

7. This will install `onoff`.

In this section, we set up a private network with our Raspberry Pi, and another node with `geth`, `nodejs`, web3, and the `onoff` library to run our example on.

When all the prerequisites have been installed, the hardware setup can be performed. For this purpose, a simple circuit will be built using a breadboard and a few electronic components.

Building the electronic circuit

As shown in the following figure, the **positive** leg (long leg) of the **LED** is connected to pin number **21** of the **breadboard**, and the **negative** (short leg) is connected to the **resistor**, which is then connected to the **ground** (**GND**) pin of the **breadboard**. Once the connections have been set up, the **ribbon cable** can be used to connect to the GPIO connector on the Raspberry Pi:

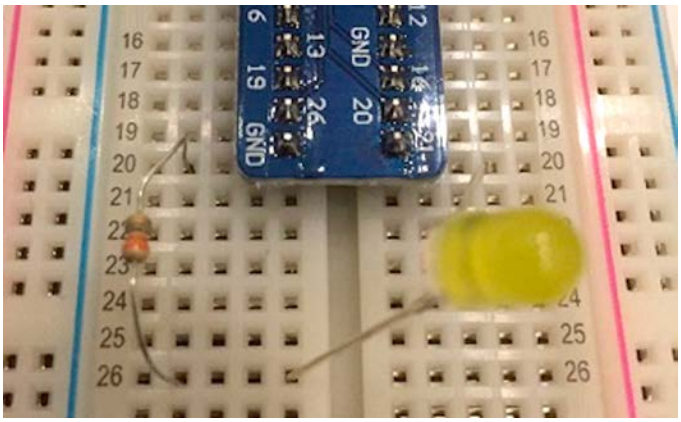


Figure 22.10: Connections for components on the breadboard

Once the connections have been set up correctly and the Raspberry Pi has been updated with the appropriate libraries and Geth, the next step is to develop a simple smart contract that expects a value. If the value provided is not what it expects, it does not trigger an event; otherwise, if the value passed matches the correct value, the event triggers, which can be read by the client JavaScript program running via Node.js.

Developing and running a Solidity contract

Of course, the **Solidity** contract can be very complicated and can also deal with the **ether** sent to it. If the amount of ether is equal to the required amount, then the event can trigger. However, in this example, the aim is to demonstrate the usage of smart contracts to trigger events, which can then be read by JavaScript running on Node.js and can then, in turn, trigger actions on IoT devices using various libraries.

The smart contract source code is shown as follows:

```
pragma solidity ^0.4.0;
contract simpleIOT {
    uint roomrent = 10;
    event roomRented(bool returnValue);
    function getRent (uint8 x) public returns (bool) {
        if (x == roomrent) {
            emit roomRented(true);
            return true;
        }
    }
}
```

The online Solidity compiler (the **Remix IDE**) can be used to run and test this contract. The **Application Binary Interface** (**ABI**) required for interacting with the contract is also available in the Remix IDE's Solidity compiler section.

The following is the ABI of the contract:

```
[
    {
        "constant": false,
        "inputs": [
            {
                "name": "x",
                "type": "uint8"
            }
        ],
        "name": "getRent",
        "outputs": [
            {
                "name": "",
                "type": "bool"
            }
        ],
        "payable": false,
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "anonymous": false,
        "inputs": [
            {
                "indexed": false,
                "name": "returnValue",
                "type": "bool"
            }
        ],
        "name": "roomRented",
        "type": "event"
    }
]
```

There are two methods by which the Raspberry Pi node can connect to the private blockchain via the web3 interface. The first is where the Raspberry Pi device is running its own `geth` client locally and maintains its ledger, as shown in the following diagram:

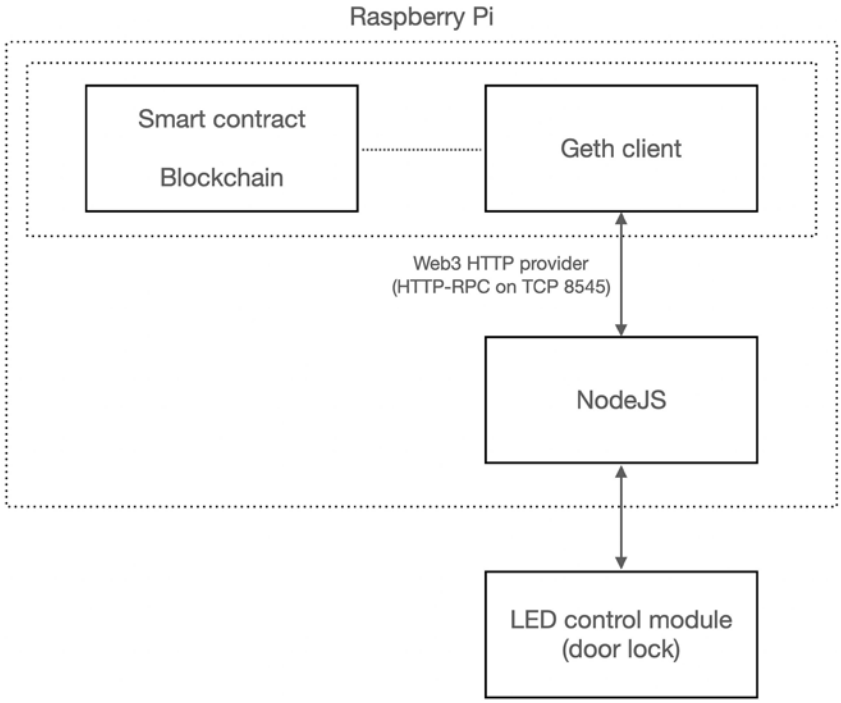


Figure 26.11: Application architecture of the room rental IoT application (an IoT device with a local ledger)

Sometimes, with resource-constrained devices, it is not possible to run a full `geth` node or even a light node. In that case, the second method, which uses a web3 provider (via an HTTP-RPC server), can be used to connect to the appropriate RPC channel. The following diagram shows the high-level architecture of an IoT application where the Raspberry Pi does not run a Geth instance. Instead, it runs only the minimum software required to provide IoT application functionality to the IoT device.

For blockchain-relevant operations, it connects to another blockchain node running on the network via HTTP-RPC:

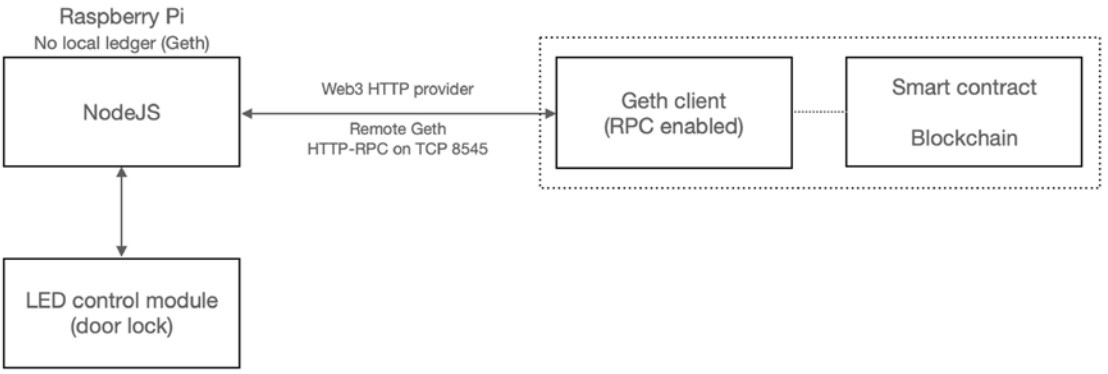


Figure 22.12: Application architecture of the room rental IoT application (an IoT device without a local ledger)

There are obvious security concerns that arise from exposing RPC interfaces publicly; therefore, it is recommended that this option is used only on private networks. If required to be used on public networks, appropriate security measures need to be put in place, such as allowing only the known IP addresses to connect to the `geth` RPC interface. This can be achieved by a combination of disabling peer discovery mechanisms and HTTP-RPC server listening interfaces.

More information about this can be found using `geth help`. Traditional network security measures such as **firewalls**, **Transport Layer Security** (**TLS**), and **certificates** can also be used but have not been discussed in this example for brevity. Now, **Truffle** can be used to deploy the contract on the private network ID `786` to which, at this point, the Raspberry Pi is connected.

Truffle `deploy` can be performed simply by using the following command; it is assumed that `truffle init` and other preliminaries discussed in *Chapter 12*, *Web3 Development Using Ethereum*, have already been performed:

```
$ truffle migrate
```

This should deploy the contract.

Once the contract has been deployed correctly, JavaScript code can be developed that will connect to the blockchain via web3, listen for the events from the smart contract in the blockchain, and turn the LED on via the Raspberry Pi. The JavaScript code required for this example is shown here. Simply write or copy and paste the code shown here into a file named `index.js`:

```
var Web3 = require('web3');
if (typeof web3 !== 'undefined')
{
    web3 = new Web3(web3.currentProvider);
}else
{
```

```
    web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:9002"));
    //http-rpc-port
}
var Gpio = require('onoff').Gpio;
var led = new Gpio(21,'out');
var coinbase = web3.eth.coinbase;
var ABIString =
'[{"constant":false,"inputs":[{"name":"x","type":"uint8"}],"name":"getRent",
"outputs":[{"name":"","type":"bool"}],
"payable":false,"stateMutability":"nonpayable",
"type":"function"},{"anonymous":false,
"inputs":[{"indexed":false,"name":"returnValue",
"type":"bool"}],"name":"roomRented","type":"event"}]';
var ABI = JSON.parse(ABIString);
var ContractAddress = '0x975881c44fbef4573fef33cccec1777a8f76669c';
web3.eth.defaultAccount = web3.eth.accounts[0];
var simpleiot = web3.eth.contract(ABI).at(ContractAddress);
var event = simpleiot.roomRented( {}, function(error, result) { if (!error)
{
    console.log("LED On");
    led.writeSync(1);
}
});
```



Note that, in the preceding code example, the contract address `0x975881c44fbef4573fef33cccec1777a8f76669c` for the `var ContractAddress` variable is specific to the deployment; it will be different when you run this example. Simply change the address in the file to what you see after deploying the contract.

Also, note the HTTP-RPC server listening port on which Geth has been started on Raspberry Pi. By default, it is TCP port `8545`. Remember to change this according to your Raspberry Pi setup and Geth configuration. It is set to `9002` in the preceding example code because Geth running on Raspberry Pi is listening on `9002` in the example. If it's listening on a different port on your Raspberry Pi, then change it to that port:

```
web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:9002"));
```

When Geth starts up, it shows which port it has the HTTP endpoint listening on. This is also configurable with `--http.port` in `geth` by specifying the port number value as a parameter for the flag.

This JavaScript code can be placed in a file on the Raspberry Pi; for example, `index.js`. It can be run by using the following command:

```
$ node index.js
```

This will start the program, which will run on Node.js and listen for events from the smart contract. Once the program is running correctly, the smart contract can be invoked by using the Truffle console. In this case, the `getRent` function is called with the parameter `10`, which is the expected value:

```
[truffle(development)> simpleiot.getRent(10)
'0x71f550949a4c5168af7b9f7f84fada99bccc20a123779642e5e8c0c0127266ee'
```

After the contract has been mined, `roomRented` will be triggered, which will turn on the LED.

In this example, it is a simple LED, but it can be any physical device, such as a room lock, that can be controlled via an actuator. If all works well, the LED will be turned on as a result of the smart contract's function invocation, as shown in the following figure:

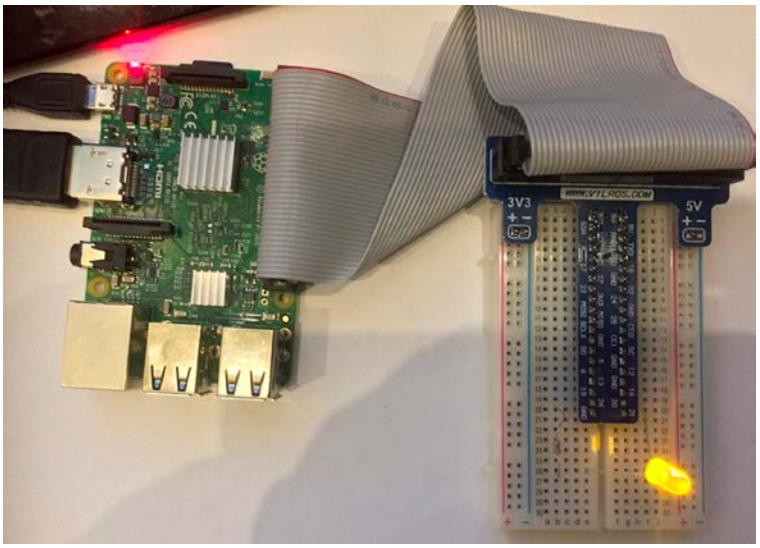


Figure 22.13: Raspberry Pi with LED control

Also, on the node side, it will display an output similar to the one shown here:

```
$ node index.js
LED On
```

As demonstrated in the preceding example, a private network of IoT devices can be built that runs a `geth` client on each of the nodes, listens for events from smart contracts, and triggers an action accordingly. The example shown is simple in design but demonstrates the underlying principles of an Ethereum network that can be built using IoT devices, along with smart contract-driven control of the physical devices.

In the next few sections, other applications of the blockchain will be discussed, such as in government, finance, media, and health.

Government

There are various applications of blockchain technology being researched currently that can support government functions and take the current model of e-government to the next level. First, in this section, some background for e-government will be provided, and then a few use cases, such as e-voting, homeland security (border control), and electronic IDs (citizen ID cards), will be discussed.

E-government, or electronic government, is a paradigm where IT and communication technology are used to deliver public services to citizens. The concept is not new and has been implemented in various countries around the world, but with blockchain technology, a new avenue of exploration has opened up. Many governments are researching the possibility of using blockchain technology for managing and delivering public services, including, but not limited to, identity cards, driving licenses, secure data sharing among various government departments, and contract management. Transparency, auditability, and integrity are attributes of blockchain technology that can go a long way in effectively managing various government functions. One of the government functions is border control, which we cover next.

Border control

Automated border control systems have been in use for decades now to thwart illegal entry into countries and prevent terrorism and human trafficking. Machine-readable travel documents, specifically **biometric passports**, have paved the way for automated border control; however, current systems are limited to a certain extent and blockchain technology can provide solutions. A **Machine-Readable Travel Document** (**MRTD**) standard is defined in document 9303 (`https://www.icao.int/publications/pages/publication.aspx?docnum=9303`) of the **International Civil Aviation Organization** (**ICAO**) and has been implemented by many countries around the world.

Each passport contains various security and identity attributes that can be used to identify the owner of the passport and circumvent attempts at tampering with these passports. These include biometric features such as retina scans, fingerprints, facial recognition, and standard ICAO-specified features, including a **Machine-Readable Zone** (**MRZ**) and other text attributes that are visible on the first page of the passport.

One key issue with current border control systems is data sharing, whereby the systems are controlled by a single entity, and data is not readily shared among law enforcement agencies. This inability to share data makes it challenging to track suspicious travel documents or individuals. Another issue is related to the immediate implementation of blacklisting a travel document; for example, when there is an immediate need to track and control suspicious travel documents. Currently, there is no mechanism available to blacklist or revoke a suspicious passport immediately and broadcast it to border control ports worldwide.

Blockchain technology can provide a solution to this problem by maintaining a blacklist in a smart contract that can be updated as required. Any changes will be immediately visible to all agencies and border control points, thus enabling immediate control over the movement of a suspected travel document.

It could be argued that traditional mechanisms like **Public Key Infrastructures** (**PKIs**) and P2P networks can also be used for this purpose, but they do not provide the benefits that a blockchain can provide. With a blockchain, the whole system can be simplified without the requirement of complex networks and PKI setups, which will also result in cost reduction. Moreover, blockchain-based systems will provide cryptographically guaranteed immutability, which helps with auditing and discourages any fraudulent activity.

The full database of all travel documents may not be stored on the blockchain currently due to inherent storage limitations, but a backend distributed database such as **BigchainDB**, **IPFS**, or **Swarm** can be used for that purpose. In this case, a hash of the travel document with the biometric ID of an individual can be stored in a simple smart contract, and a hash of the document can then be used to refer to the detailed data available on the distributed filesystem, such as IPFS. This way, when a travel document is blacklisted anywhere on the network, that information will be available immediately with the cryptographic guarantee of its authenticity and integrity throughout the distributed ledger. This functionality can also provide adequate support in anti-terrorism activities, thus playing a vital role in the homeland security function of a government.

A simple contract in **Solidity** can have an array defined for storing identities and associated biometric records. This array can be used to store the identifying information about a passport. The identity can be a hash of the MRZ of the passport or travel document concatenated with the biometric record from the RFID chip. A simple Boolean field can be used to identify blacklisted passports. Once this initial check passes, further detailed biometric verification can be performed by traditional systems. Eventually, when a decision is made regarding the entry of the passport holder, that decision can be propagated back to the blockchain, thus enabling all participants on the network to immediately share the outcome of the decision.

A high-level approach to building a blockchain-based border control system can be visualized as shown in the following diagram. In this scenario, the passport is presented for scanning to an RFID and page scanner, which reads the data page and extracts machine-readable information, along with a hash of the biometric data stored in the RFID chip. At this stage, a live photo and retina scan of the passport holder is also taken. This information is then passed on to the blockchain, where a smart contract is responsible for verifying the legitimacy of the travel document by first checking its list of blacklisted passports, and then requesting more data from the backend IPFS database for comparison. Note that the biometric data, such as a photo or retina scan, is not stored on the blockchain; instead, only a reference to this data in the backend (IPFS or BigchainDB) is stored in the blockchain.

If the data from the presented passport matches with what is held in the IPFS as files or in BigchainDB and passes the smart contract logical check, then the border gate can be opened:

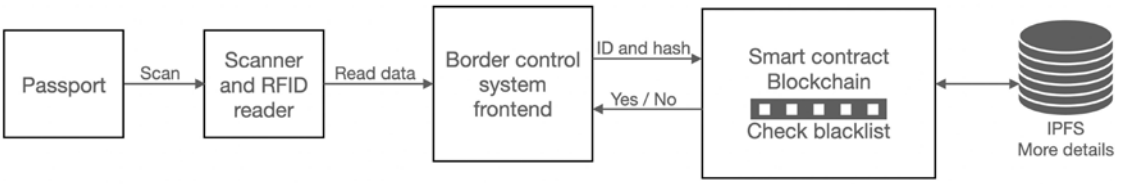


Figure 22.14: Automated border control using a blockchain

After verification, this information is propagated throughout the blockchain and is instantly available to all participants on the border control blockchain. These participants can be a worldwide consortium of homeland security departments of various nations. Another important government function is elections, which we cover next.

Elections

Voting in any government is a key function and allows citizens to participate in the democratic election process. While voting has evolved into a much more mature and secure process, it still has limitations that need to be addressed to achieve the desired level of maturity. Usually, the limitations in current voting systems revolve around fraud, weaknesses in operational processes, and especially transparency. Over the years, secure voting mechanisms (machines) have been built to ensure security and privacy, but they still have vulnerabilities that could be exploited. These vulnerabilities can lead to serious implications for the whole voting process and can result in mistrust of the government by the public.

Blockchain-based voting systems can resolve these issues by introducing end-to-end security and transparency into the process. Security is provided in the form of the guaranteed integrity and authenticity of votes by using public key cryptography, which comes as standard in a blockchain. Moreover, the immutability guaranteed by a blockchain ensures that votes cast once cannot be cast again. This can be achieved through a combination of biometric features and a smart contract maintaining a list of votes already cast. For example, a smart contract can maintain a list of already-cast votes with biometric IDs (for example, a fingerprint) and can use that to detect and prevent double-casting. Secondly, **Zero-Knowledge Proofs** (**ZKPs**) can also be used on the blockchain to protect voters' privacy. With a ZKP, a voter can remain anonymous by hiding their identity, and the vote itself can be kept confidential.



Recently, presidential elections were held in Sierra Leone using blockchain technology, making it the first country to use blockchain technology for elections: `https://www.coindesk.com/sierra-leone-secretly-holds-first-blockchain-powered-presidential-vote/`.

Another example is voting in the city of Zug. A report on this can be found here: `https://www.stadtzug.ch/_docn/1938568/eVoting_Final_Report_ENG.pdf`.

Citizen identification

Electronic IDs or national ID cards are issued by various countries around the world at present. These cards are secure and possess many security features that thwart duplication or tampering attempts. However, with the advent of blockchain technology, several improvements can be made to this process.

Digital identity is not only limited to just government-issued ID cards; it is a concept that applies to online social networks and forums, too. There can be multiple identities being used for different purposes. A blockchain-based online digital identity allows control over personal information sharing. Users can see who has used their data and for what purpose, as well as control access to it. This is not possible with the current infrastructure, which is centrally controlled.

The key benefit is that a single identity issued by the government can be used easily and transparently for multiple services via a single government blockchain. In this case, the blockchain serves as a platform where a government is providing various services, such as pensions, taxation, or benefits, and a single ID is being used to access all these services. The blockchain, in this case, provides a permanent record of every change and transaction made by a digital ID, thus ensuring the integrity and transparency of the system. Also, citizens can notarize birth certificates, marriages, deeds, and many other documents on the blockchain tied with their digital ID as proof of existence. Proof of existence provides digital evidence that something or some data exists or existed in the past at a specific date and time.

Currently, there are successful implementations of identity schemes in various countries that work well, and there is an argument that perhaps the blockchain is not required for identity management systems. Although there are several benefits, such as privacy and the control of the use of identity information, due to the current immaturity of blockchain technology, perhaps it is not ready for use in real-world identity systems. However, research is being carried out by various governments to explore the use of blockchains for identity management. We'll cover more on identity in *Chapter 20*, *Decentralized Identity*.

Moreover, laws such as the right to be forgotten can be quite difficult to incorporate into the blockchain due to its immutable nature.

Other government functions where blockchain technology can be implemented to improve cost and efficiency include the collection of taxes, benefits management and disbursement, land ownership record management, life event registration (marriages, births), motor vehicle registration, and licenses. This is not an exhaustive list, and, over time, many functions and processes of a government could be adapted to a blockchain-based model. The key benefits of blockchain, such as immutability, transparency, and decentralization, can help to bring improvements to most of the traditional government systems.

Now, let's see how the health industry can benefit from blockchain technology.

Health

The health industry has also been identified as another major industry that can benefit from adopting blockchain technology. The blockchain can provide an immutable, auditable, and transparent system that traditional P2P networks cannot. Also, the blockchain provides a simpler, more cost-effective infrastructure compared to traditional complex PKI networks. In healthcare, major issues such as privacy compromises, data breaches, high costs, and fraud can arise from a lack of interoperability, overly complex processes, transparency, auditability, and control. Another burning issue is counterfeit medicine; especially in developing countries, this is a major cause of concern.

With the adaptability of the blockchain in the health sector, several benefits can be realized, including cost savings, increased trust, faster processing of claims, high availability, no operational errors due to complexity in the operational procedures, and the prevention of the distribution of counterfeit medicine.

From another angle, blockchains that provide a digital currency as an incentive for mining can be used to provide processing power to solve scientific problems. This helps to find cures for certain diseases. Examples include **FoldingCoin**, which rewards its miners with FLDC tokens for sharing their computer's processing power for solving scientific problems that require unusually large calculations.



FoldingCoin is available at `http://foldingcoin.net/`.

Another similar project is called **CureCoin,** which is available at `https://www.curecoin.net/`. It is yet to be seen how successful these projects will be in achieving their goals, but the idea is very promising.

Next, we'll discuss media, another use case where blockchain can provide a cheap, fair, and transparent media ecosystem.

Media

Critical issues in the media industry revolve around content distribution, rights management, and royalty payments to artists. For example, digital music can be copied many times without any restrictions, and any attempts to apply copy protection have been hacked in some way or another. There is no control over the distribution of the content that a musician or songwriter produces; it can be copied as many times as needed without any restriction, and consequently has an impact on royalty payments. Also, payments are not always guaranteed and are based on traditional airtime figures. All these issues revolving around copy protection and royalty payments can be resolved by connecting consumers, artists, and all players in the industry, allowing transparency and control over the process.

The blockchain can provide a network where digital music is cryptographically guaranteed to be owned only by the consumers who pay for it. This payment mechanism is controlled by a smart contract instead of a centralized media agency or authority. The payments will be automatically made based on the logic embedded within the smart contract and the number of downloads.



A recent example of such an initiative is Musicoin: `https://musicoin.org`.

Another example of a decentralized NFT marketplace is `https://www.nftchain.tech`.

Moreover, illegally copying digital music files can be stopped altogether because everything is recorded and owned immutably in a transparent manner on the blockchain. A music file, for example, can be stored with owner information and a timestamp that can be traced throughout the blockchain network. Furthermore, consumers who own a legal copy of some content are cryptographically tied to the content they have, and it cannot be moved to another owner unless permitted by the owner. Copyrights and transfers can be managed easily via a blockchain once all digital content is immutably recorded on that blockchain. Smart contracts can then control the distribution and payment to all concerned parties.

Blockchain and AI

AI is a field of computer science that endeavors to build intelligent agents that can make rational decisions based on the scenarios and environment that they observe around them. Machine learning plays a vital role in AI technology by making use of raw data as a learning resource. A key requirement in AI-based systems is the availability of authentic data that can be used for machine learning and model building. Therefore, the explosion of data coming out of IoT devices, smartphones, and other means of data acquisition means that AI and machine learning is becoming more and more powerful. There is, however, a requirement for data authenticity, which is where the convergence with the blockchain comes in. Once consumers, producers, and other entities are on a blockchain, the data that is generated as a result of interaction between these entities can be readily used as input for machine learning engines with a guarantee of authenticity.

The convergence of the blockchain with IoT was discussed earlier in this chapter. Briefly, it can be said that due to the authenticity, integrity, privacy, and shared architecture of blockchain technology, IoT networks would benefit greatly from making use of it. This can be realized in the form of an IoT network that runs on a blockchain and makes use of a decentralized **mesh network** for communication in order to facilitate M2M communication in real time.



A mesh network is a network topology that allows all nodes on a network to connect with one another in a cooperative and dynamic fashion, to facilitate the efficient routing of data.

All of the data that is generated as a result of M2M communication can be used in **machine learning** processes to augment the functionality of artificially intelligent DAOs or simple **Autonomous Agents** (**AAs**). These AAs can act as agents in a blockchain-provided **Distributed Artificial Intelligence** (**DAI**) or distributed machine learning environment, which can learn over time using **machine learning** processes. This would enable them to make better decisions for the good of the blockchain.

Moreover, blockchain technology can play a vital role in federated learning (a sub-field of machine learning) by enhancing security, privacy, trust, and decentralization. In federated learning, data is distributed across multiple devices and used to train a shared model, without having to share raw data. By using a blockchain, the data-sharing process can be made secure and accessible only to authorized parties. Additionally, the blockchain can incentivize data providers to contribute data and computing power to train the model by rewarding them with tokens. The blockchain also ensures the transparency and immutability of the training data and model parameters, making it tamper-proof and auditable by multiple parties. The decentralized management of the model can also be achieved using a blockchain, ensuring that all participants have access to the latest version of the model and are working toward the same goal.

It could also be argued that if an IoT device were hacked, it would send malformed data to the blockchain. This issue would be mitigated using blockchain technology because an IoT device would be part of the blockchain (as a node) and would have the same security properties applied to it as a standard node in the blockchain network.

These properties include the incentivization of good behavior, the rejection of malformed transactions, the strict verification of transactions, and various other checks that are part of blockchain protocol. Therefore, even if an IoT device is hacked, it would be treated as a **Byzantine node** by the blockchain network and would not cause any adverse impact on the network. The possibility of combining intelligent oracles, intelligent smart contracts, and AAs will give rise to **Artificially Intelligent Decentralized Autonomous Organizations** (**AIDAOs**) that can act on behalf of humans to run entire organizations on their own. This is another side of AI that could potentially become normal in the future. However, more research is required to realize this vision.

The convergence of blockchain technology with various other fields, such as 3D printing, virtual reality, augmented reality, spatial computing, and the gaming industry, has also been envisaged. For example, in a multiplayer online game, a blockchain's decentralized approach allows more transparency and can ensure that no central authority is gaining an unfair advantage by manipulating game rules. Each of these topics is currently an active area of research, and more interest and development are expected. Blockchain technology is going to change the world. The revolution has already begun!

There are many applications of blockchain technology and, as discussed in this section, they can be implemented in various industries to bring about solutions to existing problems.

In the next section, we will discuss some trends that are emerging with the evolution of blockchain technology.

Some emerging trends

With the advancement of blockchain technology and the rigorous research being conducted in this space, various developments have been seen in recent years, including application-specific blockchains, new start-ups, and standardization efforts to improve adoption. Some other trends include research on interoperability between chains, scalability, privacy, and security research. Limitations around the **interoperability** of blockchains have also resulted in changes oriented toward the development of systems that can work across multiple blockchains. Some examples of these systems are Interledger, Polkadot, Cosmos, cross-chain atomic swaps, relay networks, and blockchain bridges. This trend of addressing technical challenges is expected to continue to develop in years to come, and even if almost all fundamental challenges are addressed in blockchain technology, further enhancements and optimization research will continue.

Blockchain technology has stimulated intense research interest, both in academia and the commercial sector. In recent years, interest has dramatically increased, and now major institutions and researchers around the world are exploring this technology. This growth in interest is primarily because blockchain technology can help to make businesses and their operations more efficient, cheaper to run, and more transparent. Technologies such as **ZKPs**, fully **homomorphic encryption**, and functional encryption are being researched for their potential use in blockchains. Already, ZKPs have been implemented for the first time at a practical level in the form of Zcash, which has addressed privacy issues in cryptocurrency networks. It's evident that blockchains and cryptocurrencies have helped with the advancement of cryptography, especially financial cryptography.

New fields of research are emerging with blockchains, most notably cryptoeconomics, which is the study of protocols governing the decentralized digital economy. With the continuing development of blockchains and cryptocurrencies, research in this area has also grown.

When it was realized in 2010 that existing methods were no longer efficient for mining bitcoins, miners shifted toward optimizing the available mining hardware. These initial efforts included the use of GPUs, and then **Field-Programmable Gate Arrays** (**FPGAs**) were used after GPUs reached their hash rate limit. Very quickly after that, **Application-Specific Integrated Circuits** (**ASICs**) emerged, which increased mining power significantly. This trend is expected to grow further with research into optimizing ASICs by parallelizing and decreasing their die sizes. Such optimizations will make ASICs even faster and more efficient. Moreover, GPU programming initiatives are also expected to grow, with the regular emergence of new cryptocurrencies, many of which make use of PoW algorithms that can benefit from GPU processing capabilities. For example, recently Zcash has spurred interest in GPU mining rigs and related programming using NVIDIA CUDA and OpenCL. The aim of Zcash is to use multiple GPUs in parallel to optimize mining operations. Also, some research has been done in the field of trusted computing hardware, such as Intel's **Software Guard Extensions** (**SGX**) to address security issues on blockchains. Intel's SGX has also been used in a novel consensus algorithm called **Proof of Elapsed Time** (**PoET**), which was discussed in *Chapter 14*, *Hyperledger*. Hardware research and development are expected to continue, and soon many more hardware applications will be explored. Some examples include **physical unclonable functions**, the acceleration of blockchain processes (such as cryptography), IoT hardware, and hardware security module integration for key management. Moreover, as the prover times in zero-knowledge systems are not ideal, there has been some research on building **ASICs**, FPGAs, and GPUs for improving prover times in systems like SNARKs and so on.

With the realization of security issues and vulnerabilities in smart contract programming languages, and generally in blockchain protocols, there is now a keen interest in the formal verification and testing of smart contracts and other blockchain components before production deployments. For this, various efforts are already underway, many of which we discussed in *Chapter 19*, *Blockchain Security*.

There is also an increased interest in the development of programming languages for developing smart contracts. Efforts are more focused on domain-specific languages, for example, **Solidity** for Ethereum and **Pact** for Kadena. This is just a start, and many new languages are likely to be developed as the available technology advances. Languages for writing circuits like Circom, Zokrates, Cairo, and noir are now also available. This trend will continue.

With the current maturity level of cloud platforms, many companies have started to provide **Blockchain as a Service** (**BaaS**). The most prominent examples are Microsoft's Azure, Google Cloud, Oracle, in which the blockchain platform is provided as a service, and IBM Cloud, which provides IBM's blockchain as a service. This trend is only expected to grow in the next few years, and more companies will likely emerge that provide BaaS to consumers. The convergence of other technologies with the blockchain offers major benefits. At their core, blockchains provide resilience, security, and transparency, which, when combined with other technologies, results in a very powerful complementary technology. For example, IoT stands to gain major benefits such as integrity, decentralization, and scalability when implemented via a blockchain.

AI is expected to gain numerous benefits from the implementation of blockchain technology: in fact, within blockchain technology, AI can be implemented in the form of AAs, which can make rational decisions on behalf of humans. Some ideas of the benefits that the convergence of AI and blockchain can provide are listed as follows:

- The blockchain can share machine learning models in a secure and **P2P** manner.
- The blockchain can serve as an auditing layer for decisions made by AI.
- As AI and machine learning capabilities grow, the blockchain can provide a mechanism to monitor and control any malicious behavior that AI may manifest. While AI can be used for good, it can also be used by malicious actors. In this case, the blockchain can serve as a control mechanism to thwart any attacks. For example, all agents on a blockchain can be controlled under a consensus mechanism, and malicious Byzantine behavior can be handled as per the blockchain consensus protocol.
- AI can also benefit blockchain technology by enabling artificially intelligent smart contracts.
- AI can be applied to other components of a blockchain to achieve, for example, adaptive consensus mechanisms. These can, based on network conditions, readjust fault-tolerance requirements and as a result enable a more efficient consensus mechanism.

There are of course many more examples of innovative applications of the blockchain in combination with the aforementioned technologies. Some of these were discussed in detail in the *Blockchain and AI* section of this chapter.

Some challenges

Apart from security, scalability, and privacy, which were discussed at length in earlier chapters, several other obstacles should be addressed before the broader adoption of blockchains can be achieved. We'll introduce some of these more specific areas to address in the following sections.

Regulation is considered one of the most significant challenges to blockchain development. The core issue is that blockchains, and their associated cryptocurrencies, are not recognized as legal systems or forms of currency by any government. Even though in some cases, blockchain tokens have been classified as having monetary value, for example in the US and Germany, they are still far from being accepted as regular currencies. Moreover, blockchains in their current state are not recognized by financial regulatory bodies as platforms that are stable or reliable enough to be used by financial institutions.

There have been, however, various initiatives proposed by regulatory authorities around the world to research and set regulations. Bitcoin in its current state is fully unregulated, even though some attempts have been made by governments to tax it. In the UK, under the EU VAT directive, Bitcoin transactions are exempt from **Value-Added Tax** (**VAT**). This may change after the finalization of Brexit, but **Capital Gains Tax** (**CGT**) may still be applicable in some scenarios. Some attempt by financial regulatory authorities to regulate blockchain technology is expected very soon, especially after the recent announcement by the UK's **Financial Conduct Authority** (**FCA**) that it may approve some companies that are using blockchain technology for their business services.

Another regulatory concern is that blockchain technology is not ready for production deployments. Even though the Bitcoin blockchain has evolved into a solid blockchain platform and is used in production, it is not suitable for every scenario. This is especially true in the case of sensitive environments such as finance and health. However, this situation is changing rapidly, and this chapter has already explored various examples of new blockchain projects that have been implemented in real life, such as the ASX blockchain post-trade solution. This trend is expected to grow as efforts are made to improve the associated technology and address technical limitations such as scalability and privacy.

Security is also another general concern that has been highlighted by many researchers, which is especially applicable to the finance and health sectors. A report by the **European Union Agency for Network and Information Security** (**ENISA**) has highlighted some distributed ledger-specific concerns, including the need for regulation, auditing, control, and governance. The report is available at `https://www.enisa.europa.eu/news/enisa-news/enisa-report-on-blockchain-technology-and-security`.

Some concerns highlighted in the report also include smart contract management, key management, **Anti-Money Laundering** (**AML**) regulations, and anti-fraud tools. Clearly, while blockchain is generally seen as a solution to many technical business challenges, it can also be misused. In the next section, we'll see some scenarios in which cryptocurrencies and blockchain networks are being used for illegal activities.

With the key attributes of censorship resistance and decentralization, blockchain technology can help to improve transparency and efficiency in many walks of life. However, the somewhat unregulated nature of blockchain technology means that it has the potential to be used for illegal activity. For example, consider a scenario where illegal content is published over the internet. Normally, it can be immediately shut down by approaching the relevant authorities and website service providers, but this is not possible in blockchains. Once something is there on the blockchain, it is almost impossible to remove it. This means that any unacceptable content, once published on the blockchain, cannot be removed. If the blockchain is used for distributing immoral or illegal content, then there is no way for anyone to prevent it. This fact poses a serious challenge, and it seems that some regulation and control would be beneficial in this scenario, but it provokes the critical question: how can a blockchain be regulated? In this case, it may not be prudent to create regulatory laws first and then see if blockchain technology adapts, because this might disrupt innovation and progress. It may be more sensible to let blockchain technology grow first in a similar, organic manner to the internet. Then, when its user base reaches a critical mass, governing bodies can call for the application of regulations around the implementation and usage of blockchain technology.

There are various real-life examples where the **dark web** is used in conjunction with **Bitcoin** to perform illegal activities.



The dark web is a term used to describe different networks that exist on the internet but necessitate the use of special hardware, programs, configuration, or credentials for access. More on the dark web can be found here: `https://en.wikipedia.org/wiki/Dark_web`.

One example is the Silk Road online marketplace, which was used to sell illegal drugs and other contraband over the internet. It used Bitcoin for payments and was hosted on the dark web using URLs only visible with a browser called Tor. Although the Silk Road was shut down after years of effort by law enforcement agencies, other sites have emerged that offer similar services; as such, this activity remains a major concern. Imagine that an illegal website was hosted on IPFS, a P2P distributed and decentralized storage network built on a blockchain; there would be no easy way of shutting it down. It is clear that the absence of control and regulation can encourage illegal activity, meaning criminal enterprises like Silk Road will keep appearing. Further development of totally anonymous transaction capabilities such as Zcash, while useful in various legitimate scenarios, could provide another layer of protection for criminals. Clearly, it depends on who is using the technology; anonymity can be good in many scenarios, for example, in the health industry, where patient records should be kept private and anonymous. However, it may not be appropriate if it can also be used by criminals to hide their activities. One solution might be to introduce intelligent bots, AAs, or even contracts that are programmed and embedded with regulatory logic. They would most likely be programmed by regulators and law enforcement agencies and live on the blockchain as a means to provide governance and control. For example, a blockchain could be designed in such a way that every smart contract has to go through a controller contract, which scrutinizes the code logic and provides a regulatory mechanism to control the behavior of the contract.

It may also be possible to get each smart contract's code to be inspected by regulatory authorities, and once a smart contract's code has a certain level of authenticity attached to it in the form of certificates issued by a regulator, it will be deployed on the blockchain network. This concept of **binary signing** is akin to the already established concept of **code signing**, whereby executables are digitally signed as a means to confirm that the code is bona fide and not malicious. This idea is more applicable in the context of semi-private or regulated blockchains, where a certain degree of control is required by a regulatory authority, for example, in finance. It means, however, that some degree of trust is required to be placed in a trusted third-party regulator, which may not be desirable due to the deviation from full decentralization. However, to address this, the blockchain itself can be used to provide a decentralized, transparent, and secure certificate issuing and digital signing mechanism.

There also needs to be a fine balance between privacy and transparency. Too much transparency can result in undermining personal privacy; however, too much privacy can result in the loss of transparency. The choice between privacy and transparency depends on the use case; however, a state of equilibrium is highly desirable to provide the best solution to handle both requirements.

It is envisaged that other technologies, such as IoT and AI, will converge for the mutual benefit and wider adoption of both the blockchain and the other given technology.

Summary

In this chapter, five main industries that can benefit from blockchain technology were discussed. First, IoT was discussed, which is another revolutionary technology in its own right. By combining it with blockchain technology, several limitations can be addressed, which would bring about tremendous benefits to the IoT industry. More focus has been given to IoT in this chapter, as it is the most prominent and most ready candidate for adapting blockchain technology.

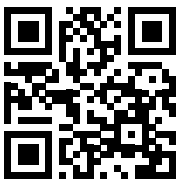
Moreover, applications in the government sector were discussed, whereby various government processes such as homeland security, identification cards, and benefits disbursements can be made transparent, secure, and more robust. Furthermore, issues in the finance sector were discussed, along with possible solutions that blockchain technology could provide. Finally, some aspects of the health sector and music industry were also discussed. All these use cases and many more stand on the pillars provided by the core attributes of blockchain technology, such as decentralization, transparency, reliability, and security. After this, a few trends were discussed that are expected to continue as the technology progresses.

In this book, we have studied the technical foundations of the blockchain and learned how to build practical real-world decentralized applications. We've explored cryptocurrencies, alternative blockchains, enterprise blockchains, scalability, security, privacy, identity, and DeFi, and examined the underlying mechanics of blockchain technology. We've combined these theoretical foundations with the practical development, design, and deployment of blockchain networks, smart contracts, and decentralized applications. We learned how to use Truffle, Ganache, Drizzle, and other tools and techniques to build DApps. Furthermore, detailed accounts of tokenization, consensus mechanisms, and recent developments such as Post-Merge Ethereum were provided.

The blockchain can arguably be considered the most innovative technology of this decade, and as we've seen throughout this book, it is one of the most active areas of study by researchers, academics... and now, you! You are now capable of applying the knowledge from this book and continuing your learning to try your hand at entry-level blockchain development or architecture. Given the material provided in this book, you are now equipped with the necessary skills and knowledge of blockchain technology to become an entry-level blockchain engineer and participate in further research and development concerning this amazing technology and be part of the blockchain revolution!

Join us on Discord!

To join the Discord community for this book – where you can share feedback, ask questions to the author, and learn about new releases – follow the QR code below:



`https://packt.link/ips2H`

Index

A

accounts 254
contract account (CA) 255
externally owned account (EOA) 254, 255
account state 268
balance 268
code hash 268
nonce 268
storage root 268
account storage trie 268
active replication 120
actuators 716
address data type 362
advanced Bitcoin addresses 163
Advanced Encryption Standard (AES) 71, 74
AddRoundKey 75
DES 74
for decryption 76, 77
for encryption 76, 77
MixColumns 75
ShiftRows 75
state 74
SubBytes 75
working 74, 75
advanced protocols, Bitcoin 202
Bitcoin Cash 204
Bitcoin Gold 205
Bitcoin Unlimited 205
Segregated Witness 202-204
Taproot 205, 206
AES-128-CTR 252, 253
Agent 670
aggregate signatures 108
aggregation based oracles 237
AI 740
airdrop hunting 631
algocracy 44
Algorand 118
altcoin 208
providing 209
Amazon Web Services (AWS) 233
reference link 514
American National Standards Institute (ANSI) 530
Analog-to-Digital Converter (ADC) 718
analysis and design, consensus algorithm 122
model 122
processes 122
timing assumptions 123
Android proof 234
anonymity phase 591
Anonymous Credentials (AnonCreds) 444, 447, 675
Anti-Money Laundering (AML) 22, 744
Apache Camel 508

app.js JavaScript file
creating 384-386

Application Binary Interface (ABI) 346, 729

Application-Specific Integrated Circuits (ASICs) 185, 742

arbitrum 575

architecture development method (ADM) 511
phases 512, 513

argument of knowledge 603

arguments (AR) 598

Aries 444, 446, 675
reference link 446

arrays 363

Artificially Intelligent Decentralized Autonomous Organizations (AIDAOs) 741

asset classes 683

asset tokenization 694

associative 80

asymmetric cryptography 80
blockchain technology 109
commitment schemes 110
encoding schemes 116
homomorphic encryption 109
integrated encryption scheme (IES) 83
private key 80, 81
public key 81, 82
secret sharing 109
verifiable random function (VRF) 117
zero-knowledge proofs (ZKPs) 111-113

asymmetric cryptography algorithms 82
discrete logarithm scheme 82
elliptic curves algorithm 83
integer factorization schemes 82

atomic broadcast 121

atomic swaps 589

attacking hash functions
birthday attack 636
collision attack 636
length extension attack 636
preimage attack 636

attacks, on cryptocurrency wallets
malware attacks 632
Man-in-the-Middle attacks 632
phishing attacks 632
security vulnerabilities, in cryptocurrency wallet code 633

attacks, on hardware wallets
malware attacks 633
Man-in-the-Middle attacks 633
physical tampering 633
supply chain attacks 633
user-level attacks 633

attacks, on layer 2 blockchains
attacks, on rollup provider 634
blockchain bridge-related vulnerabilities 634
data availability attacks 634
DSL bugs 635
state channels and side chain-related attacks 635
transaction censoring 634

attack trees 647

attribute-based encryption (ABE) 593

Augur
reference link 238

automatable 224

Automated Market Maker (AMM) 696
pros and cons 700

Autonomous Agents (AAs) 42, 740

avalanche effect 57

Aztec 576
URL 594

Azure
reference link 514

B

bad randomness 630

Bankers' Automated Clearing System (BACS) 688

base 576

base58-encoding scheme 117

base64-encoding scheme 117

Basecoin
URL 475

Beacon Chain 409, 410
beacon node 410
consensus client 411
execution client 411
features 410
proof-of-stake 415-421
validator client 411, 412

Beacon Chain nodes
versus validator nodes 414

beacon client 418

beacon node 410

Bech32 mechanism 203
reference link 203

Besu 444, 446
reference link 446

BFT algorithms 129
HotStuff 151-155
Istanbul Byzantine Fault Tolerance (IBFT) 134-137
Nakamoto consensus 144
PoW 146
Practical Byzantine Fault Tolerance (PBFT) 129-131
Tendermint 137-141

BigchainDB 736

binary signing 745

binding 592

biometric passports 735

BIP37 193

birthday attack 636

Bitcoin 9, 157, 158, 715
advanced protocols 202
altcoin 208
core client and associated tools 209
cryptographic keys 159
extended protocols 206
innovation 200
in real world 197, 198
payments 198-200
reference link 622

Bitcoin addresses 161
advanced Bitcoin addresses 163
typical Bitcoin addresses 161-163

Bitcoin APIs
URL 219

Bitcoin blockchain 176
achieving, techniques 585
data structure 176-178
fork 179, 180
genesis block 178
orphan block 179
properties 180
stale block 179

Bitcoin blockchain, techniques
anonymous signatures 593
attribute-based encryption (ABE) 593
CoinSwap 589
confidential transactions 592
Dandelion 590-592
homomorphic encryption 587
I2P 586
Indistinguishable obfuscation (IO) 586, 587
Layer 2 protocols, using 594
MimbleWimble 592
mixing protocol 588, 589
privacy managers 594
secure multiparty computation 587

Tor 586
trusted hardware-assisted confidentiality 587
TumbleBit 590
Zether 594
Zkledger 593

Bitcoin Cash (BCH) 204
reference link 204

bitcoin-cli 210
Bitcoin command-line tool, using 216
working with 214-216

Bitcoin client installation 209
bitcoin.conf, setting up 211
Bitcoin node, setting up 210
node, starting up in regtest 212-214
node, starting up in testnet 211, 212
source code, setting up 210

Bitcoin command-line tool
using 216

Bitcoin core client 209
bitcoin-cli 210
bitcoind 209
bitcoin-qt 210
download link 209

Bitcoin Core software
reference link 210

Bitcoin, cryptographic keys
private keys 159, 160
public keys 160

Bitcoin futures
reference 158

Bitcoin Gold 205
reference link 205

Bitcoin Improvement Proposals (BIPs) 201, 552
informational BIP 201
Payment 201
PaymentACK 201
PaymentRequest 201
process BIP 201
standard BIP 201

Bitcoinj
URL 219

Bitcoin Lightning 557

Bitcoin merchant solutions
URL 200

Bitcoin miner 181, 182
mining pool 186
mining systems 184
Proof of Work (PoW) 182-184
tasks, performing 181, 182

Bitcoin miner, mining systems
ASICs 185
CPU mining 184
Field Programmable Gate Arrays (FPGAs) 185
GPU 184

Bitcoin network 186, 187
bloom filter 192, 193
client software 192
protocol messages, types 187-191

Bitcoin-NG 554
microblocks 554

Bitcoin programming 219

Bitcoin testnet
reference link 191

Bitcoin transactions 163, 164, 588
bugs 175, 176
coinbase transactions 164, 165
data structure 167-169
elements 164
lifecycle 165, 166
Script language 170, 171

Bitcoin transactions, data structure
input (vin) 169
metadata 169
outputs (vout) 170
verification 170

Bitcoin transactions, lifecycle
fees 166, 167
validation 166

Bitcoin transactions, Script language
contracts 174, 175
opcodes 171
standard transaction scripts 171-174
Bitcoin Unlimited 205
reference link 205
Bitcoin wallets 194
brain wallets 195
deterministic wallets 194
hardware wallets 195
hierarchical deterministic wallets 194
mobile wallets 195
non-deterministic wallets 194
online wallets 195
paper wallets 195
types 194
BitDNS 715
BitPay
URL 200, 219
Bitswap mechanism 39
Blake hash function
reference link 287
blind signatures 10, 104
reference link 105
blob-carrying transaction 434
blob transaction
lifecycle 434
block-based DAGs 555
blockchain 11
and AI 740, 741
append-only 11
architecture 12
as layer 12-14
benefits and features 20, 21
challenges 743-745
compatibility, checking for enterprise 498
distributed ledgers 11, 23
emerging trends 741-743
fully private and proprietary blockchains 25
functioning 18, 19
generic elements 14-18
high-profile successful attacks 620
history 8, 9
in business 14
layer 1 blockchain 26
layer 2 blockchain 26
Layman's definition 11
peer-to-peer (P2P) 11
permissioned ledger 25
private blockchains 24
public blockchain 24
querying, with Geth 301
security 619-621
semi-private blockchains 24
shared ledger 24
technical definition 11
tokenized blockchains 25
tokenless blockchains 25
types 23
updatable via consensus 12
blockchain, and IoT convergence
benefits 719-721
blockchain application layer 628
DeFi attacks 631
smart contract vulnerabilities 628, 629
Blockchain as a Service (BaaS) 505, 513, 742
providers 514
blockchain, as layer
Applications layer 14
Consensus layer 14
Cryptography layer 13
Execution layer 14
Network layer 13
blockchain-based IoT implementation 722
electronic circuit, building 728, 729
first node, setting up 725
Node.js, installing 727, 728

prerequisite hardware components 722
Raspberry Pi node, setting up 726, 727
Raspberry Pi, setting up 723-725
Solidity contract, developing 729-733
Solidity contract, running 734

blockchain in finance, applications 685
financial crime prevention 686-688
insurance industry 685
payment 688, 689
post-trade settlement 685

blockchain.info
URL 219

blockchain layer 624
attacks, on consensus protocols 626
attacks, on transactions 624, 625
chain reorganization 627
double-spending 627
forking 627
selfish mining attack 627
transaction replay attacks 625, 626

blockchain layers
blockchain application layer 621, 628
blockchain protocol layer 621, 624
cryptography layer 621
hardware layer 621-623
interface layer 621, 631
network layer 621-624

Blockchain Open Ledger Operating System (BOLOS) 234

Blockchain oracle problem 235, 238

blockchain oracles
services 239, 240
types 236

blockchain oracles, types
cryptoeconomic oracles 239
inbound oracles 236
outbound oracles 238, 239

blockchain privacy 583
anonymity 584
confidentiality 584
example 610-616
Zero-knowledge, using 594, 595

blockchain privacy, zero-knowledge protocols
cryptographic commitment 595-597
proofs 597-601
ZK-SNARKs, building 601-607

blockchain services, Fabric 454
consensus service 454
distributed ledger 455
ledger storage 456
peer-to-peer protocol 455

blockchain solution
implementation strategy, establishing 498

blockchain technology
growth 1-4
limitation 23
limitations 21, 22

blockchain trilemma 546, 547
properties 546

blockchain, use cases 715, 716
government 735
health 738, 739
IoT 716
media 739

blockchain, using Geth
interacting, methods 301

Blockchain wallet
using 198

block cipher encryption function 73

block ciphers 71
operation 72

blockcracy 44

block data (transactions) 455

block encryption modes 72
cipher block chaining (CBC) mode 72
counter (CTR) mode 72
electronic code book (ECB) 72

keystream generation mode 73
message authentication mode 73
block headers 275, 455
elements 275
block interval reduction 553
block.io
URL 219
block-less DAGs 555
block metadata 455
block propagation 553
blocks 274, 444
block size
increasing 552
bloom filter 192, 193
bloXroute 551
BLS cryptography 422
Bluetooth Low Energy (BLE) 40
Blum-Blum-Shub (BBS) 55
Boba Network 576
Boneh-Lynn-Shacham (BLS) 417, 422
aggregate signatures, reference link 108
Boolean 361
boot nodes 282
brain wallets 195
Breadboard 722
bridgefy 40
Brownie 352
reference link 352
brute-force attack 635
btcd
reference link 210
BTC Relay
URL 208
bulletproofs 598
Byzantine Fault Tolerance (BFT) 122, 501
Byzantine Generals problem 5
Byzantine node 741
Byzantium 281, 348

C

Cakeshop
transaction, viewing 538
Caliper 444, 448
reference link 448
Capital Gains Tax (CGT) 743
CAP theorem 6, 7
availability 6
consistency 6
partition tolerance 6
carbon footprint, Bitcoin
reference link 147
Cardano Ouroboros 118
Casascius physical bitcoins 159
Casper 295
reference link 281
Casper the Friendly Finality Gadget (Casper-FFG) 420, 429
Cello 444, 448
reference link 448
central bank digital currency (CBDC) 22, 34
Centralized Exchange (CEX) 695
versus DEX 700, 701
Centralized Finance (CeFi) 690
centralized identity model 652, 653
centralized system 30
versus decentralized system 33
Central Limit Order Book (CLOB) 698
Certificate Authority (CA) 454, 458, 657
certificates 732
certificates, PBFT 131
CFT algorithms 124
Paxos 124-127
Raft 127-129

chain-based PoS 149
chaincode 450, 452
implementing 460, 461
chained hashing scheme 147
chainlink
URL 239
chain of blocks 11
chain reorganization 627
Chain Virtual Machine (CVM) 17
challenge-response protocols 82
challenges, enterprise blockchain
business challenges 518
compliance 518
interoperability 517
lack of standardization 517
channels, Fabric 457
checkpointing 133
Chicago Mercantile Exchange (CME) 157
cipher block chaining mode (CBC mode) 73
cipher feedback (CFB) mode 74
ciphers 52
Clearing House Automated Payment System (CHAPS) 688
clients, Fabric 457
client software 192
Clique 531
closure 80
CNexchange (CNEX) 480
Cockpit 352
code signing 745
coin 473
coinjoin 588
CoinSwap 589
collision attack 636
collision resistance 56
colored coins 206
command-line interface (CLI) 372
commit chains 559
commitment schemes 110, 111
commit phase 110
open phase 110
Pedersen commitment scheme 111
commit phase 110
binding property 110
hiding property 110
committee-based PoS 150
committing peers 457
CommonAccord
URL 228
common language for augmented contract knowledge (CLACK) 229
common reference string model 599
Common Vulnerability Scoring System (CVSS) 647
compliance 503, 504
component requisites, private net
data directory 285
genesis file 285
network ID 284
computation oracles 237
Computation Tree Logic (CTL) 642
conditional privacy 584
confidentiality 502
confidential transactions 592
confusion property 71
consensus 119
consensus algorithm
analysis and design 122
fundamental requirements 124
lottery-based 123
traditional voting-based 123

consensus algorithm, selecting factors 155
finality 155
performance 156
scalability 156
speed 156
consensus client 411, 423
consensus layer 298
consensus mechanism 643, 644
consensus mechanism, Ethereum 279, 280
consensus protocol 17, 18
consensus service 454
consensus states, IBFT
committed 136
final committed 136
new round 136
prepared 136
pre-prepared 136
round change 136
consortium chain type 497
Constant Function Market Makers (CFMMs) 700
constant mean market makers (CMMM) 698
constant product market maker (CPMM) 697
Constant sum market makers (CSMM) 697
Constellation 524
constructor function 358
contest-driven decentralization 34
contract
functions, calling 388
contract account (CA) 255
properties 255
contract creation transactions 258, 264
contracts 174
compiling, with Truffle 393-397
deploying 372-376
interacting with 398, 399
interacting with, via frontends 381, 382
migrating, with Truffle 393-397
querying, with Geth 377-380
solc, used for generating ABI and code 376, 377
testing, with Truffle 393-397
Truffle, installing and initializing 392, 393
Truffle, used for deploying and interacting with 391, 392
Web3 and Geth, used for interacting with 371, 372
contracts, via frontends
app.js JavaScript file, creating 384-386
frontend webpage, creating 387-391
functions, calling 388
web3.js JavaScript library, installing 382, 383
web3 object, creating 383
control structures, Solidity 365, 366
Coq
URL 640
Corda
reference link 24
Counter Financing of Terrorism (CFT) 22
counterparty 207, 208
armory_utxsvr 207
counter block 207
counter wallet 207
server 207
URL 208
Counterparty coins (XCPs) 207
CPU-bound PoW 146
CPU mining 184
Crash Fault Tolerance (CFT) 120, 122, 467
cross link 410
crowd wisdom-driven oracles 237, 238
cryptocurrency 250
cryptoeconomic oracles 239
cryptographic commitments 595-597
cryptographic hash function applications 63
distributed hash tables 66, 67

Merkle Patricia trie 65, 66
Merkle tree 64
cryptographic hash mode 73
cryptographic primitives 54, 55
keyless primitives 55
symmetric key primitives 67, 68
taxonomy 54
cryptography 52, 622
accountability 54
non-repudiation 53
services 52-54
cryptography layer
attacking hash functions 636
key management-related vulnerabilities and attacks 636
public key cryptography, attacking 635, 636
ZKP-related attacks 637, 638
cryptography, services
authentication 53
confidentiality 52
data origin authentication 53
entity authentication 53
integrity 52
multi-factor authentication 53
cryptojacking 622, 633
CryptoKitties 474, 480
URL 474
crypto malware 622
CryptoNote 109
crypto service provider, Fabric 459
CureCoin 739
URL 739
curl 218
reference link 218, 303
URL 381
cyclic group 80, 93

D

DAG-based chains 554
block-based DAGs 555
block-less DAGs 555
types 555
Dagger 312
DAI 45
URL 45
Dai stable coin
URL 475
DAML 521
reference link 522
Dandelion protocol 590-592
danksharding 432
DAO hack 643
DAO legality
reference link 48
DApps stats
reference link 46
dark web 744
reference link 744
data availability 634 **Data Availability Sampling (DAS) 433**
Data Encryption Standard (DES) 71, 74
data integrity service 56
data types, Solidity
reference types 361, 363
value types 361
decentralization 29-33
benefits 36, 37
contest-driven decentralization 34
disintermediation 33
methods 33
quantifying 34, 35
requirements evaluating 37, 38

trends 48
working 42

decentralized applications (DApps) 44, 46, 230, 248, 471
criteria 46
design 46-48
operations 46
Type 1 44
Type 2 45
Type 3 45

decentralized autonomous corporation (DAC) 43

decentralized autonomous initial coin offering (DAICO) 478

Decentralized Autonomous Organizations (DAOs) 3, 43, 241, 242, 250, 478, 692, 693

decentralized autonomous society (DAS) 44

decentralized consensus 31

Decentralized Exchange (DEX) 695-697
aggregator 699, 700
AMM, pros and cons 700
CMMM 698
CPMM 697
CSMM 697
order book-based DEX 698
versus CEX 700, 701

decentralized finance (DeFi) 2, 690, 691, 674
benefits 707, 708
identity 672-675
layers 692, 693
primitives 693, 694
properties 691
services 694
token, swapping 708, 709
Uniswap liquidity pool 710-714
Uniswap protocol 708

decentralized finance (DeFi), services
asset tokenization 694
Decentralized Exchanges (DEX) 695-697
derivatives 702, 703
flash loan 701, 702
insurance 704
lending and borrowing 704-707
money streaming 703
yield farming 703

decentralized identifiers (DID) 665-670
requirements 666

decentralized identity and decentralized finance (DeFi) 49

decentralized identity model 657

decentralized marketplace 45

decentralized mesh network 740

decentralized NFT marketplace
URL 739

decentralized oracle 238

Decentralized Organizations (DOs) 42, 43

Decentralized Public Key Infrastructure (DPKI) 657

decentralized storage
IPFS, using for deployment on 404-406

decentralized system 31
versus centralized system 33

decentralized web 48

decrypted private key 253

decryption 52

decryption key 253

Deep Crack 74

DeFi attacks
airdrop hunting 631
flash loan attacks 631
forged NFTs 631
identity spoofing 631
MEV/BEV 631
NFT DoS 631
sandwich attack 631
stable coins stability/security risks 631
unlimited (scarcity-free) token generation 631

delegated PoS 150
demand-side economies of scale 198
Denial of Service (DoS) attack 589, 623, 721
deployment transactions 458
derivatives 702
derivative token 476
design principles, Hyperledger
auditability 452
deterministic transactions 452
identity 451
interoperability 452
modular structure 451
portability 452
privacy and confidentiality 451
rich data queries 452
scalability 451
deterministic wallets 194
development environment
connecting, to test networks 305
private network, creating 305-307
setting up 304
DeversiFi 576
DEVP2P wire protocol 283, 290
Diem protocol
reference link 154
Differential Power Analysis (DPA) 636
difficulty time bomb 295
Diffie-Hellman algorithms 82
diffusion 591
diffusion property 71
Digital Asset Holdings (DAH) 444
Digital Asset Modeling Language (DAML) 243-245
reference link 244
digital identity 652
digital identity, models
centralized identity model 652, 653
decentralized identity model 657
federated identity model 653-657
self-sovereign identity 658
self-sovereign identity, components 659
Digital Ledger Technology (DLT) 23
Digital Rights Management (DRM) 20
digital signatures 73, 82, 98
aggregate signatures 108
blind signature 104
elliptic curve digital signature algorithm (ECDSA) 100, 101
multisignatures 105, 106
ring signatures 108, 109
RSA digital signature algorithms 98
threshold signatures 106, 107
types 104
digital tree 65
digital wallet 670, 671
governance frameworks 671, 672
verifiable data registries 671
Digix gold tokens
URL 475
Directed Acyclic Graph (DAG) 115, 312, 554, 602
Directed Acyclic Graphs (DAGs) 39
Discovery protocol 282
discrete logarithm integrated encryption scheme (DLIES) 83
discrete logarithm problem, ECC 93-95
discrete logarithm scheme 82
DiscV4 283
reference link 283
DiscV5 283
reference link 283
disintermediation 33
Distributed Artificial Intelligence (DAI) 740
distributed consensus 18
Distributed Denial of Service (DDOS) 290
distributed hash table (DHT) 66, 67

Distributed Hash Tables (DHTs) 9, 38
distributed ledgers 24, 444, 455
Besu 446
Fabric 444
Indy 446
Iroha 445
Sawtooth 445
distributed system 4-6, 30, 119
CAP theorem 6, 7
PACELC theorem 8
Distributed Validator Protocol 430
Distributed Validator Technology (DVT) 429
distributive law 80
DLS protocol 138
documentation and coding guidelines, Solidity
reference link 369
Domain-Specific Languages (DSLs) 229, 635
domain-specific projects, Hyperledger 448
Grid 448
double and add algorithm 92
double-spending attack 627
DREAD model 647
Drizzle 352

E

ECDSA digital signatures
generating 102-104
eclipse attack 623, 624
EIP-155 626
EIP155
reference link 282
EIP-1559 296-298
EIP-1559, variables
baseFeePerGas 297
maxFeePerGas 298
maxPriorityFeePerGas 298
electronic cash (e-cash) 9, 10
accountability 9
anonymity 10
Electronic Frontier Foundation (EFF) 74
Electrum
URL 195
elliptic curve 83, 87
point addition 88-90
point doubling 88-92
point multiplication 92
Elliptic Curve Cryptography (ECC) 87, 159, 258
discrete logarithm problem 93-95
keys, generating with 95-97
mathematics 87, 88
Elliptic-curve Diffie-Hellman (ECDH) 83, 525
Elliptic Curve Digital Signature Algorithm (ECDSA) 83, 100, 160, 250, 638
using 101
elliptic curve discrete logarithm problem (ECDLP) 93
Elliptic Curve Integrated Encryption Scheme (ECIES) 83, 283
reference link 283
elliptic curves algorithm 83
Embark 352
embedded consensus 207
enclave 524, 587
encoding schemes 116
base58 117
base64 117
encrypted private key 253
endorsing peers 457
enrolment certificate authority (E-CA) 454
enrolment certificates (E-Certs) 454
enterprise blockchain
architecture 507
available platforms 515-517

challenges 517
versus public blockchain 506
enterprise blockchain architecture
application layer 509
governance layer 508
integration layer 508
network layer 507
privacy layer 508
protocol layer 508
security, performance, scalability, monitoring 509
enterprise blockchain solutions
business-oriented factors 499
designing 509
limiting factors 500, 501
enterprise blockchain solutions, designing
architecture development method (ADM) 511
Architecture development method (ADM) 512
cloud solutions 513, 514
TOGAF 510
enterprise blockchains, requirements
access governance 503
better tools 506
compliance 503, 504
consistency 501
ease of use 505
integration 505
integrity 501
interoperability 504
monitoring 505
performance 502, 503
privacy 502
secure off-chain computation 505
Enterprise Blockonomics 518
enterprise dApps 503
Enterprise Ethereum Alliance (EEA) 3, 502
Enterprise Haskell 521
Enterprise Resource Planning (ERP) 499
enterprise solutions 498
enums 363
ephemeral keys 67
epidemic flooding 591
equity token offerings (ETOs) 477
ERC-20 interface
functions and events 483-485
ERC-20 token
adding, in MetaMask 493-495
building 483
contract, deploying on Remix JavaScript virtual machine 488-493
Solidity contract, building 483-488
ERC-20 token standard 480
reference link 480
ERC-223 token standard 480
ERC-721 token standard 480
ERC-777 token standard 480
ERC-884 token standard 480
ERC-1155 token standard 482
reference link 482
ERC-1400 token standard 481
ERC-1066 481
ERC-1410 481
ERC-1594 481
ERC-1643 481
ERC-1644 481
ERC-1404 token standard 481
reference link 481
ERC-4626 token standard 482, 483
reference link 482
error handling constructs, Solidity
assert 368
require 368
revert 368
throw 368
Try/Catch 368
Eth capability protocol
reference link 284

ether 250, 729

Ethereum 407, 522, 598, 625
future roadmap 440
identity 672
innovations 295
reference link 622
staking on 412-415
transactions 294

Ethereum, after The Merge 408, 409
Beacon Chain 409, 410
dimensions 408, 409
P2P interface (networking) 421, 422

Ethereum block
difficulty mechanism 278
finalization 277
processing 277, 278
validation 276, 277

Ethereum blockchain 247, 248
ecosystem architecture 249
interacting, with MetaMask 321
programming languages 287, 344
state transition function 267

Ethereum blockchain, elements 249
accounts 254
cryptocurrency 250
Ethereum network 281
EVM 270-272
keys and addresses 250-254
messages 265, 266
miners 279
nodes 279
precompiled smart contracts 286, 287
transactions 255
wallets 289

Ethereum blocks
elements 274

Ethereum Classic (ETC) 250, 625

Ethereum environment (EOAs) 265

Ethereum Go client
URL 305

Ethereum Homestead 264

Ethereum improvement proposals (EIPS) 295, 432
reference link 480
URL 295

Ethereum, innovations
difficulty time bomb 295
EIP-1559 296-298
merge and upgrades 298

Ethereum Name Service (ENS) 508

Ethereum network 281
DEVP2P 283
Discovery protocol 282, 283
main net 282
private nets 282
RLPx 283
sub-protocols 283
test nets 282

Ethereum Virtual Machine (EVM) 241, 247, 270-272, 278, 344, 411
execution environment 273
machine state 273, 274

Ethereum WebAssembly (ewasm) 272
reference link 272

Ethereum yellow paper
reference link 248

Ethlance 44
URL 45

European Union Agency for Network and Information Security (ENISA) 744

events, Solidity 366

EVM networks
reference link 272

EVMs (ZK-EVMs) 569

execution client 411, 423

execution layer 298

extendable-output functions (XOFs) 61
extended protocols, Bitcoin 206
colored coins 206, 207
counterparty 207, 208
external function calls 357
external functions 356
externally owned account (EOA) 254, 265
properties 254, 255
External Owned Accounts (EOAs) 693

F

Fabric 444, 452, 515
core capabilities 453
messages 456
reference link 444
Fabric 2.0 465
new chaincode application patterns 466, 467
new chaincode lifecycle management 465, 466
reference link 465
Fabric, applications 459
application model 462
chaincode implementation 460, 461
Fabric, components
channels 457
clients 457
crypto service provider 459
membership service provider 458
nodes 457
peers 457
private data collections (PDCs) 458
smart contracts 459
transactions 458
world state database 457, 458
Fabric, consensus mechanism
ordering 462
transaction endorsement 462
validation and commitment 462
Fabric, key concepts 452, 453
APIs and CLIs 456
blockchain services 454
membership service 453, 454
smart contract services 456
Fabric, transaction flow
steps 463, 464
failure detectors 121
fallback functions 358
faster consensus mechanisms 555
Fast Reed-Solomon IOP of Proximity (FRI) 597
fault tolerance 120
fault-tolerant algorithms
Byzantine fault-tolerance (BFT) 120
crash fault-tolerance (CFT) 120
Federal Information Security Management Act (FISMA) 517
federated identity model 653-657
Feistel cipher 71
Fiat-Shamir (FS) 609
field
cardinality 80
finite field 80
order 80
prime field 80
Field Programmable Gate Arrays (FPGAs) 185, 742
Filament 721
Filecoin 39
Financial Conduct Authority (FCA) 504, 743
financial instruments
attributes 683
financial markets 681
exchanges 682
trading 681

financial markets, exchanges
order management and routing systems 683
orders and order properties 682, 683
trade, components 683
trade lifecycle 684, 685
finite field 80
firewalls 732
flash loan 701, 702
flash loan attack 631
FLP impossibility result 121
Fluff phase 591
FoldingCoin 739
URL 739
foreign exchange (forex) 474
forged NFTs 631
forking 627
forks 179, 281
hard fork 180
soft fork 180
temporary forks 180
types 180
formal specifications 639
formal verification 639, 644
model checking 641-643
of smart contracts 640, 641
Frama-C
URL 644
frontend attacks
account hacking 633
DoS attacks 633
malicious scripts 633
misaligned frontend 633
frontend webpage
creating 387-391
Frontier 281
Fuel v1 576
full-ecosystem decentralization 38
communication 39, 40
power, computing 40-42
storage 38, 39
fully private blockchain 25
function modifiers, Solidity
override 359
payable 358
pure 358
view 358
virtual 359
functions, Solidity 355
constructor functions 358
external function calls 357
external functions 356
fallback functions 358
function modifiers 358
function signature 356
function visibility specifiers 358
input parameters 356
internal function calls 357
internal functions 356
modifier functions 358
output parameters 357
syntax 356
function visibility specifiers, Solidity
external 358
internal 358
private 358
public 358
fungible tokens 473
divisible, working principle 473
indistinguishable, working principle 473
interchangeable, working principle 473
working principle 473

G

Galois counter (GCM) mode 74
Galois fields 80

Ganache 348
ganache-cli 348
ganache-ui 349-351
gas 262, 263
gas limit field 258
Gasper PoS 410
gas price field 258
Gemini Dollar (GUSD)
URL 475
General-Purpose I/O (GPIO) pins 722
general-purpose programming languages (GPLs) 230
genesis block 178, 276
genesis file 15, 285
parameters 285
genesis transaction 227
Geth
account, creating 299-301
POST requests, used for interacting with 380, 381
used, for interacting with contracts 371, 372
used, for querying blockchain 301
used, for querying contracts 377-380
Geth attach 301, 302
Geth client 248, 289
configuring 299
installing 299
Geth console 301
Geth JavaScript console
experimenting with 310, 311
Geth JSON RPC 301-304
Geth RPC APIs
reference link 304
global stabilization time (GST) 123
global variables 359, 360
Gluon 576
Goerli test network 322
Golem 45
URL 45
Google RPC (gRPC) 455
GoQuorum plugins
reference link 542
governance frameworks 671, 672
government functions, blockchain
border control 735-737
citizen identification 737, 738
elections 737
GPU 184
Greediest Heaviest Observed SubTree (GHOST) 279, 421, 553
GreenAddress 195
Grid 444, 448
reference link 448
group signatures 593

H

Hadamard Product Relation (HPR) 602
handshaking 283
hard fork 180
hardware description languages (HDLs) 185
hardware device-assisted proofs 234
Android proof 234
Ledger proof 234
trusted hardware-assisted proofs 235, 236
hardware layer, blockchain 622, 623
hardware oracles 236
Hardware Security Module (HSM) 194, 637
Hardware Security Modules (HSMs) 451, 505
hardware wallets 195
hash-based commitment schemes 596
hash-based MACs (HMACs) 69
hash collision attack 240

hash functions 56, 57
applications 63
message digest (MD) functions 57
messages, encrypting with SHA-256 62, 63
properties 56
secure hash algorithms (SHAs) 57, 58
security properties 56
hash pointer 19
hexadecimal literals 363
hierarchical deterministic wallets 194, 195
High-Performance Computing (HPC) 505
Homestead 281, 348
homomorphic encryption 109, 587, 741
fully homomorphic system 109
partially homomorphic system 109
Horn solving 641
HotStuff 151, 152
chain quality 151
commit phase 153
decide phase 153
linear view change 151
liveness 154
optimistic responsiveness 151
pre-commit phase 153
prepare phase 153
safety 154
HTTP REST interface
using 218, 219
hybrid encryption schemes 83
Hyperledger
APIs and SDKs 450
communication 450
consensus 450
data store 450
design principles 451, 452
policy services 450
projects 444
reference architecture 449, 450
security and crypto 450
smart contracts 450
URL 444
Hyperledger Fabric blockchain
capabilities 453
Hyperledger Indy 675
Hyperledger Landscape
URL 444
Hyperledger project 515
Hyperledger Ursa 675

I

IBD node 190
IBFT 531
Quorum network, setting up with 532
IBM 444
reference link 514
identity 651, 652
in DeFi 672-675
in Ethereum 672
in Metaverse 672-675
in Web3 672
in world of Web3 673-675
Identity and Access Management Systems (IAMs) 655
challenges 656
identity mixer 454
Identity Provider (IDP) 655
identity spoofing 631
iExec
URL 239
Immutable X 576
impersonation attack 635
inbound connections 421
inbound oracles 236

inbound oracles, types
aggregation based oracles 237
computation oracles 237
crowd wisdom-driven oracles 237, 238
decentralized oracle 238
hardware oracles 236
smart oracles 238
software oracles 236
indistinguishable obfuscation (IO) 586, 587
Indy 444, 446
reference link 446
Infrastructure as a Service (IaaS) 513
Infura
URL 321
inheritance, Solidity 367
Initial Block Download (IBD) node 189
Initial Coin Offering (ICO) 476, 716
initial currency offering (ICO) 476, 477
initial exchange offering (IEO) 477
initialization vector (IV) 68, 253
initial public offerings (IPOs) 476, 477
inner padding (ipad) 69
insecure key storage 637
Institute of electrical and electronics engineers (IEEE) 3
integer factorization schemes 82
integer literals 363
integer overflow and underflow 628, 629
integers 361, 362
Integrated Development Environments (IDEs) 318
integrated encryption scheme (IES) 83
discrete logarithm integrated encryption scheme (DLIES) 83
elliptic curve integrated encryption scheme (ECIES) 83
IntelliSense 353
Intel Software Guard Extensions (Intel SGX) 445
Interactive Oracle Proof (IOP) 597, 605
classes, categorizing 605
interface layer, blockchain
attacks on wallets 632, 633
oracle attacks/oracle manipulation attacks 632
internal function calls 357
internal functions 356
International Civil Aviation Organization (ICAO) 735
international standards organization (ISO) 3
Internet of Things (IoT) 231, 716
Internet Service Providers (ISPs) 39
interoperability 504
Inter-Planetary File System (IPFS) 39, 49
IRL 49
Inter-Process Communication (IPC) 309
Invertible Bloom Lookup Tables (IBLTs) 553
Invisible Internet Project (I2P) 586
invocation transactions 458
IoT, and blockchain convergence
benefits 719-721
IoT architecture 716, 717
application layer 718, 719
device layer 718
management layer 718
network layer 718
physical object layer 718
IPFS 352, 736
used, for deployment on decentralized storage 404-406
IPv6 719
Iroha 444, 445
reference link 445
Isabelle
URL 640
Istanbul 348

Istanbul Byzantine Fault Tolerance (IBFT) 119, 120, 134-137, 430
consensus states 136
references 137
versus Practical Byzantine Fault Tolerance (PBFT) 134, 135
working 135

J

JavaScript Object Notation (JSON) 302
JavaScript runtime environment (JSRE) 398
Jaxx
URL 195
JSON RPC
URL 302
JSON RPC interface
using 217, 218
JSON Web Signature (JWS) 664
Just a Bunch of Key wallets 194

K

Kadcast 551
Keccak 61
key derivation function (KDF) 68, 252, 253
keyed hash functions 68
key escrow attack 637
key establishment mechanisms 82
keyless primitives 55
hash functions 56, 57
random numbers 55
key loss or theft 637
key management-related vulnerabilities and attacks
insecure key storage 637
key escrow attack 637
key loss or theft 637
unauthorized key sharing 637
key reuse attack 636
keys
generating, with ECC 95-97
keystream 72
key stretching 195
Know Your Customer (KYC) 197, 503

L

languages, Ethereum blockchain
Low-level Lisp-like Language (LLL) 344
Mutan 344
Serpent 344
Solidity 344
Vyper 344
Yul 344
Last in, First Out (LIFO) 170, 270
Latest Message Driven Greediest Heaviest Observed SubTree (LMD GHOST) 419, 421, 438
layer 1 blockchain 26
monolithic blockchain 26
polylithic blockchain 26
layer 2 blockchain 26
sidechains 26
Layer2 Finance 576
LED 722
Ledger Blue 234
ledger decoupling 445
Ledger Nano S 234
Ledger proof 234
ledger storage 456
Legal Knowledge Interchange Format (LKIF) 225
length extension attack 636
Libbitcoin
URL 219

Libra
URL 475
libraries, Hyperledger project 446
Anonymous Credentials (AnonCreds) 447
Aries 446
Transact 447
Ursa 447
libraries, Solidity 367
LibUrsa 447
LibZmix 447
light clients 289
Light Emitting Diode (LED) 722
Light Ethereum Sub-protocol (LES) 283
Lightweight Directory Access Protocol (LDAP) 454
Linear Temporal Logic (LTL) 642
literals 363
enums 363
hexadecimal literals 363
integer literals 363
string literals 363
loan mechanism
actors 704, 705
local variables 359
log replication 129
log series 266
London 348
Loopring 576
lower bound result 122
Low-level Lisp-like Language (LLL) 287, 344
low watermark 133

M

mac 253
machine learning 740
Machine-Readable Travel Document (MRTD) 735
Machine-Readable Zone (MRZ) 735
main net 282
Man-in-the-Middle (MITM) attack 635
Manticore
reference link 645
mappings 364
MasterCoin 716
master key 67
mathematical concepts 79
field 80
group 80
modular arithmetic 79
sets 80
mathematical puzzle 181
Mauve Paper
reference link 554
Maximal Extractable Value (MEV) 434
maximum achievable decentralization (MAD) 35
maximum value 237
mean value 237
median value 237
membership service, Fabric 453, 454
membership service provider (MSP) 453, 458
memory-bound PoW 146
memory hard computational puzzles 147
memory pool (mempool) 205
merged fee market 435
Merkelized Alternative Script Tree (MAST) 206
Merkle-Damgard construction 57
Merkle Patricia Tree (MPT) 65, 66, 256, 257, 572
nodes, types 65
Merkle Patricia trie (MPT) 523
Merkle root 15, 64, 256
ReceiptsRoot 257
StateRoot 257
TransactionsRoot 257

Merkle trees 64, 65, 256
mesh communication topology 152
mesh network 740
message authentication 53
message authentication codes (MACs) 53, 68
hash-based message authentication codes 69
message call transactions 258, 265
message digest (MD) functions 57
message passing 119
messages 265
components 265, 266
messages, Fabric
consensus messages 456
discovery messages 456
synchronization messages 456
transaction messages 456
messages, PBFT 132
messages, Tendermint 143
pre-commit 142
pre-vote 142
proposal 142
MetaMask 483
accounts, importing with keystore files 328-331
custom network, adding 325-327
custom network, adding to connect with Remix IDE 325-327
installing 321, 322
Remix IDE, used for interacting contract through 336-342
used, for creating account 322-324
used, for deploying contract 331-335
used, for funding account 322-324
used, for interacting with Ethereum Blockchain 321
using, to deploy smart contract 324
MetaMask wallet 289
Metaverse
identity 672-675
metaverses 49
Metis Andromeda 576
MEV/BEV 631
MimbleWimble 592
Miner Extractable Value (MEV) 436, 700
miner node
functions 279
minikey 159
minimum feasible decentralization (MFD) 35
mining 19
mining algorithm
steps 182
mining pool 186
mining systems 184
mini private key format 159
Miniscript 175
reference link 175
mixing protocol 588, 589
mnemonic code 194
mobile wallets 195
model checking 641-643
modifier functions 358
modulus 79
Monero 109, 110
money streaming 703
MPC-based approach 600
Multichain blockchains
reference link 106
multichain solutions 549
multiparty computation (MPC) 600
multi-party non-repudiation (MPNR) protocols 54
Multi-Signature (M of N) 431
multisignatures 105, 106
MultiSig (Pay to MultiSig) 172

Musicoin
URL 739
Mutan 344
MVC-B architecture 462
blockchain logic 462
control logic 462
data model 462
view logic 462

N

Nakamoto coefficient 35
reference link 35
Nakamoto consensus 120, 144
properties 146
versus traditional consensus 145
Namecoin 42, 715
reference link 715
National Institute of Standards and Technology (NIST) 57
native contracts 286
Near Field Communication (NFC) 195, 588
network effect 198
network ID 284
networking functions 428
network layer, blockchain
DoS attack 623
eclipse attack 623, 624
network spoofing 624
Sybil attack 623
network model 138
network spoofing 624
new chaincode application patterns, Fabric 2.0 466, 467
new chaincode lifecycle management, Fabric 2.0 465, 466
New Out Of Box Software (NOOBS) 723
download link 723
New York Stock Exchange (NYSE) 682
NFT DoS 631
node 4
Node.js 347
installation link 532
URL 347
nodes, Fabric
orderer nodes 457
node types, Merkle Patricia tree
branch nodes 65
extension nodes 65
leaf nodes 65
null nodes 65
nonce 15, 68, 258
non-deterministic wallets 194
Non-Fungible Tokens (NFTs) 2, 473, 706
indivisible 474
non-interchangeable 474
unique 474
non-outsourceable puzzles 151
not colored coins 207
nothing at stake problem 150, 627
Null data/OP_RETURN 172
Nxt
URL 150

O

off-chain solutions 546, 555, 556
OMG Network 576
Ommers validation 277
OMNI network
URL 45
on-chain scaling solutions 551
on-chain solutions 546
one-way pegged sidechain 26
Onion Router 586
online wallets 195

opcodes 170, 171, 288
reference link 289
Open Authorization (OAuth) 654
OpenBazaar
URL 228
Openchain
reference link 106
OpenID 655
open phase 110
Open Web Application Security Project (OWASP) 647
OpenZeppelin toolkit 353
reference link 353
OpenZepplin 495
reference link 495
optimal decentralization point (ODP) 35
optimistic rollups 563, 564, 575
advantages 564
disadvantages 564
vs ZK-rollups 573
Oracle 685
reference link 514
oracle-as-a-service platforms 239
oracle attacks
bribing oracles 632
Denial of Service (DoS) attack 632
freeloading attacks 632
oracle censorship 632
Sybil attacks 632
tampering with data sources 632
oracle, is Truebit
URL 237
oracle manipulation attack 629, 630
preventing 630
oracles 121, 231-233
standard mechanics 232
use cases 231
order book-based DEX 698
orderer 454
orderer nodes 457
orphan block 179
Ouroboros PoS consensus mechanism, Cardano
URL 150
outbound connections 421
outbound oracle 238
outer padding (opad) 69
Out of Gas (OOG) 264
output feedback (OFB) mode 74
Over-the-Counter (OTC) 681
Oyente 644, 645
URL 645

P

P2P interface (networking) 421, 422
elements 421
P2P networks 736
PACELC theorem 8
Pacemaker 151, 154
Pact 742
PageSigner project
reference link 233
paper wallets 195
parallel computing 31
partially homomorphic encryptions (PHEs) 109
passive replication 120
Password-Based Key Derivation Function 1 (PBKDF1) 68
Paxos (PAX) 123-125
URL 475
working 126, 127
Pay2Taproot (P2TR) 206
payment channels 556
Pay-to-Public-Key Hash (P2PKH) 172
Pay-to-Script Hash (P2SH) 172

PBFT, in Hyperledger Sawtooth
reference link 134
Pedersen commitments 592, 596
Pedersen commitment scheme 111
reference link 111
Peercoin
URL 150
peers, Fabric
committing peers 457
endorsing peers 457
Peer-to-Peer (P2P) 186, 446, 507, 689, 719
protocol 455
pegged sidechains 26, 209
Peggy character 111
permissioned ledger 25
Personal Package Archives (PPAs) 344
Petersburg 348
physical unclonable functions 742
Plasma 558, 575
URL 559
vs Sidechains 559
Platform as a Service (PaaS) 721
point addition 88, 89
example 89, 90
point doubling 91
example 92
point multiplication 92
Point of Sale (POS) terminals 200
policies 454
Polkadot 549-551
Polkadot BABE 118
Polygon 576
scalability solutions 576, 577
Polygon Po 577-579
polynomial commitment scheme (PCS) 605
categories 606
polynomial commitment schemes 596
Polynomial Interactive Oracle Proof (PIOP) 605, 606
post-quantum cryptography 638
POST requests
used, for interacting with Geth 380, 381
post-trade settlement 685
PoW, alternatives 147, 148
non-outsourceable puzzles 151
Proof of Activity (PoA) 150
Proof of stake (PoS) 148-150
Proof of Storage 148
Practical Algorithm to Retrieve Information Coded in Alphanumeric (PATRICIA) 65
Practical Byzantine Fault Tolerance (PBFT) 119, 120, 123, 129, 131, 643
certificates 131
checkpointing protocol 133
checkpointing subprotocol 129
commit phase 130
Istanbul Byzantine Fault Tolerance (IBFT) 135
limitations 134
messages 132
normal operation subprotocol 129
prepare phase 130
pre-prepare phase 129
strengths 134
versus Istanbul Byzantine Fault Tolerance (IBFT) 134
view change protocol 132, 133
view change subprotocol 129
working 130
precompiled contracts 286, 287
preimage attack 636
preimage resistance 56
prime field 80
privacy
anonymity 502
confidentiality 502

private blockchains 24, 553
private data collections (PDCs), Fabric 458
private graph construction phase 591
private key 67, 80, 159, 160, 250-252
private net 282
components, requisites 284, 285
private network
creating 305-307
Geth JavaScript console, experimenting with 310, 311
initializing 307-310
transactions, mining 312-318
transactions, sending 312-318
private transaction manager 525
Proactive Market Maker (PMM) model 700
Probabilistic Polynomial Time (PPT) 596
Program Counter (PC) 273
programming languages, Ethereum blockchain
opcodes 288, 289
runtime bytecode 288
Solidity 287, 288
projects, Hyperledger 444
distributed ledgers 444
domain-specific 448
libraries 446
tools 447
Proof of Activity (PoA) 150
proof of authenticity 233
Proof of Authority (PoA) 523
Proof of Burn (PoB) 148, 209
proof of coinage 148
Proof of Concept (PoC) 498
Proof of Deposit 148
Proof of Elapsed Time (PoET) 445, 742
proof of ownership 209
proof of retrievability 148
proof of stake (PoS) 26, 46, 119, 148-150, 295, 409, 503, 553, 626
proof-of-stake (PoS) 415-421, 430
Proof of Storage 148
proof of validity 233
Proof of Work (PoW) 7, 46, 63, 119, 144, 165, 181, 182, 183, 184, 204, 295, 500
CPU-bound PoW 146
memory-bound PoW 146
working 145
Proofs of Knowledge (PoK) 447
Propose Builder Separation (PBS) 435, 437
proprietary blockchain 25
protocol messages
types 187
provable
URL 239
pseudorandom number generators (PRNGs) 55
public blockchain 24
versus enterprise blockchain 506
public key 81, 251
public key cryptography 10, 80-82, 635
brute-force attack 635
impersonation attack 635
key reuse attack 636
Man-in-the-Middle (MITM) attack 635
side-channel attack 636
Public Key Infrastructure (PKI) 108, 131, 451, 657, 736
public keys 67, 160
identifying, by prefixes 160
public-key schemes 83
puzzle-promise 590
Pycoin
URL 219

Q

quadratic arithmetic program (QAP) 115

quantum key distribution (QKD) 638

quantum-safe signature schemes 638

Quorum 515, 522
access control, with permissioning 529, 530, 531
architecture 522
cryptography 525
performance 531
pluggable architecture 542, 543
pluggable consensus 531
privacy 525-527
projects 542, 543
reference link 543

Quorum network, setting up with IBFT 532
Geth, attaching to nodes 535-537
investigating, with Geth 539-542
private transaction, running 535
Quorum Wizard, installing 532-535
transaction, viewing in Cakeshop 538

Quorum, on Azure
reference link 543

Quorum, on Kaleido
reference link 543

Quorum privacy
enclave decryption 528, 529
enclave encryption 527
transaction propagation 527

Quorum Wizard
installing 532-535

R

RACE Integrity Primitives Evaluation Message Digest (RIPEMD) 58

Radio-Frequency Identification (RFID) tags 718

Radix tree 65

Raft 467, 531

Raft protocol 127-129
subproblems 127

Raiden 557

RANDAO 426

randomized algorithms 122

random line of nodes 591

random number generators (RNGs) 55

random numbers 55
random strings, generating 55, 56

range proof 585

Rank 1 Constraint System (R1CS) 115

Raspberry Pi 722
URL 722

Raspbian
installation link 723

Realitio project
URL 239

real randomness 55

Recursive Length Prefix (RLP) 261
reference link 261

Redundant Byzantine Fault Tolerance (RBFT) 446

Reed-Solomon error correction
using 159

reentrancy bug 241, 629

reference architecture, Hyperledger 449, 450

reference types 361, 363
arrays 363
mappings 364
structs 363, 364

refund balance 267

regulatory compliance 649

Relying Parties (RPs) 655

Remix IDE 318-321, 483, 729
custom network, connecting with 325-327
URL 483

used, for interacting contract through MetaMask 336-342
using, to deploy smart contract 324
Remix plugin
reference link 542
Remote Procedure Calls (RPCs) 190, 248
replay attack 625
replicated state machine (RSM) 127, 521
replication 120
active replication 120
passive replication 120
state machine replication (SMR) 120
resistor 722
restricted private transactions 502
return on investment (ROI) 476
reverse oracle 238
reward application 277
ribbon cable connector 722
Ricardian contract, objects
code 228
parameters 228
prose 228
Ricardian contracts 225-229
properties 225
ring signatures 108, 109, 593
Ripple labs (codius) 238
RLPx 283
reference link 283
Role-Based Access Control (RBAC) 503
reference link 530
rollups 559, 560
data availability 560, 561
data validity 560
optimistic rollups 563, 564
types 563
working with 561, 562
Ropsten 354
round functions 71
RSA 83
decrypting with 85-87
encrypting with 85-87
key generation process 83
RSA digital signatures 98
authenticity 99
generating 100
non-reusability property 99
operation 99
unforgeability property 99
RSA puzzle solver 590
runtime bytecode 288

S

Sabre 447
safe curves
reference link 95
SAFE Network
reference link 45
SafetyNet
reference link 234
salt 68
sandwich attack 631
Sawtooth 444, 445, 515
reference link 445
S-boxes 71
scalability 545, 546
blockchain trilemma 546-548
categories 549
improving, methods 548
multichain solutions 549
off-chain solutions 555, 556
on-chain scaling solutions 551
Polkadot 549-551
rollup solutions 559, 560

scalability, off-chain solutions
commit chains 559
Plasma 558
Plasma chains, versus sidechains 559
sidechains 557
state channels 556, 557
sub-chains 557
tree chains 558
trusted hardware-assisted scalability 559
scalability, on-chain scaling solutions
Bitcoin-NG 554
block interval reduction 553
block propagation 553, 554
block size, increasing 552
bloXroute 551
DAG-based chains 554
faster consensus mechanisms 555
Invertible Bloom Lookup Tables 553
kadcast 551
private blockchains 553
sharding 553
transaction parallelization 552
scalability, rollup solutions
data availability 560, 561
data validity 560
example 577
fraud proof-based classification 575-577
multilayer solutions 579, 580
optimistic rollups 563, 564
optimistic rollups, versus ZK-rollups 573
Polygon PoS 577-579
types 563
usage 561, 562
validity proof-based classification 575-577
ZK-EVM 570-573
ZK-Rollups 564-566
ZK-Rollups, building technologies 566-569
ZK-ZK-rollups 573
scalability trilemma 440
scalar point multiplication 92
Schnorr signatures 205
Script 16, 170, 222
Scrypt 68
second pre-image resistance 56
secret key ciphers 69
block ciphers 71
stream ciphers 70
secret key cryptography 67
secret key (KEY) 73
secret prefix 69
secret sharing scheme 109
secret suffix 69
secure element (SE) 195
secure hash algorithms (SHAs) 57, 58
RIPEMD 58
SHA-0 57
SHA-1 58
SHA-2 58
SHA-3 58
SHA-3 (Keccak) 61, 62
SHA-256 58-60
Whirlpool 58
secure multiparty computation (SMPC) 502, 587
security analysis tools and mechanism 638, 639
formal verification 639, 640
smart contract security 644, 645
Security Assertion Markup Language (SAML) 654
security, blockchain 619-621
security protocol 54
security token offerings (STOs) 476, 477
seed 55
Segregated Witness (SegWit) 164, 202, 552
improvements 202, 203
transactions 203, 204
selective disclosure 584

self-destruct set 266
selfish mining attack 627
Self-Sovereign Identity (SSI) 658
components 659
decentralized identifiers 665-670
digital wallet 670, 671
verifiable credentials 659
semi-private blockchains 24
send fail issue 628
separation of concerns 528
Sepolia
URL 489
Sepolia test net
reference link 305
serialization 261
Serpent 344
Seth 447
SHA3-256 hash function 253
SHA-3 (Keccak) 61, 62
SHA-256 58-60
hash computation 59
messages, encrypting with 62
pre-processing 58
Shapella 440
sharding 432-440, 553
shared key cryptography 67
shared ledger 24
shared memory 119
sidechains 26, 557
side-channel attack 636
Silk Road marketplace 593
Simple Payment Verification (SPV)
clients 289
nodes 192
Simple Serialize (SSZ) 422
simple transaction 258, 264
simulator 603
Single Board Computer (SBC) 722
single-factor authentication 53
Single Sign On (SSO) 508, 654
slashing 413
Slither 645
smart contract engines 447
smart contract, oracles
hardware device-assisted proofs 234
software and network-assisted proofs 233
smart contracts 37, 42, 174, 221, 222
deploying 240, 241, 354
deploying, with MetaMask 324
deploying, with Remix IDE 324
oracles 231-233
properties 222-224
real-world application 224, 225
technology 242
testing 353
Truffle, used for testing and deploying 399-404
writing 353
smart contract security 644
smart contract services, Fabric 456
events 456
secure container 456
secure registry 456
smart contract, technology
Digital Asset Modeling Language 243-245
Solana Sealevel 242
smart contract templates 229, 230
smart contract vulnerabilities 628
integer underflow and overflow 629
oracle manipulation attack 629, 630
reentrancy 629
send fail issue 628
timestamp dependency bugs 628
unguarded selfdestruct 629
unprivileged write to storage 629

smart oracles 238
SMR problem 9
SMT (Satisfiability Modulo Theories) 641
SNARKs
types 601
soft fork 180
software and network-assisted proofs 233
TLS-N-based mechanism 233
TLSNotary 233
Software Guard Extensions (SGX) 587, 742
software oracles 236
Solana Sealevel
reference link 242
solc 287
experimenting with 345-347
installing 344, 345
used, for generating ABI and code 376, 377
Solgraph 646
URL 646
Solidity 287, 344, 354, 628, 736, 742
control structures 365, 366
data types 361
error handling 368
events 366
features 354, 355
functions 355-359
inheritance 367
libraries 367
reference link 288
variables 359
Solidity compiler 344
Solidity language 529
speed 502
sponge and squeeze construction 61
spreading phase 591
SSI-specific blockchain projects 675
AnonCreds 675
Aries 675
challenges 676, 677
Hyperledger Indy 675
initiatives 676
other projects 676
Ursa 675
SSI stack
four-layer model 671
SSL stripping 635
stable coins stability 631
stable tokens 474
algorithmically stable 475
commodity collateralized 475
crypto collateralized 475
fiat collateralized 475
stake grinding attack 627
stale block 179
standard transaction scripts 171
state and nonce validation 277
state channels 556
performing, steps 556
state machine replication (SMR) 6, 120
states, Ethereum blockchain
account state 268
world state 268
state variables 360
state variables, modifiers
constant 361
immutable 361
state variables, Tendermint 143
lockedRound 143
lockedValue 143
step 143
validRound 143
validValue 143
state variables, visibility scope
internal 360
private 360
public 360

static keys 67
status 49
reference link 49
Steemit 49
URL 49
Stem phase 591
storage root 268
stream ciphers 69, 71
operation 70
types 70
STRIDE model 647
benefits 647
implementing 648
string literals 363
structs 363, 364
structured reference string (SRS) 601
sub-chains 557
reference link 557
subnets 417
substitution-permutation network (SPN) 71
subverted approach 600
Succinct Non-Interactive Argument of Knowledge 598
suicide set 266
Swarm 283, 290, 291, 352, 736
reference link 291
Sybil attack 144, 623
symbolic execution 645
symmetric cryptography 51, 67, 68
symmetric-key schemes 83
synchronization modes
full 299
light 299
snap 299
synchrony assumptions 122
sync node 189
system model, Tendermint
network model 138
processes 138
security and cryptography 139
state machine replication 139
timing assumptions 139

T

Taproot 205
Merkelized Alternative Script Tree (MAST) 206
Pay2Taproot (P2TR) 206
Schnorr signatures 205
T-Certs 454
temporary forks 180
Tendermint 137, 138
messages 142, 143
properties 139
state transition 139
state variables 143
usage 140, 141
Tendermint Core
URL 144
Tessera 524
test nets 282
test networks
connecting to 305
Tether
reference link 45
Tether gold
URL 475
The Merge 422-431
clients 423
The Open Group Architecture Framework (TOGAF)
reference link 510
The Surge
phases 433

thin block 205
threat matrix 647
threat modeling 646, 647
threshold signatures 106, 107
timestamp 15
timestamp dependence 628
timestamp dependency bugs 628
time to live (TTL) 290
timing assumptions, consensus algorithm
asynchrony 123
partial synchrony 123
synchrony 123
TLS-N-based mechanism 233
TLSNotary 233
token engineering 495
reference link 496
tokenization
advantages 470, 471
disadvantages 472
on blockchain 470
process 475, 476
tokenized blockchains 25
tokenless blockchains 25
token offerings 476-478
decentralized autonomous initial coin offering 478
equity token offerings 477
initial coin offerings 476, 477
initial exchange offerings 477
security token offerings 477
tokenomics (token economics) 495
tokens 469
fungible tokens 473
non-fungible tokens (NFTs) 473
security tokens 475
stable tokens 474
taxonomy 496
types 473
token standards 479
Token Taxonomy Framework (TTF) 496
reference link 496
tools, Hyperledger project 447
Caliper 448
Cello 448
Tor 586
total order broadcast 121
touched accounts 267
Town Crier
URL 239
trade lifecycle
steps 685
trading instruments 683
traditional consensus
properties 146
versus Nakamoto consensus 145
Traditional Finance (TradFi) 690
Transact 444, 447
reference link 447
transaction execution 266
transaction families 445
transaction flow, Fabric
steps 463, 464
transaction manager 524, 527
transaction order dependence 630
transaction ordering dependency bug 628
transaction parallelization 552
transaction pools 279
transaction receipts 269, 270
transaction replay attack 625, 626
transactions 16, 255, 264, 279, 458
components 257-260
contract creation transactions 258, 264
message call transactions 258, 265
simple transactions 258, 264
transactions per second (TPS) 531, 549

transactions, SegWit
P2SH-P2WPKH 204
P2SH-P2WSH 204
P2WPKH 203
P2WSH 204
transaction substate 266
transaction trie 260
transaction validation 266, 277
transparent setups 600
transparent SNARKs 601
Transport Layer Security (TLS) 233, 732
tree chains 558
Trezor
URL 195
Triple DES (3DES) 74
TrueBit
URL 239
Truffle 351-353, 732
installing and initializing 392, 393
URL 351
used, for compiling contracts 393-397
used, for deploying with contracts 391, 392
used, for interacting with contracts 391, 392
used, for migrating contracts 393-397
used, for testing contracts 393-397
used, for testing smart contracts 399-404
used, for deploying smart contracts 399-404
Trusted Execution Environment (TEE) 234, 445, 508, 588
trusted hardware-assisted confidentiality 587
trusted hardware-assisted proofs 235, 236
trusted hardware-assisted scalability 559
trusted model 600
trusted non-universal setup 601
trusted universal setup 601
T-shaped cobbler 722
TumbleBit 590
phases 590
tumbler 590
two-phase commit (2PC) 125
two-way peg 209, 557
two-way pegged sidechain 26
tx.origin 630
typical Bitcoin addresses 161, 163

U

UK Jurisdiction Taskforce (UKJT) 224
unauthorized key sharing 637
uncle block 274
unconditional privacy 584
unguarded selfdestruct 629
uniform reference string (URS) 600
Uniform Resource Identifier (URI) 200, 662
unrestricted private transactions 502
unsolvability results 121
Unspent Transaction Output (UTXO) 169, 203
Ursa 444, 447
reference link 447
USDT (Tether)
URL 475

V

validator node 411, 412
status 412
validator nodes
versus Beacon Chain nodes 414
validium (validia) 575
Value-Added Tax (VAT) 743
value types 361
address 362
Boolean 361

integers 361, 362
literals 363
variables, Solidity 359
global variables 359, 360
local variables 359
state variables 360
Verifiable Credentials (VCs) 659
architecture, components 660
benefits 659
ecosystem 662
structure 662-664
verifiable presentation 665
Verifiable Data Registries (VDRs) 661, 671
verifiable presentation (VP) 665
verifiable random function (VRF) 117, 150
URL 150
view change protocol 132, 133
virtual machine 17
virtual mining 148
virtual read-only memory (virtual ROM) 271
Visual Studio Code 353
VMware Blockchain (VMBC) 518
architecture 520, 521
components 519
consensus protocol 519
for Ethereum 522
reference link 521
VR headsets 4
Vyper 344

W

Waffle 353
reference link 353
Wallet Import Format (WIF) 159
wallets 289
weak collision resistance 56
Web3
identity 672-675
used, for interacting with contracts 371, 372
web3.js JavaScript library
installing 382, 383
web3 object
creating 383
WebAssembly (Wasm) 272
Web evolution, reviewing
Web 1 49
Web 2 49
Web 3 49
web layer 48
Weierstrass equation 87
Whisper 283, 290, 291, 352
reference link 290
whistleblowing validator 413
whitepapers, Hyperledger
reference link 449
Why3 644
Wired Equivalent Privacy (WEP) 636
Wireshark
URL 191
witness 603
witnet
URL 239
world computer 407
World of Accountancy 227
World of Law 227
world state 268
database 457, 458
world state trie 268

X

XOR (exclusive OR) 61

Y

yield farming 703

YUL
reference link 272

Z

Zcash 110, 113
URL 21, 111

zero-knowledge proofs (ZKPs) 111, 112, 502, 584, 597-601, 737, 741
Ali Baba's Cave analogy 111
challenge phase 113
completeness property 111
response phase 113
soundness property 111
witness phase 113
zero-knowledge property 111
zero-knowledge range proofs (ZKRPs) 116
zk-SNARKs 113-116

zero-knowledge range proofs (ZKRPs) 116

zero-knowledge Succinct Transparent Argument of Knowledge (zk-STARK) 113, 598
building 601-607
commitment 607
evaluation 608
evaluation proof, verifying 608, 609
limitation 115
proof generation 115
properties 114
setup 607
versus, zk-SNARKs 116

zero-knowledge (ZK) rollups 564-566
cons 569
technologies, used for building 566-569

Zether 594

ZK-EVM 570-572
categories 572
reference link 573
types 572, 573

Zkledger 593

ZKP-related attacks
attacks on privacy 637
digital signature vulnerabilities 637
inadequate bit security 637
proof malleability 638
quantum threats 637
setup vulnerabilities 638

ZK-rollups 575
vs optimistic rollups 573

zk-SNARK construction
arithmetic circuit 115
QAP 115
R1CS 115

ZKSpace 577

ZKSwap 577

zkSync 577

ZK-ZK-rollups 573

Zooko's Triangle 42

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



`https://packt.link/free-ebook/9781803241067`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly