

Mastering Bitcoin

Table of Contents

Preface	1
Writing the Bitcoin Book	1
Intended Audience	1
Why Are There Bugs on the Cover?.....	1
Conventions Used in This Book	2
Code Examples	2
Using Code Examples	3
Safari® Books Online	3
How to Contact Us	3
Acknowledgments	4
Quick Glossary	7
Introduction	9
What Is Bitcoin?.....	9
History of Bitcoin.....	12
Bitcoin Uses, Users, and Their Stories	12
Getting Started	13
How Bitcoin Works	21
Transactions, Blocks, Mining, and the Blockchain	21
Bitcoin Transactions	24
Constructing a Transaction.....	27
Bitcoin Mining	31
Mining Transactions in Blocks	32
Spending the Transaction	34
The Bitcoin Client.....	36
Bitcoin Core: The Reference Implementation	36
Using Bitcoin Core’s JSON-RPC API from the Command Line.....	44
Alternative Clients, Libraries, and Toolkits.....	55
Keys, Addresses, Wallets	61
Introduction	61
Bitcoin Addresses	69
Implementing Keys and Addresses in Python	82
Wallets	86
Advanced Keys and Addresses	97
Transactions	108
Introduction	108
Transaction Lifecycle	108
Transaction Structure	110
Transaction Outputs and Inputs	110

Transaction Chaining and Orphan Transactions	119
Transaction Scripts and Script Language	120
Standard Transactions	125
The Bitcoin Network	135
Peer-to-Peer Network Architecture.....	135
Nodes Types and Roles.....	135
The Extended Bitcoin Network	137
Network Discovery	140
Full Nodes	144
Exchanging "Inventory".....	144
Simplified Payment Verification (SPV) Nodes	145
Bloom Filters	149
Bloom Filters and Inventory Updates	153
Transaction Pools	153
Alert Messages	154
The Blockchain	156
Introduction	156
Structure of a Block	157
Block Header	157
Block Identifiers: Block Header Hash and Block Height	158
The Genesis Block	159
Linking Blocks in the Blockchain	160
Merkle Trees.....	161
Merkle Trees and Simplified Payment Verification (SPV).....	168
Mining and Consensus	169
Introduction	169
Decentralized Consensus.....	172
Independent Verification of Transactions	173
Mining Nodes	174
Aggregating Transactions into Blocks	175
Constructing the Block Header	182
Mining the Block	184
Successfully Mining the Block	193
Validating a New Block	193
Assembling and Selecting Chains of Blocks	194
Mining and the Hashing Race	199
Consensus Attacks.....	205
Alternative Chains, Currencies, and Applications	208
A Taxonomy of Alternative Currencies and Chains	208
Meta Coin Platforms	209
Alt Coins.....	211

Noncurrency Alt Chains.....	217
Future of Currencies.....	220
Bitcoin Security	221
Security Principles	221
User Security Best Practices	223
Conclusion.....	225
Appendix A: Transaction Script Language Operators, Constants, and Symbols	226
Appendix B: Bitcoin Improvement Proposals.....	233
Appendix C: pycoin, ku, and tx.....	240
Key Utility (KU)	240
Appendix D: Bitcoin Explorer (bx) Commands.....	252
Examples of bx command use	253

Preface

Writing the Bitcoin Book

I first stumbled upon bitcoin in mid-2011. My immediate reaction was more or less "Pfft! Nerd money!" and I ignored it for another six months, failing to grasp its importance. This is a reaction that I have seen repeated among many of the smartest people I know, which gives me some consolation. The second time I came across bitcoin, in a mailing list discussion, I decided to read the white paper written by Satoshi Nakamoto, to study the authoritative source and see what it was all about. I still remember the moment I finished reading those nine pages, when I realized that bitcoin was not simply a digital currency, but a network of trust that could also provide the basis for so much more than just currencies. The realization that "this isn't money, it's a decentralized trust network," started me on a four-month journey to devour every scrap of information about bitcoin I could find. I became obsessed and enthralled, spending 12 or more hours each day glued to a screen, reading, writing, coding, and learning as much as I could. I emerged from this state of fugue, more than 20 pounds lighter from lack of consistent meals, determined to dedicate myself to working on bitcoin.

Two years later, after creating a number of small startups to explore various bitcoin-related services and products, I decided that it was time to write my first book. Bitcoin was the topic that had driven me into a frenzy of creativity and consumed my thoughts; it was the most exciting technology I had encountered since the Internet. It was now time to share my passion about this amazing technology with a broader audience.

Intended Audience

This book is mostly intended for coders. If you can use a programming language, this book will teach you how cryptographic currencies work, how to use them, and how to develop software that works with them. The first few chapters are also suitable as an in-depth introduction to bitcoin for noncoders—those trying to understand the inner workings of bitcoin and cryptocurrencies.

Why Are There Bugs on the Cover?

The leafcutter ant is a species that exhibits highly complex behavior in a colony super-organism, but each individual ant operates on a set of simple rules driven by social interaction and the exchange of chemical scents (pheromones). Per Wikipedia: "Next to humans, leafcutter ants form the largest and most complex animal societies on Earth." Leafcutter ants don't actually eat leaves, but rather use them to farm a fungus, which is the central food source for the colony. Get that? These ants are farming!

Although ants form a caste-based society and have a queen for producing offspring, there is no central authority or leader in an ant colony. The highly intelligent and sophisticated behavior exhibited by a multimillion-member colony is an emergent property from the interaction of the individuals in a social network.

Nature demonstrates that decentralized systems can be resilient and can produce emergent complexity and incredible sophistication without the need for a central authority, hierarchy, or

complex parts.

Bitcoin is a highly sophisticated decentralized trust network that can support a myriad of financial processes. Yet, each node in the bitcoin network follows a few simple mathematical rules. The interaction between many nodes is what leads to the emergence of the sophisticated behavior, not any inherent complexity or trust in any single node. Like an ant colony, the bitcoin network is a resilient network of simple nodes following simple rules that together can do amazing things without any central coordination.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.

TIP | This icon signifies a tip, suggestion, or general note.

WARNING | This icon indicates a warning or caution.

Code Examples

The examples are illustrated in Python, C++, and using the command line of a Unix-like operating system such as Linux or Mac OS X. All code snippets are available in the [GitHub repository](#) in the *code* subdirectory of the main repo. Fork the book code, try the code examples, or submit corrections via GitHub.

All the code snippets can be replicated on most operating systems with a minimal installation of compilers and interpreters for the corresponding languages. Where necessary, we provide basic installation instructions and step-by-step examples of the output of those instructions.

Some of the code snippets and code output have been reformatted for print. In all such cases, the lines have been split by a backslash (\) character, followed by a newline character. When transcribing the examples, remove those two characters and join the lines again and you should see identical results as shown in the example.

All the code snippets use real values and calculations where possible, so that you can build from example to example and see the same results in any code you write to calculate the same values. For example, the private keys and corresponding public keys and addresses are all real. The sample transactions, blocks, and blockchain references have all been introduced in the actual bitcoin blockchain and are part of the public ledger, so you can review them on any bitcoin system.

Using Code Examples

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Mastering Bitcoin* by Andreas M. Antonopoulos (O'Reilly). Copyright 2015 Andreas M. Antonopoulos, 978-1-449-37404-4."

Some editions of this book are offered under an open source license, such as CC-BY-NC (creativecommons.org), in which case the terms of that license apply.

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online

NOTE

Safari Books Online is an on-demand digital library that delivers expert [content](#) in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/mastering_bitcoin.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

This book represents the efforts and contributions of many people. I am grateful for all the help I received from friends, colleagues, and even complete strangers, who joined me in this effort to write the definitive technical book on cryptocurrencies and bitcoin.

It is impossible to make a distinction between the bitcoin technology and the bitcoin community, and this book is as much a product of that community as it is a book on the technology. My work on this book was encouraged, cheered on, supported, and rewarded by the entire bitcoin community from the very beginning until the very end. More than anything, this book has allowed me to be part of a wonderful community for two years and I can't thank you enough for accepting me into this community. There are far too many people to mention by name—people I've met at conferences, events, seminars, meetups, pizza gatherings, and small private gatherings, as well as many who communicated with me by Twitter, on reddit, on bitcointalk.org, and on GitHub who have had an impact on this book. Every idea, analogy, question, answer, and explanation you find in this book was at some point inspired, tested, or improved through my interactions with the community. Thank you all for your support; without you this book would not have happened. I am forever grateful.

The journey to becoming an author starts long before the first book, of course. My first language (and schooling) was Greek, so I had to take a remedial English writing course in my first year of university. I owe thanks to Diana Kordas, my English writing teacher, who helped me build confidence and skills that year. Later, as a professional, I developed my technical writing skills on the topic of data centers, writing for *Network World* magazine. I owe thanks to John Dix and John Gallant, who gave me my first writing job as a columnist at *Network World* and to my editor Michael Cooney and my colleague Johna Till Johnson who edited my columns and made them fit for publication. Writing 500 words a week for four years gave me enough experience to eventually consider becoming an author. Thanks to Jean de Vera for her early encouragement to become an author and for always believing and insisting that I had a book in me.

Thanks also to those who supported me when I submitted my book proposal to O'Reilly, by providing references and reviewing the proposal. Specifically, thanks to John Gallant, Gregory Ness, Richard Stiennon, Joel Snyder, Adam B. Levine, Sandra Gittlen, John Dix, Johna Till Johnson, Roger Ver, and Jon Matonis. Special thanks to Richard Kagan and Tymon Mattoszk, who reviewed early versions of the proposal and Matthew Owain Taylor, who copyedited the proposal.

Thanks to Cricket Liu, author of the O'Reilly title *DNS and BIND*, who introduced me to O'Reilly. Thanks also to Michael Loukides and Allyson MacDonald at O'Reilly, who worked for months to help make this book happen. Allyson was especially patient when deadlines were missed and deliverables delayed as life intervened in our planned schedule.

The first few drafts of the first few chapters were the hardest, because bitcoin is a difficult subject to unravel. Every time I pulled on one thread of the bitcoin technology, I had to pull in the whole

thing. I repeatedly got stuck and a bit despondent as I struggled to make the topic easy to understand and create a narrative around such a dense technical subject. Eventually, I decided to tell the story of bitcoin through the stories of the people using bitcoin and the whole book became a lot easier to write. I owe thanks to my friend and mentor, Richard Kagan, who helped me unravel the story and get past the moments of writer's block, and Pamela Morgan, who reviewed early drafts of each chapter and asked the hard questions to make them better. Also, thanks to the developers of the San Francisco Bitcoin Developers Meetup group and Taariq Lewis, the group's co-founder, for helping to test the early material.

During the development of the book, I made early drafts available on GitHub and invited public comments. More than a hundred comments, suggestions, corrections, and contributions were submitted in response. Those contributions are explicitly acknowledged, with my thanks, in [Early Release Draft \(GitHub Contributions\)](#). Special thanks to Minh T. Nguyen, who volunteered to manage the GitHub contributions and added many significant contributions himself. Thanks also to Andrew Naugler for infographic design.

Once the book was drafted, it went through several rounds of technical review. Thanks to Cricket Liu and Lorne Lantz for their thorough review, comments, and support.

Several bitcoin developers contributed code samples, reviews, comments, and encouragement. Thanks to Amir Taaki and Eric Voskuil for example code snippets and many great comments; Vitalik Buterin and Richard Kiss for help with elliptic curve math and code contributions; Gavin Andresen for corrections, comments, and encouragement; Michalis Kargakis for comments, contributions, and btcd writeup; and Robin Inge for errata submissions improving the second print.

I owe my love of words and books to my mother, Theresa, who raised me in a house with books lining every wall. My mother also bought me my first computer in 1982, despite being a self-described technophobe. My father, Menelaos, a civil engineer who just published his first book at 80 years old, was the one who taught me logical and analytical thinking and a love of science and engineering.

Thank you all for supporting me throughout this journey.

Early Release Draft (GitHub Contributions)

Many contributors offered comments, corrections, and additions to the early-release draft on GitHub. Thank you all for your contributions to this book. Following is a list of notable GitHub contributors, including their GitHub ID in parentheses:

- Minh T. Nguyen, GitHub contribution editor (enderminh)
- Ed Eykholt (edeykholt)
- Michalis Kargakis (kargakis)
- Erik Wahlström (erikwam)
- Richard Kiss (richardkiss)
- Eric Winchell (winchell)
- Sergej Kotliar (ziggamon)
- Nagaraj Hubli (nagarajhubli)

- ethers
- Alex Waters (alexwaters)
- Mihail Russu (MihailRussu)
- Ish Ot Jr. (ishotjr)
- James Addison (jayaddison)
- Nekomata (nekomata-3)
- Simon de la Rouviere (simondlr)
- Chapman Shoop (belovachap)
- Holger Schinzel (schinzelh)
- effectsToCause (vericoin)
- Stephan Oeste (Emzy)
- Joe Bauers (joebauers)
- Jason Bisterfeldt (jbisterfeldt)
- Ed Leafe (EdLeafe)

Quick Glossary

This quick glossary contains many of the terms used in relation to bitcoin. These terms are used throughout the book, so bookmark this for a quick reference.

address

A bitcoin address looks like `1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV`. It consists of a string of letters and numbers starting with a "1" (number one). Just like you ask others to send an email to your email address, you would ask others to send you bitcoin to your bitcoin address.

bip

Bitcoin Improvement Proposals. A set of proposals that members of the bitcoin community have submitted to improve bitcoin. For example, BIP0021 is a proposal to improve the bitcoin uniform resource identifier (URI) scheme.

bitcoin

The name of the currency unit (the coin), the network, and the software.

block

A grouping of transactions, marked with a timestamp, and a fingerprint of the previous block. The block header is hashed to produce a proof of work, thereby validating the transactions. Valid blocks are added to the main blockchain by network consensus.

blockchain

A list of validated blocks, each linking to its predecessor all the way to the genesis block.

confirmations

Once a transaction is included in a block, it has one confirmation. As soon as *another* block is mined on the same blockchain, the transaction has two confirmations, and so on. Six or more confirmations is considered sufficient proof that a transaction cannot be reversed.

difficulty

A network-wide setting that controls how much computation is required to produce a proof of work.

difficulty target

A difficulty at which all the computation in the network will find blocks approximately every 10 minutes.

difficulty retargeting

A network-wide recalculation of the difficulty that occurs once every 2,106 blocks and considers the hashing power of the previous 2,106 blocks.

fees

The sender of a transaction often includes a fee to the network for processing the requested transaction. Most transactions require a minimum fee of 0.5 mBTC.

hash

A digital fingerprint of some binary input.

genesis block

The first block in the blockchain, used to initialize the cryptocurrency.

miner

A network node that finds valid proof of work for new blocks, by repeated hashing.

network

A peer-to-peer network that propagates transactions and blocks to every bitcoin node on the network.

Proof-Of-Work

A piece of data that requires significant computation to find. In bitcoin, miners must find a numeric solution to the SHA256 algorithm that meets a network-wide target, the difficulty target.

reward

An amount included in each new block as a reward by the network to the miner who found the Proof-Of-Work solution. It is currently 25BTC per block.

secret key (aka private key)

The secret number that unlocks bitcoins sent to the corresponding address. A secret key looks like `5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh`.

transaction

In simple terms, a transfer of bitcoins from one address to another. More precisely, a transaction is a signed data structure expressing a transfer of value. Transactions are transmitted over the bitcoin network, collected by miners, and included into blocks, made permanent on the blockchain.

wallet

Software that holds all your bitcoin addresses and secret keys. Use it to send, receive, and store your bitcoin.

Introduction

What Is Bitcoin?

Bitcoin is a collection of concepts and technologies that form the basis of a digital money ecosystem. Units of currency called bitcoins are used to store and transmit value among participants in the bitcoin network. Bitcoin users communicate with each other using the bitcoin protocol primarily via the Internet, although other transport networks can also be used. The bitcoin protocol stack, available as open source software, can be run on a wide range of computing devices, including laptops and smartphones, making the technology easily accessible.

Users can transfer bitcoins over the network to do just about anything that can be done with conventional currencies, including buy and sell goods, send money to people or organizations, or extend credit. Bitcoins can be purchased, sold, and exchanged for other currencies at specialized currency exchanges. Bitcoin in a sense is the perfect form of money for the Internet because it is fast, secure, and borderless.

Unlike traditional currencies, bitcoins are entirely virtual. There are no physical coins or even digital coins per se. The coins are implied in transactions that transfer value from sender to recipient. Users of bitcoin own keys that allow them to prove ownership of transactions in the bitcoin network, unlocking the value to spend it and transfer it to a new recipient. Those keys are often stored in a digital wallet on each user's computer. Possession of the key that unlocks a transaction is the only prerequisite to spending bitcoins, putting the control entirely in the hands of each user.

Bitcoin is a distributed, peer-to-peer system. As such there is no "central" server or point of control. Bitcoins are created through a process called "mining," which involves competing to find solutions to a mathematical problem while processing bitcoin transactions. Any participant in the bitcoin network (i.e., anyone using a device running the full bitcoin protocol stack) may operate as a miner, using their computer's processing power to verify and record transactions. Every 10 minutes on average, someone is able to validate the transactions of the past 10 minutes and is rewarded with brand new bitcoins. Essentially, bitcoin mining decentralizes the currency-issuance and clearing functions of a central bank and replaces the need for any central bank with this global competition.

The bitcoin protocol includes built-in algorithms that regulate the mining function across the network. The difficulty of the processing task that miners must perform—to successfully record a block of transactions for the bitcoin network—is adjusted dynamically so that, on average, someone succeeds every 10 minutes regardless of how many miners (and CPUs) are working on the task at any moment. The protocol also halves the rate at which new bitcoins are created every four years, and limits the total number of bitcoins that will be created to a fixed total of 21 million coins. The result is that the number of bitcoins in circulation closely follows an easily predictable curve that reaches 21 million by the year 2140. Due to bitcoin's diminishing rate of issuance, over the long term, the bitcoin currency is deflationary. Furthermore, bitcoin cannot be inflated by "printing" new money above and beyond the expected issuance rate.

Behind the scenes, bitcoin is also the name of the protocol, a network, and a distributed computing innovation. The bitcoin currency is really only the first application of this invention. As a developer, I see bitcoin as akin to the Internet of money, a network for propagating value and

securing the ownership of digital assets via distributed computation. There's a lot more to bitcoin than first meets the eye.

In this chapter we'll get started by explaining some of the main concepts and terms, getting the necessary software, and using bitcoin for simple transactions. In following chapters we'll start unwrapping the layers of technology that make bitcoin possible and examine the inner workings of the bitcoin network and protocol.

Digital Currencies Before Bitcoin

The emergence of viable digital money is closely linked to developments in cryptography. This is not surprising when one considers the fundamental challenges involved with using bits to represent value that can be exchanged for goods and services. Two basic questions for anyone accepting digital money are:

1. Can I trust the money is authentic and not counterfeit?
2. Can I be sure that no one else can claim that this money belongs to them and not me? (Aka the “double-spend” problem.)

Issuers of paper money are constantly battling the counterfeiting problem by using increasingly sophisticated papers and printing technology. Physical money addresses the double-spend issue easily because the same paper note cannot be in two places at once. Of course, conventional money is also often stored and transmitted digitally. In these cases, the counterfeiting and double-spend issues are handled by clearing all electronic transactions through central authorities that have a global view of the currency in circulation. For digital money, which cannot take advantage of esoteric inks or holographic strips, cryptography provides the basis for trusting the legitimacy of a user’s claim to value. Specifically, cryptographic digital signatures enable a user to sign a digital asset or transaction proving the ownership of that asset. With the appropriate architecture, digital signatures also can be used to address the double-spend issue.

When cryptography started becoming more broadly available and understood in the late 1980s, many researchers began trying to use cryptography to build digital currencies. These early digital currency projects issued digital money, usually backed by a national currency or precious metal such as gold.

Although these earlier digital currencies worked, they were centralized and, as a result, they were easy to attack by governments and hackers. Early digital currencies used a central clearinghouse to settle all transactions at regular intervals, just like a traditional banking system. Unfortunately, in most cases these nascent digital currencies were targeted by worried governments and eventually litigated out of existence. Some failed in spectacular crashes when the parent company liquidated abruptly. To be robust against intervention by antagonists, whether legitimate governments or criminal elements, a decentralized digital currency was needed to avoid a single point of attack. Bitcoin is such a system, completely decentralized by design, and free of any central authority or point of control that can be attacked or corrupted.

Bitcoin represents the culmination of decades of research in cryptography and distributed systems and includes four key innovations brought together in a unique and powerful combination. Bitcoin consists of:

- A decentralized peer-to-peer network (the bitcoin protocol)
- A public transaction ledger (the blockchain)
- A decentralized mathematical and deterministic currency issuance (distributed mining)
- A decentralized transaction verification system (transaction script)

History of Bitcoin

Bitcoin was invented in 2008 with the publication of a paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System," written under the alias of Satoshi Nakamoto. Nakamoto combined several prior inventions such as b-money and HashCash to create a completely decentralized electronic cash system that does not rely on a central authority for currency issuance or settlement and validation of transactions. The key innovation was to use a distributed computation system (called a "proof-of-work" algorithm) to conduct a global "election" every 10 minutes, allowing the decentralized network to arrive at *consensus* about the state of transactions. This elegantly solves the issue of double-spend where a single currency unit can be spent twice. Previously, the double-spend problem was a weakness of digital currency and was addressed by clearing all transactions through a central clearinghouse.

The bitcoin network started in 2009, based on a reference implementation published by Nakamoto and since revised by many other programmers. The distributed computation that provides security and resilience for bitcoin has increased exponentially, and now exceeds that combined processing capacity of the world's top super-computers. Bitcoin's total market value is estimated at between 5 billion and 10 billion US dollars, depending on the bitcoin-to-dollar exchange rate. The largest transaction processed so far by the network was 150 million US dollars, transmitted instantly and processed without any fees.

Satoshi Nakamoto withdrew from the public in April of 2011, leaving the responsibility of developing the code and network to a thriving group of volunteers. The identity of the person or people behind bitcoin is still unknown. However, neither Satoshi Nakamoto nor anyone else exerts control over the bitcoin system, which operates based on fully transparent mathematical principles. The invention itself is groundbreaking and has already spawned new science in the fields of distributed computing, economics, and econometrics.

A Solution to a Distributed Computing Problem

Satoshi Nakamoto's invention is also a practical and novel solution to a problem in distributed computing, known as the "Byzantine Generals' Problem." Briefly, the problem consists of trying to agree on a course of action by exchanging information over an unreliable and potentially compromised network. Satoshi Nakamoto's solution, which uses the concept of proof-of-work to achieve consensus without a central trusted authority, represents a breakthrough in distributed computing science and has wide applicability beyond currency. It can be used to achieve consensus on decentralized networks to prove the fairness of elections, lotteries, asset registries, digital notarization, and more.

Bitcoin Uses, Users, and Their Stories

Bitcoin is a technology, but it expresses money that is fundamentally a language for exchanging value between people. Let's look at the people who are using bitcoin and some of the most common uses of the currency and protocol through their stories. We will reuse these stories throughout the book to illustrate the real-life uses of digital money and how they are made possible by the various technologies that are part of bitcoin.

North American low-value retail

Alice lives in Northern California's Bay Area. She has heard about bitcoin from her techie friends and wants to start using it. We will follow her story as she learns about bitcoin, acquires some, and then spends some of her bitcoin to buy a cup of coffee at Bob's Cafe in Palo Alto. This story will introduce us to the software, the exchanges, and basic transactions from the perspective of a retail consumer.

North American high-value retail

Carol is an art gallery owner in San Francisco. She sells expensive paintings for bitcoin. This story will introduce the risks of a "51%" consensus attack for retailers of high-value items.

Offshore contract services

Bob, the cafe owner in Palo Alto, is building a new website. He has contracted with an Indian web developer, Gopesh, who lives in Bangalore, India. Gopesh has agreed to be paid in bitcoin. This story will examine the use of bitcoin for outsourcing, contract services, and international wire transfers.

Charitable donations

Eugenia is the director of a children's charity in the Philippines. Recently she has discovered bitcoin and wants to use it to reach a whole new group of foreign and domestic donors to fundraise for her charity. She's also investigating ways to use bitcoin to distribute funds quickly to areas of need. This story will show the use of bitcoin for global fundraising across currencies and borders and the use of an open ledger for transparency in charitable organizations.

Import/export

Mohammed is an electronics importer in Dubai. He's trying to use bitcoin to buy electronics from the US and China for import into the UAE to accelerate the process of payments for imports. This story will show how bitcoin can be used for large business-to-business international payments tied to physical goods.

Mining for bitcoin

Jing is a computer engineering student in Shanghai. He has built a "mining" rig to mine for bitcoins, using his engineering skills to supplement his income. This story will examine the "industrial" base of bitcoin: the specialized equipment used to secure the bitcoin network and issue new currency.

Each of these stories is based on real people and real industries that are currently using bitcoin to create new markets, new industries, and innovative solutions to global economic issues.

Getting Started

To join the bitcoin network and start using the currency, all a user has to do is download an application or use a web application. Because bitcoin is a standard, there are many implementations of the bitcoin client software. There is also a reference implementation, also known as the Satoshi client, which is managed as an open source project by a team of developers and is derived from the original implementation written by Satoshi Nakamoto.

The three main forms of bitcoin clients are:

Full client

A full client, or "full node," is a client that stores the entire history of bitcoin transactions (every transaction by every user, ever), manages the users' wallets, and can initiate transactions directly on the bitcoin network. This is similar to a standalone email server, in that it handles all aspects of the protocol without relying on any other servers or third-party services.

Lightweight client

A lightweight client stores the user's wallet but relies on third-party-owned servers for access to the bitcoin transactions and network. The light client does not store a full copy of all transactions and therefore must trust the third-party servers for transaction validation. This is similar to a standalone email client that connects to a mail server for access to a mailbox, in that it relies on a third party for interactions with the network.

Web client

Web clients are accessed through a web browser and store the user's wallet on a server owned by a third party. This is similar to webmail in that it relies entirely on a third-party server.

Mobile Bitcoin

Mobile clients for smartphones, such as those based on the Android system, can either operate as full clients, lightweight clients, or web clients. Some mobile clients are synchronized with a web or desktop client, providing a multiplatform wallet across multiple devices but with a common source of funds.

The choice of bitcoin client depends on how much control the user wants over funds. A full client will offer the highest level of control and independence for the user, but in turn puts the burden of backups and security on the user. On the other end of the range of choices, a web client is the easiest to set up and use, but the trade-off with a web client is that counterparty risk is introduced because security and control is shared with the user and the owner of the web service. If a web-wallet service is compromised, as many have been, the users can lose all their funds. Conversely, if users have a full client without adequate backups, they might lose their funds through a computer mishap.

For the purposes of this book, we will be demonstrating the use of a variety of downloadable bitcoin clients, from the reference implementation (the Satoshi client) to web wallets. Some of the examples will require the use of the reference client, which, in addition to being a full client, also exposes APIs to the wallet, network, and transaction services. If you are planning to explore the programmatic interfaces into the bitcoin system, you will need the reference client.

Quick Start

Alice, who we introduced in [Bitcoin Uses, Users, and Their Stories](#), is not a technical user and only recently heard about bitcoin from a friend. She starts her journey by visiting the official website bitcoin.org, where she finds a broad selection of bitcoin clients. Following the advice on the bitcoin.org site, she chooses the lightweight bitcoin client Multibit.

Alice follows a link from the bitcoin.org site to download and install Multibit on her desktop.

Multibit is available for Windows, Mac OS, and Linux desktops.

WARNING

A bitcoin wallet must be protected by a password or passphrase. There are many bad actors attempting to break weak passwords, so take care to select one that cannot be easily broken. Use a combination of upper and lowercase characters, numbers, and symbols. Avoid personal information such as birth dates or names of sports teams. Avoid any words commonly found in dictionaries, in any language. If you can, use a password generator to create a completely random password that is at least 12 characters in length. Remember: bitcoin is money and can be instantly moved anywhere in the world. If it is not well protected, it can be easily stolen.

Once Alice has downloaded and installed the Multibit application, she runs it and is greeted by a Welcome screen, as shown in [The Multibit bitcoin client Welcome screen](#).

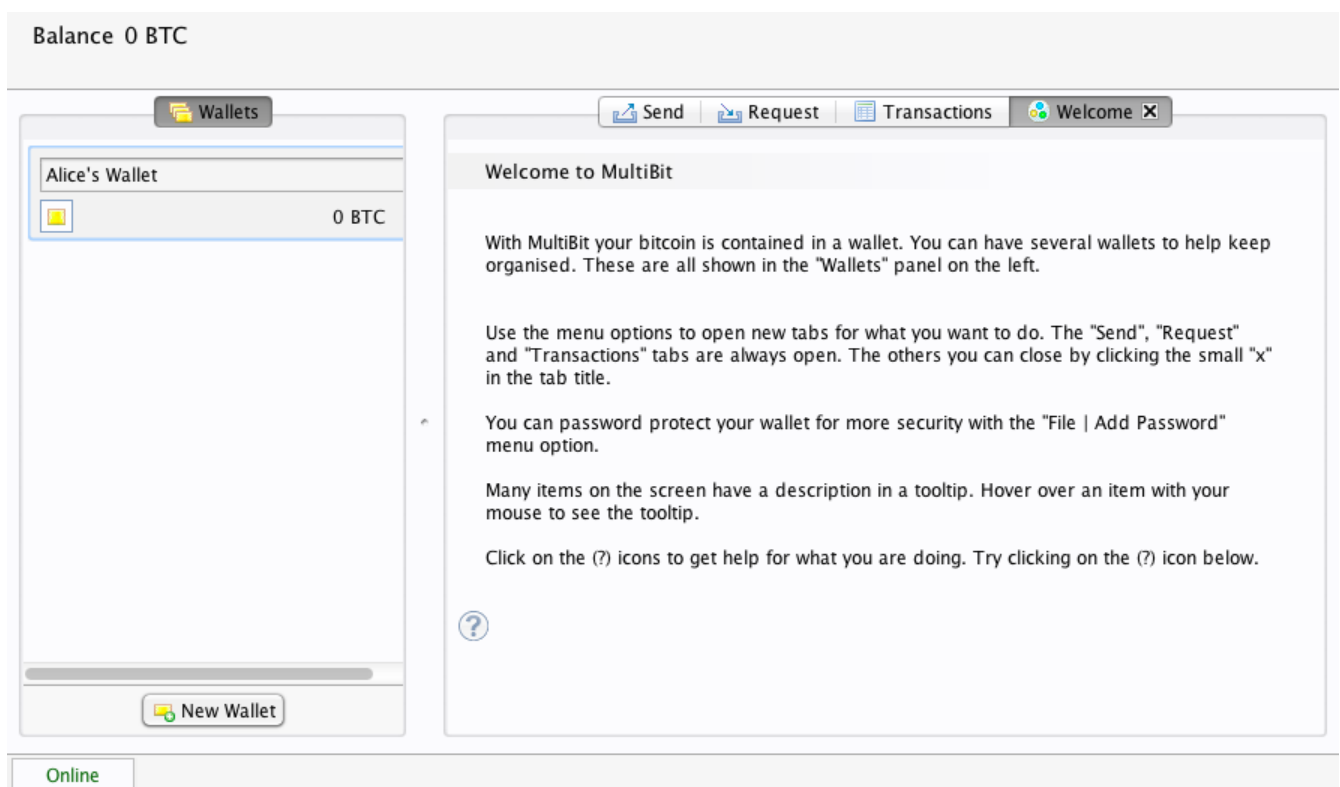


Figure 1. The Multibit bitcoin client Welcome screen

Multibit automatically creates a wallet and a new bitcoin address for Alice, which Alice can see by clicking the Request tab shown in [Alice's new bitcoin address, in the Request tab of the Multibit client](#).

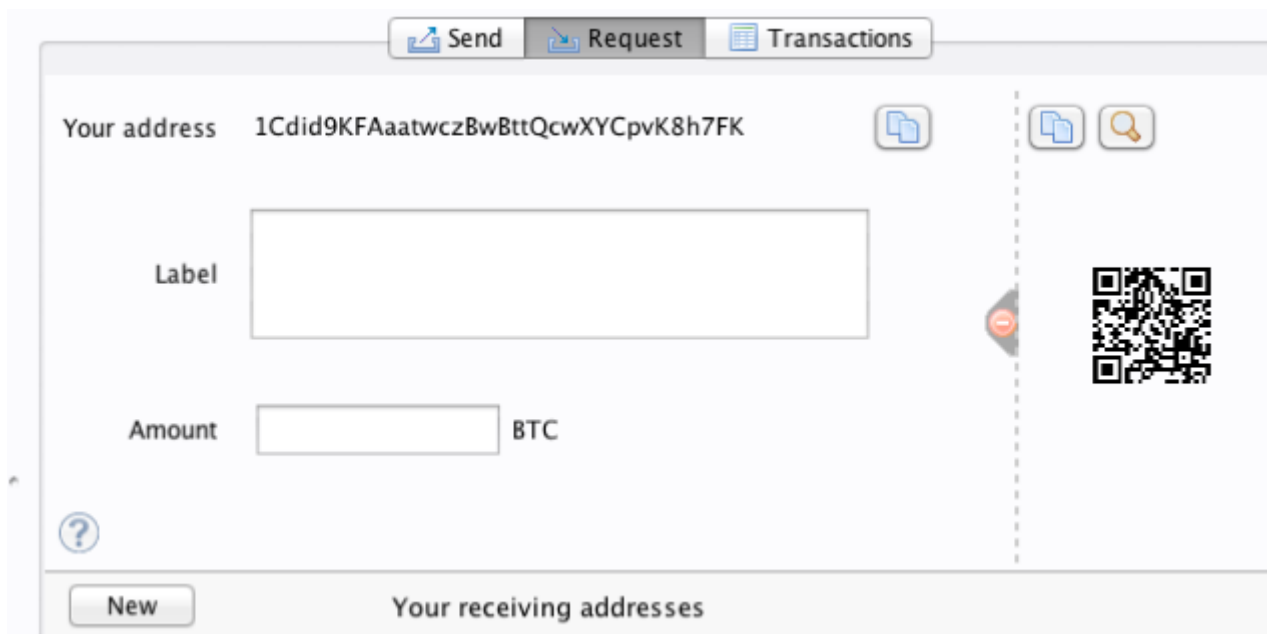


Figure 2. Alice's new bitcoin address, in the Request tab of the Multibit client

The most important part of this screen is Alice's *bitcoin address*. Like an email address, Alice can share this address and anyone can use it to send money directly to her new wallet. On the screen it appears as a long string of letters and numbers: `1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK`. Next to the wallet's bitcoin address is a QR code, a form of barcode that contains the same information in a format that can be scanned by a smartphone camera. The QR code is the black-and-white square on the right side of the window. Alice can copy the bitcoin address or the QR code onto her clipboard by clicking the copy button adjacent to each of them. Clicking the QR code itself will magnify it, so that it can be easily scanned by a smartphone camera.

Alice can also print the QR code as a way to easily give her address to others without them having to type the long string of letters and numbers.

TIP

Bitcoin addresses start with the digit 1 or 3. Like email addresses, they can be shared with other bitcoin users who can use them to send bitcoin directly to your wallet. Unlike email addresses, you can create new addresses as often as you like, all of which will direct funds to your wallet. A wallet is simply a collection of addresses and the keys that unlock the funds within. You can increase your privacy by using a different address for every transaction. There is practically no limit to the number of addresses a user can create.

Alice is now ready to start using her new bitcoin wallet.

Getting Your First Bitcoins

It is not possible to buy bitcoins at a bank or foreign exchange kiosks at this time. As of 2014, it is still quite difficult to acquire bitcoins in most countries. There are a number of specialized currency exchanges where you can buy and sell bitcoin in exchange for a local currency. These operate as web-based currency markets and include:

Bitstamp

A European currency market that supports several currencies including euros (EUR) and US

dollars (USD) via wire transfer.

Coinbase

A US-based bitcoin wallet and platform where merchants and consumers can transact in bitcoin. Coinbase makes it easy to buy and sell bitcoin, allowing users to connect to US checking accounts via the ACH system.

Cryptocurrency exchanges such as these operate at the intersection of national currencies and cryptocurrencies. As such, they are subject to national and international regulations, and are often specific to a single country or economic area and specialize in the national currencies of that area. Your choice of currency exchange will be specific to the national currency you use and limited to the exchanges that operate within the legal jurisdiction of your country. Similar to opening a bank account, it takes several days or weeks to set up the necessary accounts with these services because they require various forms of identification to comply with KYC (know your customer) and AML (anti-money laundering) banking regulations. Once you have an account on a bitcoin exchange, you can then buy or sell bitcoins quickly just as you could with foreign currency with a brokerage account.

You can find a more complete list at [bitcoin charts](#), a site that offers price quotes and other market data across many dozens of currency exchanges.

There are four other methods for getting bitcoins as a new user:

- Find a friend who has bitcoins and buy some from him directly. Many bitcoin users start this way.
- Use a classified service such as [localbitcoins.com](#) to find a seller in your area to buy bitcoins for cash in an in-person transaction.
- Sell a product or service for bitcoin. If you're a programmer, sell your programming skills.
- Use a bitcoin ATM in your city. Find a bitcoin ATM close to you using an online map from [CoinDesk](#).

Alice was introduced to bitcoin by a friend and so she has an easy way of getting her first bitcoins while she waits for her account on a California currency market to be verified and activated.

Sending and Receiving Bitcoins

Alice has created her bitcoin wallet and she is now ready to receive funds. Her wallet application randomly generated a private key (described in more detail in [Private Keys](#)) together with its corresponding bitcoin address. At this point, her bitcoin address is not known to the bitcoin network or "registered" with any part of the bitcoin system. Her bitcoin address is simply a number that corresponds to a key that she can use to control access to the funds. There is no account or association between that address and an account. Until the moment this address is referenced as the recipient of value in a transaction posted on the bitcoin ledger (the blockchain), it is simply part of the vast number of possible addresses that are "valid" in bitcoin. Once it has been associated with a transaction, it becomes part of the known addresses in the network and Alice can check its balance on the public ledger.

Alice meets her friend Joe, who introduced her to bitcoin, at a local restaurant so they can exchange

some US dollars and put some bitcoins into her account. She has brought a printout of her address and the QR code as displayed in her bitcoin wallet. There is nothing sensitive, from a security perspective, about the bitcoin address. It can be posted anywhere without risking the security of her account.

Alice wants to convert just 10 US dollars into bitcoin, so as not to risk too much money on this new technology. She gives Joe a \$10 bill and the printout of her address so that Joe can send her the equivalent amount of bitcoin.

Next, Joe has to figure out the exchange rate so that he can give the correct amount of bitcoin to Alice. There are hundreds of applications and websites that can provide the current market rate. Here are some of the most popular:

Bitcoin Charts

A market data listing service that shows the market rate of bitcoin across many exchanges around the globe, denominated in different local currencies

Bitcoin Average

A site that provides a simple view of the volume-weighted-average for each currency

ZeroBlock

A free Android and iOS application that can display a bitcoin price from different exchanges (see [ZeroBlock, a bitcoin market-rate application for Android and iOS](#))

Bitcoin Wisdom

Another market data listing service

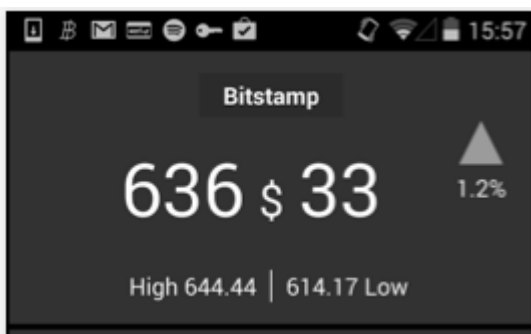


Figure 3. ZeroBlock, a bitcoin market-rate application for Android and iOS

Using one of the applications or websites just listed, Joe determines the price of bitcoin to be approximately 100 US dollars per bitcoin. At that rate he should give Alice 0.10 bitcoin, also known as 100 millibits, in return for the 10 US dollars she gave him.

Once Joe has established a fair exchange price, he opens his mobile wallet application and selects to "send" bitcoin. For example, if using the Blockchain mobile wallet on an Android phone, he would see a screen requesting two inputs, as shown in [Blockchain mobile wallet's bitcoin send screen](#).

- The destination bitcoin address for the transaction
- The amount of bitcoin to send

In the input field for the bitcoin address, there is a small icon that looks like a QR code. This allows

Joe to scan the barcode with his smartphone camera so that he doesn't have to type in Alice's bitcoin address (1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK), which is quite long and difficult to type. Joe taps the QR code icon and activates the smartphone camera, scanning the QR code from Alice's printed wallet that she brought with her. The mobile wallet application fills in the bitcoin address and Joe can check that it scanned correctly by comparing a few digits from the address with the address printed by Alice.

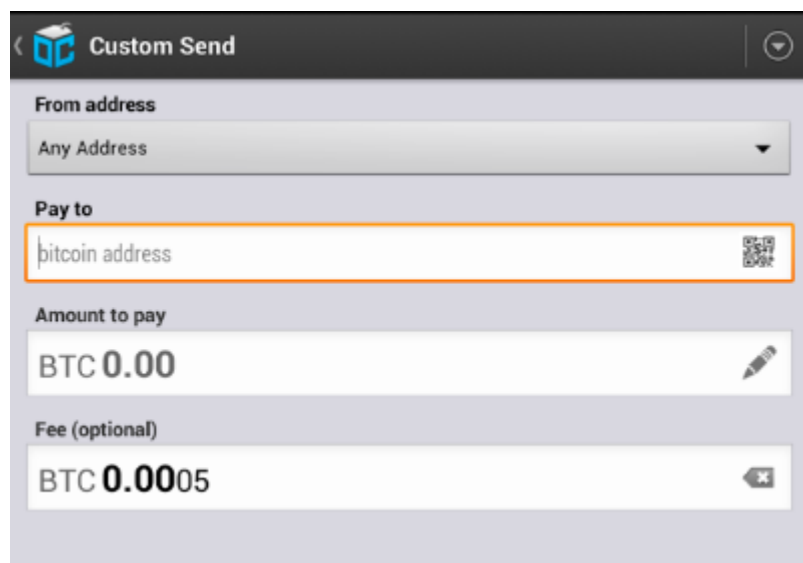


Figure 4. Blockchain mobile wallet's bitcoin send screen

Joe then enters the bitcoin value for the transaction, 0.10 bitcoin. He carefully checks to make sure he has entered the correct amount, because he is about to transmit money and any mistake could be costly. Finally, he presses Send to transmit the transaction. Joe's mobile bitcoin wallet constructs a transaction that assigns 0.10 bitcoin to the address provided by Alice, sourcing the funds from Joe's wallet and signing the transaction with Joe's private keys. This tells the bitcoin network that Joe has authorized a transfer of value from one of his addresses to Alice's new address. As the transaction is transmitted via the peer-to-peer protocol, it quickly propagates across the bitcoin network. In less than a second, most of the well-connected nodes in the network receive the transaction and see Alice's address for the first time.

If Alice has a smartphone or laptop with her, she will also be able to see the transaction. The bitcoin ledger—a constantly growing file that records every bitcoin transaction that has ever occurred—is public, meaning that all she has to do is look up her own address and see if any funds have been sent to it. She can do this quite easily at the blockchain.info website by entering her address in the search box. The website will show her a [page](#) listing all the transactions to and from that address. If Alice is watching that page, it will update to show a new transaction transferring 0.10 bitcoin to her balance soon after Joe hits Send.

Confirmations

At first, Alice's address will show the transaction from Joe as "Unconfirmed." This means that the transaction has been propagated to the network but has not yet been included in the bitcoin transaction ledger, known as the blockchain. To be included, the transaction must be "picked up" by a miner and included in a block of transactions. Once a new block is created, in approximately 10 minutes, the transactions within the block will be accepted as "confirmed" by the network and can be spent. The transaction is seen by all instantly, but it is only "trusted" by all when it is included in a newly mined block.

Alice is now the proud owner of 0.10 bitcoin that she can spend. In the next chapter we will look at her first purchase with bitcoin, and examine the underlying transaction and propagation technologies in more detail.

How Bitcoin Works

Transactions, Blocks, Mining, and the Blockchain

The bitcoin system, unlike traditional banking and payment systems, is based on de-centralized trust. Instead of a central trusted authority, in bitcoin, trust is achieved as an emergent property from the interactions of different participants in the bitcoin system. In this chapter, we will examine bitcoin from a high level by tracking a single transaction through the bitcoin system and watch as it becomes "trusted" and accepted by the bitcoin mechanism of distributed consensus and is finally recorded on the blockchain, the distributed ledger of all transactions.

Each example is based on an actual transaction made on the bitcoin network, simulating the interactions between the users (Joe, Alice, and Bob) by sending funds from one wallet to another. While tracking a transaction through the bitcoin network and blockchain, we will use a *blockchain explorer* site to visualize each step. A blockchain explorer is a web application that operates as a bitcoin search engine, in that it allows you to search for addresses, transactions, and blocks and see the relationships and flows between them.

Popular blockchain explorers include:

- [Blockchain info](#)
- [Bitcoin Block Explorer](#)
- [insight](#)
- [blockr Block Reader](#)

Each of these has a search function that can take an address, transaction hash, or block number and find the equivalent data on the bitcoin network and blockchain. With each example, we will provide a URL that takes you directly to the relevant entry, so you can study it in detail.

Bitcoin Overview

In the overview diagram shown in [Bitcoin overview](#), we see that the bitcoin system consists of users with wallets containing keys, transactions that are propagated across the network, and miners who produce (through competitive computation) the consensus blockchain, which is the authoritative ledger of all transactions. In this chapter, we will trace a single transaction as it travels across the network and examine the interactions between each part of the bitcoin system, at a high level. Subsequent chapters will delve into the technology behind wallets, mining, and merchant systems.

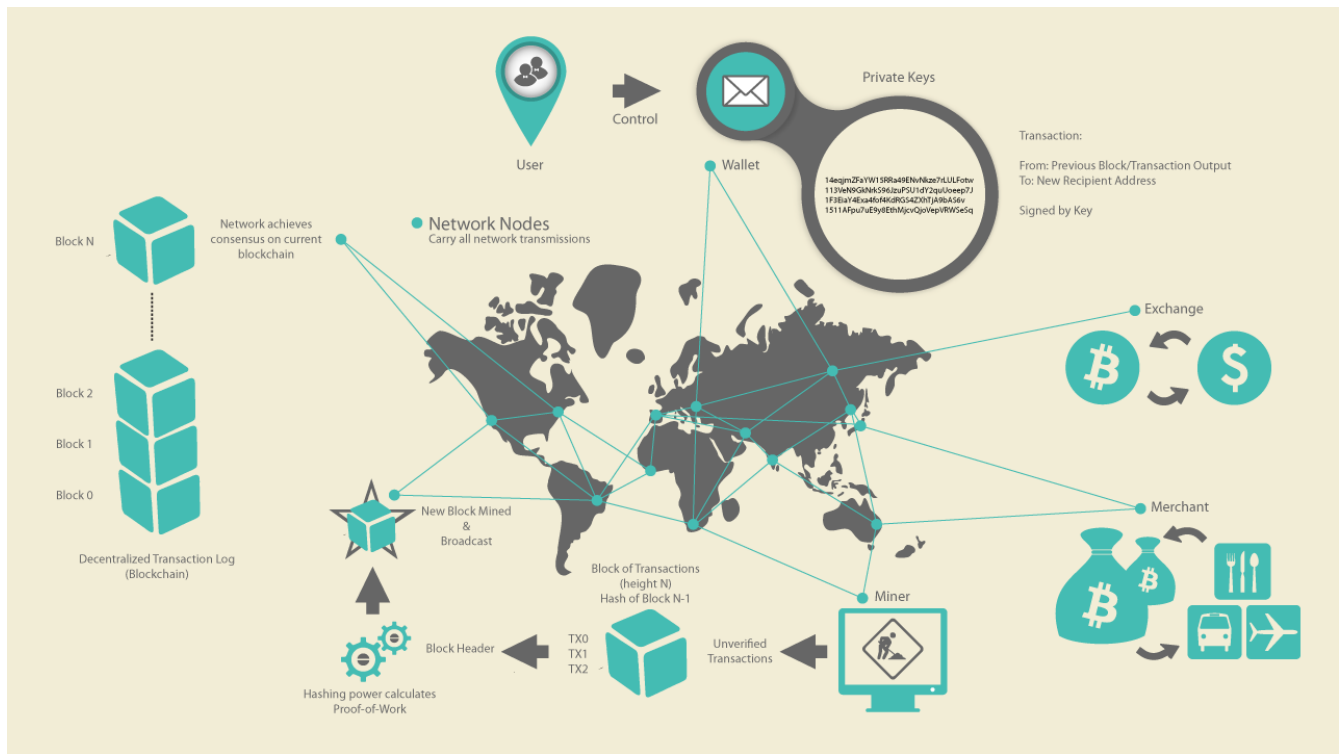


Figure 5. Bitcoin overview

Buying a Cup of Coffee

Alice, introduced in the previous chapter, is a new user who has just acquired her first bitcoin. In [Getting Your First Bitcoins](#), Alice met with her friend Joe to exchange some cash for bitcoin. The transaction created by Joe funded Alice's wallet with 0.10 BTC. Now Alice will make her first retail transaction, buying a cup of coffee at Bob's coffee shop in Palo Alto, California. Bob's coffee shop recently started accepting bitcoin payments, by adding a bitcoin option to his point-of-sale system. The prices at Bob's Cafe are listed in the local currency (US dollars), but at the register, customers have the option of paying in either dollars or bitcoin. Alice places her order for a cup of coffee and Bob enters the transaction at the register. The point-of-sale system will convert the total price from US dollars to bitcoins at the prevailing market rate and display the prices in both currencies, as well as show a QR code containing a *payment request* for this transaction (see [Payment request QR code \(Hint: Try to scan this!\)](#)):

Total:
\$1.50 USD
0.015 BTC



Figure 6. Payment request QR code (Hint: Try to scan this!)

The payment request QR code encodes the following URL, defined in BIP0021:

```
bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?  
amount=0.015&  
label=Bob%27s%20Cafe&  
message=Purchase%20at%20Bob%27s%20Cafe
```

Components of the URL

A bitcoin address: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"

The payment amount: "0.015"

A label for the recipient address: "Bob's Cafe"

A description for the payment: "Purchase at Bob's Cafe"

TIP

Unlike a QR code that simply contains a destination bitcoin address, a payment request is a QR-encoded URL that contains a destination address, a payment amount, and a generic description such as "Bob's Cafe." This allows a bitcoin wallet application to prefill the information used to send the payment while showing a human-readable description to the user. You can scan the QR code with a bitcoin wallet application to see what Alice would see.

Bob says, "That's one-dollar-fifty, or fifteen millibits."

Alice uses her smartphone to scan the barcode on display. Her smartphone shows a payment of **0.0150 BTC** to **Bob's Cafe** and she selects **Send** to authorize the payment. Within a few seconds (about the same amount of time as a credit card authorization), Bob would see the transaction on the register, completing the transaction.

In the following sections we will examine this transaction in more detail, see how Alice's wallet constructed it, how it was propagated across the network, how it was verified, and finally, how Bob can spend that amount in subsequent transactions.

NOTE

The bitcoin network can transact in fractional values, e.g., from milli-bitcoins (1/1000th of a bitcoin) down to 1/100,000,000th of a bitcoin, which is known as a satoshi. Throughout this book we'll use the term "bitcoin" to refer to any quantity of bitcoin currency, from the smallest unit (1 satoshi) to the total number (21,000,000) of all bitcoins that will ever be mined.

Bitcoin Transactions

In simple terms, a transaction tells the network that the owner of a number of bitcoins has authorized the transfer of some of those bitcoins to another owner. The new owner can now spend these bitcoins by creating another transaction that authorizes transfer to another owner, and so on, in a chain of ownership.

Transactions are like lines in a double-entry bookkeeping ledger. In simple terms, each transaction contains one or more "inputs," which are debits against a bitcoin account. On the other side of the transaction, there are one or more "outputs," which are credits added to a bitcoin account. The inputs and outputs (debits and credits) do not necessarily add up to the same amount. Instead, outputs add up to slightly less than inputs and the difference represents an implied "transaction fee," which is a small payment collected by the miner who includes the transaction in the ledger. A bitcoin transaction is shown as a bookkeeping ledger entry in [Transaction as double-entry bookkeeping](#).

The transaction also contains proof of ownership for each amount of bitcoin (inputs) whose value is transferred, in the form of a digital signature from the owner, which can be independently validated by anyone. In bitcoin terms, "spending" is signing a transaction that transfers value from a previous transaction over to a new owner identified by a bitcoin address.

TIP

Transactions move value from *transaction inputs* to *transaction outputs*. An input is where the coin value is coming from, usually a previous transaction's output. A transaction output assigns a new owner to the value by associating it with a key. The destination key is called an *encumbrance*. It imposes a requirement for a signature for the funds to be redeemed in future transactions. Outputs from one transaction can be used as inputs in a new transaction, thus creating a chain of ownership as the value is moved from address to address (see [A chain of transactions, where the output of one transaction is the input of the next transaction](#)).

received bitcoin from her friend Joe in return for cash. That transaction has a number of bitcoins locked (encumbered) against Alice's key. Her new transaction to Bob's Cafe references the previous transaction as an input and creates new outputs to pay for the cup of coffee and receive change. The transactions form a chain, where the inputs from the latest transaction correspond to outputs from previous transactions. Alice's key provides the signature that unlocks those previous transaction outputs, thereby proving to the bitcoin network that she owns the funds. She attaches the payment for coffee to Bob's address, thereby "encumbering" that output with the requirement that Bob produces a signature in order to spend that amount. This represents a transfer of value between Alice and Bob. This chain of transactions, from Joe to Alice to Bob, is illustrated in [A chain of transactions, where the output of one transaction is the input of the next transaction](#).

Common Transaction Forms

The most common form of transaction is a simple payment from one address to another, which often includes some "change" returned to the original owner. This type of transaction has one input and two outputs and is shown in [Most common transaction](#).

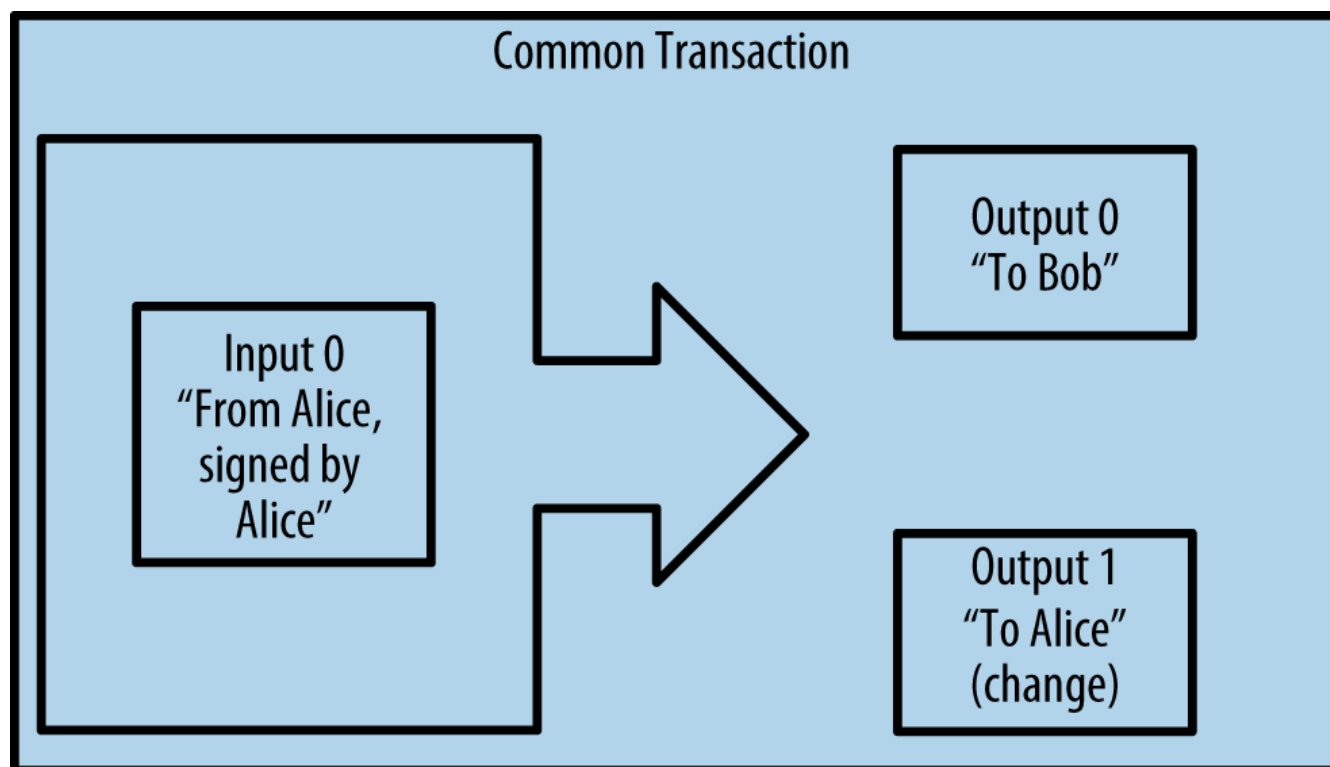


Figure 9. Most common transaction

Another common form of transaction is one that aggregates several inputs into a single output (see [Transaction aggregating funds](#)). This represents the real-world equivalent of exchanging a pile of coins and currency notes for a single larger note. Transactions like these are sometimes generated by wallet applications to clean up lots of smaller amounts that were received as change for payments.

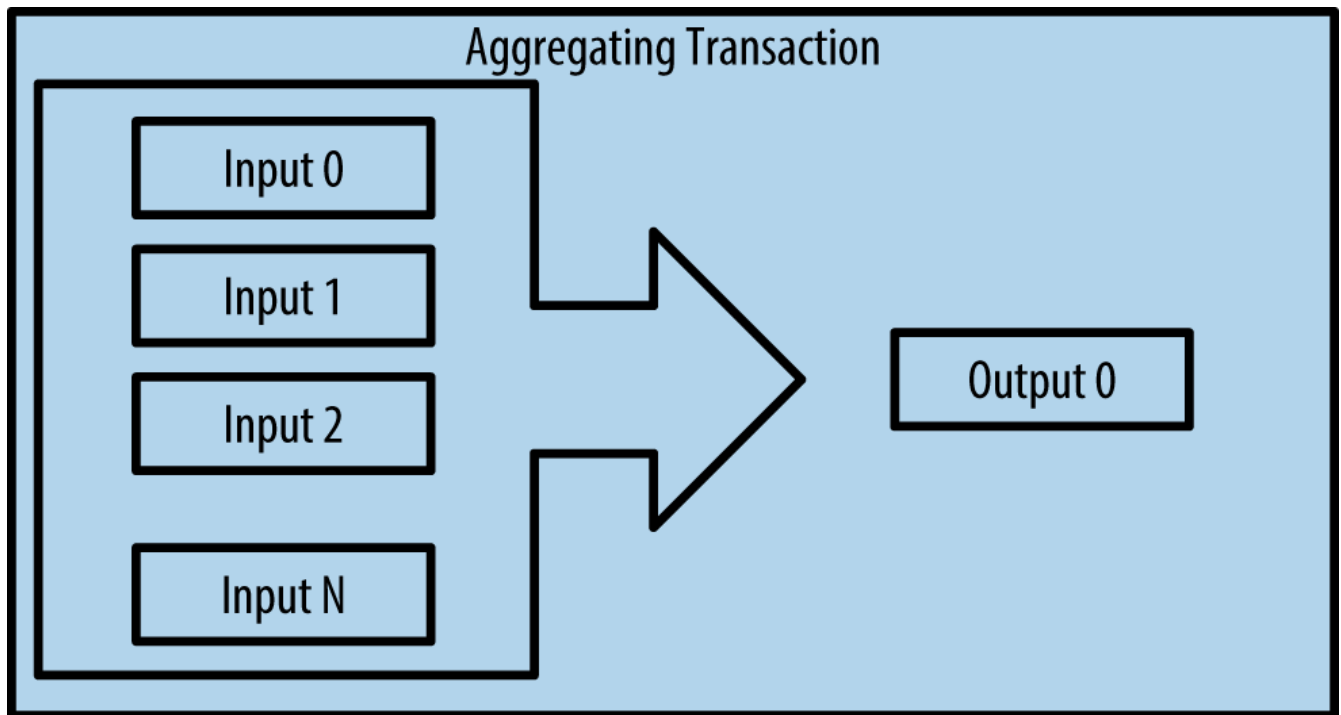


Figure 10. Transaction aggregating funds

Finally, another transaction form that is seen often on the bitcoin ledger is a transaction that distributes one input to multiple outputs representing multiple recipients (see [Transaction distributing funds](#)). This type of transaction is sometimes used by commercial entities to distribute funds, such as when processing payroll payments to multiple employees.

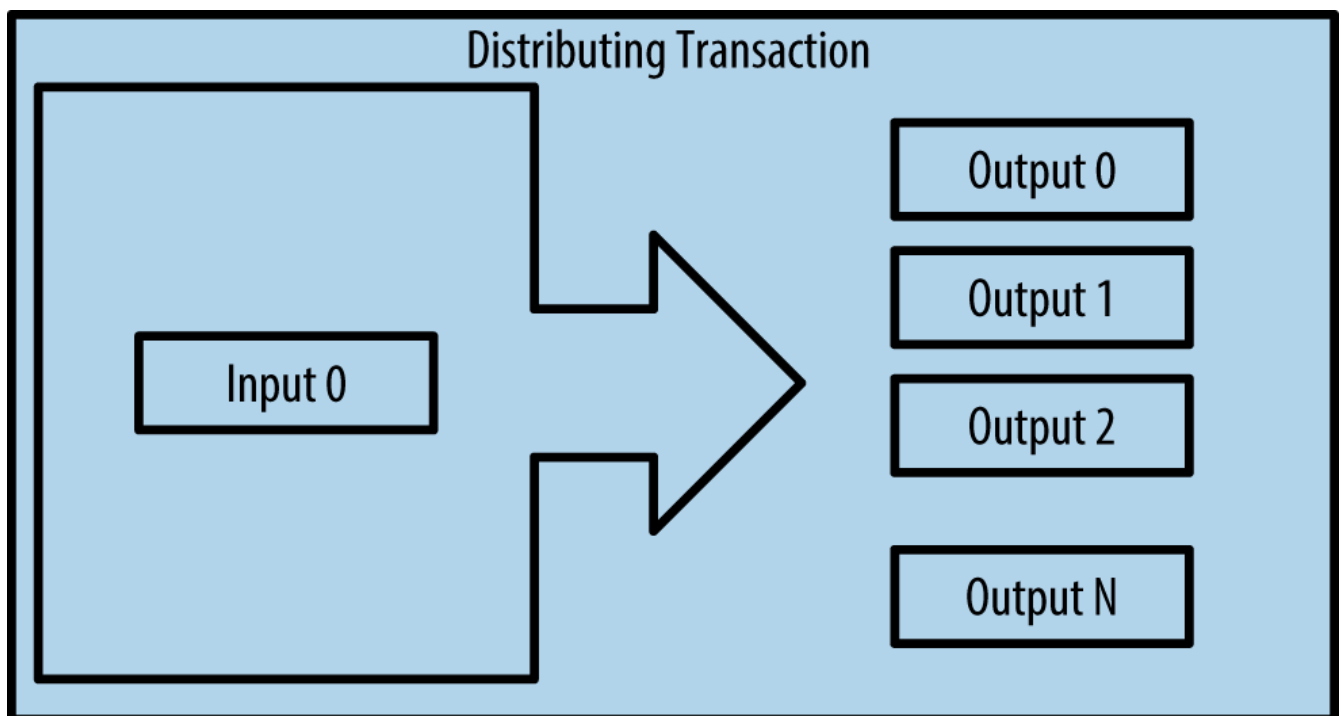


Figure 11. Transaction distributing funds

Constructing a Transaction

Alice's wallet application contains all the logic for selecting appropriate inputs and outputs to build a transaction to Alice's specification. Alice only needs to specify a destination and an amount and

the rest happens in the wallet application without her seeing the details. Importantly, a wallet application can construct transactions even if it is completely offline. Like writing a check at home and later sending it to the bank in an envelope, the transaction does not need to be constructed and signed while connected to the bitcoin network. It only has to be sent to the network eventually for it to be executed.

Getting the Right Inputs

Alice's wallet application will first have to find inputs that can pay for the amount she wants to send to Bob. Most wallet applications keep a small database of "unspent transaction outputs" that are locked (encumbered) with the wallet's own keys. Therefore, Alice's wallet would contain a copy of the transaction output from Joe's transaction, which was created in exchange for cash (see [Getting Your First Bitcoins](#)). A bitcoin wallet application that runs as a full-index client actually contains a copy of every unspent output from every transaction in the blockchain. This allows a wallet to construct transaction inputs as well as quickly verify incoming transactions as having correct inputs. However, because a full-index client takes up a lot of disk space, most user wallets run "lightweight" clients that track only the user's own unspent outputs.

If the wallet application does not maintain a copy of unspent transaction outputs, it can query the bitcoin network to retrieve this information, using a variety of APIs available by different providers or by asking a full-index node using the bitcoin JSON RPC API. [Look up all the unspent outputs for Alice's bitcoin address](#) shows a RESTful API request, constructed as an HTTP GET command to a specific URL. This URL will return all the unspent transaction outputs for an address, giving any application the information it needs to construct transaction inputs for spending. We use the simple command-line HTTP client *cURL* to retrieve the response.

Example 1. Look up all the unspent outputs for Alice's bitcoin address

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
```

Example 2. Response to the lookup

```
{
  "unspent_outputs": [
    {
      "tx_hash": "186f9f998a5...2836dd734d2804fe65fa35779",
      "tx_index": 104810202,
      "tx_output_n": 0,
      "script": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
      "value": 10000000,
      "value_hex": "00989680",
      "confirmations": 0
    }
  ]
}
```

The response in [Response to the lookup](#) shows one unspent output (one that has not been redeemed yet) under the ownership of Alice's address `1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK`. The response includes the reference to the transaction in which this unspent output is contained (the payment from Joe) and its value in satoshis, at 10 million, equivalent to 0.10 bitcoin. With this information, Alice's wallet application can construct a transaction to transfer that value to new owner addresses.

TIP | View the [transaction from Joe to Alice](#).

As you can see, Alice's wallet contains enough bitcoins in a single unspent output to pay for the cup of coffee. Had this not been the case, Alice's wallet application might have to "rummage" through a pile of smaller unspent outputs, like picking coins from a purse until it could find enough to pay for coffee. In both cases, there might be a need to get some change back, which we will see in the next section, as the wallet application creates the transaction outputs (payments).

Creating the Outputs

A transaction output is created in the form of a script that creates an encumbrance on the value and can only be redeemed by the introduction of a solution to the script. In simpler terms, Alice's transaction output will contain a script that says something like, "This output is payable to whoever can present a signature from the key corresponding to Bob's public address." Because only Bob has the wallet with the keys corresponding to that address, only Bob's wallet can present such a signature to redeem this output. Alice will therefore "encumber" the output value with a demand for a signature from Bob.

This transaction will also include a second output, because Alice's funds are in the form of a 0.10 BTC output, too much money for the 0.015 BTC cup of coffee. Alice will need 0.085 BTC in change. Alice's change payment is created *by Alice's wallet* in the very same transaction as the payment to Bob. Essentially, Alice's wallet breaks her funds into two payments: one to Bob, and one back to

herself. She can then use the change output in a subsequent transaction, thus spending it later.

Finally, for the transaction to be processed by the network in a timely fashion, Alice's wallet application will add a small fee. This is not explicit in the transaction; it is implied by the difference between inputs and outputs. If instead of taking 0.085 in change, Alice creates only 0.0845 as the second output, there will be 0.0005 BTC (half a millibitcoin) left over. The input's 0.10 BTC is not fully spent with the two outputs, because they will add up to less than 0.10. The resulting difference is the *transaction fee* that is collected by the miner as a fee for including the transaction in a block and putting it on the blockchain ledger.

The resulting transaction can be seen using a blockchain explorer web application, as shown in [Alice's transaction to Bob's Cafe](#).

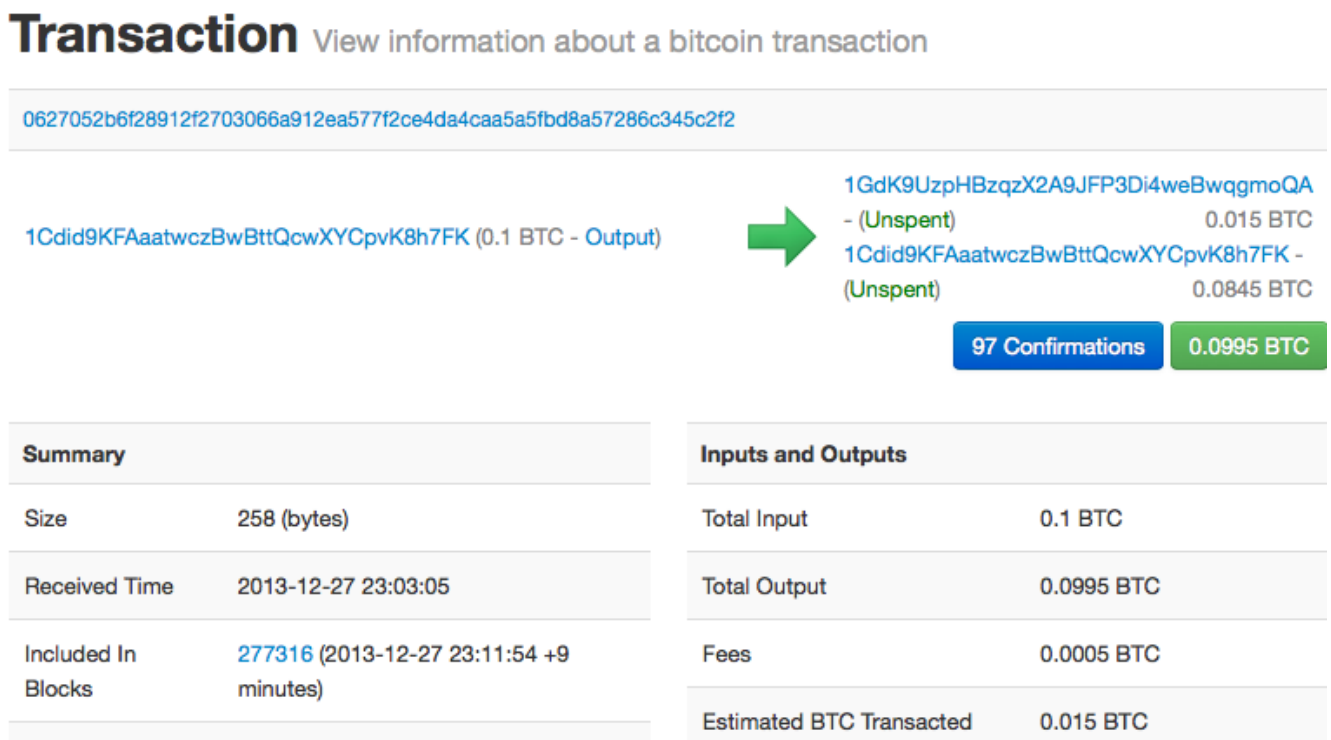


Figure 12. Alice's transaction to Bob's Cafe

TIP View the [transaction from Alice to Bob's Cafe](#).

Adding the Transaction to the Ledger

The transaction created by Alice's wallet application is 258 bytes long and contains everything necessary to confirm ownership of the funds and assign new owners. Now, the transaction must be transmitted to the bitcoin network where it will become part of the distributed ledger (the blockchain). In the next section we will see how a transaction becomes part of a new block and how the block is "mined." Finally, we will see how the new block, once added to the blockchain, is increasingly trusted by the network as more blocks are added.

Transmitting the transaction

Because the transaction contains all the information necessary to process, it does not matter how or where it is transmitted to the bitcoin network. The bitcoin network is a peer-to-peer network, with

each bitcoin client participating by connecting to several other bitcoin clients. The purpose of the bitcoin network is to propagate transactions and blocks to all participants.

How it propagates

Alice's wallet application can send the new transaction to any of the other bitcoin clients it is connected to over any Internet connection: wired, WiFi, or mobile. Her bitcoin wallet does not have to be connected to Bob's bitcoin wallet directly and she does not have to use the Internet connection offered by the cafe, though both those options are possible, too. Any bitcoin network node (other client) that receives a valid transaction it has not seen before will immediately forward it to other nodes to which it is connected. Thus, the transaction rapidly propagates out across the peer-to-peer network, reaching a large percentage of the nodes within a few seconds.

Bob's view

If Bob's bitcoin wallet application is directly connected to Alice's wallet application, Bob's wallet application might be the first node to receive the transaction. However, even if Alice's wallet sends the transaction through other nodes, it will reach Bob's wallet within a few seconds. Bob's wallet will immediately identify Alice's transaction as an incoming payment because it contains outputs redeemable by Bob's keys. Bob's wallet application can also independently verify that the transaction is well formed, uses previously unspent inputs, and contains sufficient transaction fees to be included in the next block. At this point Bob can assume, with little risk, that the transaction will shortly be included in a block and confirmed.

TIP

A common misconception about bitcoin transactions is that they must be "confirmed" by waiting 10 minutes for a new block, or up to 60 minutes for a full six confirmations. Although confirmations ensure the transaction has been accepted by the whole network, such a delay is unnecessary for small-value items such as a cup of coffee. A merchant may accept a valid small-value transaction with no confirmations, with no more risk than a credit card payment made without an ID or a signature, as merchants routinely accept today.

Bitcoin Mining

The transaction is now propagated on the bitcoin network. It does not become part of the shared ledger (the *blockchain*) until it is verified and included in a block by a process called *mining*. See [Mining and Consensus](#) for a detailed explanation.

The bitcoin system of trust is based on computation. Transactions are bundled into *blocks*, which require an enormous amount of computation to prove, but only a small amount of computation to verify as proven. The mining process serves two purposes in bitcoin:

- Mining creates new bitcoins in each block, almost like a central bank printing new money. The amount of bitcoin created per block is fixed and diminishes with time.
- Mining creates trust by ensuring that transactions are only confirmed if enough computational power was devoted to the block that contains them. More blocks mean more computation, which means more trust.

A good way to describe mining is like a giant competitive game of sudoku that resets every time someone finds a solution and whose difficulty automatically adjusts so that it takes approximately 10 minutes to find a solution. Imagine a giant sudoku puzzle, several thousand rows and columns in size. If I show you a completed puzzle you can verify it quite quickly. However, if the puzzle has a few squares filled and the rest are empty, it takes a lot of work to solve! The difficulty of the sudoku can be adjusted by changing its size (more or fewer rows and columns), but it can still be verified quite easily even if it is very large. The "puzzle" used in bitcoin is based on a cryptographic hash and exhibits similar characteristics: it is asymmetrically hard to solve but easy to verify, and its difficulty can be adjusted.

In [Bitcoin Uses, Users, and Their Stories](#), we introduced Jing, a computer engineering student in Shanghai. Jing is participating in the bitcoin network as a miner. Every 10 minutes or so, Jing joins thousands of other miners in a global race to find a solution to a block of transactions. Finding such a solution, the so-called proof of work, requires quadrillions of hashing operations per second across the entire bitcoin network. The algorithm for proof of work involves repeatedly hashing the header of the block and a random number with the SHA256 cryptographic algorithm until a solution matching a predetermined pattern emerges. The first miner to find such a solution wins the round of competition and publishes that block into the blockchain.

Jing started mining in 2010 using a very fast desktop computer to find a suitable proof of work for new blocks. As more miners started joining the bitcoin network, the difficulty of the problem increased rapidly. Soon, Jing and other miners upgraded to more specialized hardware, such as high-end dedicated graphical processing units (GPUs) cards such as those used in gaming desktops or consoles. At the time of this writing, the difficulty is so high that it is profitable only to mine with application-specific integrated circuits (ASIC), essentially hundreds of mining algorithms printed in hardware, running in parallel on a single silicon chip. Jing also joined a "mining pool," which much like a lottery pool allows several participants to share their efforts and the rewards. Jing now runs two USB-connected ASIC machines to mine for bitcoin 24 hours a day. He pays his electricity costs by selling the bitcoin he is able to generate from mining, creating some income from the profits. His computer runs a copy of bitcoind, the reference bitcoin client, as a backend to his specialized mining software.

Mining Transactions in Blocks

A transaction transmitted across the network is not verified until it becomes part of the global distributed ledger, the blockchain. Every 10 minutes on average, miners generate a new block that contains all the transactions since the last block. New transactions are constantly flowing into the network from user wallets and other applications. As these are seen by the bitcoin network nodes, they get added to a temporary pool of unverified transactions maintained by each node. As miners build a new block, they add unverified transactions from this pool to a new block and then attempt to solve a very hard problem (a.k.a., proof of work) to prove the validity of that new block. The process of mining is explained in detail in [Introduction](#).

Transactions are added to the new block, prioritized by the highest-fee transactions first and a few other criteria. Each miner starts the process of mining a new block of transactions as soon as he receives the previous block from the network, knowing he has lost that previous round of competition. He immediately creates a new block, fills it with transactions and the fingerprint of the previous block, and starts calculating the proof of work for the new block. Each miner includes

a special transaction in his block, one that pays his own bitcoin address a reward of newly created bitcoins (currently 25 BTC per block). If he finds a solution that makes that block valid, he "wins" this reward because his successful block is added to the global blockchain and the reward transaction he included becomes spendable. Jing, who participates in a mining pool, has set up his software to create new blocks that assign the reward to a pool address. From there, a share of the reward is distributed to Jing and other miners in proportion to the amount of work they contributed in the last round.

Alice's transaction was picked up by the network and included in the pool of unverified transactions. Because it had sufficient fees, it was included in a new block generated by Jing's mining pool. Approximately five minutes after the transaction was first transmitted by Alice's wallet, Jing's ASIC miner found a solution for the block and published it as block #277316, containing 419 other transactions. Jing's ASIC miner published the new block on the bitcoin network, where other miners validated it and started the race to generate the next block.

You can see the block that includes [Alice's transaction](#).

A few minutes later, a new block, #277317, is mined by another miner. Because this new block is based on the previous block (#277316) that contained Alice's transaction, it added even more computation on top of that block, thereby strengthening the trust in those transactions. The block containing Alice's transaction is counted as one "confirmation" of that transaction. Each block mined on top of the one containing the transaction is an additional confirmation. As the blocks pile on top of each other, it becomes exponentially harder to reverse the transaction, thereby making it more and more trusted by the network.

In the diagram in [Alice's transaction included in block #277316](#) we can see block #277316, which contains Alice's transaction. Below it are 277,316 blocks (including block #0), linked to each other in a chain of blocks (blockchain) all the way back to block #0, known as the *genesis block*. Over time, as the "height" in blocks increases, so does the computation difficulty for each block and the chain as a whole. The blocks mined after the one that contains Alice's transaction act as further assurance, as they pile on more computation in a longer and longer chain. By convention, any block with more than six confirmations is considered irrevocable, because it would require an immense amount of computation to invalidate and recalculate six blocks. We will examine the process of mining and the way it builds trust in more detail in [Mining and Consensus](#).

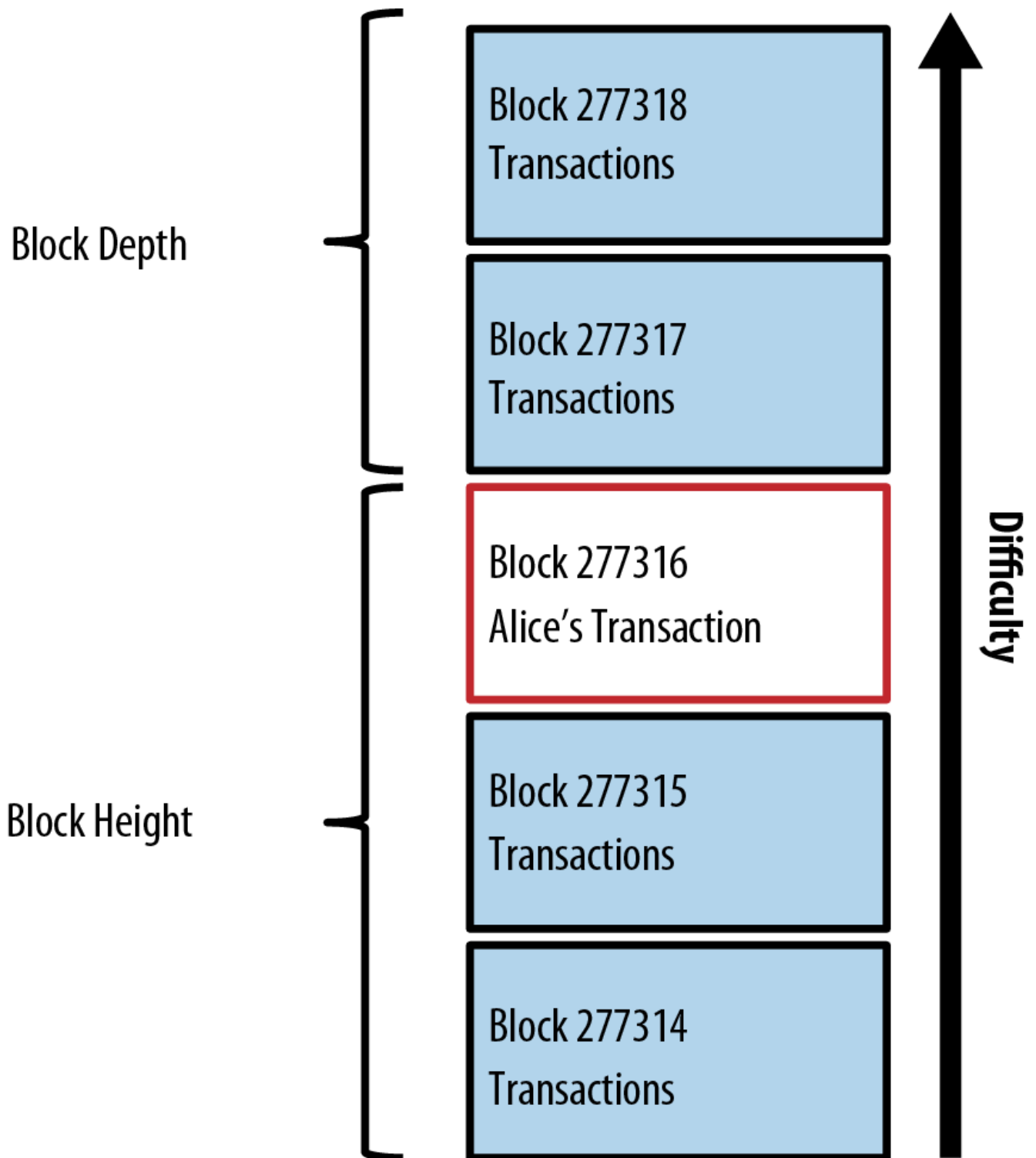


Figure 13. Alice's transaction included in block #277316

Spending the Transaction

Now that Alice's transaction has been embedded in the blockchain as part of a block, it is part of the distributed ledger of bitcoin and visible to all bitcoin applications. Each bitcoin client can independently verify the transaction as valid and spendable. Full-index clients can track the source of the funds from the moment the bitcoins were first generated in a block, incrementally from transaction to transaction, until they reach Bob's address. Lightweight clients can do what is called a simplified payment verification (see [Simplified Payment Verification \(SPV\) Nodes](#)) by confirming that the transaction is in the blockchain and has several blocks mined after it, thus providing

assurance that the network accepts it as valid.

Bob can now spend the output from this and other transactions, by creating his own transactions that reference these outputs as their inputs and assign them new ownership. For example, Bob can pay a contractor or supplier by transferring value from Alice's coffee cup payment to these new owners. Most likely, Bob's bitcoin software will aggregate many small payments into a larger payment, perhaps concentrating all the day's bitcoin revenue into a single transaction. This would move the various payments into a single address, used as the store's general "checking" account. For a diagram of an aggregating transaction, see [Transaction aggregating funds](#).

As Bob spends the payments received from Alice and other customers, he extends the chain of transactions, which in turn are added to the global blockchain ledger for all to see and trust. Let's assume that Bob pays his web designer Gopesh in Bangalore for a new website page. Now the chain of transactions will look like [Alice's transaction as part of a transaction chain from Joe to Gopesh](#).

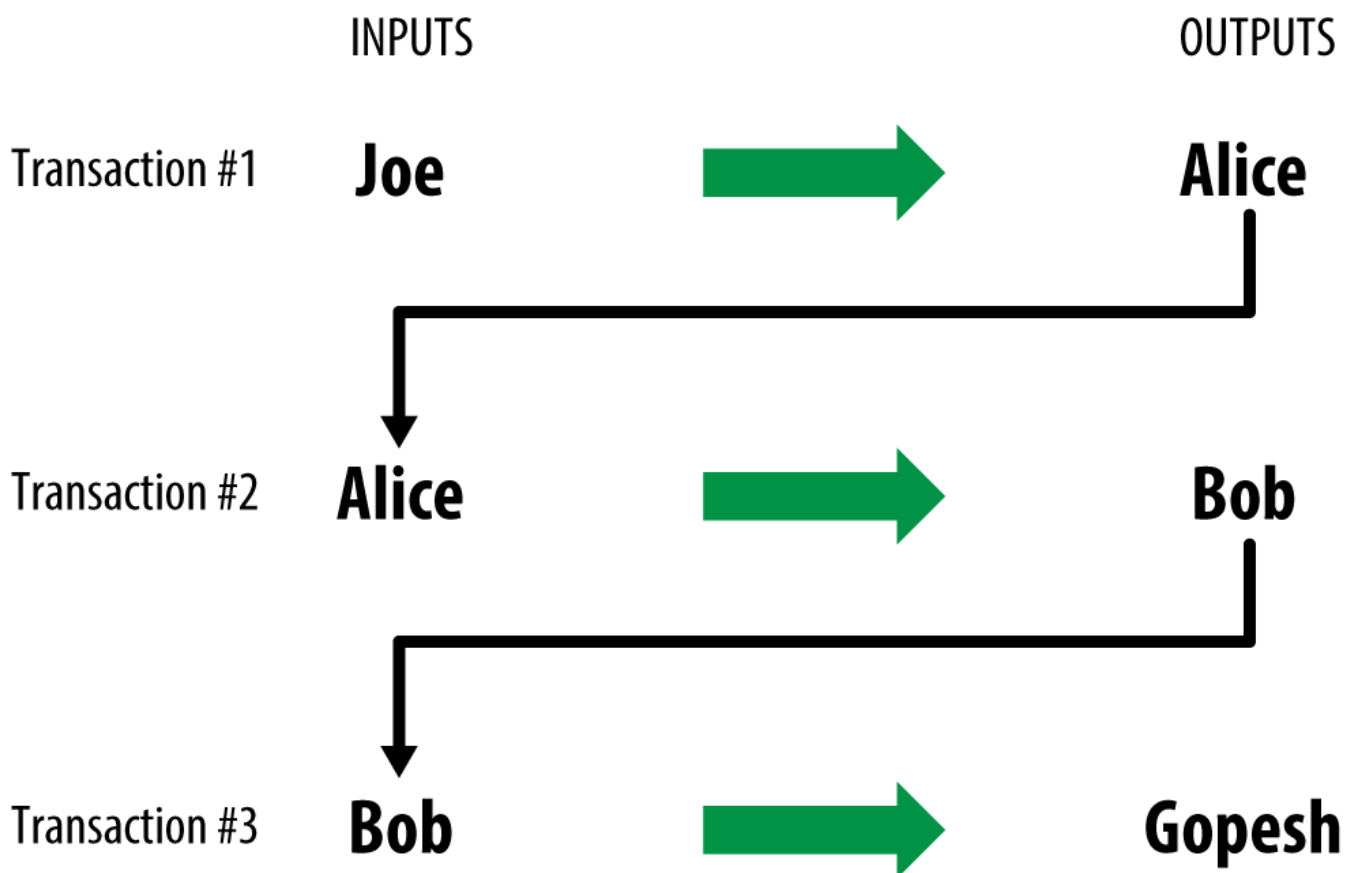


Figure 14. Alice's transaction as part of a transaction chain from Joe to Gopesh

The Bitcoin Client

Bitcoin Core: The Reference Implementation

You can download the reference client *Bitcoin Core*, also known as the "Satoshi client," from bitcoin.org. The reference client implements all aspects of the bitcoin system, including wallets, a transaction verification engine with a full copy of the entire transaction ledger (blockchain), and a full network node in the peer-to-peer bitcoin network.

On [Bitcoin's Choose Your Wallet page](https://bitcoin.org/en/choose-your-wallet), select Bitcoin Core to download the reference client. Depending on your operating system, you will download an executable installer. For Windows, this is either a ZIP archive or an .exe executable. For Mac OS it is a .dmg disk image. Linux versions include a PPA package for Ubuntu or a tar.gz archive. The [bitcoin.org](https://bitcoin.org/en/choose-your-wallet) page that lists recommended bitcoin clients is shown in [Choosing a bitcoin client at bitcoin.org](https://bitcoin.org/en/choose-your-wallet).

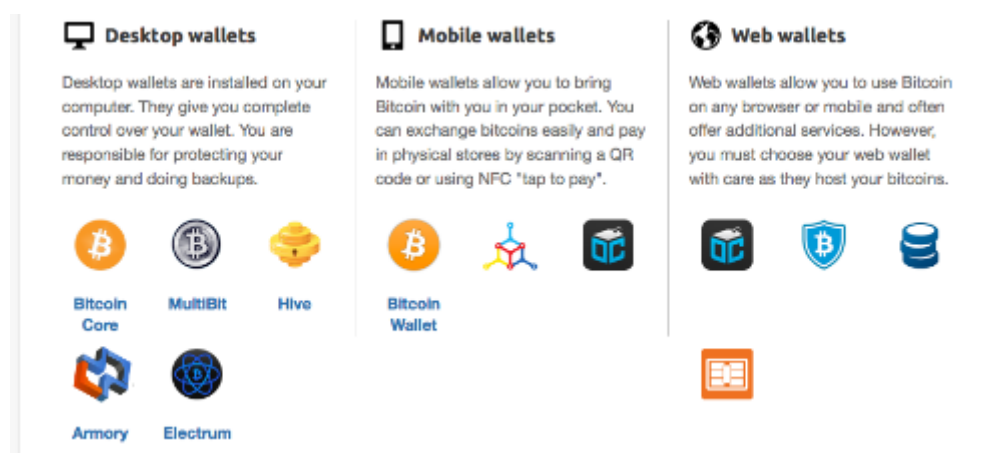


Figure 15. Choosing a bitcoin client at bitcoin.org

Running Bitcoin Core for the First Time

If you download an installable package, such as an .exe, .dmg, or PPA, you can install it the same way as any application on your operating system. For Windows, run the .exe and follow the step-by-step instructions. For Mac OS, launch the .dmg and drag the Bitcoin-Qt icon into your *Applications* folder. For Ubuntu, double-click the PPA in your File Explorer and it will open the package manager to install the package. Once you have completed installation you should have a new application called Bitcoin-Qt in your application list. Double-click the icon to start the bitcoin client.

The first time you run Bitcoin Core it will start downloading the blockchain, a process that might take several days (see [Bitcoin Core screen during the blockchain initialization](#)). Leave it running in the background until it displays "Synchronized" and no longer shows "out of sync" next to the balance.

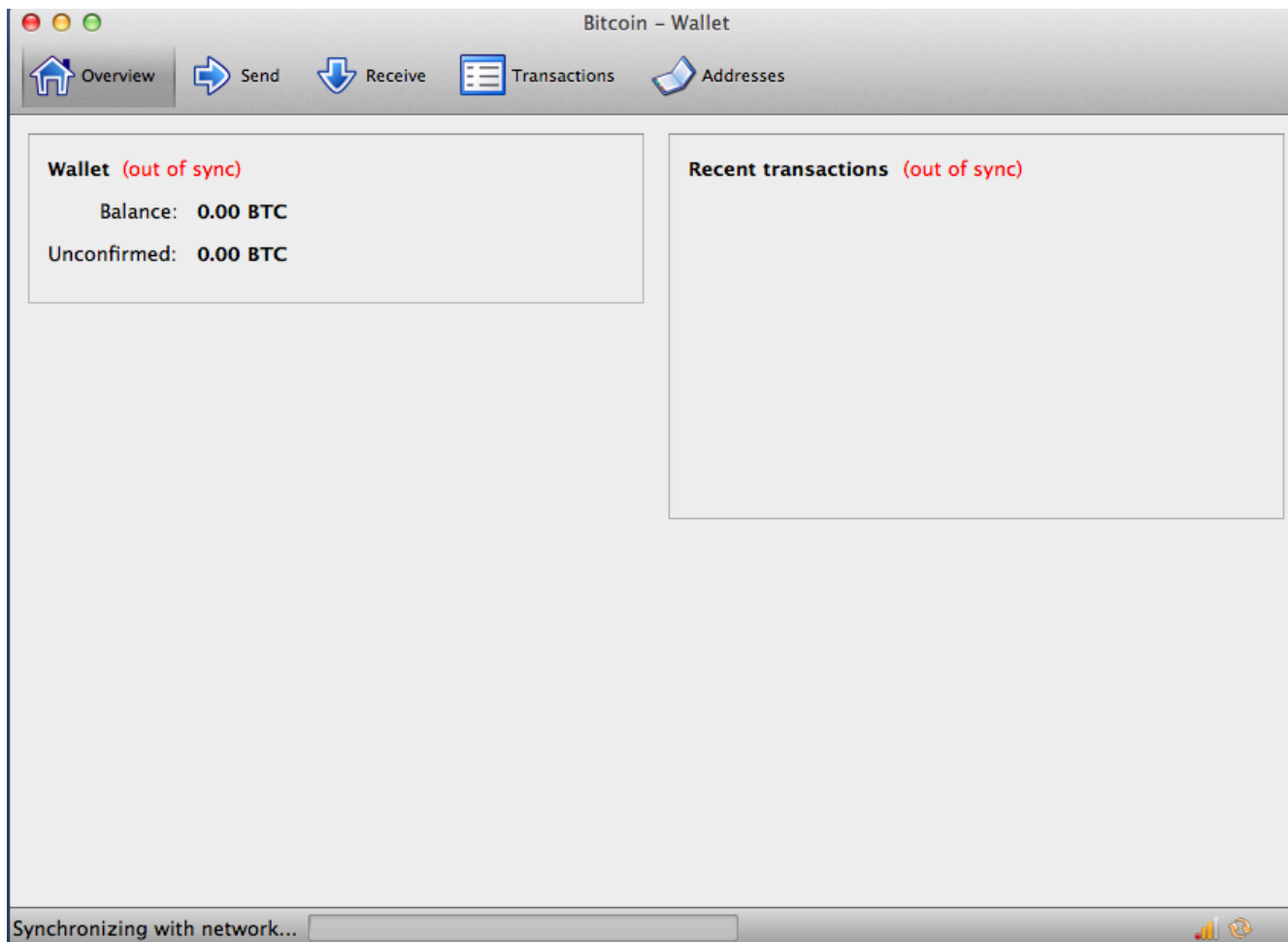


Figure 16. Bitcoin Core screen during the blockchain initialization

TIP

Bitcoin Core keeps a full copy of the transaction ledger (blockchain), with every transaction that has ever occurred on the bitcoin network since its inception in 2009. This dataset is several gigabytes in size (approximately 16 GB in late 2013) and is downloaded incrementally over several days. The client will not be able to process transactions or update account balances until the full blockchain dataset is downloaded. During that time, the client will display "out of sync" next to the account balances and show "Synchronizing" in the footer. Make sure you have enough disk space, bandwidth, and time to complete the initial synchronization.

Compiling Bitcoin Core from the Source Code

For developers, there is also the option to download the full source code as a ZIP archive or by cloning the authoritative source repository from GitHub. On the [GitHub bitcoin page](#), select Download ZIP from the sidebar. Alternatively, use the git command line to create a local copy of the source code on your system. In the following example, we are cloning the source code from a Unix-like command line, in Linux or Mac OS:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

TIP

The instructions and resulting output might vary from version to version. Follow the documentation that comes with the code even if it differs from the instructions you see here, and don't be surprised if the output displayed on your screen is slightly different from the examples here.

When the git cloning operation has completed, you will have a complete local copy of the source code repository in the directory *bitcoin*. Change to this directory by typing `cd bitcoin` at the prompt:

```
$ cd bitcoin
```

By default, the local copy will be synchronized with the most recent code, which might be an unstable or beta version of bitcoin. Before compiling the code, select a specific version by checking out a release *tag*. This will synchronize the local copy with a specific snapshot of the code repository identified by a keyword tag. Tags are used by the developers to mark specific releases of the code by version number. First, to find the available tags, we use the `git tag` command:

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... many more tags ...]

v0.8.4rc2
v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

The list of tags shows all the released versions of bitcoin. By convention, *release candidates*, which are intended for testing, have the suffix "rc". Stable releases that can be run on production systems have no suffix. From the preceding list, select the highest version release, which at this writing was v0.9.0rc1. To synchronize the local code with this version, use the `git checkout` command:

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

The source code includes documentation, which can be found in a number of files. Review the main documentation located in *README.md* in the bitcoin directory by typing `more README.md` at the prompt and using the space bar to progress to the next page. In this chapter, we will build the command-line bitcoin client, also known as *bitcoind* on Linux. Review the instructions for compiling the bitcoind command-line client on your platform by typing `more doc/build-unix.md`. Alternative instructions for Mac OS X and Windows can be found in the *doc* directory, as *build-osx.md* or *build-msw.md*, respectively.

Carefully review the build prerequisites, which are in the first part of the build documentation. These are libraries that must be present on your system before you can begin to compile bitcoin. If these prerequisites are missing, the build process will fail with an error. If this happens because you missed a prerequisite, you can install it and then resume the build process from where you left off. Assuming the prerequisites are installed, you start the build process by generating a set of build scripts using the *autogen.sh* script.

TIP

The Bitcoin Core build process was changed to use the autogen/configure/make system starting with version 0.9. Older versions use a simple Makefile and work slightly differently from the following example. Follow the instructions for the version you want to compile. The autogen/configure/make introduced in 0.9 is likely to be the build system used for all future versions of the code and is the system demonstrated in the following examples.

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

The *autogen.sh* script creates a set of automatic configuration scripts that will interrogate your system to discover the correct settings and ensure you have all the necessary libraries to compile the code. The most important of these is the *configure* script that offers a number of different options to customize the build process. Type `./configure --help` to see the various options:

```
$ ./configure --help
```

'configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.

Usage: ./configure [OPTION]... [VAR=VALUE]...

To assign environment variables (e.g., CC, CFLAGS...), specify them as VAR=VALUE. See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:

-h, --help	display this help and exit
--help=short	display options specific to this package
--help=recursive	display the short help of all the included packages
-V, --version	display version information and exit

[... many more options and variables are displayed below ...]

Optional Features:

--disable-option-checking	ignore unrecognized --enable/--with options
--disable-FEATURE	do not include FEATURE (same as --enable-FEATURE=no)
--enable-FEATURE[=ARG]	include FEATURE [ARG=yes]

[... more options ...]

Use these variables to override the choices made by 'configure' or to help it to find libraries and programs with nonstandard names/locations.

Report bugs to <info@bitcoin.org>.

```
$
```

The **configure** script allows you to enable or disable certain features of bitcoind through the use of the **--enable-FEATURE** and **--disable-FEATURE** flags, where **FEATURE** is replaced by the feature name, as listed in the help output. In this chapter, we will build the bitcoind client with all the default features. We won't be using the configuration flags, but you should review them to understand what optional features are part of the client. Next, run the **configure** script to automatically discover all the necessary libraries and create a customized build script for your system:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... many more system features are tested ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

If all goes well, the **configure** command will end by creating the customized build scripts that will allow us to compile bitcoind. If there are any missing libraries or errors, the **configure** command will terminate with an error instead of creating the build scripts. If an error occurs, it is most likely because of a missing or incompatible library. Review the build documentation again and make sure you install the missing prerequisites. Then run **configure** again and see if that fixes the error. Next, you will compile the source code, a process that can take up to an hour to complete. During the compilation process you should see output every few seconds or every few minutes, or an error if something goes wrong. The compilation process can be resumed at any time if interrupted. Type **make** to start compiling:

```

$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX      addrman.o
  CXX      alert.o
  CXX      rpcserver.o
  CXX      bloom.o
  CXX      chainparams.o

[... many more compilation messages follow ...]

  CXX      test_bitcoin-wallet_tests.o
  CXX      test_bitcoin-rpc_wallet_tests.o
  CXXLD    test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$

```

If all goes well, bitcoind is now compiled. The final step is to install the bitcoind executable into the system path using the **make** command:

```

$ sudo make install
Making install in src
Making install in .
  /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make install-am
  /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$

```

You can confirm that bitcoin is correctly installed by asking the system for the path of the two executables, as follows:

```
$ which bitcoind
/usr/local/bin/bitcoind

$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

The default installation of bitcoind puts it in `/usr/local/bin`. When you first run bitcoind, it will remind you to create a configuration file with a strong password for the JSON-RPC interface. Run bitcoind by typing `bitcoind` into the terminal:

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configuration
file:
/home/ubuntu/.bitcoin/bitcoin.conf
It is recommended you use the following random password:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDewEY2nM6M4H9Tx5dFjoAVVbK
(you do not need to remember this password)
The username and password MUST NOT be the same.
If the file does not exist, create it with owner-readable-only file permissions.
It is also recommended to set alertnotify so you are notified of problems;
for example: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

Edit the configuration file in your preferred editor and set the parameters, replacing the password with a strong password as recommended by bitcoind. Do *not* use the password shown here. Create a file inside the `.bitcoin` directory so that it is named `.bitcoin/bitcoin.conf` and enter a username and password:

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDewEY2nM6M4H9Tx5dFjoAVVbK
```

While you're editing this configuration file, you might want to set a few other options, such as `txindex` (see [Transaction Database Index and txindex Option](#)). For a full listing of the available options, type `bitcoind --help`.

Now, run the Bitcoin Core client. The first time you run it, it will rebuild the bitcoin blockchain by downloading all the blocks. This is a multigigabyte file and will take an average of two days to download in full. You can shorten the blockchain initialization time by downloading a partial copy of the blockchain using a BitTorrent client from [SourceForge](#).

Run bitcoind in the background with the option `-daemon`:

```
$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... more startup messages ...]
```

Using Bitcoin Core's JSON-RPC API from the Command Line

The Bitcoin Core client implements a JSON-RPC interface that can also be accessed using the command-line helper **bitcoin-cli**. The command line allows us to experiment interactively with the capabilities that are also available programmatically via the API. To start, invoke the **help** command to see a list of the available bitcoin RPC commands:

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
decoderawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
getblockchaininfo
getblockcount
getblockhash index
```

```

getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" (
[{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...]
["privatekey1",...] sighashtype )
stop
submitblock "hexdata" ( "jsonparametersobject" )

```

```
validateaddress "bitcoinaddress"  
verifychain ( checklevel numblocks )  
verifymessage "bitcoinaddress" "signature" "message"  
walletlock  
walletpassphrase "passphrase" timeout  
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

Getting Information on the Bitcoin Core Client Status

Commands: `getinfo`

Bitcoin's `getinfo` RPC command displays basic information about the status of the bitcoin network node, the wallet, and the blockchain database. Use `bitcoin-cli` to run it:

```
$ bitcoin-cli getinfo
```

```
{  
  "version" : 90000,  
  "protocolversion" : 70002,  
  "walletversion" : 60000,  
  "balance" : 0.00000000,  
  "blocks" : 286216,  
  "timeoffset" : -72,  
  "connections" : 4,  
  "proxy" : "",  
  "difficulty" : 2621404453.06461525,  
  "testnet" : false,  
  "keypoololdest" : 1374553827,  
  "keypoolsize" : 101,  
  "paytxfee" : 0.00000000,  
  "errors" : ""  
}
```

The data is returned in JavaScript Object Notation (JSON), a format that can easily be "consumed" by all programming languages but is also quite human-readable. Among this data we see the version numbers for the bitcoin software client (90000), protocol (70002), and wallet (60000). We see the current balance contained in the wallet, which is zero. We see the current block height, showing us how many blocks are known to this client (286216). We also see various statistics about the bitcoin network and the settings related to this client. We will explore these settings in more detail in the rest of this chapter.

TIP

It will take some time, perhaps more than a day, for the bitcoind client to "catch up" to the current blockchain height as it downloads blocks from other bitcoin clients. You can check its progress using `getinfo` to see the number of known blocks.

Wallet Setup and Encryption

Commands: `encryptwallet`, `walletpassphrase`

Before you proceed with creating keys and other commands, you should first encrypt the wallet with a password. For this example, you will use the `encryptwallet` command with the password "foo". Obviously, replace "foo" with a strong and complex password!

```
$ bitcoin-cli encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The
keypool has been flushed, you need to make a new backup.
$
```

You can verify the wallet has been encrypted by running `getinfo` again. This time you will notice a new entry called `unlocked_until`. This is a counter showing how long the wallet decryption password will be stored in memory, keeping the wallet unlocked. At first this will be set to zero, meaning the wallet is locked:

```
$ bitcoin-cli getinfo
```

```
{
  "version" : 90000,
  #[... other information...]
  "unlocked_until" : 0,
  "errors" : ""
}
$
```

To unlock the wallet, issue the `walletpassphrase` command, which takes two parameters—the password and a number of seconds until the wallet is locked again automatically (a time counter):

```
$ bitcoin-cli walletpassphrase foo 360
$
```

You can confirm the wallet is unlocked and see the timeout by running `getinfo` again:

```
$ bitcoin-cli getinfo
```

```
{
  "version" : 90000,

  #[...] other information ...]

  "unlocked_until" : 1392580909,
  "errors" : ""
}
```

Wallet Backup, Plain-text Dump, and Restore

Commands: `backupwallet`, `importwallet`, `dumpwallet`

Next, we will practice creating a wallet backup file and then restoring the wallet from the backup file. Use the `backupwallet` command to back up, providing the filename as the parameter. Here we back up the wallet to the file *wallet.backup*:

```
$ bitcoin-cli backupwallet wallet.backup
$
```

Now, to restore the backup file, use the `importwallet` command. If your wallet is locked, you will need to unlock it first (see `walletpassphrase` in the preceding section) in order to import the backup file:

```
$ bitcoin-cli importwallet wallet.backup
$
```

The `dumpwallet` command can be used to dump the wallet into a text file that is human-readable:

```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(0000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 #
addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 #
addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[... many more keys ...]

$
```

Wallet Addresses and Receiving Transactions

Commands: `getnewaddress`, `getreceivedbyaddress`, `listtransactions`, `getaddressesbyaccount`, `getbalance`

The bitcoin reference client maintains a pool of addresses, the size of which is displayed by `keypoolsize` when you use the command `getinfo`. These addresses are generated automatically and can then be used as public receiving addresses or change addresses. To get one of these addresses, use the `getnewaddress` command:

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

Now, we can use this address to send a small amount of bitcoin to our bitcoind wallet from an external wallet (assuming you have some bitcoin in an exchange, web wallet, or other bitcoind wallet held elsewhere). For this example, we will send 50 millibits (0.050 bitcoin) to the preceding address.

We can now query the bitcoind client for the amount received by this address, and specify how many confirmations are required before an amount is counted in that balance. For this example, we will specify zero confirmations. A few seconds after sending the bitcoin from another wallet, we will see it reflected in the wallet. We use `getreceivedbyaddress` with the address and the number of confirmations set to zero (0):

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

If we omit the zero from the end of this command, we will only see the amounts that have at least `minconf` confirmations, where `minconf` is the setting for the minimum number of confirmations before a transaction is listed in the balance. The `minconf` setting is specified in the bitcoind configuration file. Because the transaction sending this bitcoin was only sent in the last few seconds, it has still not confirmed and therefore we will see it list a zero balance:

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

The transactions received by the entire wallet can also be displayed using the `listtransactions` command:

```
$ bitcoin-cli listtransactions
```

```
[
  {
    "account" : "",
    "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "category" : "receive",
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908
  }
]
```

We can list all addresses in the entire wallet using the `getaddressesbyaccount` command:

```
$ bitcoin-cli getaddressesbyaccount ""
```

```
[
  "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
  "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
  "1FvRHWWhBBZA8cGRRsGiAeqEzUmjJkJQWR",
  "1NVJK3JJsL41BF1KyxrUyJW5XHjunjfp2jz",
  "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
  "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
  "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
  "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
  "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
  "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
  "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
  "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
  "1LqJZz1D9yHxG4cLkdujngG5jNNGmPeAMD"
]
```

Finally, the command `getbalance` will show the total balance of the wallet, adding up all transactions confirmed with at least `minconf` confirmations:

```
$ bitcoin-cli getbalance
0.05000000
```

TIP

If the transaction has not yet confirmed, the balance returned by `getbalance` will be zero. The configuration option "minconf" determines the minimum number of confirmations that are required before a transaction shows in the balance.

Exploring and Decoding Transactions

Commands: `gettransaction`, `getrawtransaction`, `decoderawtransaction`

We'll now explore the incoming transaction that was listed previously using the `gettransaction` command. We can retrieve a transaction by its transaction hash, shown at `txid` earlier, with the `gettransaction` command:

```
{
  "amount" : 0.05000000,
  "confirmations" : 0,
  "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
  "time" : 1392660908,
  "timereceived" : 1392660908,
  "details" : [
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjB8JU7nBsCSfEVQX5u9CL",
      "category" : "receive",
      "amount" : 0.05000000
    }
  ]
}
```

TIP

Transaction IDs are not authoritative until a transaction has been confirmed. Absence of a transaction hash in the blockchain does not mean the transaction was not processed. This is known as "transaction malleability," because transaction hashes can be modified prior to confirmation in a block. After confirmation, the txid is immutable and authoritative.

The transaction form shown with the command `gettransaction` is the simplified form. To retrieve the full transaction code and decode it, we will use two commands: `getrawtransaction` and `decoderawtransaction`. First, `getrawtransaction` takes the *transaction hash (txid)* as a parameter and returns the full transaction as a "raw" hex string, exactly as it exists on the bitcoin network:

To decode this hex string, use the `decoderawtransaction` command. Copy and paste the hex as the first parameter of `decoderawtransaction` to get the full contents interpreted as a JSON data structure (for formatting reasons the hex string is shortened in the following example):

The transaction decode shows all the components of this transaction, including the transaction inputs and outputs. In this case we see that the transaction that credited our new address with 50 millibits used one input and generated two outputs. The input to this transaction was the output from a previously confirmed transaction (shown as the vin txid starting with `d3c7`). The two outputs correspond to the 50 millibit credit and an output with change back to the sender.

We can further explore the blockchain by examining the previous transaction referenced by its txid in this transaction using the same commands (e.g., `gettransaction`). Jumping from transaction to transaction we can follow a chain of transactions back as the coins are transmitted from owner address to owner address.

Once the transaction we received has been confirmed by inclusion in a block, the `gettransaction` command will return additional information, showing the *block hash (identifier)* in which the transaction was included:

Here, we see the new information in the entries `blockhash` (the hash of the block in which the transaction was included), and `blockindex` with value 18 (indicating that our transaction was the 18th transaction in that block).

Transaction Database Index and `txindex` Option

By default, Bitcoin Core builds a database containing *only* the transactions related to the user's wallet. If you want to be able to access *any* transaction with commands like `gettransaction`, you need to configure Bitcoin Core to build a complete transaction index, which can be achieved with the `txindex` option. Set `txindex=1` in the Bitcoin Core configuration file (usually found in your home directory under `.bitcoin/bitcoin.conf`). Once you change this parameter, you need to restart bitcoind and wait for it to rebuild the index.

Exploring Blocks

Commands: `getblock`, `getblockhash`

Now that we know which block our transaction was included in, we can query that block. We use the `getblock` command with the block hash as the parameter:

The block contains 367 transactions and as you can see, the 18th transaction listed (`9ca8f9...`) is the txid of the one crediting 50 millibits to our address. The `height` entry tells us this is the 286384th block in the blockchain.

We can also retrieve a block by its block height using the `getblockhash` command, which takes the block height as the parameter and returns the block hash for that block:

Here, we retrieve the block hash of the "genesis block," the first block mined by Satoshi Nakamoto, at height zero. Retrieving this block shows:

The `getblock`, `getblockhash`, and `gettransaction` commands can be used to explore the blockchain database, programmatically.

Creating, Signing, and Submitting Transactions Based on Unspent Outputs

Commands: `listunspent`, `gettxout`, `createrawtransaction`, `decoderawtransaction`, `signrawtransaction`, `sendrawtransaction`

Bitcoin's transactions are based on the concept of spending "outputs," which are the result of previous transactions, to create a transaction chain that transfers ownership from address to address. Our wallet has now received a transaction that assigned one such output to our address. Once this is confirmed, we can spend that output.

First, we use the `listunspent` command to show all the unspent *confirmed* outputs in our wallet:

```
$ bitcoin-cli listunspent
```

We see that the transaction `9ca8f9...` created an output (with vout index 0) assigned to the address `1hvzSo...` for the amount of 50 millibits, which at this point has received seven confirmations. Transactions use previously created outputs as their inputs by referring to them by the previous txid and vout index. We will now create a transaction that will spend the 0th vout of the txid `9ca8f9...` as its input and assign it to a new output that sends value to a new address.

First, let's look at the specific output in more detail. We use `gettxout` to get the details of this unspent output. Transaction outputs are always referenced by txid and vout, and these are the parameters we pass to `gettxout`:

What we see here is the output that assigned 50 millibits to our address `1hvz...`. To spend this output we will create a new transaction. First, let's make an address to which we will send the money:

```
$ bitcoin-cli getnewaddress  
1LnFTndy3qzXGN19Jwscj1T8LR3MVe3JDb
```

We will send 25 millibits to the new address `1LnFTn...` we just created in our wallet. In our new transaction, we will spend the 50 millibit output and send 25 millibits to this new address. Because we have to spend the *whole* output from the previous transaction, we must also generate some change. We will generate change back to the `1hvz...` address, sending the change back to the address from which the value originated. Finally, we will also have to pay a fee for this transaction. To pay the fee, we will reduce the change output by 0.5 millibits, and return 24.5 millibits in change. The difference between the sum of the new outputs (25 mBTC + 24.5 mBTC = 49.5 mBTC) and the input (50 mBTC) will be collected as a transaction fee by the miners.

We use `createrawtransaction` to create this transaction. As parameters to `createrawtransaction` we provide the transaction input (the 50 millibit unspent output from our confirmed transaction) and the two transaction outputs (money sent to the new address and change sent back to the previous address):

The `createrawtransaction` command produces a raw hex string that encodes the transaction details we supplied. Let's confirm everything is correct by decoding this raw string using the `decoderawtransaction` command:

That looks correct! Our new transaction "consumes" the unspent output from our confirmed transaction and then spends it in two outputs, one for 25 millibits to our new address and one for 24.5 millibits as change back to the original address. The difference of 0.5 millibits represents the transaction fee and will be credited to the miner who finds the block that includes our transaction.

As you might notice, the transaction contains an empty `scriptSig` because we haven't signed it yet. Without a signature, this transaction is meaningless; we haven't yet proven that we *own* the address from which the unspent output is sourced. By signing, we remove the lock on the output and prove that we own this output and can spend it. We use the `signrawtransaction` command to sign the transaction. It takes the raw transaction hex string as the parameter:

TIP

An encrypted wallet must be unlocked before a transaction is signed because signing requires access to the secret keys in the wallet.

The `signrawtransaction` command returns another hex-encoded raw transaction. We decode it to see what changed, with `decoderawtransaction`:

Now, the inputs used in the transaction contain a `scriptSig`, which is a digital signature proving ownership of address `1hvz...` and removing the lock on the output so that it can be spent. The signature makes this transaction verifiable by any node in the bitcoin network.

Now it's time to submit the newly created transaction to the network. We do that with the command `sendrawtransaction`, which takes the raw hex string produced by `signrawtransaction`. This is the same string we just decoded:

The command `sendrawtransaction` returns a *transaction hash (txid)* as it submits the transaction on the network. We can now query that transaction ID with `gettransaction`:

```

{
  "amount" : 0.00000000,
  "fee" : -0.00050000,
  "confirmations" : 0,
  "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
  "time" : 1392666702,
  "timereceived" : 1392666702,
  "details" : [
    {
      "account" : "",
      "address" : "1LnFTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "category" : "send",
      "amount" : -0.02500000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "category" : "send",
      "amount" : -0.02450000,
      "fee" : -0.00050000
    },
    {
      "account" : "",
      "address" : "1LnFTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
      "category" : "receive",
      "amount" : 0.02500000
    },
    {
      "account" : "",
      "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
      "category" : "receive",
      "amount" : 0.02450000
    }
  ]
}

```

As before, we can also examine this in more detail using the `getrawtransaction` and `decodetransaction` commands. These commands will return the exact same hex string that we produced and decoded previously just before we sent it on the network.

Alternative Clients, Libraries, and Toolkits

Beyond the reference client (bitcoind), other clients and libraries can be used to interact with the bitcoin network and data structures. These are implemented in a variety of programming languages, offering programmers native interfaces in their own language.

Alternative implementations include:

libbitcoin

Bitcoin Cross-Platform C++ Development Toolkit

bitcoin explorer

Bitcoin Command Line Tool

bitcoin server

Bitcoin Full Node and Query Server

bitcoinj

A Java full-node client library

btcd

A Go language full-node bitcoin client

Bits of Proof (BOP)

A Java enterprise-class implementation of bitcoin

picocoin

A C implementation of a lightweight client library for bitcoin

pybitcointools

A Python bitcoin library

pycoin

Another Python bitcoin library

Many more libraries exist in a variety of other programming languages and more are created all the time.

Libbitcoin and Bitcoin Explorer

The libbitcoin library is a cross-platform C++ development toolkit that supports the libbitcoin-server full node and the Bitcoin Explorer (bx) command line tool.

The bx commands offer many of the same capabilities as the bitcoind client commands we illustrated in this chapter. The bx commands also offer some key management and manipulation tools that are not offered by bitcoind, including type-2 deterministic keys and mnemonic key encoding, as well as stealth address, payment, and query support.

Installing Bitcoin Explorer

To use Bitcoin Explorer, simply [download the signed executable for your operating system](#). Builds are available for mainnet and testnet for Linux, OS X, and Windows.

Type **bx** with no parameters to display the list of all available commands (see [Bitcoin Explorer \(bx\) Commands](#)).

Bitcoin Explorer also provides an installer for [building from sources on Linux and OS X, as well as](#)

[Visual Studio projects for Windows](#). Sources can also be built manually using Autotools. These also install the libbitcoin library dependency.

TIP Bitcoin Explorer offers many useful commands for encoding and decoding addresses, and converting to and from different formats and representations. Use them to explore the various formats such as Base16 (hex), Base58, Base58Check, Base64, etc.

Installing Libbitcoin

The libbitcoin library provides an installer for [building from sources on Linux and OS X, as well as Visual Studio projects for Windows](#). Sources can also be built manually using Autotools.

TIP The Bitcoin Explorer installer installs both bx and the libbitcoin library, so if you have built bx from sources, you can skip this step.

pycoin

The Python library [pycoin](#), originally written and maintained by Richard Kiss, is a Python-based library that supports manipulation of bitcoin keys and transactions, even supporting the scripting language enough to properly deal with nonstandard transactions.

The pycoin library supports both Python 2 (2.7.x) and Python 3 (after 3.3), and comes with some handy command-line utilities, ku and tx. To install pycoin 0.42 under Python 3 in a virtual environment (venv), use the following:

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Downloading/unpacking pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded
  Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info for package
pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

Here's a sample Python script to fetch and spend some bitcoin using the pycoin library:

```
#!/usr/bin/env python

from pycoin.key import Key

from pycoin.key.validate import is_address_valid, is_wif_valid
from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address)
print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False),
key.address(use_uncompressed=True)):
    print("** WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())
```

For examples using the command-line utilities ku and tx, see [pycoin](#), [ku](#), and [tx](#).

btcd

btcd is a full-node bitcoin implementation written in Go. It currently downloads, validates, and serves the blockchain using the exact rules (including bugs) for block acceptance as the reference implementation, bitcoind. It also properly relays newly mined blocks, maintains a transaction pool, and relays individual transactions that have not yet made it into a block. It ensures that all individual transactions admitted to the pool follow the rules required and also includes the vast majority of the more strict checks that filter transactions based on miner requirements ("standard" transactions).

One key difference between btcd and bitcoind is that btcd does not include wallet functionality, and this was a very intentional design decision. This means you can't actually make or receive payments directly with btcd. That functionality is provided by the btcwallet and btcgui projects, which are both under active development. Other notable differences between btcd and bitcoind include btcd support for both HTTP POST requests (such as bitcoind) and the preferred Websockets, and the fact that btcd's RPC connections are TLS-enabled by default.

Installing btcd

To install btcd for Windows, download and run the msi available at [GitHub](#), or run the following command on Linux, assuming you already have installed the Go language:

```
$ go get github.com/conformal/btcd/...
```

To update btcd to the latest version, just run:

```
$ go get -u -v github.com/conformal/btcd/...
```

Controlling btcd

btcd has a number of configuration options, which you can view by running:

```
$ btcd --help
```

btcd comes prepackaged with some goodies such as btcctl, which is a command-line utility that can be used to both control and query btcd via RPC. btcd does not enable its RPC server by default; you must configure at minimum both an RPC username and password in the following configuration files:

- *btcd.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

- *btccctl.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

Or if you want to override the configuration files from the command line:

```
$ btcd -u myuser -P SomeDecentp4ssw0rd
$ btccctl -u myuser -P SomeDecentp4ssw0rd
```

For a list of available options, run the following:

```
$ btccctl --help
```

Keys, Addresses, Wallets

Introduction

Ownership of bitcoin is established through *digital keys*, *bitcoin addresses*, and *digital signatures*. The digital keys are not actually stored in the network, but are instead created and stored by users in a file, or simple database, called a *wallet*. The digital keys in a user's wallet are completely independent of the bitcoin protocol and can be generated and managed by the user's wallet software without reference to the blockchain or access to the Internet. Keys enable many of the interesting properties of bitcoin, including de-centralized trust and control, ownership attestation, and the cryptographic-proof security model.

Every bitcoin transaction requires a valid signature to be included in the blockchain, which can only be generated with valid digital keys; therefore, anyone with a copy of those keys has control of the bitcoin in that account. Keys come in pairs consisting of a private (secret) key and a public key. Think of the public key as similar to a bank account number and the private key as similar to the secret PIN, or signature on a check that provides control over the account. These digital keys are very rarely seen by the users of bitcoin. For the most part, they are stored inside the wallet file and managed by the bitcoin wallet software.

In the payment portion of a bitcoin transaction, the recipient's public key is represented by its digital fingerprint, called a *bitcoin address*, which is used in the same way as the beneficiary name on a check (i.e., "Pay to the order of"). In most cases, a bitcoin address is generated from and corresponds to a public key. However, not all bitcoin addresses represent public keys; they can also represent other beneficiaries such as scripts, as we will see later in this chapter. This way, bitcoin addresses abstract the recipient of funds, making transaction destinations flexible, similar to paper checks: a single payment instrument that can be used to pay into people's accounts, pay into company accounts, pay for bills, or pay to cash. The bitcoin address is the only representation of the keys that users will routinely see, because this is the part they need to share with the world.

In this chapter we will introduce wallets, which contain cryptographic keys. We will look at how keys are generated, stored, and managed. We will review the various encoding formats used to represent private and public keys, addresses, and script addresses. Finally, we will look at special uses of keys: to sign messages, to prove ownership, and to create vanity addresses and paper wallets.

Public Key Cryptography and Cryptocurrency

Public key cryptography was invented in the 1970s and is a mathematical foundation for computer and information security.

Since the invention of public key cryptography, several suitable mathematical functions, such as prime number exponentiation and elliptic curve multiplication, have been discovered. These mathematical functions are practically irreversible, meaning that they are easy to calculate in one direction and infeasible to calculate in the opposite direction. Based on these mathematical functions, cryptography enables the creation of digital secrets and unforgeable digital signatures. Bitcoin uses elliptic curve multiplication as the basis for its public key cryptography.

In bitcoin, we use public key cryptography to create a key pair that controls access to bitcoins. The key pair consists of a private key and—derived from it—a unique public key. The public key is used to receive bitcoins, and the private key is used to sign transactions to spend those bitcoins.

There is a mathematical relationship between the public and the private key that allows the private key to be used to generate signatures on messages. This signature can be validated against the public key without revealing the private key.

When spending bitcoins, the current bitcoin owner presents her public key and a signature (different each time, but created from the same private key) in a transaction to spend those bitcoins. Through the presentation of the public key and signature, everyone in the bitcoin network can verify and accept the transaction as valid, confirming that the person transferring the bitcoins owned them at the time of the transfer.

TIP In most wallet implementations, the private and public keys are stored together as a *key pair* for convenience. However, the public key can be calculated from the private key, so storing only the private key is also possible.

Private and Public Keys

A bitcoin wallet contains a collection of key pairs, each consisting of a private key and a public key. The private key (k) is a number, usually picked at random. From the private key, we use elliptic curve multiplication, a one-way cryptographic function, to generate a public key (K). From the public key (K), we use a one-way cryptographic hash function to generate a bitcoin address (A). In this section, we will start with generating the private key, look at the elliptic curve math that is used to turn that into a public key, and finally, generate a bitcoin address from the public key. The relationship between private key, public key, and bitcoin address is shown in [Private key, public key, and bitcoin address](#).

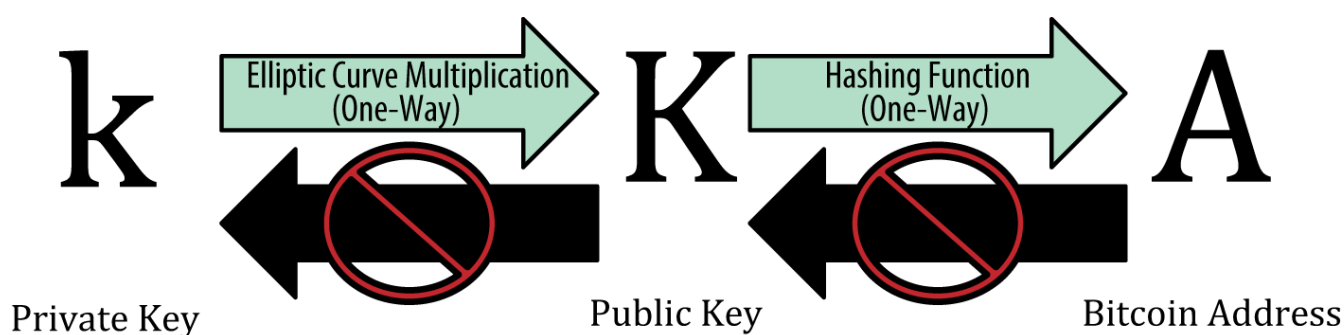


Figure 17. Private key, public key, and bitcoin address

Private Keys

A private key is simply a number, picked at random. Ownership and control over the private key is the root of user control over all funds associated with the corresponding bitcoin address. The private key is used to create signatures that are required to spend bitcoins by proving ownership of funds used in a transaction. The private key must remain secret at all times, because revealing it to third parties is equivalent to giving them control over the bitcoins secured by that key. The private key must also be backed up and protected from accidental loss, because if it's lost it cannot be recovered and the funds secured by it are forever lost, too.

TIP

The bitcoin private key is just a number. You can pick your private keys randomly using just a coin, pencil, and paper: toss a coin 256 times and you have the binary digits of a random private key you can use in a bitcoin wallet. The public key can then be generated from the private key.

Generating a private key from a random number

The first and most important step in generating keys is to find a secure source of entropy, or randomness. Creating a bitcoin key is essentially the same as "Pick a number between 1 and 2^{256} ." The exact method you use to pick that number does not matter as long as it is not predictable or repeatable. Bitcoin software uses the underlying operating system's random number generators to produce 256 bits of entropy (randomness). Usually, the OS random number generator is initialized by a human source of randomness, which is why you may be asked to wiggle your mouse around for a few seconds. For the truly paranoid, nothing beats dice, pencil, and paper.

More accurately, the private key can be any number between 1 and $n - 1$, where n is a constant ($n = 1.158 * 10^{77}$, slightly less than 2^{256}) defined as the order of the elliptic curve used in bitcoin (see [Elliptic Curve Cryptography Explained](#)). To create such a key, we randomly pick a 256-bit number and check that it is less than $n - 1$. In programming terms, this is usually achieved by feeding a larger string of random bits, collected from a cryptographically secure source of randomness, into the SHA256 hash algorithm that will conveniently produce a 256-bit number. If the result is less than $n - 1$, we have a suitable private key. Otherwise, we simply try again with another random number.

TIP

Do not write your own code to create a random number or use a "simple" random number generator offered by your programming language. Use a cryptographically secure pseudo-random number generator (CSPRNG) with a seed from a source of sufficient entropy. Study the documentation of the random number generator library you choose to make sure it is cryptographically secure. Correct implementation of the CSPRNG is critical to the security of the keys.

The following is a randomly generated private key (k) shown in hexadecimal format (256 binary digits shown as 64 hexadecimal digits, each 4 bits):

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

TIP

The size of bitcoin's private key space, 2^{256} is an unfathomably large number. It is approximately 10^{77} in decimal. The visible universe is estimated to contain 10^{80} atoms.

To generate a new key with the Bitcoin Core client (see [The Bitcoin Client](#)), use the `getnewaddress` command. For security reasons it displays the public key only, not the private key. To ask bitcoind to expose the private key, use the `dumpprivkey` command. The `dumpprivkey` command shows the private key in a Base58 checksum-encoded format called the *Wallet Import Format* (WIF), which we will examine in more detail in [Private key formats](#). Here's an example of generating and displaying a private key using these two commands:

```
$ bitcoind getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwWCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

The `dumpprivkey` command opens the wallet and extracts the private key that was generated by the `getnewaddress` command. It is not otherwise possible for `bitcoind` to know the private key from the public key, unless they are both stored in the wallet.

TIP

The `dumpprivkey` command is not generating a private key from a public key, as this is impossible. The command simply reveals the private key that is already known to the wallet and which was generated by the `getnewaddress` command.

You can also use the Bitcoin Explorer command-line tool (see [Libbitcoin](#) and [Bitcoin Explorer](#)) to generate and display private keys with the commands `seed`, `ec-new` and `ec-to-wif`:

```
$ bx seed | bx ec-new | bx ec-to-wif
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Public Keys

The public key is calculated from the private key using elliptic curve multiplication, which is irreversible: $(K = k * G)$ where k is the private key, G is a constant point called the *generator point* and K is the resulting public key. The reverse operation, known as "finding the discrete logarithm"—calculating k if you know K —is as difficult as trying all possible values of k , i.e., a brute-force search. Before we demonstrate how to generate a public key from a private key, let's look at elliptic curve cryptography in a bit more detail.

Elliptic Curve Cryptography Explained

Elliptic curve cryptography is a type of asymmetric or public-key cryptography based on the discrete logarithm problem as expressed by addition and multiplication on the points of an elliptic curve.

[An elliptic curve](#) is an example of an elliptic curve, similar to that used by bitcoin.

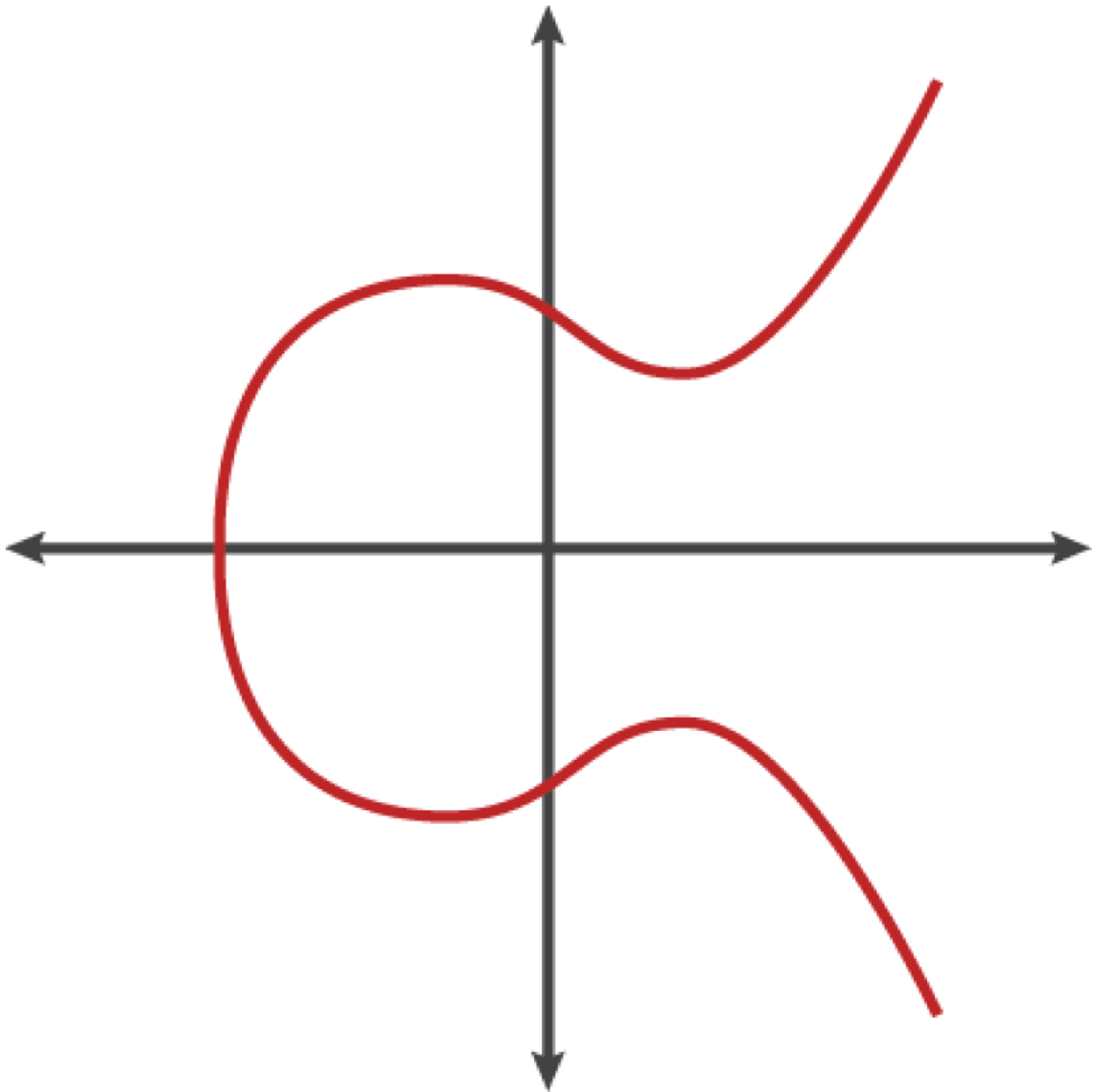


Figure 18. An elliptic curve

Bitcoin uses a specific elliptic curve and set of mathematical constants, as defined in a standard called **secp256k1**, established by the National Institute of Standards and Technology (NIST). The **secp256k1** curve is defined by the following function, which produces an elliptic curve:

or

The *mod p* (modulo prime number p) indicates that this curve is over a finite field of prime order p , also written as $\mathbb{F}_p$, where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, a very large prime number.

Because this curve is defined over a finite field of prime order instead of over the real numbers, it looks like a pattern of dots scattered in two dimensions, which makes it difficult to visualize. However, the math is identical as that of an elliptic curve over the real numbers. As an example, [Elliptic curve cryptography: visualizing an elliptic curve over \$\mathbb{F}\(p\)\$, with \$p=17\$](#) shows the same elliptic curve over a much smaller finite field of prime order 17, showing a pattern of dots on a grid.

The `secp256k1` bitcoin elliptic curve can be thought of as a much more complex pattern of dots on a unfathomably large grid.

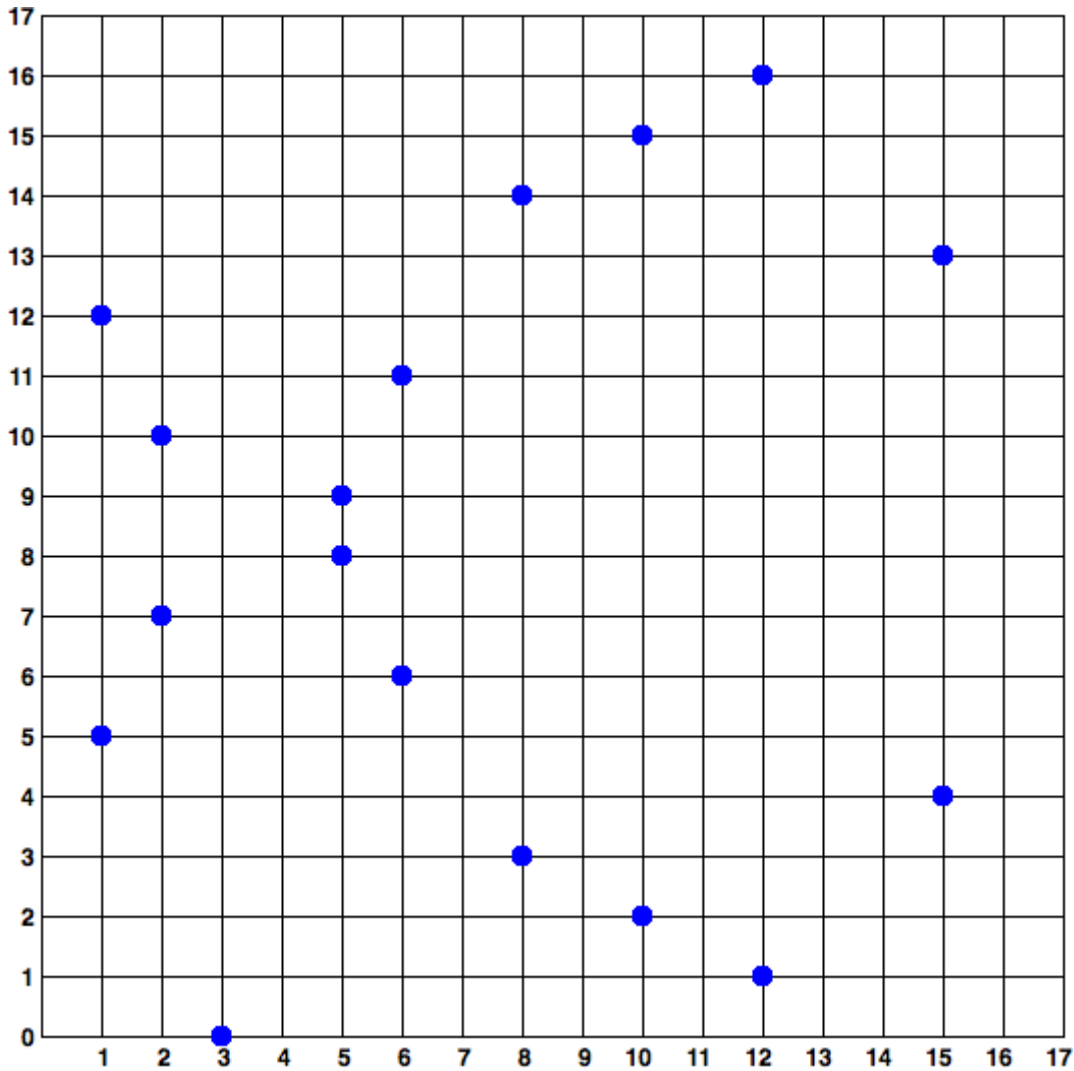


Figure 19. Elliptic curve cryptography: visualizing an elliptic curve over $F(p)$, with $p=17$

So, for example, the following is a point P with coordinates (x,y) that is a point on the `secp256k1` curve. You can check this yourself using Python:

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240,  
32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

```

Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p =
115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x =
55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y =
32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0

```

In elliptic curve math, there is a point called the "point at infinity," which roughly corresponds to the role of 0 in addition. On computers, it's sometimes represented by $x = y = 0$ (which doesn't satisfy the elliptic curve equation, but it's an easy separate case that can be checked).

There is also a $+$ operator, called "addition," which has some properties similar to the traditional addition of real numbers that grade school children learn. Given two points P_1 and P_2 on the elliptic curve, there is a third point $P_3 = P_1 + P_2$, also on the elliptic curve.

Geometrically, this third point P_3 is calculated by drawing a line between P_1 and P_2 . This line will intersect the elliptic curve in exactly one additional place. Call this point $P_3' = (x, y)$. Then reflect in the x-axis to get $P_3 = (x, -y)$.

There are a couple of special cases that explain the need for the "point at infinity."

If P_1 and P_2 are the same point, the line "between" P_1 and P_2 should extend to be the tangent on the curve at this point P_1 . This tangent will intersect the curve in exactly one new point. You can use techniques from calculus to determine the slope of the tangent line. These techniques curiously work, even though we are restricting our interest to points on the curve with two integer coordinates!

In some cases (i.e., if P_1 and P_2 have the same x values but different y values), the tangent line will be exactly vertical, in which case $P_3 =$ "point at infinity."

If P_1 is the "point at infinity," then the sum $P_1 + P_2 = P_2$. Similarly, if P_2 is the point at infinity, then $P_1 + P_2 = P_1$. This shows how the point at infinity plays the role of 0.

It turns out that $+$ is associative, which means that $(A + B) + C = A + (B + C)$. That means we can write $A + B + C$ without parentheses without any ambiguity.

Now that we have defined addition, we can define multiplication in the standard way that extends addition. For a point P on the elliptic curve, if k is a whole number, then $kP = P + P + P + \dots + P$ (k times). Note that k is sometimes confusingly called an "exponent" in this case.

Generating a Public Key

Starting with a private key in the form of a randomly generated number k , we multiply it by a predetermined point on the curve called the *generator point* G to produce another point somewhere else on the curve, which is the corresponding public key K . The generator point is specified as part of the `secp256k1` standard and is always the same for all keys in bitcoin:

where k is the private key, G is the generator point, and K is the resulting public key, a point on the curve. Because the generator point is always the same for all bitcoin users, a private key k multiplied with G will always result in the same public key K . The relationship between k and K is fixed, but can only be calculated in one direction, from k to K . That's why a bitcoin address (derived from K) can be shared with anyone and does not reveal the user's private key (k).

TIP

A private key can be converted into a public key, but a public key cannot be converted back into a private key because the math only works one way.

Implementing the elliptic curve multiplication, we take the private key k generated previously and multiply it with the generator point G to find the public key K :

```
K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G
```

Public Key K is defined as a point $K = (x, y)$:

$K = (x, y)$

where,

$x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A$

$y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB$

To visualize multiplication of a point with an integer, we will use the simpler elliptic curve over the real numbers—remember, the math is the same. Our goal is to find the multiple kG of the generator point G . That is the same as adding G to itself, k times in a row. In elliptic curves, adding a point to itself is the equivalent of drawing a tangent line on the point and finding where it intersects the curve again, then reflecting that point on the x -axis.

[Elliptic curve cryptography: Visualizing the multiplication of a point \$G\$ by an integer \$k\$ on an elliptic curve](#) shows the process for deriving G , $2G$, $4G$, as a geometric operation on the curve.

TIP

Most bitcoin implementations use the [OpenSSL cryptographic library](#) to do the elliptic curve math. For example, to derive the public key, the function `EC_POINT_mul()` is used.

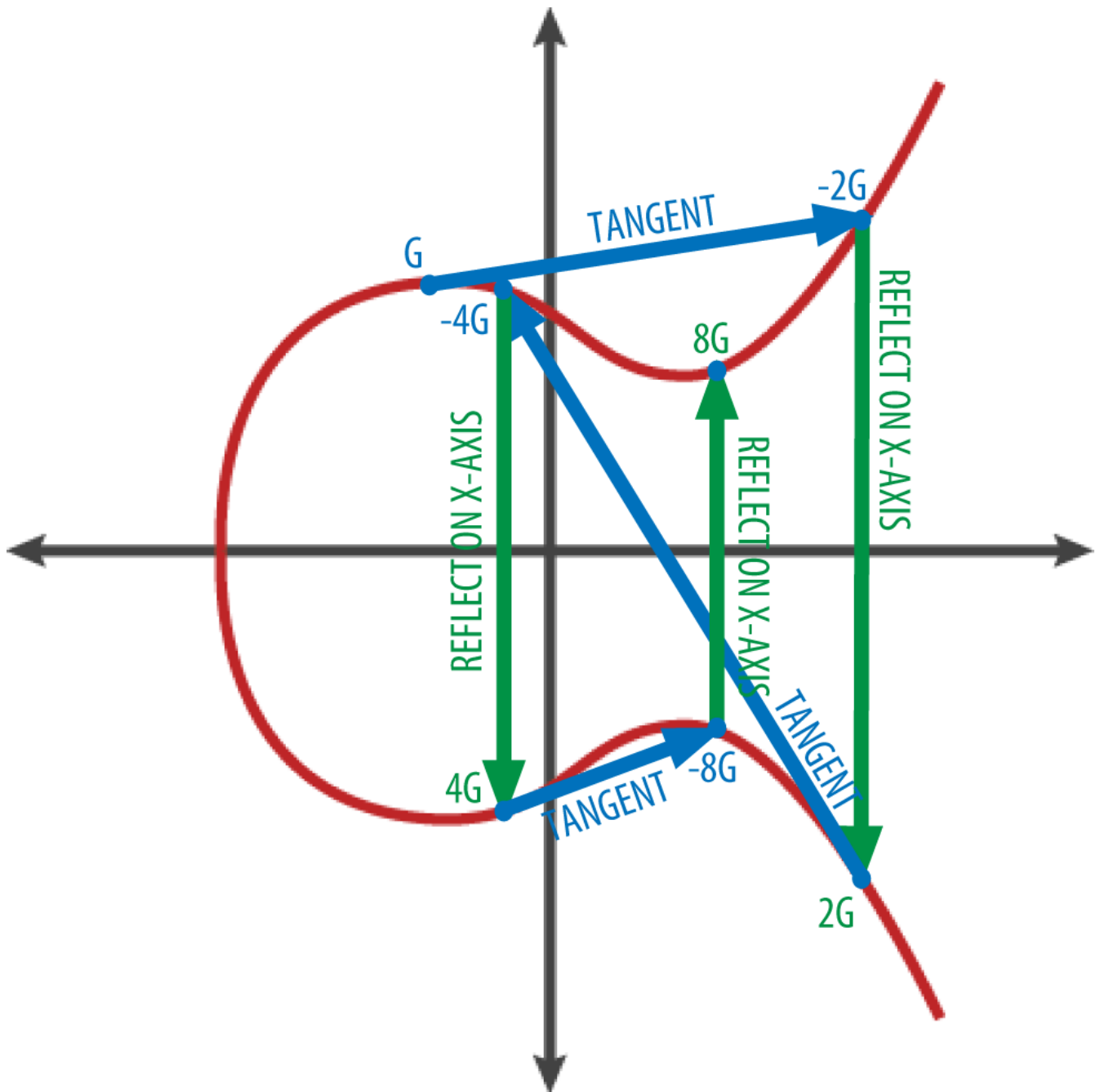


Figure 20. Elliptic curve cryptography: Visualizing the multiplication of a point G by an integer k on an elliptic curve

Bitcoin Addresses

A bitcoin address is a string of digits and characters that can be shared with anyone who wants to send you money. Addresses produced from public keys consist of a string of numbers and letters, beginning with the digit "1". Here's an example of a bitcoin address:

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

The bitcoin address is what appears most commonly in a transaction as the "recipient" of the funds. If we were to compare a bitcoin transaction to a paper check, the bitcoin address is the beneficiary, which is what we write on the line after "Pay to the order of." On a paper check, that beneficiary

can sometimes be the name of a bank account holder, but can also include corporations, institutions, or even cash. Because paper checks do not need to specify an account, but rather use an abstract name as the recipient of funds, that makes paper checks very flexible as payment instruments. Bitcoin transactions use a similar abstraction, the bitcoin address, to make them very flexible. A bitcoin address can represent the owner of a private/public key pair, or it can represent something else, such as a payment script, as we will see in [Pay-to-Script-Hash \(P2SH\)](#). For now, let's examine the simple case, a bitcoin address that represents, and is derived from, a public key.

The bitcoin address is derived from the public key through the use of one-way cryptographic hashing. A "hashing algorithm" or simply "hash algorithm" is a one-way function that produces a fingerprint or "hash" of an arbitrary-sized input. Cryptographic hash functions are used extensively in bitcoin: in bitcoin addresses, in script addresses, and in the mining proof-of-work algorithm. The algorithms used to make a bitcoin address from a public key are the Secure Hash Algorithm (SHA) and the RACE Integrity Primitives Evaluation Message Digest (RIPEMD), specifically SHA256 and RIPEMD160.

Starting with the public key K , we compute the SHA256 hash and then compute the RIPEMD160 hash of the result, producing a 160-bit (20-byte) number:

where K is the public key and A is the resulting bitcoin address.

TIP

A bitcoin address is *not* the same as a public key. Bitcoin addresses are derived from a public key using a one-way function.

Bitcoin addresses are almost always presented to users in an encoding called "Base58Check" (see [Base58 and Base58Check Encoding](#)), which uses 58 characters (a Base58 number system) and a checksum to help human readability, avoid ambiguity, and protect against errors in address transcription and entry. Base58Check is also used in many other ways in bitcoin, whenever there is a need for a user to read and correctly transcribe a number, such as a bitcoin address, a private key, an encrypted key, or a script hash. In the next section we will examine the mechanics of Base58Check encoding and decoding, and the resulting representations. [Public key to bitcoin address: conversion of a public key into a bitcoin address](#) illustrates the conversion of a public key into a bitcoin address.

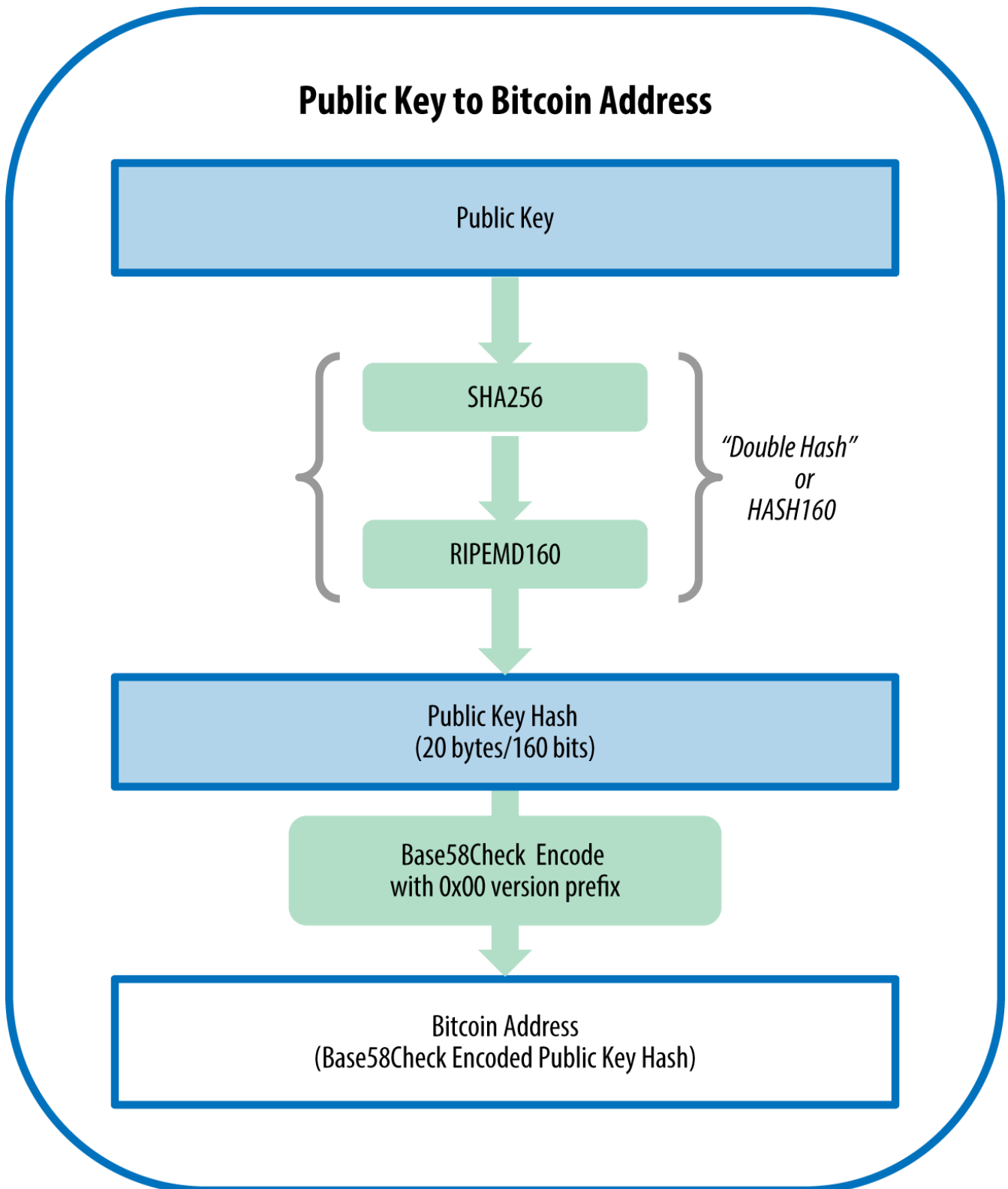


Figure 21. Public key to bitcoin address: conversion of a public key into a bitcoin address

Base58 and Base58Check Encoding

In order to represent long numbers in a compact way, using fewer symbols, many computer systems use mixed-alphanumeric representations with a base (or radix) higher than 10. For example, whereas the traditional decimal system uses the 10 numerals 0 through 9, the hexadecimal system uses 16, with the letters A through F as the six additional symbols. A number represented in hexadecimal format is shorter than the equivalent decimal representation. Even more compact, Base-64 representation uses 26 lower-case letters, 26 capital letters, 10 numerals,

and two more characters such as "+" and "/" to transmit binary data over text-based media such as email. Base-64 is most commonly used to add binary attachments to email. Base58 is a text-based binary-encoding format developed for use in bitcoin and used in many other cryptocurrencies. It offers a balance between compact representation, readability, and error detection and prevention. Base58 is a subset of Base64, using the upper- and lowercase letters and numbers, but omitting some characters that are frequently mistaken for one another and can appear identical when displayed in certain fonts. Specifically, Base58 is Base64 without the 0 (number zero), O (capital o), l (lower L), I (capital i), and the symbols "\" and "/". Or, more simply, it is a set of lower and capital letters and numbers without the four (0, O, l, I) just mentioned.

Example 3. bitcoin's Base58 alphabet

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

To add extra security against typos or transcription errors, Base58Check is a Base58 encoding format, frequently used in bitcoin, which has a built-in error-checking code. The checksum is an additional four bytes added to the end of the data that is being encoded. The checksum is derived from the hash of the encoded data and can therefore be used to detect and prevent transcription and typing errors. When presented with a Base58Check code, the decoding software will calculate the checksum of the data and compare it to the checksum included in the code. If the two do not match, that indicates that an error has been introduced and the Base58Check data is invalid. For example, this prevents a mistyped bitcoin address from being accepted by the wallet software as a valid destination, an error that would otherwise result in loss of funds.

To convert data (a number) into a Base58Check format, we first add a prefix to the data, called the "version byte," which serves to easily identify the type of data that is encoded. For example, in the case of a bitcoin address the prefix is zero (0x00 in hex), whereas the prefix used when encoding a private key is 128 (0x80 in hex). A list of common version prefixes is shown in [Base58Check version prefix and encoded result examples](#).

Next, we compute the "double-SHA" checksum, meaning we apply the SHA256 hash-algorithm twice on the previous result (prefix and data):

```
checksum = SHA256(SHA256(prefix+data))
```

From the resulting 32-byte hash (hash-of-a-hash), we take only the first four bytes. These four bytes serve as the error-checking code, or checksum. The checksum is concatenated (appended) to the end.

The result is composed of three items: a prefix, the data, and a checksum. This result is encoded using the Base58 alphabet described previously. [Base58Check encoding: a Base58, versioned, and checksummed format for unambiguously encoding bitcoin data](#) illustrates the Base58Check encoding process.

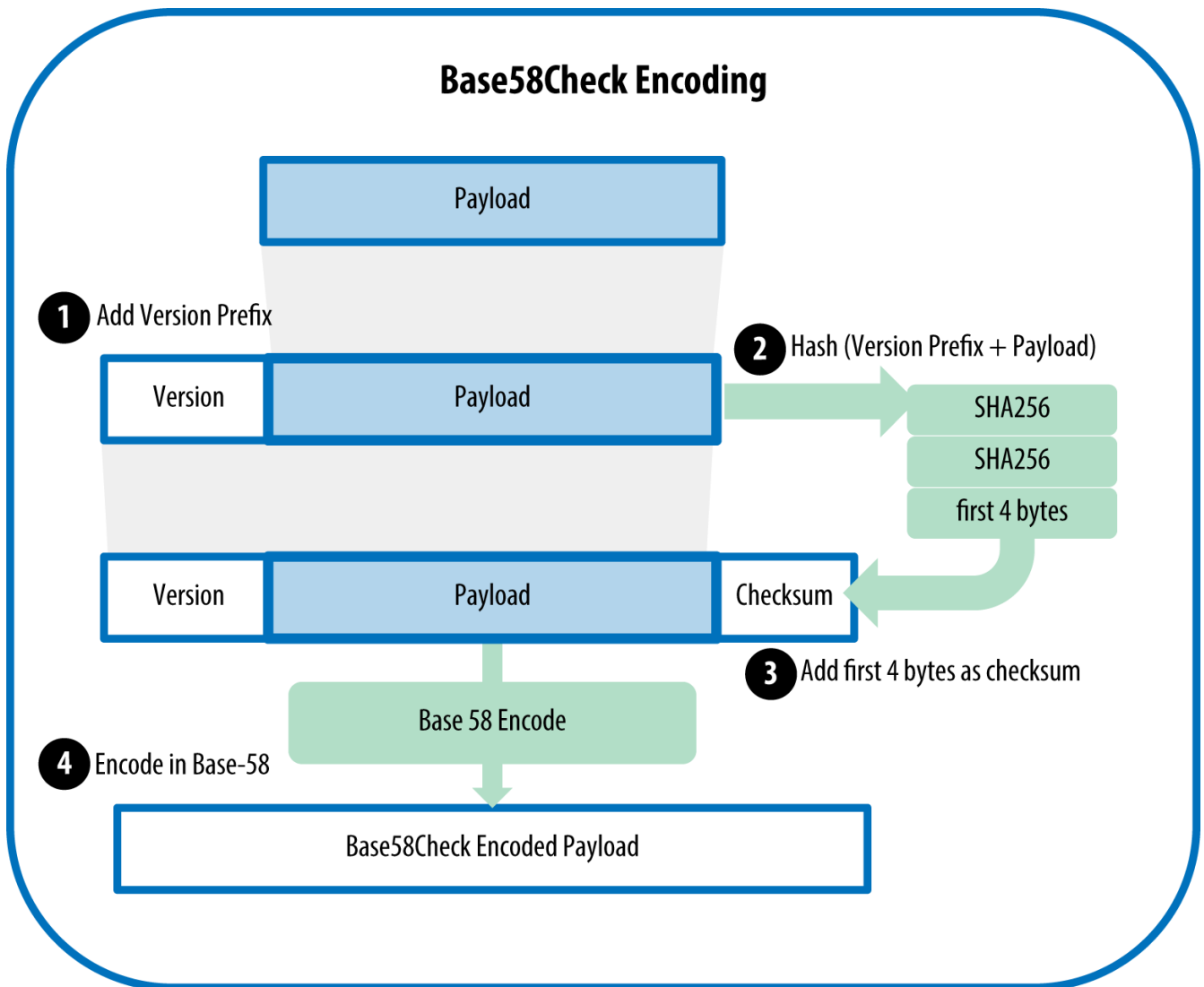


Figure 22. Base58Check encoding: a Base58, versioned, and checksummed format for unambiguously encoding bitcoin data

In bitcoin, most of the data presented to the user is Base58Check-encoded to make it compact, easy to read, and easy to detect errors. The version prefix in Base58Check encoding is used to create easily distinguishable formats, which when encoded in Base58 contain specific characters at the beginning of the Base58Check-encoded payload. These characters make it easy for humans to identify the type of data that is encoded and how to use it. This is what differentiates, for example, a Base58Check-encoded bitcoin address that starts with a 1 from a Base58Check-encoded private key WIF format that starts with a 5. Some example version prefixes and the resulting Base58 characters are shown in [Base58Check version prefix and encoded result examples](#).

Table 1. Base58Check version prefix and encoded result examples

Type	Version prefix (hex)	Base58 result prefix
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K or L
BIP38 Encrypted Private Key	0x0142	6P

Type	Version prefix (hex)	Base58 result prefix
BIP32 Extended Public Key	0x0488B21E	xpub

Let's look at the complete process of creating a bitcoin address, from a private key, to a public key (a point on the elliptic curve), to a double-hashed address and finally, the Base58Check encoding. The C++ code in [Creating a Base58Check-encoded bitcoin address from a private key](#) shows the complete step-by-step process, from private key to Base58Check-encoded bitcoin address. The code example uses the libbitcoin library introduced in [Alternative Clients, Libraries, and Toolkits](#) for some helper functions.

Example 4. Creating a Base58Check-encoded bitcoin address from a private key

```
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Private secret key.
    bc::ec_secret secret;
    bool success = bc::decode_base16(secret,
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
    assert(success);
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //   bc::payment_address payaddr;
    //   bc::set_public_key(payaddr, public_key);
    //   const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);

    bc::data_chunk unencoded_address;
    // Reserve 25 bytes
    //   [ version:1 ]
    //   [ hash:20   ]
    //   [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).
    unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash.
    bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding
    assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);

    std::cout << "Address: " << address << std::endl;
    return 0;
}
```

The code uses a predefined private key so that it produces the same bitcoin address every time it is run, as shown in [Compiling and running the addr code](#).

Example 5. Compiling and running the addr code

```
# Compile the addr.cpp code
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# Run the addr executable
$ ./addr
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK
```

Key Formats

Both private and public keys can be represented in a number of different formats. These representations all encode the same number, even though they look different. These formats are primarily used to make it easy for people to read and transcribe keys without introducing errors.

Private key formats

The private key can be represented in a number of different formats, all of which correspond to the same 256-bit number. [Private key representations \(encoding formats\)](#) shows three common formats used to represent private keys.

Table 2. Private key representations (encoding formats)

Type	Prefix	Description
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding: Base58 with version prefix of 128 and 32-bit checksum
WIF-compressed	K or L	As above, with added suffix 0x01 before encoding

[Example: Same key, different formats](#) shows the private key generated in these three formats.

Table 3. Example: Same key, different formats

Format	Private Key
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpb nkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGm f6YwgdGWZgawvrtJ

All of these representations are different ways of showing the same number, the same private key. They look different, but any one format can easily be converted to any other format.

We use the **wif-to-ec** command from Bitcoin Explorer (see [Libbitcoin and Bitcoin Explorer](#)) to show that both WIF keys represent the same private key:

```
$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

$ bx wif-to-ec KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
```

Decode from Base58Check

The Bitcoin Explorer commands (see [Libbitcoin and Bitcoin Explorer](#)) make it easy to write shell scripts and command-line "pipes" that manipulate bitcoin keys, addresses, and transactions. You can use Bitcoin Explorer to decode the Base58Check format on the command line.

We use the **base58check-decode** command to decode the uncompressed key:

```
$ bx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbkeyhfsYB1Jcn
wrapper
{
  checksum 4286807748
  payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
  version 128
}
```

The result contains the key as payload, the Wallet Import Format (WIF) version prefix 128, and a checksum.

Notice that the "payload" of the compressed key is appended with the suffix **01**, signalling that the derived public key is to be compressed.

```
$ bx base58check-decode KxFC1jmwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
wrapper
{
  checksum 2339607926
  payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01
  version 128
}
```

Encode from hex to Base58Check

To encode into Base58Check (the opposite of the previous command), we use the **base58check-encode** command from Bitcoin Explorer (see [Libbitcoin and Bitcoin Explorer](#)) and provide the hex private key, followed by the Wallet Import Format (WIF) version prefix 128:

```
bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
--version 128
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Encode from hex (compressed key) to Base58Check

To encode into Base58Check as a "compressed" private key (see [Compressed private keys](#)), we append the suffix `01` to the hex key and then encode as above:

```
$ bx base58check-encode
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 --version 128
KxFC1jmwvCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

The resulting WIF-compressed format starts with a "K". This denotes that the private key within has a suffix of "01" and will be used to produce compressed public keys only (see [Compressed public keys](#)).

Public key formats

Public keys are also presented in different ways, most importantly as either *compressed* or *uncompressed* public keys.

As we saw previously, the public key is a point on the elliptic curve consisting of a pair of coordinates (x, y) . It is usually presented with the prefix `04` followed by two 256-bit numbers, one for the x coordinate of the point, the other for the y coordinate. The prefix `04` is used to distinguish uncompressed public keys from compressed public keys that begin with a `02` or a `03`.

Here's the public key generated by the private key we created earlier, shown as the coordinates x and y :

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Here's the same public key shown as a 520-bit number (130 hex digits) with the prefix `04` followed by x and then y coordinates, as `04 x y`:

Compressed public keys

Compressed public keys were introduced to bitcoin to reduce the size of transactions and conserve disk space on nodes that store the bitcoin blockchain database. Most transactions include the public key, required to validate the owner's credentials and spend the bitcoin. Each public key requires 520 bits (prefix `\+` x `\+` y), which when multiplied by several hundred transactions per block, or tens of thousands of transactions per day, adds a significant amount of data to the blockchain.

As we saw in the section [Public Keys](#), a public key is a point (x, y) on an elliptic curve. Because the curve expresses a mathematical function, a point on the curve represents a solution to the equation and, therefore, if we know the x coordinate we can calculate the y coordinate by solving the

equation $y^2 \bmod p = (x^3 + 7) \bmod p$. That allows us to store only the x coordinate of the public key point, omitting the y coordinate and reducing the size of the key and the space required to store it by 256 bits. An almost 50% reduction in size in every transaction adds up to a lot of data saved over time!

Whereas uncompressed public keys have a prefix of `04`, compressed public keys start with either a `02` or a `03` prefix. Let's look at why there are two possible prefixes: because the left side of the equation is y^2 , that means the solution for y is a square root, which can have a positive or negative value. Visually, this means that the resulting y coordinate can be above the x -axis or below the x -axis. As you can see from the graph of the elliptic curve in [An elliptic curve](#), the curve is symmetric, meaning it is reflected like a mirror by the x -axis. So, while we can omit the y coordinate we have to store the *sign* of y (positive or negative), or in other words, we have to remember if it was above or below the x -axis because each of those options represents a different point and a different public key. When calculating the elliptic curve in binary arithmetic on the finite field of prime order p , the y coordinate is either even or odd, which corresponds to the positive/negative sign as explained earlier. Therefore, to distinguish between the two possible values of y , we store a compressed public key with the prefix `02` if the y is even, and `03` if it is odd, allowing the software to correctly deduce the y coordinate from the x coordinate and uncompress the public key to the full coordinates of the point. Public key compression is illustrated in [Public key compression](#).

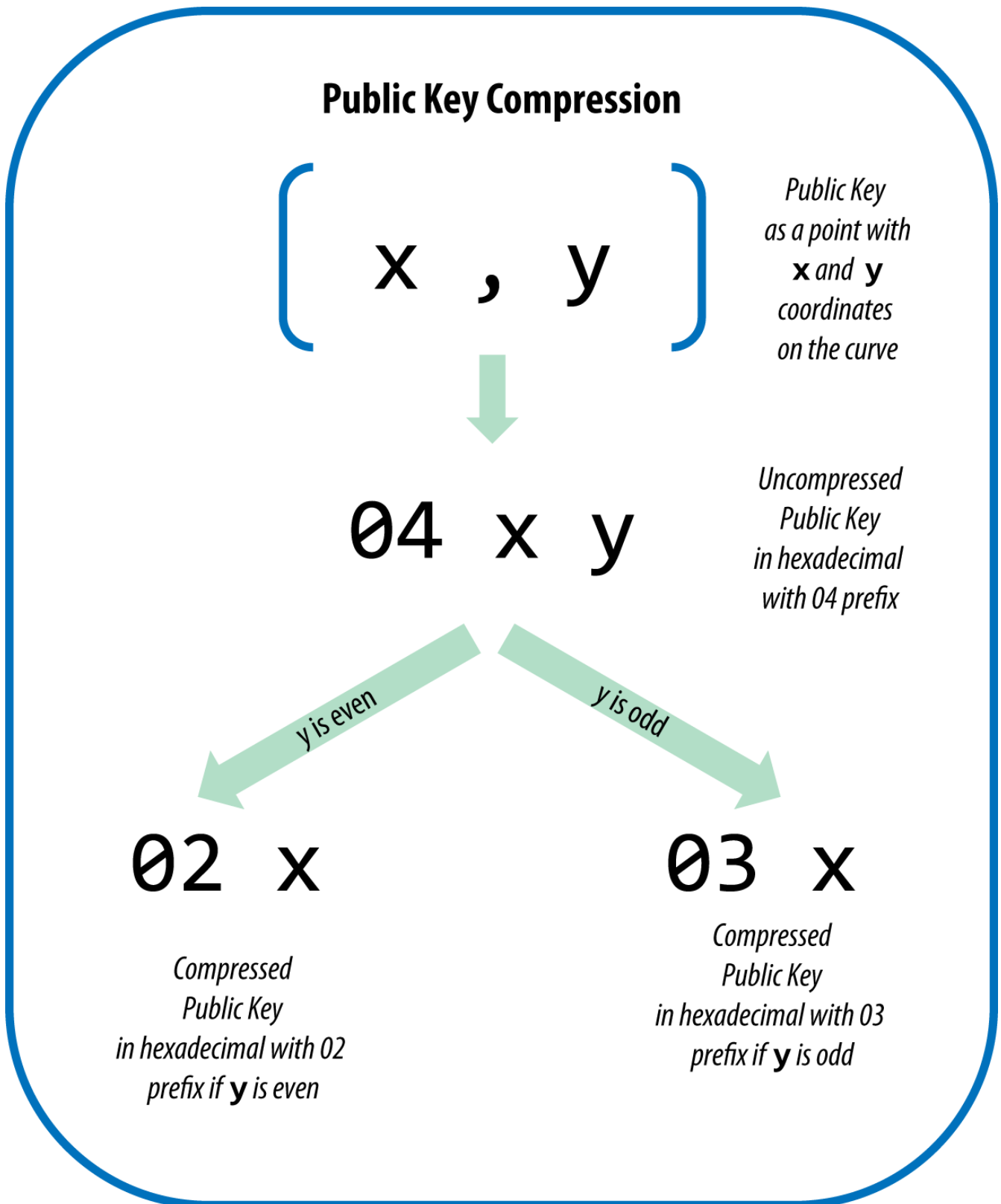


Figure 23. Public key compression

Here's the same public key generated previously, shown as a compressed public key stored in 264 bits (66 hex digits) with the prefix **03** indicating the y coordinate is odd:

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

This compressed public key corresponds to the same private key, meaning that it is generated from

the same private key. However, it looks different from the uncompressed public key. More importantly, if we convert this compressed public key to a bitcoin address using the double-hash function (`RIPEMD160(SHA256(K))`) it will produce a *different* bitcoin address. This can be confusing, because it means that a single private key can produce a public key expressed in two different formats (compressed and uncompressed) that produce two different bitcoin addresses. However, the private key is identical for both bitcoin addresses.

Compressed public keys are gradually becoming the default across bitcoin clients, which is having a significant impact on reducing the size of transactions and therefore the blockchain. However, not all clients support compressed public keys yet. Newer clients that support compressed public keys have to account for transactions from older clients that do not support compressed public keys. This is especially important when a wallet application is importing private keys from another bitcoin wallet application, because the new wallet needs to scan the blockchain to find transactions corresponding to these imported keys. Which bitcoin addresses should the bitcoin wallet scan for? The bitcoin addresses produced by uncompressed public keys, or the bitcoin addresses produced by compressed public keys? Both are valid bitcoin addresses, and can be signed for by the private key, but they are different addresses!

To resolve this issue, when private keys are exported from a wallet, the Wallet Import Format that is used to represent them is implemented differently in newer bitcoin wallets, to indicate that these private keys have been used to produce *compressed* public keys and therefore *compressed* bitcoin addresses. This allows the importing wallet to distinguish between private keys originating from older or newer wallets and search the blockchain for transactions with bitcoin addresses corresponding to the uncompressed, or the compressed, public keys, respectively. Let's look at how this works in more detail, in the next section.

Compressed private keys

Ironically, the term "compressed private key" is misleading, because when a private key is exported as WIF-compressed it is actually one byte *longer* than an "uncompressed" private key. That is because it has the added 01 suffix, which signifies it comes from a newer wallet and should only be used to produce compressed public keys. Private keys are not compressed and cannot be compressed. The term "compressed private key" really means "private key from which compressed public keys should be derived," whereas "uncompressed private key" really means "private key from which uncompressed public keys should be derived." You should only refer to the export format as "WIF-compressed" or "WIF" and not refer to the private key as "compressed" to avoid further confusion.

Remember, these formats are *not* used interchangeably. In a newer wallet that implements compressed public keys, the private keys will only ever be exported as WIF-compressed (with a K or L prefix). If the wallet is an older implementation and does not use compressed public keys, the private keys will only ever be exported as WIF (with a 5 prefix). The goal here is to signal to the wallet importing these private keys whether it must search the blockchain for compressed or uncompressed public keys and addresses.

If a bitcoin wallet is able to implement compressed public keys, it will use those in all transactions. The private keys in the wallet will be used to derive the public key points on the curve, which will be compressed. The compressed public keys will be used to produce bitcoin addresses and those will be used in transactions. When exporting private keys from a new wallet that implements

compressed public keys, the Wallet Import Format is modified, with the addition of a one-byte suffix **01** to the private key. The resulting Base58Check-encoded private key is called a "Compressed WIF" and starts with the letter K or L, instead of starting with "5" as is the case with WIF-encoded (non-compressed) keys from older wallets.

Example: [Same key, different formats](#) shows the same key, encoded in WIF and WIF-compressed formats.

Table 4. Example: Same key, different formats

Format	Private Key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpb nkeyhfsYB1Jcn
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD_01_
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGm f6YwgdGWZgawvrtJ

TIP

"Compressed private keys" is a misnomer! They are not compressed; rather, the WIF-compressed format signifies that they should only be used to derive compressed public keys and their corresponding bitcoin addresses. Ironically, a "WIF-compressed" encoded private key is one byte longer because it has the added 01 suffix to distinguish it from an "uncompressed" one.

Implementing Keys and Addresses in Python

The most comprehensive bitcoin library in Python is [pybitcointools](#) by Vitalik Buterin. In [Key and address generation and formatting with the pybitcointools library](#), we use the pybitcointools library (imported as "bitcoin") to generate and display keys and addresses in various formats.

Example 6. Key and address generation and formatting with the pybitcointools library

```

import bitcoin

# Generate a random private key
valid_private_key = False
while not valid_private_key:
    private_key = bitcoin.random_key()
    decoded_private_key = bitcoin.decode_privkey(private_key, 'hex')
    valid_private_key = 0 < decoded_private_key < bitcoin.N

print "Private Key (hex) is: ", private_key
print "Private Key (decimal) is: ", decoded_private_key

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
print "Private Key (WIF) is: ", wif_encoded_private_key

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print "Private Key Compressed (hex) is: ", compressed_private_key

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print "Private Key (WIF-Compressed) is: ", wif_compressed_private_key

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
print "Public Key (x,y) coordinates is:", public_key

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key, 'hex')
print "Public Key (hex) is:", hex_encoded_public_key

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key
if (public_key_y % 2) == 0:
    compressed_prefix = '02'
else:
    compressed_prefix = '03'
hex_compressed_public_key = compressed_prefix + bitcoin.encode(public_key_x, 16)
print "Compressed Public Key (hex) is:", hex_compressed_public_key

# Generate bitcoin address from public key
print "Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key)

# Generate compressed bitcoin address from compressed public key
print "Compressed Bitcoin Address (b58check) is:", \
    bitcoin.pubkey_to_address(hex_compressed_public_key)

```

Running [key-to-address-ecc-example.py](#) shows the output from running this code.

Example 7. Running key-to-address-ecc-example.py

A [script demonstrating elliptic curve math used for bitcoin keys](#) is another example, using the Python ECDSA library for the elliptic curve math and without using any specialized bitcoin libraries.

Example 8. A script demonstrating elliptic curve math used for bitcoin keys

```
import ecdsa
import os
from ecdsa.util import string_to_number, number_to_string

# secp256k1, http://www.oid-info.com/get/1.3.132.0.10
_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEC2FL
_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L
_b = 0x0000000000000000000000000000000000000000000000000000000000000007L
_a = 0x000000000000000000000000000000000000000000000000000000000000000L
_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L
_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L
curve_secp256k1 = ecdsa.ellipticcurve.CurveFp(_p, _a, _b)
generator_secp256k1 = ecdsa.ellipticcurve.Point(curve_secp256k1, _Gx, _Gy, _r)
oid_secp256k1 = (1, 3, 132, 0, 10)
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve_secp256k1, generator_secp256k1,
oid_secp256k1)
ec_order = _r

curve = curve_secp256k1
generator = generator_secp256k1

def random_secret():
    convert_to_int = lambda array: int("".join(array).encode("hex"), 16)

    # Collect 256 bits of random data from the OS's cryptographically secure
    random generator
    byte_array = os.urandom(32)

    return convert_to_int(byte_array)

def get_point_pubkey(point):
    if point.y() & 1:
        key = '03' + '%064x' % point.x()
    else:
        key = '02' + '%064x' % point.x()
    return key.decode('hex')

def get_point_pubkey_uncompressed(point):
    key = '04' + \
```

```

        '%064x' % point.x() + \
        '%064x' % point.y()
    return key.decode('hex')

# Generate a new private key.
secret = random_secret()
print "Secret: ", secret

# Get the public key point.
point = secret * generator
print "EC point:", point

print "BTC public key:", get_point_pubkey(point).encode("hex")

# Given the point (x, y) we can create the object using:
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec_order)
assert point1 == point

```

Installing the Python ECDSA library and running the `ec_math.py` script shows the output produced by running this script.

NOTE

The example above uses `os.urandom`, which reflects a cryptographically secure random number generator (CSRNG) provided by the underlying operating system. In the case of an UNIX-like operating system such as Linux, it draws from `/dev/urandom`; and in the case of Windows, calls `CryptGenRandom()`. If a suitable randomness source is not found, `NotImplementedError` will be raised. While the random number generator used here is for demonstration purposes, it is *not* appropriate for generating production-quality bitcoin keys as it is not implemented with sufficient security.

Example 9. Installing the Python ECDSA library and running the `ec_math.py` script

```

$ # Install Python PIP package manager
$ sudo apt-get install python-pip
$ # Install the Python ECDSA library
$ sudo pip install ecdsa
$ # Run the script
$ python ec-math.py
Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873

```

Wallets

Wallets are containers for private keys, usually implemented as structured files or simple databases. Another method for making keys is *deterministic key generation*. Here you derive each new private key, using a one-way hash function from a previous private key, linking them in a sequence. As long as you can re-create that sequence, you only need the first key (known as a *seed* or *master* key) to generate them all. In this section we will examine the different methods of key generation and the wallet structures that are built around them.

TIP

Bitcoin wallets contain keys, not coins. Each user has a wallet containing keys. Wallets are really keychains containing pairs of private/public keys (see [Private and Public Keys](#)). Users sign transactions with the keys, thereby proving they own the transaction outputs (their coins). The coins are stored on the blockchain in the form of transaction-outputs (often noted as vout or txout).

Nondeterministic (Random) Wallets

In the first bitcoin clients, wallets were simply collections of randomly generated private keys. This type of wallet is called a *Type-0 nondeterministic wallet*. For example, the Bitcoin Core client pregenerates 100 random private keys when first started and generates more keys as needed, using each key only once. This type of wallet is nicknamed "Just a Bunch Of Keys," or JBOK, and such wallets are being replaced with deterministic wallets because they are cumbersome to manage, back up, and import. The disadvantage of random keys is that if you generate many of them you must keep copies of all of them, meaning that the wallet must be backed up frequently. Each key must be backed up, or the funds it controls are irrevocably lost if the wallet becomes inaccessible. This conflicts directly with the principle of avoiding address re-use, by using each bitcoin address for only one transaction. Address re-use reduces privacy by associating multiple transactions and addresses with each other. A Type-0 nondeterministic wallet is a poor choice of wallet, especially if you want to avoid address re-use because that means managing many keys, which creates the need for frequent backups. Although the Bitcoin Core client includes a Type-0 wallet, using this wallet is discouraged by developers of Bitcoin Core. [Type-0 nondeterministic \(random\) wallet: a collection of randomly generated keys](#) shows a nondeterministic wallet, containing a loose collection of random keys.

Deterministic (Seeded) Wallets

Deterministic, or "seeded" wallets are wallets that contain private keys that are all derived from a common seed, through the use of a one-way hash function. The seed is a randomly generated number that is combined with other data, such as an index number or "chain code" (see [Hierarchical Deterministic Wallets \(BIP0032/BIP0044\)](#)) to derive the private keys. In a deterministic wallet, the seed is sufficient to recover all the derived keys, and therefore a single backup at creation time is sufficient. The seed is also sufficient for a wallet export or import, allowing for easy migration of all the user's keys between different wallet implementations.

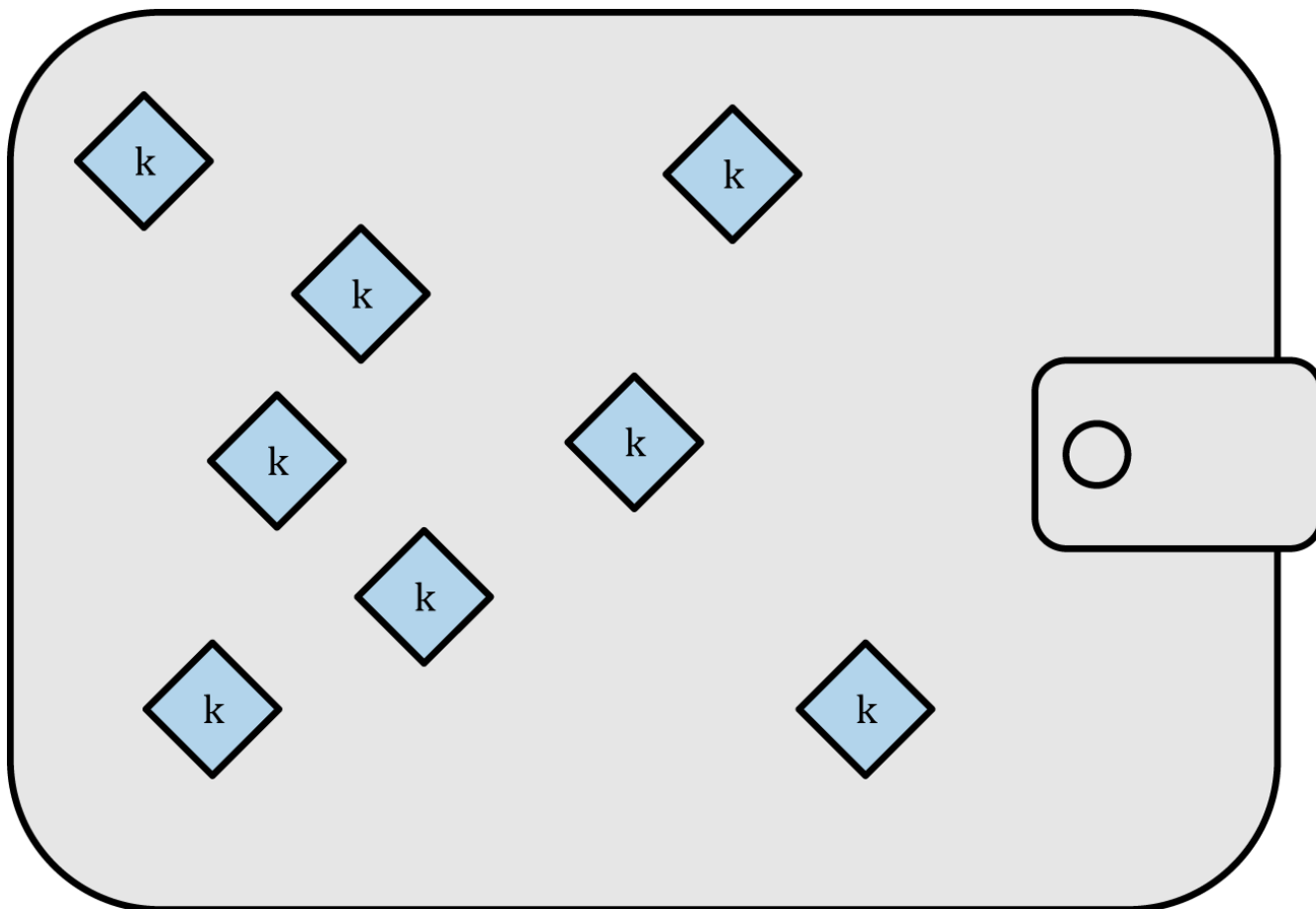


Figure 24. Type-0 nondeterministic (random) wallet: a collection of randomly generated keys

Mnemonic Code Words

Mnemonic codes are English word sequences that represent (encode) a random number used as a seed to derive a deterministic wallet. The sequence of words is sufficient to re-create the seed and from there re-create the wallet and all the derived keys. A wallet application that implements deterministic wallets with mnemonic code will show the user a sequence of 12 to 24 words when first creating a wallet. That sequence of words is the wallet backup and can be used to recover and re-create all the keys in the same or any compatible wallet application. Mnemonic code words make it easier for users to back up wallets because they are easy to read and correctly transcribe, as compared to a random sequence of numbers.

Mnemonic codes are defined in Bitcoin Improvement Proposal 39 (see [\[bip0039\]](#)), currently in Draft status. Note that BIP0039 is a draft proposal and not a standard. Specifically, there is a different standard, with a different set of words, used by the Electrum wallet and predating BIP0039. BIP0039 is used by the Trezor wallet and a few other wallets but is incompatible with Electrum's implementation.

BIP0039 defines the creation of a mnemonic code and seed as follows:

1. Create a random sequence (entropy) of 128 to 256 bits.
2. Create a checksum of the random sequence by taking the first few bits of its SHA256 hash.
3. Add the checksum to the end of the random sequence.
4. Divide the sequence into sections of 11 bits, using those to index a dictionary of 2048 predefined

words.

5. Produce 12 to 24 words representing the mnemonic code.

Mnemonic codes: entropy and word length shows the relationship between the size of entropy data and the length of mnemonic codes in words.

Table 5. Mnemonic codes: entropy and word length

Entropy (bits)	Checksum (bits)	Entropy+checksum	Word length
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

The mnemonic code represents 128 to 256 bits, which are used to derive a longer (512-bit) seed through the use of the key-stretching function PBKDF2. The resulting seed is used to create a deterministic wallet and all of its derived keys.

Tables #table_4-6 and #table_4-7 show some examples of mnemonic codes and the seeds they produce.

Table 6. 128-bit entropy mnemonic code and resulting seed

Entropy input (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemonic (12 words)	army van defense carry jealous true garbage claim echo media make crunch
Seed (512 bits)	3338a6d2ee71c7f28eb5b882159634cd46a898463 e9d2d0980f8e80dfbba5b0fa0291e5fb88 8a599b44b93187be6ee3ab5fd3ead7dd646341b2c db8d08d13bf7

Table 7. 256-bit entropy mnemonic code and resulting seed

Entropy input (256 bits)	2041546864449caff939d32d574753fe684d3c947c 3346713dd8423e74abcf8c
Mnemonic (24 words)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Seed (512 bits)	3972e432e99040f75ebe13a660110c3e29d131a2c8 08c7ee5f1631d0a977fcf473bee22 fce540af281bf7cdeade0dd2c1c795bd02f1e4049e2 05a0158906c343

Hierarchical Deterministic Wallets (BIP0032/BIP0044)

Deterministic wallets were developed to make it easy to derive many keys from a single "seed." The most advanced form of deterministic wallets is the *hierarchical deterministic wallet* or *HD wallet* defined by the BIP0032 standard. Hierarchical deterministic wallets contain keys derived in a tree structure, such that a parent key can derive a sequence of children keys, each of which can derive a sequence of grandchildren keys, and so on, to an infinite depth. This tree structure is illustrated in [Type-2 hierarchical deterministic wallet: a tree of keys generated from a single seed](#).

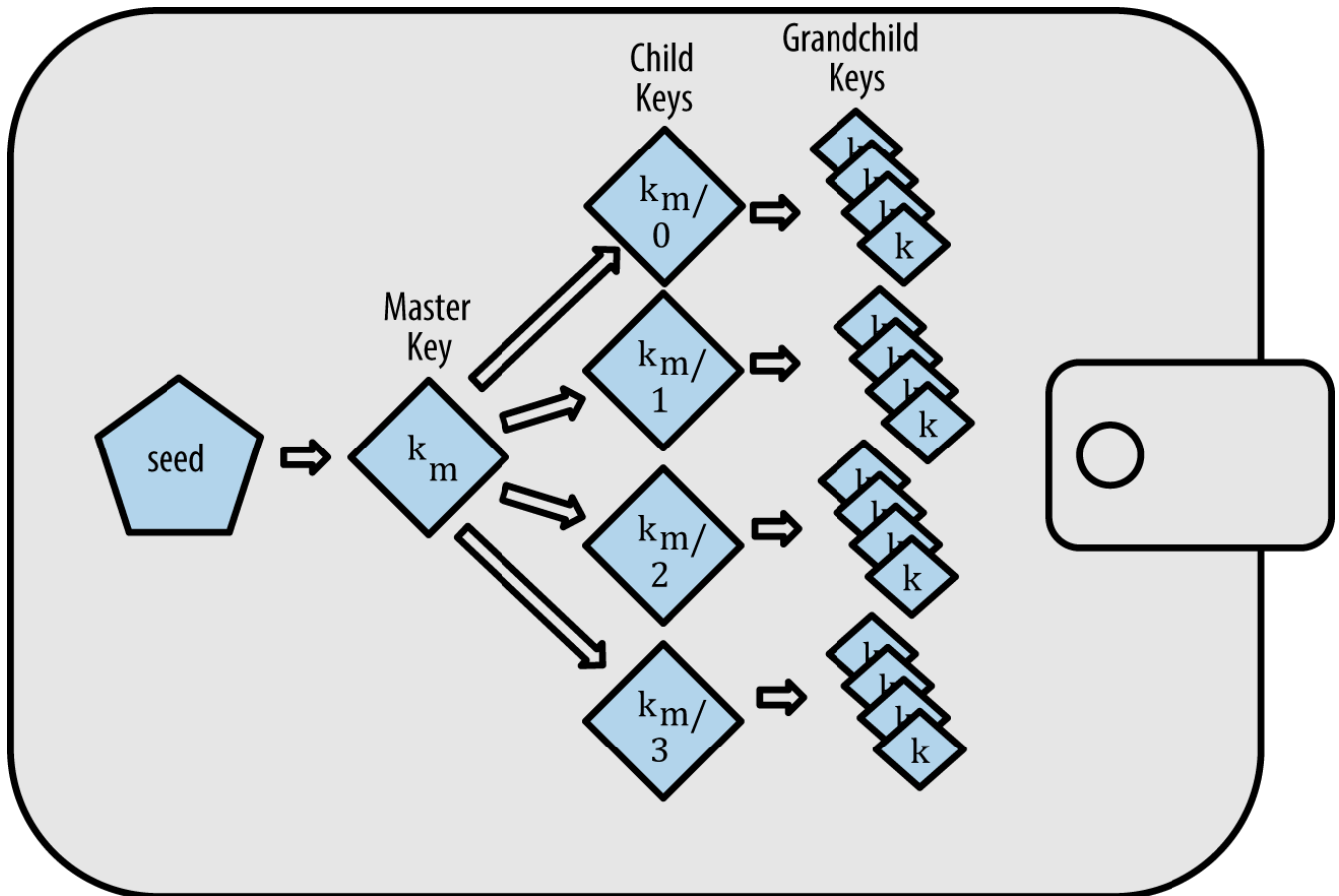


Figure 25. Type-2 hierarchical deterministic wallet: a tree of keys generated from a single seed

TIP

If you are implementing a bitcoin wallet, it should be built as an HD wallet following the BIP0032 and BIP0044 standards.

HD wallets offer two major advantages over random (nondeterministic) keys. First, the tree structure can be used to express additional organizational meaning, such as when a specific branch of subkeys is used to receive incoming payments and a different branch is used to receive change from outgoing payments. Branches of keys can also be used in a corporate setting, allocating different branches to departments, subsidiaries, specific functions, or accounting categories.

The second advantage of HD wallets is that users can create a sequence of public keys without having access to the corresponding private keys. This allows HD wallets to be used on an insecure server or in a receive-only capacity, issuing a different public key for each transaction. The public keys do not need to be preloaded or derived in advance, yet the server doesn't have the private keys that can spend the funds.

HD wallet creation from a seed

HD wallets are created from a single *root seed*, which is a 128-, 256-, or 512-bit random number. Everything else in the HD wallet is deterministically derived from this root seed, which makes it possible to re-create the entire HD wallet from that seed in any compatible HD wallet. This makes it easy to back up, restore, export, and import HD wallets containing thousands or even millions of keys by simply transferring only the root seed. The root seed is most often represented by a *mnemonic word sequence*, as described in the previous section [Mnemonic Code Words](#), to make it easier for people to transcribe and store it.

The process of creating the master keys and master chain code for an HD wallet is shown in [Creating master keys and chain code from a root seed](#).

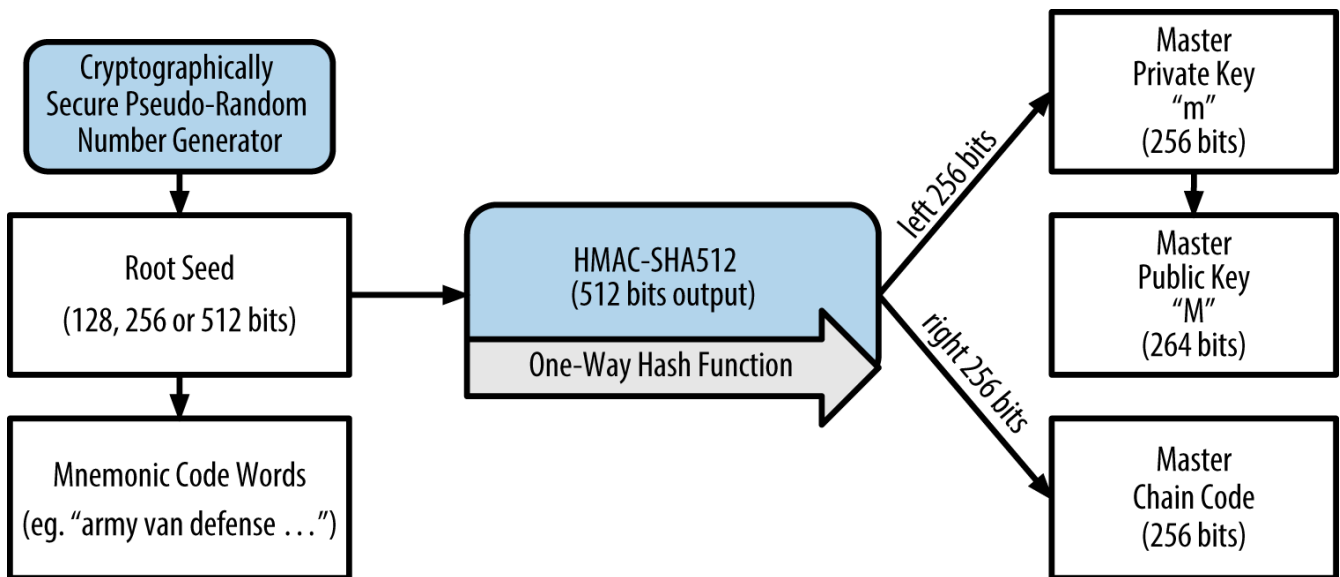


Figure 26. Creating master keys and chain code from a root seed

The root seed is input into the HMAC-SHA512 algorithm and the resulting hash is used to create a *master private key* (m) and a *master chain code*. The master private key (m) then generates a corresponding master public key (M), using the normal elliptic curve multiplication process $m * G$ that we saw earlier in this chapter. The chain code is used to introduce entropy in the function that creates child keys from parent keys, as we will see in the next section.

Private child key derivation

Hierarchical deterministic wallets use a *child key derivation* (CKD) function to derive children keys from parent keys.

The child key derivation functions are based on a one-way hash function that combines:

- A parent private or public key (ECDSA uncompressed key)
- A seed called a chain code (256 bits)
- An index number (32 bits)

The chain code is used to introduce seemingly random data to the process, so that the index is not sufficient to derive other child keys. Thus, having a child key does not make it possible to find its siblings, unless you also have the chain code. The initial chain code seed (at the root of the tree) is

made from random data, while subsequent chain codes are derived from each parent chain code.

These three items are combined and hashed to generate children keys, as follows.

The parent public key, chain code, and the index number are combined and hashed with the HMAC-SHA512 algorithm to produce a 512-bit hash. The resulting hash is split into two halves. The right-half 256 bits of the hash output become the chain code for the child. The left-half 256 bits of the hash and the index number are added to the parent private key to produce the child private key. In [Extending a parent private key to create a child private key](#), we see this illustrated with the index set to 0 to produce the 0th (first by index) child of the parent.

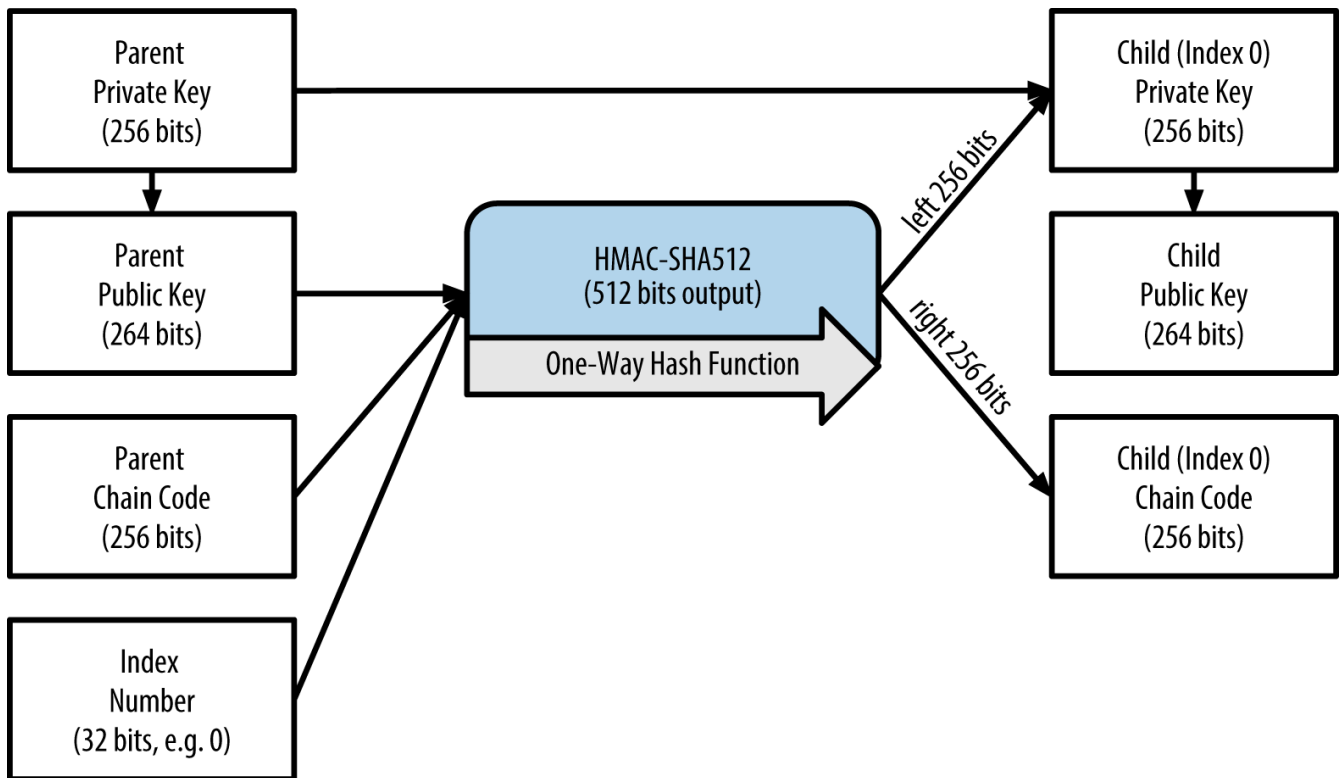


Figure 27. Extending a parent private key to create a child private key

Changing the index allows us to extend the parent and create the other children in the sequence, e.g., Child 0, Child 1, Child 2, etc. Each parent key can have 2 billion children keys.

Repeating the process one level down the tree, each child can in turn become a parent and create its own children, in an infinite number of generations.

Using derived child keys

Child private keys are indistinguishable from nondeterministic (random) keys. Because the derivation function is a one-way function, the child key cannot be used to find the parent key. The child key also cannot be used to find any siblings. If you have the n_{th} child, you cannot find its siblings, such as the $n-1$ child or the $n+1$ child, or any other children that are part of the sequence. Only the parent key and chain code can derive all the children. Without the child chain code, the child key cannot be used to derive any grandchildren either. You need both the child private key and the child chain code to start a new branch and derive grandchildren.

So what can the child private key be used for on its own? It can be used to make a public key and a bitcoin address. Then, it can be used to sign transactions to spend anything paid to that address.

TIP

A child private key, the corresponding public key, and the bitcoin address are all indistinguishable from keys and addresses created randomly. The fact that they are part of a sequence is not visible, outside of the HD wallet function that created them. Once created, they operate exactly as "normal" keys.

Extended keys

As we saw earlier, the key derivation function can be used to create children at any level of the tree, based on the three inputs: a key, a chain code, and the index of the desired child. The two essential ingredients are the key and chain code, and combined these are called an *extended key*. The term "extended key" could also be thought of as "extensible key" because such a key can be used to derive children.

Extended keys are stored and represented simply as the concatenation of the 256-bit key and 256-bit chain code into a 512-bit sequence. There are two types of extended keys. An extended private key is the combination of a private key and chain code and can be used to derive child private keys (and from them, child public keys). An extended public key is a public key and chain code, which can be used to create child public keys, as described in [Generating a Public Key](#).

Think of an extended key as the root of a branch in the tree structure of the HD wallet. With the root of the branch, you can derive the rest of the branch. The extended private key can create a complete branch, whereas the extended public key can only create a branch of public keys.

TIP

An extended key consists of a private or public key and chain code. An extended key can create children, generating its own branch in the tree structure. Sharing an extended key gives access to the entire branch.

Extended keys are encoded using Base58Check, to easily export and import between different BIP0032-compatible wallets. The Base58Check coding for extended keys uses a special version number that results in the prefix "xprv" and "xpub" when encoded in Base58 characters, to make them easily recognizable. Because the extended key is 512 or 513 bits, it is also much longer than other Base58Check-encoded strings we have seen previously.

Here's an example of an extended private key, encoded in Base58Check:

```
xprv9tyUQV64JT5qs3RSTJkXCWKMMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6Go  
NMKUga5biW6Hx4tws2six3b9c
```

Here's the corresponding extended public key, also encoded in Base58Check:

```
xpub67xpozcx8pe95XVuZLHXZeG6WXHpGq6Qv5cmNfi7cS5mtjJ2tgyeQbBs2UAR6KECeeMVKZBPLrtJunSD  
MstweyLXhRgPxdp14sk9tJPW9
```

Public child key derivation

As mentioned previously, a very useful characteristic of hierarchical deterministic wallets is the ability to derive public child keys from public parent keys, *without* having the private keys. This

gives us two ways to derive a child public key: either from the child private key, or directly from the parent public key.

An extended public key can be used, therefore, to derive all of the *public* keys (and only the public keys) in that branch of the HD wallet structure.

This shortcut can be used to create very secure public-key-only deployments where a server or application has a copy of an extended public key and no private keys whatsoever. That kind of deployment can produce an infinite number of public keys and bitcoin addresses, but cannot spend any of the money sent to those addresses. Meanwhile, on another, more secure server, the extended private key can derive all the corresponding private keys to sign transactions and spend the money.

One common application of this solution is to install an extended public key on a web server that serves an ecommerce application. The web server can use the public key derivation function to create a new bitcoin address for every transaction (e.g., for a customer shopping cart). The web server will not have any private keys that would be vulnerable to theft. Without HD wallets, the only way to do this is to generate thousands of bitcoin addresses on a separate secure server and then preload them on the ecommerce server. That approach is cumbersome and requires constant maintenance to ensure that the ecommerce server doesn't "run out" of keys.

Another common application of this solution is for cold-storage or hardware wallets. In that scenario, the extended private key can be stored on a paper wallet or hardware device (such as a Trezor hardware wallet), while the extended public key can be kept online. The user can create "receive" addresses at will, while the private keys are safely stored offline. To spend the funds, the user can use the extended private key on an offline signing bitcoin client or sign transactions on the hardware wallet device (e.g., Trezor). [Extending a parent public key to create a child public key](#) illustrates the mechanism for extending a parent public key to derive child public keys.

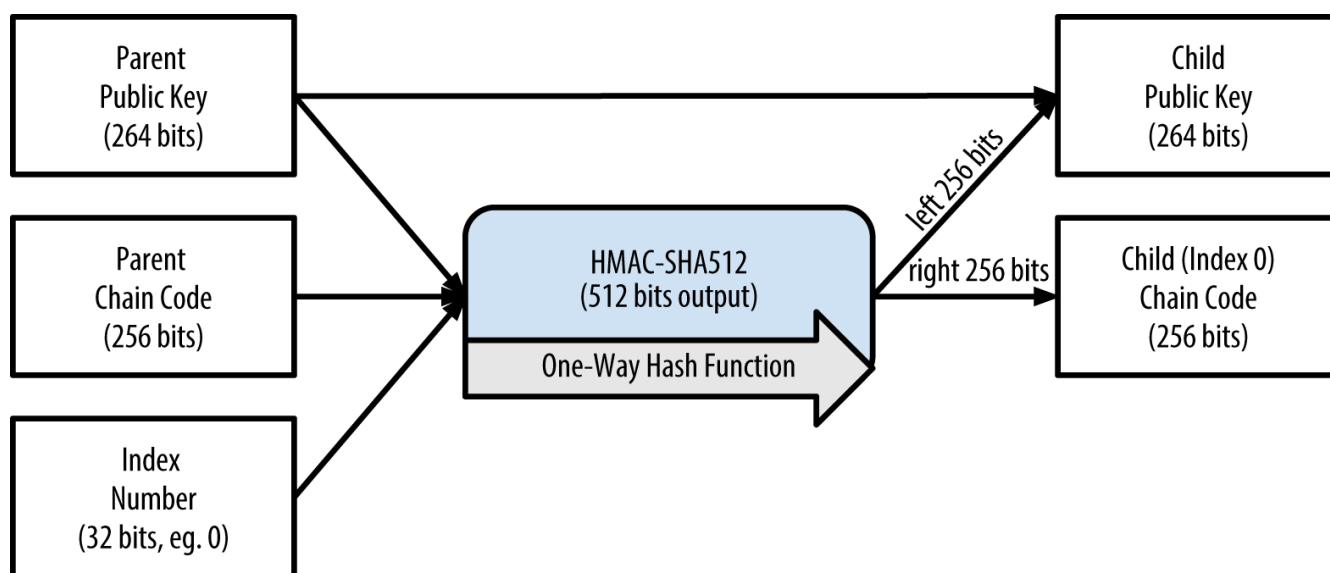


Figure 28. Extending a parent public key to create a child public key

Hardened child key derivation

The ability to derive a branch of public keys from an extended public key is very useful, but it comes with a potential risk. Access to an extended public key does not give access to child private keys. However, because the extended public key contains the chain code, if a child private key is

known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children. Worse, the child private key together with a parent chain code can be used to deduce the parent private key.

To counter this risk, HD wallets use an alternative derivation function called *hardened derivation*, which "breaks" the relationship between parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a "firewall" in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key. The hardened derivation function looks almost identical to the normal child private key derivation, except that the parent private key is used as input to the hash function, instead of the parent public key, as shown in the diagram in [Hardened derivation of a child key; omits the parent public key](#).

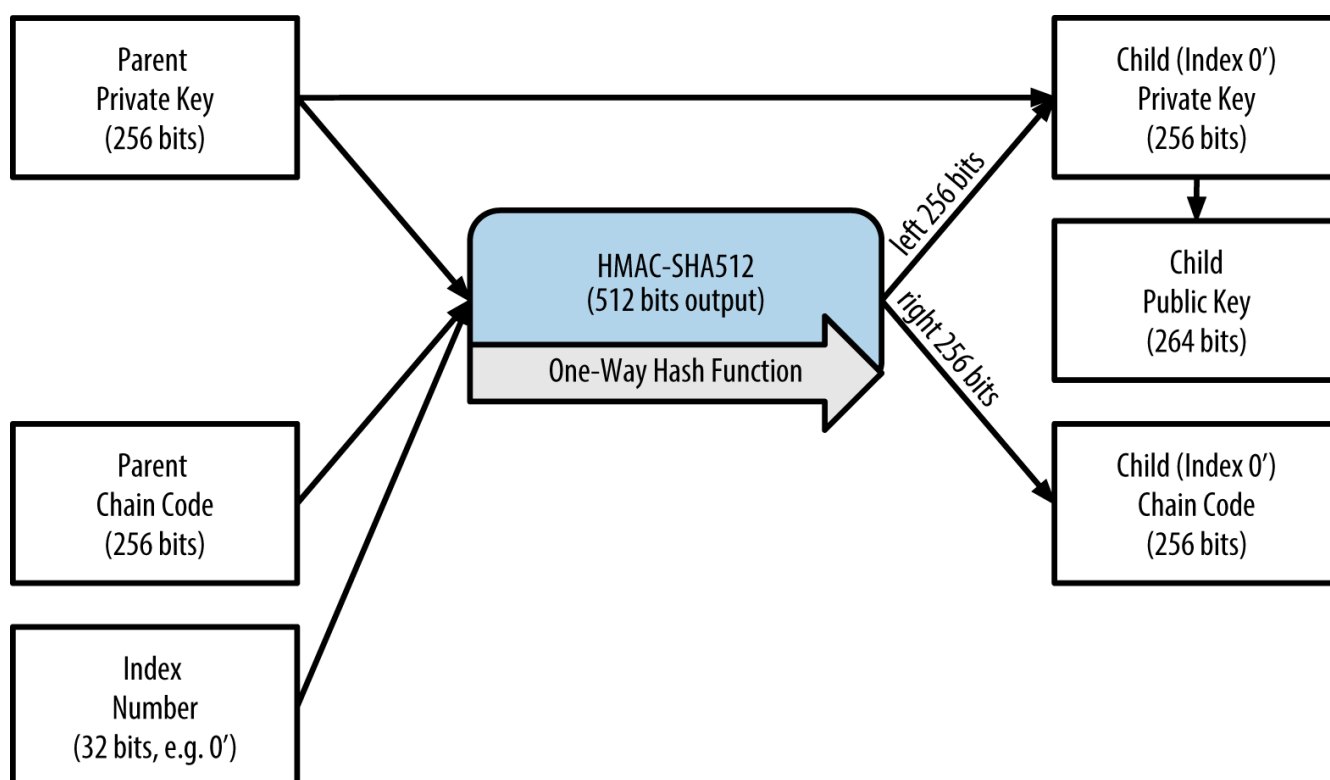


Figure 29. Hardened derivation of a child key; omits the parent public key

When the hardened private derivation function is used, the resulting child private key and chain code are completely different from what would result from the normal derivation function. The resulting "branch" of keys can be used to produce extended public keys that are not vulnerable, because the chain code they contain cannot be exploited to reveal any private keys. Hardened derivation is therefore used to create a "gap" in the tree above the level where extended public keys are used.

In simple terms, if you want to use the convenience of an extended public key to derive branches of public keys, without exposing yourself to the risk of a leaked chain code, you should derive it from a hardened parent, rather than a normal parent. As a best practice, the level-1 children of the master keys are always derived through the hardened derivation, to prevent compromise of the master keys.

Index numbers for normal and hardened derivation

The index number used in the derivation function is a 32-bit integer. To easily distinguish between keys derived through the normal derivation function versus keys derived through hardened derivation, this index number is split into two ranges. Index numbers between 0 and $2^{31}-1$ (0x0 to 0x7FFFFFFF) are used *only* for normal derivation. Index numbers between 2^{31} and $2^{32}-1$ (0x80000000 to 0xFFFFFFFF) are used *only* for hardened derivation. Therefore, if the index number is less than 2^{31} , that means the child is normal, whereas if the index number is equal or above 2^{31} , the child is hardened.

To make the index number easier to read and display, the index number for hardened children is displayed starting from zero, but with a prime symbol. The first normal child key is therefore displayed as 0, whereas the first hardened child (index 0x80000000) is displayed as 0'. In sequence then, the second hardened key would have index 0x80000001 and would be displayed as 1', and so on. When you see an HD wallet index i' , that means $2^{31}+i$.

HD wallet key identifier (path)

Keys in an HD wallet are identified using a "path" naming convention, with each level of the tree separated by a slash (/) character (see [HD wallet path examples](#)). Private keys derived from the master private key start with "m". Public keys derived from the master public key start with "M". Therefore, the first child private key of the master private key is m/0. The first child public key is M/0. The second grandchild of the first child is m/0/1, and so on.

The "ancestry" of a key is read from right to left, until you reach the master key from which it was derived. For example, identifier m/x/y/z describes the key that is the z-th child of key m/x/y, which is the y-th child of key m/x, which is the x-th child of m.

Table 8. HD wallet path examples

HD path	Key described
m/0	The first (0) child private key from the master private key (m)
m/0/0	The first grandchild private key of the first child (m/0)
m/0'/0	The first normal grandchild of the first <i>hardened</i> child (m/0')
m/1/0	The first grandchild private key of the second child (m/1)
M/23/17/0/0	The first great-great-grandchild public key of the first great-grandchild of the 18th grandchild of the 24th child

Navigating the HD wallet tree structure

The HD wallet tree structure offers tremendous flexibility. Each parent extended key can have 4 billion children: 2 billion normal children and 2 billion hardened children. Each of those children can have another 4 billion children, and so on. The tree can be as deep as you want, with an infinite

number of generations. With all that flexibility, however, it becomes quite difficult to navigate this infinite tree. It is especially difficult to transfer HD wallets between implementations, because the possibilities for internal organization into branches and subbranches are endless.

Two Bitcoin Improvement Proposals (BIPs) offer a solution to this complexity, by creating some proposed standards for the structure of HD wallet trees. BIP0043 proposes the use of the first hardened child index as a special identifier that signifies the "purpose" of the tree structure. Based on BIP0043, an HD wallet should use only one level-1 branch of the tree, with the index number identifying the structure and namespace of the rest of the tree by defining its purpose. For example, an HD wallet using only branch `m/i/` is intended to signify a specific purpose and that purpose is identified by index number "i".

Extending that specification, BIP0044 proposes a multiaccount structure as "purpose" number `44'` under BIP0043. All HD wallets following the BIP0044 structure are identified by the fact that they only used one branch of the tree: `m/44/`.

BIP0044 specifies the structure as consisting of five predefined tree levels:

`m / purpose' / coin_type' / account' / change / address_index`

The first-level "purpose" is always set to `44'`. The second-level "coin_type" specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its own subtree under the second level. There are three currencies defined for now: Bitcoin is `m/44'/0'`, Bitcoin Testnet is `m/44'/1'`; and Litecoin is `m/44'/2'`.

The third level of the tree is "account," which allows users to subdivide their wallets into separate logical subaccounts, for accounting or organizational purposes. For example, an HD wallet might contain two bitcoin "accounts": `m/44'/0'/0'` and `m/44'/0'/1'`. Each account is the root of its own subtree.

On the fourth level, "change," an HD wallet has two subtrees, one for creating receiving addresses and one for creating change addresses. Note that whereas the previous levels used hardened derivation, this level uses normal derivation. This is to allow this level of the tree to export extended public keys for use in a nonsecured environment. Usable addresses are derived by the HD wallet as children of the fourth level, making the fifth level of the tree the "address_index." For example, the third receiving address for bitcoin payments in the primary account would be `M/44'/0'/0'/0/2`. [BIP0044 HD wallet structure examples](#) shows a few more examples.

Table 9. BIP0044 HD wallet structure examples

HD path	Key described
<code>M/44'/0'/0'/0/2</code>	The third receiving public key for the primary bitcoin account
<code>M/44'/0'/3'/1/14</code>	The fifteenth change-address public key for the fourth bitcoin account
<code>m/44'/2'/0'/0/1</code>	The second private key in the Litecoin main account, for signing transactions

Experimenting with HD wallets using Bitcoin Explorer

Using the Bitcoin Explorer command-line tool introduced in [The Bitcoin Client](#), you can experiment with generating and extending BIP0032 deterministic keys, as well as displaying them in different formats:

```
$ bx seed | bx hd-new > m # create a new master private key from a seed and store
in file "m"
$ cat m # show the master extended private key
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdFgvzghDt39DHPFpovvtWZaRgY5uPwV7RpEgHs7c
vdgfiSjLjjbuGKGcjRyU7RGGSS8Xa
$ cat m | bx hd-public # generate the M/0 extended public key
xpub67xpozcx8pe95XVuZLHXZeG6WXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJ
unSDMstweyLXhRgPxdp14sk9tJPW9
$ cat m | bx hd-private # generate the m/0 extended private key
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9
i6GoNMKUga5biW6Hx4tws2six3b9c
$ cat m | bx hd-private | bx hd-to-wif # show the private key of m/0 as a WIF
L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN
$ cat m | bx hd-public | bx hd-to-address # show the bitcoin address of M/0
1CHCnCjgMnb6digimckNQ6TBVcTWBAmPHK
$ cat m | bx hd-private | bx hd-private --index 12 --hard | bx hd-private --index
4 # generate m/0/12'/4
xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqqwoS6YZBLatR
ZRyUeAkRPe1kLR1P6Mn7jUrXFquUt
```

Advanced Keys and Addresses

In the following sections we will look at advanced forms of keys and addresses, such as encrypted private keys, script and multisignature addresses, vanity addresses, and paper wallets.

Encrypted Private Keys (BIP0038)

Private keys must remain secret. The need for *confidentiality* of the private keys is a truism that is quite difficult to achieve in practice, because it conflicts with the equally important security objective of *availability*. Keeping the private key private is much harder when you need to store backups of the private key to avoid losing it. A private key stored in a wallet that is encrypted by a password might be secure, but that wallet needs to be backed up. At times, users need to move keys from one wallet to another—to upgrade or replace the wallet software, for example. Private key backups might also be stored on paper (see [Paper Wallets](#)) or on external storage media, such as a USB flash drive. But what if the backup itself is stolen or lost? These conflicting security goals led to the introduction of a portable and convenient standard for encrypting private keys in a way that can be understood by many different wallets and bitcoin clients, standardized by Bitcoin Improvement Proposal 38 or BIP0038 (see [\[bip0038\]](#)).

BIP0038 proposes a common standard for encrypting private keys with a passphrase and encoding them with Base58Check so that they can be stored securely on backup media, transported securely

between wallets, or kept in any other conditions where the key might be exposed. The standard for encryption uses the Advanced Encryption Standard (AES), a standard established by the National Institute of Standards and Technology (NIST) and used broadly in data encryption implementations for commercial and military applications.

A BIP0038 encryption scheme takes as input a bitcoin private key, usually encoded in the Wallet Import Format (WIF), as a Base58Check string with a prefix of "5". Additionally, the BIP0038 encryption scheme takes a passphrase—a long password—usually composed of several words or a complex string of alphanumeric characters. The result of the BIP0038 encryption scheme is a Base58Check-encoded encrypted private key that begins with the prefix **6P**. If you see a key that starts with **6P**, that means it is encrypted and requires a passphrase in order to convert (decrypt) it back into a WIF-formatted private key (prefix **5**) that can be used in any wallet. Many wallet applications now recognize BIP0038-encrypted private keys and will prompt the user for a passphrase to decrypt and import the key. Third-party applications, such as the incredibly useful browser-based [Bit Address](#) (Wallet Details tab), can be used to decrypt BIP0038 keys.

The most common use case for BIP0038 encrypted keys is for paper wallets that can be used to back up private keys on a piece of paper. As long as the user selects a strong passphrase, a paper wallet with BIP0038 encrypted private keys is incredibly secure and a great way to create offline bitcoin storage (also known as "cold storage").

Test the encrypted keys in [Example of BIP0038 encrypted private key](#) using [bitaddress.org](#) to see how you can get the decrypted key by entering the passphrase.

Table 10. Example of BIP0038 encrypted private key

Private Key (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpb nkeyhfsYB1Jcn
Passphrase	MyTestPassphrase
Encrypted Key (BIP0038)	6PRTLH6mWa48xSopbU1cKrVjpKbBZxcLRRCdct Lj3z5yx87MobKoXdTsJ

Pay-to-Script Hash (P2SH) and Multi-Sig Addresses

As we know, traditional bitcoin addresses begin with the number "1" and are derived from the public key, which is derived from the private key. Although anyone can send bitcoin to a "1" address, that bitcoin can only be spent by presenting the corresponding private key signature and public key hash.

Bitcoin addresses that begin with the number "3" are pay-to-script hash (P2SH) addresses, sometimes erroneously called multi-signature or multi-sig addresses. They designate the beneficiary of a bitcoin transaction as the hash of a script, instead of the owner of a public key. The feature was introduced in January 2012 with Bitcoin Improvement Proposal 16, or BIP0016 (see [\[bip0016\]](#)), and is being widely adopted because it provides the opportunity to add functionality to the address itself. Unlike transactions that "send" funds to traditional "1" bitcoin addresses, also known as pay-to-public-key-hash (P2PKH), funds sent to "3" addresses require something more than the presentation of one public key hash and one private key signature as proof of ownership. The requirements are designated at the time the address is created, within the script, and all inputs

to this address will be encumbered with the same requirements.

A pay-to-script hash address is created from a transaction script, which defines who can spend a transaction output (for more detail, see [Pay-to-Script-Hash \(P2SH\)](#)). Encoding a pay-to-script hash address involves using the same double-hash function as used during creation of a bitcoin address, only applied on the script instead of the public key:

```
script hash = RIPEMD160(SHA256(script))
```

The resulting "script hash" is encoded with Base58Check with a version prefix of 5, which results in an encoded address starting with a **3**. An example of a P2SH address is **3F6i6kwkevjr7AsAd4te2YB2zZyASEm1HM**, which can be derived using the Bitcoin Explorer commands **script-encode**, **sha256**, **ripemd160**, and **base58check-encode** (see [Libbitcoin](#) and [Bitcoin Explorer](#)) as follows:

```
$ echo dup hash160 [ 89abcdefabbaabbaabbaabbaabbaabbaabbaabba ] equalverify checksig >
script
$ bx script-encode < script | bx sha256 | bx ripemd160 | bx base58check-encode
--version 5
3F6i6kwkevjr7AsAd4te2YB2zZyASEm1HM
```

TIP

P2SH is not necessarily the same as a multi-signature standard transaction. A P2SH address *most often* represents a multi-signature script, but it might also represent a script encoding other types of transactions.

Multi-signature addresses and P2SH

Currently, the most common implementation of the P2SH function is the multi-signature address script. As the name implies, the underlying script requires more than one signature to prove ownership and therefore spend funds. The bitcoin multi-signature feature is designed to require M signatures (also known as the “threshold”) from a total of N keys, known as an M-of-N multi-sig, where M is equal to or less than N. For example, Bob the coffee shop owner from [Introduction](#) could use a multi-signature address requiring 1-of-2 signatures from a key belonging to him and a key belonging to his spouse, ensuring either of them could sign to spend a transaction output locked to this address. This would be similar to a “joint account” as implemented in traditional banking where either spouse can spend with a single signature. Or Gopesh, the web designer paid by Bob to create a website, might have a 2-of-3 multi-signature address for his business that ensures that no funds can be spent unless at least two of the business partners sign a transaction.

We will explore how to create transactions that spend funds from P2SH (and multi-signature) addresses in [Transactions](#).

Vanity Addresses

Vanity addresses are valid bitcoin addresses that contain human-readable messages. For example, **1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33** is a valid address that contains the letters forming the word “Love” as the first four Base-58 letters. Vanity addresses require generating and testing billions of

candidate private keys, until one derives a bitcoin address with the desired pattern. Although there are some optimizations in the vanity generation algorithm, the process essentially involves picking a private key at random, deriving the public key, deriving the bitcoin address, and checking to see if it matches the desired vanity pattern, repeating billions of times until a match is found.

Once a vanity address matching the desired pattern is found, the private key from which it was derived can be used by the owner to spend bitcoins in exactly the same way as any other address. Vanity addresses are no less or more secure than any other address. They depend on the same Elliptic Curve Cryptography (ECC) and Secure Hash Algorithm (SHA) as any other address. You can no more easily find the private key of an address starting with a vanity pattern than you can any other address.

In [Introduction](#), we introduced Eugenia, a children's charity director operating in the Philippines. Let's say that Eugenia is organizing a bitcoin fundraising drive and wants to use a vanity bitcoin address to publicize the fundraising. Eugenia will create a vanity address that starts with "1Kids" to promote the children's charity fundraiser. Let's see how this vanity address will be created and what it means for the security of Eugenia's charity.

Generating vanity addresses

It's important to realize that a bitcoin address is simply a number represented by symbols in the Base58 alphabet. The search for a pattern like "1Kids" can be seen as searching for an address in the range from **`1KIds111111111111111111111111`** to **`1KIdszzzzzzzzzzzzzzzzzzzzzzzzzzzzzz`**. There are approximately 58^{29} (approximately 1.4×10^{51}) addresses in that range, all starting with "1Kids".
The range of vanity addresses starting with "1Kids" shows the range of addresses that have the prefix 1Kids.

Table 11. The range of vanity addresses starting with "1Kids"

[illegible]

Let's look at the pattern "1Kids" as a number and see how frequently we might find this pattern in a bitcoin address (see [The frequency of a vanity pattern \(1KidsCharity\) and average time-to-find on a desktop PC](#)). An average desktop computer PC, without any specialized hardware, can search approximately 100,000 keys per second.

Table 12. The frequency of a vanity pattern (1KidsCharity) and average time-to-find on a desktop PC

Length	Pattern	Frequency	Average search time
1	1K	1 in 58 keys	< 1 milliseconds
2	1Ki	1 in 3,364	50 milliseconds
3	1Kid	1 in 195,000	< 2 seconds

Length	Pattern	Frequency	Average search time
4	1Kids	1 in 11 million	1 minute
5	1KidsC	1 in 656 million	1 hour
6	1KidsCh	1 in 38 billion	2 days
7	1KidsCha	1 in 2.2 trillion	3–4 months
8	1KidsChar	1 in 128 trillion	13–18 years
9	1KidsChari	1 in 7 quadrillion	800 years
10	1KidsCharit	1 in 400 quadrillion	46,000 years
11	1KidsCharity	1 in 23 quintillion	2.5 million years

As you can see, Eugenia won't be creating the vanity address "1KidsCharity" any time soon, even if she had access to several thousand computers. Each additional character increases the difficulty by a factor of 58. Patterns with more than seven characters are usually found by specialized hardware, such as custom-built desktops with multiple graphical processing units (GPUs). These are often repurposed bitcoin mining "rigs" that are no longer profitable for bitcoin mining but can be used to find vanity addresses. Vanity searches on GPU systems are many orders of magnitude faster than on a general-purpose CPU.

Another way to find a vanity address is to outsource the work to a pool of vanity miners, such as the pool at [Vanity Pool](#). A pool is a service that allows those with GPU hardware to earn bitcoin searching for vanity addresses for others. For a small payment (0.01 bitcoin or approximately \$5 at the time of this writing), Eugenia can outsource the search for a seven-character pattern vanity address and get results in a few hours instead of having to run a CPU search for months.

Generating a vanity address is a brute-force exercise: try a random key, check the resulting address to see if it matches the desired pattern, repeat until successful. [Vanity address miner](#) shows an example of a "vanity miner," a program designed to find vanity addresses, written in C++. The example uses the libbitcoin library, which we introduced in [Alternative Clients, Libraries, and Toolkits](#).

Example 10. Vanity address miner

```
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine);
// Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
bool match_found(const std::string& address);
```

```

int main()
{
    // random_device on Linux uses "/dev/urandom"
    // CAUTION: Depending on implementation this RNG may not be secure enough!
    // Do not use vanity keys generated by this example in production
    std::random_device random;
    std::default_random_engine engine(random());

    // Loop continuously...
    while (true)
    {
        // Generate a random secret.
        bc::ec_secret secret = random_secret(engine);
        // Get the address.
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_hex(secret) << std::endl;
            return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value...
    for (uint8_t& byte: secret)
        byte = engine() % std::numeric_limits<uint8_t>::max();
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr;
    bc::set_public_key(payaddr, pubkey);
    // Return encoded form.
    return payaddr.encoded();
}

bool match_found(const std::string& address)

```

```

{
    auto addr_it = address.begin();
    // Loop through the search string comparing it to the lower case
    // character of the supplied address.
    for (auto it = search.begin(); it != search.end(); ++it, ++addr_it)
        if (*it != std::tolower(*addr_it))
            return false;
    // Reached end of search string, so address matches.
    return true;
}

```

NOTE

The example above uses `std::random_device`. Depending on the implementation it may reflect a cryptographically secure random number generator (CSRNG) provided by the underlying operating system. In the case of UNIX-like operating system such as Linux, it draws from `/dev/urandom`. While the random number generator used here is for demonstration purposes, it is *not* appropriate for generating production-quality bitcoin keys as it is not implemented with sufficient security.

The example code must be compiled using a C compiler and linked against the libbitcoin library (which must be first installed on that system). To run the example, run the `vanity-miner++` executable with no parameters (see [Compiling and running the vanity-miner example](#)) and it will attempt to find a vanity address starting with "1kid".

Example 11. Compiling and running the vanity-miner example

```

$ # Compile the code with g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the example
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Run it again for a different result
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Use "time" to see how long it takes to find a result
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real    0m8.868s
user    0m8.828s
sys     0m0.035s

```

The example code will take a few seconds to find a match for the three-character pattern "kid", as we can see when we use the `time` Unix command to measure the execution time. Change the `search`

pattern in the source code and see how much longer it takes for four- or five-character patterns!

Vanity address security

Vanity addresses can be used to enhance *and* to defeat security measures; they are truly a double-edged sword. Used to improve security, a distinctive address makes it harder for adversaries to substitute their own address and fool your customers into paying them instead of you. Unfortunately, vanity addresses also make it possible for anyone to create an address that *resembles* any random address, or even another vanity address, thereby fooling your customers.

Eugenia could advertise a randomly generated address (e.g., `1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy`) to which people can send their donations. Or, she could generate a vanity address that starts with 1Kids, to make it more distinctive.

In both cases, one of the risks of using a single fixed address (rather than a separate dynamic address per donor) is that a thief might be able to infiltrate your website and replace it with his own address, thereby diverting donations to himself. If you have advertised your donation address in a number of different places, your users may visually inspect the address before making a payment to ensure it is the same one they saw on your website, on your email, and on your flyer. In the case of a random address like `1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy`, the average user will perhaps inspect the first few characters "1J7mdg" and be satisfied that the address matches. Using a vanity address generator, someone with the intent to steal by substituting a similar-looking address can quickly generate addresses that match the first few characters, as shown in [Generating vanity addresses to match a random address](#).

Table 13. Generating vanity addresses to match a random address

Original Random Address	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
Vanity (4 character match)	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
Vanity (5 character match)	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
Vanity (6 character match)	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

So does a vanity address increase security? If Eugenia generates the vanity address `1Kids33q44erFfpeXrmDSz7zEqG2FesZEN`, users are likely to look at the vanity pattern word *and* a few characters beyond, for example noticing the "1Kids33" part of the address. That would force an attacker to generate a vanity address matching at least six characters (two more), expending an effort that is 3,364 times (58⁵; 58) higher than the effort Eugenia expended for her four-character vanity. Essentially, the effort Eugenia expends (or pays a vanity pool for) "pushes" the attacker into having to produce a longer pattern vanity. If Eugenia pays a pool to generate an 8-character vanity address, the attacker would be pushed into the realm of 10 characters, which is infeasible on a personal computer and expensive even with a custom vanity-mining rig or vanity pool. What is affordable for Eugenia becomes unaffordable for the attacker, especially if the potential reward of fraud is not high enough to cover the cost of the vanity address generation.

Paper Wallets

Paper wallets are bitcoin private keys printed on paper. Often the paper wallet also includes the

corresponding bitcoin address for convenience, but this is not necessary because it can be derived from the private key. Paper wallets are a very effective way to create backups or offline bitcoin storage, also known as "cold storage." As a backup mechanism, a paper wallet can provide security against the loss of key due to a computer mishap such as a hard drive failure, theft, or accidental deletion. As a "cold storage" mechanism, if the paper wallet keys are generated offline and never stored on a computer system, they are much more secure against hackers, key-loggers, and other online computer threats.

Paper wallets come in many shapes, sizes, and designs, but at a very basic level are just a key and an address printed on paper. [Simplest form of a paper wallet—a printout of the bitcoin address and private key](#) shows the simplest form of a paper wallet.

Table 14. Simplest form of a paper wallet—a printout of the bitcoin address and private key.

Public Address	Private Key (WIF)
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpb nkeyhfsYB1Jcn

Paper wallets can be generated easily using a tool such as the client-side JavaScript generator at [bitaddress.org](#). This page contains all the code necessary to generate keys and paper wallets, even while completely disconnected from the Internet. To use it, save the HTML page on your local drive or on an external USB flash drive. Disconnect from the Internet and open the file in a browser. Even better, boot your computer using a pristine operating system, such as a CD-ROM bootable Linux OS. Any keys generated with this tool while offline can be printed on a local printer over a USB cable (not wirelessly), thereby creating paper wallets whose keys exist only on the paper and have never been stored on any online system. Put these paper wallets in a fireproof safe and "send" bitcoin to their bitcoin address, to implement a simple yet highly effective "cold storage" solution. [An example of a simple paper wallet from bitaddress.org](#) shows a paper wallet generated from the bitaddress.org site.



Figure 30. An example of a simple paper wallet from bitaddress.org

The disadvantage of the simple paper wallet system is that the printed keys are vulnerable to theft. A thief who is able to gain access to the paper can either steal it or photograph the keys and take control of the bitcoins locked with those keys. A more sophisticated paper wallet storage system uses BIP0038 encrypted private keys. The keys printed on the paper wallet are protected by a

passphrase that the owner has memorized. Without the passphrase, the encrypted keys are useless. Yet, they still are superior to a passphrase-protected wallet because the keys have never been online and must be physically retrieved from a safe or other physically secured storage. [An example of an encrypted paper wallet from bitaddress.org](#). The passphrase is "test." shows a paper wallet with an encrypted private key (BIP0038) created on the bitaddress.org site.



Figure 31. An example of an encrypted paper wallet from bitaddress.org. The passphrase is "test."

WARNING

Although you can deposit funds into a paper wallet several times, you should withdraw all funds only once, spending everything. This is because in the process of unlocking and spending funds some wallets might generate a change address if you spend less than the whole amount. Additionally, if the computer you use to sign the transaction is compromised, you risk exposing the private key. By spending the entire balance of a paper wallet only once, you reduce the risk of key compromise. If you need only a small amount, send any remaining funds to a new paper wallet in the same transaction.

Paper wallets come in many designs and sizes, with many different features. Some are intended to be given as gifts and have seasonal themes, such as Christmas and New Year's themes. Others are designed for storage in a bank vault or safe with the private key hidden in some way, either with opaque scratch-off stickers, or folded and sealed with tamper-proof adhesive foil. Figures [<a data-type="xref" href="#paper_wallet_bpw" data-xrefstyle="select: labelnumber">#paper_wallet_bpw](#) through [<a data-type="xref" href="#paper_wallet_spw" data-xrefstyle="select: labelnumber">#paper_wallet_spw](#) show various examples of paper wallets with security and backup features.

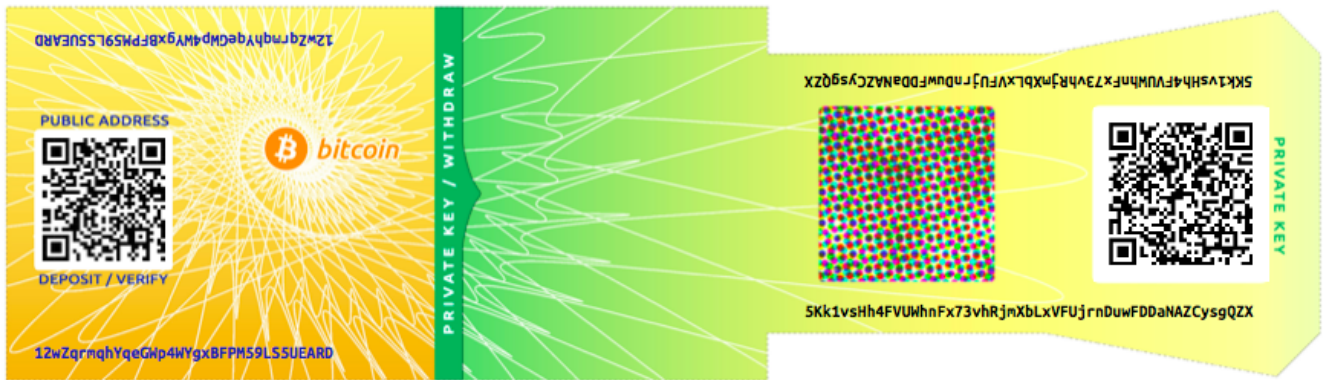


Figure 32. An example of a paper wallet from bitcoinpaperwallet.com with the private key on a folding flap.



Figure 33. The bitcoinpaperwallet.com paper wallet with the private key concealed.

Other designs feature additional copies of the key and address, in the form of detachable stubs similar to ticket stubs, allowing you to store multiple copies to protect against fire, flood, or other natural disasters.



Figure 34. An example of a paper wallet with additional copies of the keys on a backup "stub."

Transactions

Introduction

Transactions are the most important part of the bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the bitcoin system. Each transaction is a public entry in bitcoin's blockchain, the global double-entry bookkeeping ledger.

In this chapter we will examine all the various forms of transactions, what they contain, how to create them, how they are verified, and how they become part of the permanent record of all transactions.

Transaction Lifecycle

A transaction's lifecycle starts with the transaction's creation, also known as *origination*. The transaction is then signed with one or more signatures indicating the authorization to spend the funds referenced by the transaction. The transaction is then broadcast on the bitcoin network, where each network node (participant) validates and propagates the transaction until it reaches (almost) every node in the network. Finally, the transaction is verified by a mining node and included in a block of transactions that is recorded on the blockchain.

Once recorded on the blockchain and confirmed by sufficient subsequent blocks (confirmations), the transaction is a permanent part of the bitcoin ledger and is accepted as valid by all participants. The funds allocated to a new owner by the transaction can then be spent in a new transaction, extending the chain of ownership and beginning the lifecycle of a transaction again.

Creating Transactions

In some ways it helps to think of a transaction in the same way as a paper check. Like a check, a transaction is an instrument that expresses the intent to transfer money and is not visible to the financial system until it is submitted for execution. Like a check, the originator of the transaction does not have to be the one signing the transaction.

Transactions can be created online or offline by anyone, even if the person creating the transaction is not an authorized signer on the account. For example, an accounts payable clerk might process payable checks for signature by the CEO. Similarly, an accounts payable clerk can create bitcoin transactions and then have the CEO apply digital signatures to make them valid. Whereas a check references a specific account as the source of the funds, a bitcoin transaction references a specific previous transaction as its source, rather than an account.

Once a transaction has been created, it is signed by the owner (or owners) of the source funds. If it is properly formed and signed, the signed transaction is now valid and contains all the information needed to execute the transfer of funds. Finally, the valid transaction has to reach the bitcoin network so that it can be propagated until it reaches a miner for inclusion in the public ledger (the blockchain).

Broadcasting Transactions to the Bitcoin Network

First, a transaction needs to be delivered to the bitcoin network so that it can be propagated and included in the blockchain. In essence, a bitcoin transaction is just 300 to 400 bytes of data and has to reach any one of tens of thousands of bitcoin nodes. The senders do not need to trust the nodes they use to broadcast the transaction, as long as they use more than one to ensure that it propagates. The nodes don't need to trust the sender or establish the sender's "identity." Because the transaction is signed and contains no confidential information, private keys, or credentials, it can be publicly broadcast using any underlying network transport that is convenient. Unlike credit card transactions, for example, which contain sensitive information and can only be transmitted on encrypted networks, a bitcoin transaction can be sent over any network. As long as the transaction can reach a bitcoin node that will propagate it into the bitcoin network, it doesn't matter how it is transported to the first node.

Bitcoin transactions can therefore be transmitted to the bitcoin network over insecure networks such as WiFi, Bluetooth, NFC, Chirp, barcodes, or by copying and pasting into a web form. In extreme cases, a bitcoin transaction could be transmitted over packet radio, satellite relay, or shortwave using burst transmission, spread spectrum, or frequency hopping to evade detection and jamming. A bitcoin transaction could even be encoded as smileys (emoticons) and posted in a public forum or sent as a text message or Skype chat message. Bitcoin has turned money into a data structure, making it virtually impossible to stop anyone from creating and executing a bitcoin transaction.

Propagating Transactions on the Bitcoin Network

Once a bitcoin transaction is sent to any node connected to the bitcoin network, the transaction will be validated by that node. If valid, that node will propagate it to the other nodes to which it is connected, and a success message will be returned synchronously to the originator. If the transaction is invalid, the node will reject it and synchronously return a rejection message to the originator.

The bitcoin network is a peer-to-peer network, meaning that each bitcoin node is connected to a few other bitcoin nodes that it discovers during startup through the peer-to-peer protocol. The entire network forms a loosely connected mesh without a fixed topology or any structure, making all nodes equal peers. Messages, including transactions and blocks, are propagated from each node to all the peers to which it is connected, a process called "flooding." A new validated transaction injected into any node on the network will be sent to all of the nodes connected to it (neighbors), each of which will send the transaction to all its neighbors, and so on. In this way, within a few seconds a valid transaction will propagate in an exponentially expanding ripple across the network until all nodes in the network have received it.

The bitcoin network is designed to propagate transactions and blocks to all nodes in an efficient and resilient manner that is resistant to attacks. To prevent spamming, denial-of-service attacks, or other nuisance attacks against the bitcoin system, every node independently validates every transaction before propagating it further. A malformed transaction will not get beyond one node. The rules by which transactions are validated are explained in more detail in [Independent Verification of Transactions](#).

Transaction Structure

A transaction is a *data structure* that encodes a transfer of value from a source of funds, called an *input*, to a destination, called an *output*. Transaction inputs and outputs are not related to accounts or identities. Instead, you should think of them as bitcoin amounts—chunks of bitcoin—being locked with a specific secret that only the owner, or person who knows the secret, can unlock. A transaction contains a number of fields, as shown in [The structure of a transaction](#).

Table 15. The structure of a transaction

Size	Field	Description
4 bytes	Version	Specifies which rules this transaction follows
1–9 bytes (VarInt)	Input Counter	How many inputs are included
Variable	Inputs	One or more transaction inputs
1–9 bytes (VarInt)	Output Counter	How many outputs are included
Variable	Outputs	One or more transaction outputs
4 bytes	Locktime	A Unix timestamp or block number

Transaction Locktime

Locktime, also known as `nLockTime` from the variable name used in the reference client, defines the earliest time that a transaction is valid and can be relayed on the network or added to the blockchain. It is set to zero in most transactions to indicate immediate propagation and execution. If locktime is nonzero and below 500 million, it is interpreted as a block height, meaning the transaction is not valid and is not relayed or included in the blockchain prior to the specified block height. If it is above 500 million, it is interpreted as a Unix Epoch timestamp (seconds since Jan-1-1970) and the transaction is not valid prior to the specified time. Transactions with locktime specifying a future block or time must be held by the originating system and transmitted to the bitcoin network only after they become valid. The use of locktime is equivalent to postdating a paper check.

Transaction Outputs and Inputs

The fundamental building block of a bitcoin transaction is an *unspent transaction output*, or UTXO. UTXO are indivisible chunks of bitcoin currency locked to a specific owner, recorded on the blockchain, and recognized as currency units by the entire network. The bitcoin network tracks all available (unspent) UTXO currently numbering in the millions. Whenever a user receives bitcoin, that amount is recorded within the blockchain as a UTXO. Thus, a user's bitcoin might be scattered as UTXO amongst hundreds of transactions and hundreds of blocks. In effect, there is no such thing as a stored balance of a bitcoin address or account; there are only scattered UTXO, locked to specific owners. The concept of a user's bitcoin balance is a derived construct created by the wallet application. The wallet calculates the user's balance by scanning the blockchain and aggregating all

UTXO belonging to that user.

TIP

There are no accounts or balances in bitcoin; there are only *unspent transaction outputs* (UTXO) scattered in the blockchain.

A UTXO can have an arbitrary value denominated as a multiple of satoshis. Just like dollars can be divided down to two decimal places as cents, bitcoins can be divided down to eight decimal places as satoshis. Although UTXO can be any arbitrary value, once created it is indivisible just like a coin that cannot be cut in half. If a UTXO is larger than the desired value of a transaction, it must still be consumed in its entirety and change must be generated in the transaction. In other words, if you have a 20 bitcoin UTXO and want to pay 1 bitcoin, your transaction must consume the entire 20 bitcoin UTXO and produce two outputs: one paying 1 bitcoin to your desired recipient and another paying 19 bitcoin in change back to your wallet. As a result, most bitcoin transactions will generate change.

Imagine a shopper buying a \$1.50 beverage, reaching into her wallet and trying to find a combination of coins and bank notes to cover the \$1.50 cost. The shopper will choose exact change if available (a dollar bill and two quarters), or a combination of smaller denominations (six quarters), or if necessary, a larger unit such as a five dollar bank note. If she hands too much money, say \$5, to the shop owner, she will expect \$3.50 change, which she will return to her wallet and have available for future transactions.

Similarly, a bitcoin transaction must be created from a user's UTXO in whatever denominations that user has available. Users cannot cut a UTXO in half any more than they can cut a dollar bill in half and use it as currency. The user's wallet application will typically select from the user's available UTXO various units to compose an amount greater than or equal to the desired transaction amount.

As with real life, the bitcoin application can use several strategies to satisfy the purchase amount: combining several smaller units, finding exact change, or using a single unit larger than the transaction value and making change. All of this complex assembly of spendable UTXO is done by the user's wallet automatically and is invisible to users. It is only relevant if you are programmatically constructing raw transactions from UTXO.

The UTXO consumed by a transaction are called transaction inputs, and the UTXO created by a transaction are called transaction outputs. This way, chunks of bitcoin value move forward from owner to owner in a chain of transactions consuming and creating UTXO. Transactions consume UTXO by unlocking it with the signature of the current owner and create UTXO by locking it to the bitcoin address of the new owner.

The exception to the output and input chain is a special type of transaction called the *coinbase* transaction, which is the first transaction in each block. This transaction is placed there by the "winning" miner and creates brand-new bitcoin payable to that miner as a reward for mining. This is how bitcoin's money supply is created during the mining process, as we will see in [Mining and Consensus](#).

TIP

What comes first? Inputs or outputs, the chicken or the egg? Strictly speaking, outputs come first because coinbase transactions, which generate new bitcoin, have no inputs and create outputs from nothing.

Transaction Outputs

Every bitcoin transaction creates outputs, which are recorded on the bitcoin ledger. Almost all of these outputs, with one exception (see [Data Output \(OP_RETURN\)](#)) create spendable chunks of bitcoin called *unspent transaction outputs* or UTXO, which are then recognized by the whole network and available for the owner to spend in a future transaction. Sending someone bitcoin is creating an unspent transaction output (UTXO) registered to their address and available for them to spend.

UTXO are tracked by every full-node bitcoin client as a data set called the *UTXO set* or *UTXO pool*, held in a database. New transactions consume (spend) one or more of these outputs from the UTXO set.

Transaction outputs consist of two parts:

- An amount of bitcoin, denominated in *satoshis*, the smallest bitcoin unit
- A *locking script*, also known as an "encumbrance" that "locks" this amount by specifying the conditions that must be met to spend the output

The transaction scripting language, used in the locking script mentioned previously, is discussed in detail in [Transaction Scripts and Script Language](#). [The structure of a transaction output](#) shows the structure of a transaction output.

Table 16. The structure of a transaction output

Size	Field	Description
8 bytes	Amount	Bitcoin value in satoshis (10^{-8} bitcoin)
1-9 bytes (VarInt)	Locking-Script Size	Locking-Script length in bytes, to follow
Variable	Locking-Script	A script defining the conditions needed to spend the output

In [A script that calls the blockchain.info API to find the UTXO related to an address](#), we use the blockchain.info API to find the unspent outputs (UTXO) of a specific address.

Example 12. A script that calls the blockchain.info API to find the UTXO related to an address

```
# get unspent outputs from blockchain API

import json
import requests

# example address
address = '1Dorian4RoXcnBv9hnQ4Y2C1an6NJ4UrxX'

# The API URL is https://blockchain.info/unspent?active=<address>
# It returns a JSON object with a list "unspent_outputs", containing UTXO, like
this:
#{  "unspent_outputs":[
#    {
#
#      "tx_hash":"ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167",
#      "tx_index":51919767,
#      "tx_output_n": 1,
#      "script":"76a9148c7e252f8d64b0b6e313985915110fcfefcf4a2d88ac",
#      "value": 8000000,
#      "value_hex": "7a1200",
#      "confirmations":28691
#    },
#    ...
#  ]}

resp = requests.get('https://blockchain.info/unspent?active=%s' % address)
utxo_set = json.loads(resp.text)["unspent_outputs"]

for utxo in utxo_set:
    print "%s:%d - %ld Satoshis" % (utxo['tx_hash'], utxo['tx_output_n'],
    utxo['value'])
```

Running the script, we see a list of transaction IDs, a colon, the index number of the specific unspent transaction output (UTXO), and the value of that UTXO in satoshis. The locking script is not shown in the output in [Running the get-utxo.py script](#).

Example 13. Running the `get-utxo.py` script

```
$ python get-utxo.py
ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167:1 - 8000000
Satoshis
6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 - 16050000
Satoshis
74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0:0 - 5000000
Satoshis
b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4:0 - 10000000
Satoshis
...
```

Spending conditions (encumbrances)

Transaction outputs associate a specific amount (in satoshis) to a specific *encumbrance* or locking script that defines the condition that must be met to spend that amount. In most cases, the locking script will lock the output to a specific bitcoin address, thereby transferring ownership of that amount to the new owner. When Alice paid Bob's Cafe for a cup of coffee, her transaction created a 0.015 bitcoin output *encumbered* or locked to the cafe's bitcoin address. That 0.015 bitcoin output was recorded on the blockchain and became part of the Unspent Transaction Output set, meaning it showed in Bob's wallet as part of the available balance. When Bob chooses to spend that amount, his transaction will release the encumbrance, unlocking the output by providing an unlocking script containing a signature from Bob's private key.

Transaction Inputs

In simple terms, transaction inputs are pointers to UTXO. They point to a specific UTXO by reference to the transaction hash and sequence number where the UTXO is recorded in the blockchain. To spend UTXO, a transaction input also includes unlocking scripts that satisfy the spending conditions set by the UTXO. The unlocking script is usually a signature proving ownership of the bitcoin address that is in the locking script.

When users make a payment, their wallet constructs a transaction by selecting from the available UTXO. For example, to make a 0.015 bitcoin payment, the wallet app may select a 0.01 UTXO and a 0.005 UTXO, using them both to add up to the desired payment amount.

In [A script for calculating how much total bitcoin will be issued](#), we show the use of a "greedy" algorithm to select from available UTXO in order to make a specific payment amount. In the example, the available UTXO are provided as a constant array, but in reality, the available UTXO would be retrieved with an RPC call to Bitcoin Core, or to a third-party API as shown in [A script that calls the blockchain.info API to find the UTXO related to an address](#).

Example 14. A script for calculating how much total bitcoin will be issued

```
# Selects outputs from a UTXO list using a greedy algorithm.
```

```

from sys import argv

class OutputInfo:

    def __init__(self, tx_hash, tx_index, value):
        self.tx_hash = tx_hash
        self.tx_index = tx_index
        self.value = value

    def __repr__(self):
        return "<%s:%s with %s Satoshis>" % (self.tx_hash, self.tx_index,
                                              self.value)

# Select optimal outputs for a send from unspent outputs list.
# Returns output list and remaining change to be sent to
# a change address.
def select_outputs_greedy(unspent, min_value):
    # Fail if empty.
    if not unspent:
        return None
    # Partition into 2 lists.
    lessers = [utxo for utxo in unspent if utxo.value < min_value]
    greater = [utxo for utxo in unspent if utxo.value >= min_value]
    key_func = lambda utxo: utxo.value
    if greater:
        # Not-empty. Find the smallest greater.
        min_greater = min(greater)
        change = min_greater.value - min_value
        return [min_greater], change
    # Not found in greater. Try several lessers instead.
    # Rearrange them from biggest to smallest. We want to use the least
    # amount of inputs as possible.
    lessers.sort(key=key_func, reverse=True)
    result = []
    accum = 0
    for utxo in lessers:
        result.append(utxo)
        accum += utxo.value
        if accum >= min_value:
            change = accum - min_value
            return result, "Change: %d Satoshis" % change
    # No results found.
    return None, 0

def main():
    unspent = [
        OutputInfo("ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167", 1,
                  8000000),
        OutputInfo("6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf", 0,

```

```

16050000),

OutputInfo("b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4", 0,
10000000),

OutputInfo("7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1", 0,
25000000),

OutputInfo("55ea01bd7e9afd3d3ab9790199e777d62a0709cf0725e80a7350fdb22d7b8ec6", 17,
5470541),

OutputInfo("12b6a7934c1df821945ee9ee3b3326d07ca7a65fd6416ea44ce8c3db0c078c64", 0,
10000000),

OutputInfo("7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818", 0,
16100000),
]

if len(argv) > 1:
    target = long(argv[1])
else:
    target = 55000000

print "For transaction amount %d Satoshis (%f bitcoin) use: " % (target,
target/10.0**8)
print select_outputs_greedy(unspent, target)

if __name__ == "__main__":
    main()

```

If we run the *select-utxo.py* script without a parameter, it will attempt to construct a set of UTXO (and change) for a payment of 55,000,000 satoshis (0.55 bitcoin). If you provide a target payment amount as a parameter, the script will select UTXO to make that target payment amount. In [Running the select-utxo.py script](#), we run the script trying to make a payment of 0.5 bitcoin or 50,000,000 satoshis.

Example 15. Running the select-utxo.py script

```

$ python select-utxo.py 50000000
For transaction amount 50000000 Satoshis (0.500000 bitcoin) use:
(<7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1:0 with
25000000 Satoshis>,
<7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818:0 with 16100000
Satoshis>, <6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0
with 16050000 Satoshis>], 'Change: 7150000 Satoshis')

```

Once the UTXO is selected, the wallet then produces unlocking scripts containing signatures for

each of the UTXO, thereby making them spendable by satisfying their locking script conditions. The wallet adds these UTXO references and unlocking scripts as inputs to the transaction. [The structure of a transaction input](#) shows the structure of a transaction input.

Table 17. The structure of a transaction input

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent; first one is 0
1-9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script.
4 bytes	Sequence Number	Currently disabled Tx-replacement feature, set to 0xFFFFFFFF

NOTE

The sequence number is used to override a transaction prior to the expiration of the transaction locktime, which is a feature that is currently disabled in bitcoin. Most transactions set this value to the maximum integer value (0xFFFFFFFF) and it is ignored by the bitcoin network. If the transaction has a nonzero locktime, at least one of its inputs must have a sequence number below 0xFFFFFFFF in order to enable locktime.

Transaction Fees

Most transactions include transaction fees, which compensate the bitcoin miners for securing the network. Mining and the fees and rewards collected by miners are discussed in more detail in [Mining and Consensus](#). This section examines how transaction fees are included in a typical transaction. Most wallets calculate and include transaction fees automatically. However, if you are constructing transactions programmatically, or using a command-line interface, you must manually account for and include these fees.

Transaction fees serve as an incentive to include (mine) a transaction into the next block and also as a disincentive against "spam" transactions or any kind of abuse of the system, by imposing a small cost on every transaction. Transaction fees are collected by the miner who mines the block that records the transaction on the blockchain.

Transaction fees are calculated based on the size of the transaction in kilobytes, not the value of the transaction in bitcoin. Overall, transaction fees are set based on market forces within the bitcoin network. Miners prioritize transactions based on many different criteria, including fees, and might even process transactions for free under certain circumstances. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next-most-mined block, whereas a transaction with insufficient or no fees might be delayed,

processed on a best-effort basis after a few blocks, or not processed at all. Transaction fees are not mandatory, and transactions without fees might be processed eventually; however, including transaction fees encourages priority processing.

Over time, the way transaction fees are calculated and the effect they have on transaction prioritization has been evolving. At first, transaction fees were fixed and constant across the network. Gradually, the fee structure has been relaxed so that it may be influenced by market forces, based on network capacity and transaction volume. The current minimum transaction fee is fixed at 0.0001 bitcoin or a tenth of a milli-bitcoin per kilobyte, recently decreased from one milli-bitcoin. Most transactions are less than one kilobyte; however, those with multiple inputs or outputs can be larger. In future revisions of the bitcoin protocol, it is expected that wallet applications will use statistical analysis to calculate the most appropriate fee to attach to a transaction based on the average fees of recent transactions.

The current algorithm used by miners to prioritize transactions for inclusion in a block based on their fees is examined in detail in [Mining and Consensus](#).

Adding Fees to Transactions

The data structure of transactions does not have a field for fees. Instead, fees are implied as the difference between the sum of inputs and the sum of outputs. Any excess amount that remains after all outputs have been deducted from all inputs is the fee that is collected by the miners.

Transaction fees are implied, as the excess of inputs minus outputs:

$$\text{Fees} = \text{Sum}(\text{Inputs}) - \text{Sum}(\text{Outputs})$$

This is a somewhat confusing element of transactions and an important point to understand, because if you are constructing your own transactions you must ensure you do not inadvertently include a very large fee by underspending the inputs. That means that you must account for all inputs, if necessary by creating change, or you will end up giving the miners a very big tip!

For example, if you consume a 20-bitcoin UTXO to make a 1-bitcoin payment, you must include a 19-bitcoin change output back to your wallet. Otherwise, the 19-bitcoin "leftover" will be counted as a transaction fee and will be collected by the miner who mines your transaction in a block. Although you will receive priority processing and make a miner very happy, this is probably not what you intended.

WARNING

If you forget to add a change output in a manually constructed transaction, you will be paying the change as a transaction fee. "Keep the change!" might not be what you intended.

Let's see how this works in practice, by looking at Alice's coffee purchase again. Alice wants to spend 0.015 bitcoin to pay for coffee. To ensure this transaction is processed promptly, she will want to include a transaction fee, say 0.001. That will mean that the total cost of the transaction will be 0.016. Her wallet must therefore source a set of UTXO that adds up to 0.016 bitcoin or more and, if necessary, create change. Let's say her wallet has a 0.2-bitcoin UTXO available. It will therefore need to consume this UTXO, create one output to Bob's Cafe for 0.015, and a second output with

0.184 bitcoin in change back to her own wallet, leaving 0.001 bitcoin unallocated, as an implicit fee for the transaction.

Now let's look at a different scenario. Eugenia, our children's charity director in the Philippines, has completed a fundraiser to purchase school books for the children. She received several thousand small donations from people all around the world, totaling 50 bitcoin, so her wallet is full of very small payments (UTXO). Now she wants to purchase hundreds of school books from a local publisher, paying in bitcoin.

As Eugenia's wallet application tries to construct a single larger payment transaction, it must source from the available UTXO set, which is composed of many smaller amounts. That means that the resulting transaction will source from more than a hundred small-value UTXO as inputs and only one output, paying the book publisher. A transaction with that many inputs will be larger than one kilobyte, perhaps 2 to 3 kilobytes in size. As a result, it will require a higher fee than the minimal network fee of 0.0001 bitcoin.

Eugenia's wallet application will calculate the appropriate fee by measuring the size of the transaction and multiplying that by the per-kilobyte fee. Many wallets will overpay fees for larger transactions to ensure the transaction is processed promptly. The higher fee is not because Eugenia is spending more money, but because her transaction is more complex and larger in size—the fee is independent of the transaction's bitcoin value.

Transaction Chaining and Orphan Transactions

As we have seen, transactions form a chain, whereby one transaction spends the outputs of the previous transaction (known as the parent) and creates outputs for a subsequent transaction (known as the child). Sometimes an entire chain of transactions depending on each other—say a parent, child, and grandchild transaction—are created at the same time, to fulfill a complex transactional workflow that requires valid children to be signed before the parent is signed. For example, this is a technique used in CoinJoin transactions where multiple parties join transactions together to protect their privacy.

When a chain of transactions is transmitted across the network, they don't always arrive in the same order. Sometimes, the child might arrive before the parent. In that case, the nodes that see a child first can see that it references a parent transaction that is not yet known. Rather than reject the child, they put it in a temporary pool to await the arrival of its parent and propagate it to every other node. The pool of transactions without parents is known as the *orphan transaction pool*. Once the parent arrives, any orphans that reference the UTXO created by the parent are released from the pool, revalidated recursively, and then the entire chain of transactions can be included in the transaction pool, ready to be mined in a block. Transaction chains can be arbitrarily long, with any number of generations transmitted simultaneously. The mechanism of holding orphans in the orphan pool ensures that otherwise valid transactions will not be rejected just because their parent has been delayed and that eventually the chain they belong to is reconstructed in the correct order, regardless of the order of arrival.

There is a limit to the number of orphan transactions stored in memory, to prevent a denial-of-service attack against bitcoin nodes. The limit is defined as `MAX_ORPHAN_TRANSACTIONS` in the source code of the bitcoin reference client. If the number of orphan transactions in the pool exceeds `MAX_ORPHAN_TRANSACTIONS`, one or more randomly selected orphan transactions are evicted from the

pool, until the pool size is back within limits.

Transaction Scripts and Script Language

Bitcoin clients validate transactions by executing a script, written in a Forth-like scripting language. Both the locking script (encumbrance) placed on a UTXO and the unlocking script that usually contains a signature are written in this scripting language. When a transaction is validated, the unlocking script in each input is executed alongside the corresponding locking script to see if it satisfies the spending condition.

Today, most transactions processed through the bitcoin network have the form "Alice pays Bob" and are based on the same script called a Pay-to-Public-Key-Hash script. However, the use of scripts to lock outputs and unlock inputs means that through use of the programming language, transactions can contain an infinite number of conditions. Bitcoin transactions are not limited to the "Alice pays Bob" form and pattern.

This is only the tip of the iceberg of possibilities that can be expressed with this scripting language. In this section, we will demonstrate the components of the bitcoin transaction scripting language and show how it can be used to express complex conditions for spending and how those conditions can be satisfied by unlocking scripts.

TIP

Bitcoin transaction validation is not based on a static pattern, but instead is achieved through the execution of a scripting language. This language allows for a nearly infinite variety of conditions to be expressed. This is how bitcoin gets the power of "programmable money."

Script Construction (Lock + Unlock)

Bitcoin's transaction validation engine relies on two types of scripts to validate transactions: a locking script and an unlocking script.

A locking script is an encumbrance placed on an output, and it specifies the conditions that must be met to spend the output in the future. Historically, the locking script was called a *scriptPubKey*, because it usually contained a public key or bitcoin address. In this book we refer to it as a "locking script" to acknowledge the much broader range of possibilities of this scripting technology. In most bitcoin applications, what we refer to as a locking script will appear in the source code as *scriptPubKey*.

An unlocking script is a script that "solves," or satisfies, the conditions placed on an output by a locking script and allows the output to be spent. Unlocking scripts are part of every transaction input, and most of the time they contain a digital signature produced by the user's wallet from his or her private key. Historically, the unlocking script is called *scriptSig*, because it usually contained a digital signature. In most bitcoin applications, the source code refers to the unlocking script as *scriptSig*. In this book, we refer to it as an "unlocking script" to acknowledge the much broader range of locking script requirements, because not all unlocking scripts must contain signatures.

Every bitcoin client will validate transactions by executing the locking and unlocking scripts together. For each input in the transaction, the validation software will first retrieve the UTXO

referenced by the input. That UTXO contains a locking script defining the conditions required to spend it. The validation software will then take the unlocking script contained in the input that is attempting to spend this UTXO and execute the two scripts.

In the original bitcoin client, the unlocking and locking scripts were concatenated and executed in sequence. For security reasons, this was changed in 2010, because of a vulnerability that allowed a malformed unlocking script to push data onto the stack and corrupt the locking script. In the current implementation, the scripts are executed separately with the stack transferred between the two executions, as described next.

First, the unlocking script is executed, using the stack execution engine. If the unlocking script executed without errors (e.g., it has no "dangling" operators left over), the main stack (not the alternate stack) is copied and the locking script is executed. If the result of executing the locking script with the stack data copied from the unlocking script is "TRUE," the unlocking script has succeeded in resolving the conditions imposed by the locking script and, therefore, the input is a valid authorization to spend the UTXO. If any result other than "TRUE" remains after execution of the combined script, the input is invalid because it has failed to satisfy the spending conditions placed on the UTXO. Note that the UTXO is permanently recorded in the blockchain, and therefore is invariable and is unaffected by failed attempts to spend it by reference in a new transaction. Only a valid transaction that correctly satisfies the conditions of the UTXO results in the UTXO being marked as "spent" and removed from the set of available (unspent) UTXO.

Combining `scriptSig` and `scriptPubKey` to evaluate a transaction script is an example of the unlocking and locking scripts for the most common type of bitcoin transaction (a payment to a public key hash), showing the combined script resulting from the concatenation of the unlocking and locking scripts prior to script validation.

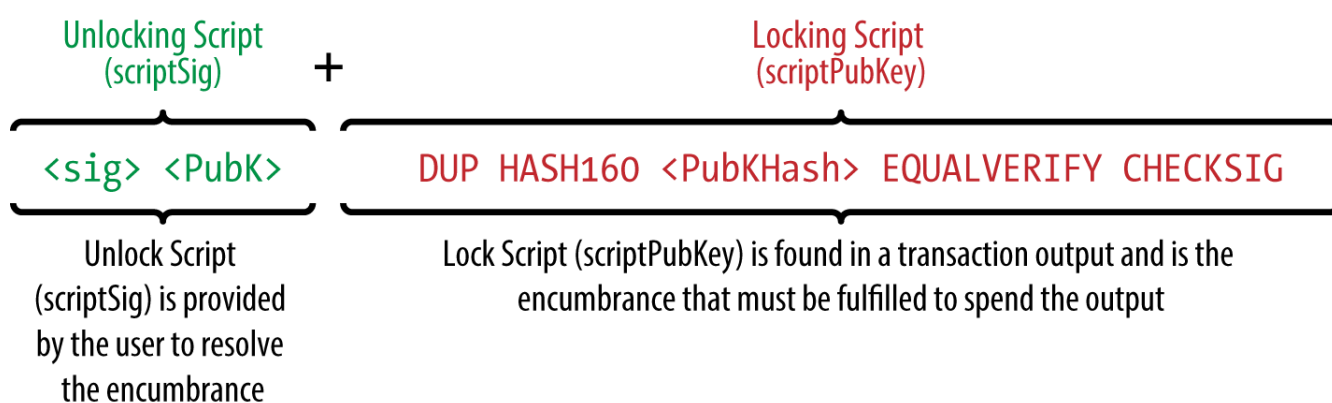


Figure 35. Combining `scriptSig` and `scriptPubKey` to evaluate a transaction script

Scripting Language

The bitcoin transaction script language, called *Script*, is a Forth-like reverse-polish notation stack-based execution language. If that sounds like gibberish, you probably haven't studied 1960's programming languages. Script is a very simple language that was designed to be limited in scope and executable on a range of hardware, perhaps as simple as an embedded device, such as a handheld calculator. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. In the case of programmable money, that is a deliberate security feature.

Bitcoin's scripting language is called a stack-based language because it uses a data structure called a *stack*. A stack is a very simple data structure, which can be visualized as a stack of cards. A stack allows two operations: push and pop. Push adds an item on top of the stack. Pop removes the top item from the stack.

The scripting language executes the script by processing each item from left to right. Numbers (data constants) are pushed onto the stack. Operators push or pop one or more parameters from the stack, act on them, and might push a result onto the stack. For example, `OP_ADD` will pop two items from the stack, add them, and push the resulting sum onto the stack.

Conditional operators evaluate a condition, producing a boolean result of TRUE or FALSE. For example, `OP_EQUAL` pops two items from the stack and pushes TRUE (TRUE is represented by the number 1) if they are equal or FALSE (represented by zero) if they are not equal. Bitcoin transaction scripts usually contain a conditional operator, so that they can produce the TRUE result that signifies a valid transaction.

In [Bitcoin's script validation doing simple math](#), the script `2 3 OP_ADD 5 OP_EQUAL` demonstrates the arithmetic addition operator `OP_ADD`, adding two numbers and putting the result on the stack, followed by the conditional operator `OP_EQUAL`, which checks that the resulting sum is equal to 5. For brevity, the `OP_` prefix is omitted in the step-by-step example.

The following is a slightly more complex script, which calculates $2 + 7 - 3 + 1$. Notice that when the script contains several operators in a row, the stack allows the results of one operator to be acted upon by the next operator:

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

Try validating the preceding script yourself using pencil and paper. When the script execution ends, you should be left with the value TRUE on the stack.

Although most locking scripts refer to a bitcoin address or public key, thereby requiring proof of ownership to spend the funds, the script does not have to be that complex. Any combination of locking and unlocking scripts that results in a TRUE value is valid. The simple arithmetic we used as an example of the scripting language is also a valid locking script that can be used to lock a transaction output.

Use part of the arithmetic example script as the locking script:

```
3 OP_ADD 5 OP_EQUAL
```

which can be satisfied by a transaction containing an input with the unlocking script:

```
2
```

The validation software combines the locking and unlocking scripts and the resulting script is:

```
2 3 OP_ADD 5 OP_EQUAL
```

As we saw in the step-by-step example in [Bitcoin's script validation doing simple math](#), when this script is executed, the result is `OP_TRUE`, making the transaction valid. Not only is this a valid transaction output locking script, but the resulting UTXO could be spent by anyone with the arithmetic skills to know that the number 2 satisfies the script.

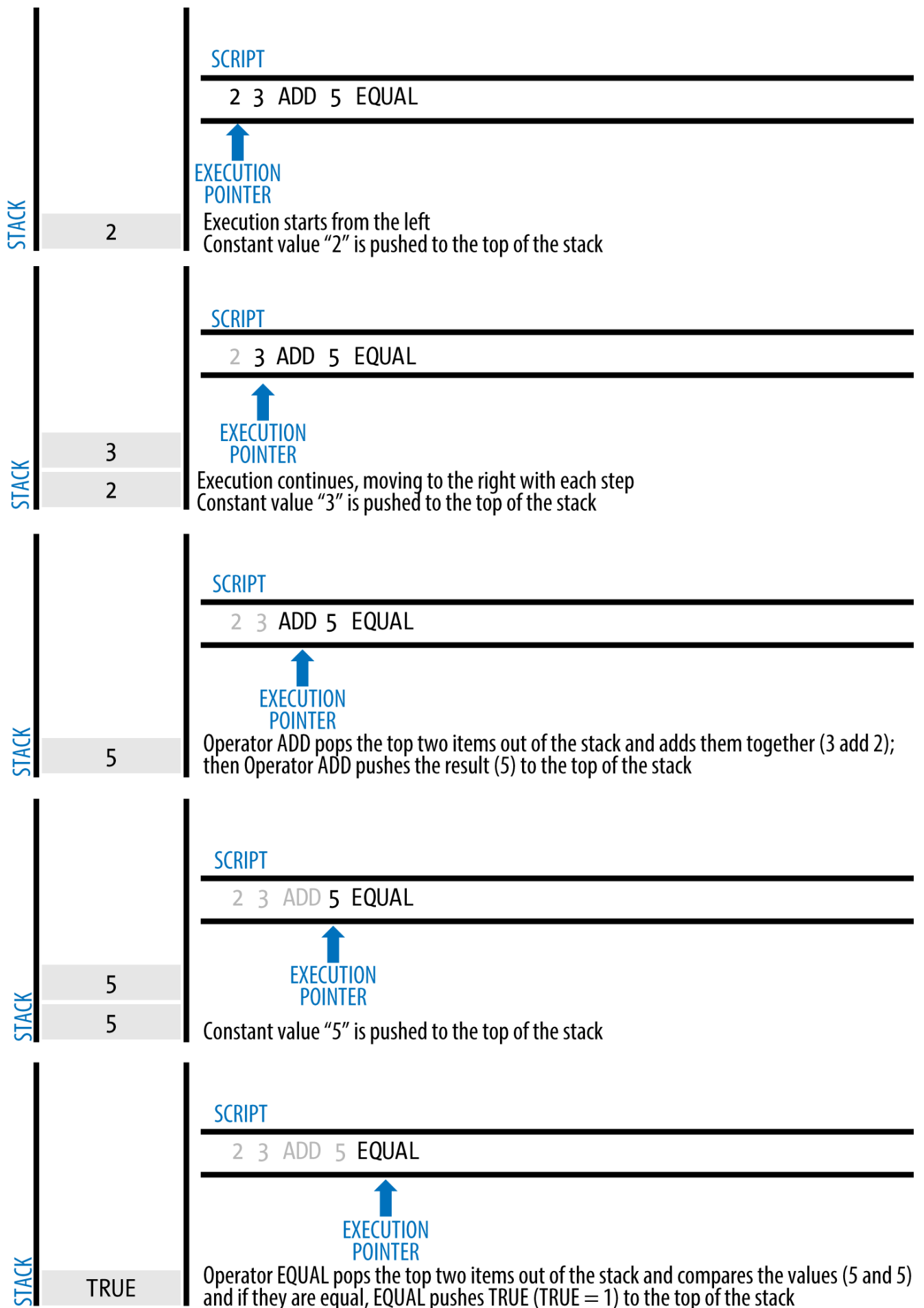


Figure 36. Bitcoin's script validation doing simple math

TIP

Transactions are valid if the top result on the stack is TRUE (noted as `{0x01}`), any other non-zero value or if the stack is empty after script execution. Transactions are invalid if the top value on the stack is FALSE (a zero-length empty value, noted as `{}`) or if script execution is halted explicitly by an operator, such as OP_VERIFY, OP_RETURN, or a conditional terminator such as OP_ENDIF. See [Transaction Script Language Operators, Constants, and Symbols](#) for details.

Turing Incompleteness

The bitcoin transaction script language contains many operators, but is deliberately limited in one important way—there are no loops or complex flow control capabilities other than conditional flow control. This ensures that the language is not *Turing Complete*, meaning that scripts have limited complexity and predictable execution times. Script is not a general-purpose language. These limitations ensure that the language cannot be used to create an infinite loop or other form of "logic bomb" that could be embedded in a transaction in a way that causes a denial-of-service attack against the bitcoin network. Remember, every transaction is validated by every full node on the bitcoin network. A limited language prevents the transaction validation mechanism from being used as a vulnerability.

Stateless Verification

The bitcoin transaction script language is stateless, in that there is no state prior to execution of the script, or state saved after execution of the script. Therefore, all the information needed to execute a script is contained within the script. A script will predictably execute the same way on any system. If your system verifies a script, you can be sure that every other system in the bitcoin network will also verify the script, meaning that a valid transaction is valid for everyone and everyone knows this. This predictability of outcomes is an essential benefit of the bitcoin system.

Standard Transactions

In the first few years of bitcoin's development, the developers introduced some limitations in the types of scripts that could be processed by the reference client. These limitations are encoded in a function called `isStandard()`, which defines five types of "standard" transactions. These limitations are temporary and might be lifted by the time you read this. Until then, the five standard types of transaction scripts are the only ones that will be accepted by the reference client and most miners who run the reference client. Although it is possible to create a nonstandard transaction containing a script that is not one of the standard types, you must find a miner who does not follow these limitations to mine that transaction into a block.

Check the source code of the Bitcoin Core client (the reference implementation) to see what is currently allowed as a valid transaction script.

The five standard types of transaction scripts are pay-to-public-key-hash (P2PKH), public-key, multi-signature (limited to 15 keys), pay-to-script-hash (P2SH), and data output (OP_RETURN), which are described in more detail in the following sections.

Pay-to-Public-Key-Hash (P2PKH)

The vast majority of transactions processed on the bitcoin network are P2PKH transactions. These contain a locking script that encumbers the output with a public key hash, more commonly known as a bitcoin address. Transactions that pay a bitcoin address contain P2PKH scripts. An output locked by a P2PKH script can be unlocked (spent) by presenting a public key and a digital signature created by the corresponding private key.

For example, let's look at Alice's payment to Bob's Cafe again. Alice made a payment of 0.015 bitcoin to the cafe's bitcoin address. That transaction output would have a locking script of the form:

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

The **Cafe Public Key Hash** is equivalent to the bitcoin address of the cafe, without the Base58Check encoding. Most applications would show the *public key hash* in hexadecimal encoding and not the familiar bitcoin address Base58Check format that begins with a "1".

The preceding locking script can be satisfied with an unlocking script of the form:

```
<Cafe Signature> <Cafe Public Key>
```

The two scripts together would form the following combined validation script:

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160  
<Cafe Public Key Hash> OP_EQUAL OP_CHECKSIG
```

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In other words, the result will be TRUE if the unlocking script has a valid signature from the cafe's private key that corresponds to the public key hash set as an encumbrance.

Figures <#P2PubKHash1> and <#P2PubKHash2> show (in two parts) a step-by-step execution of the combined script, which will prove this is a valid transaction.

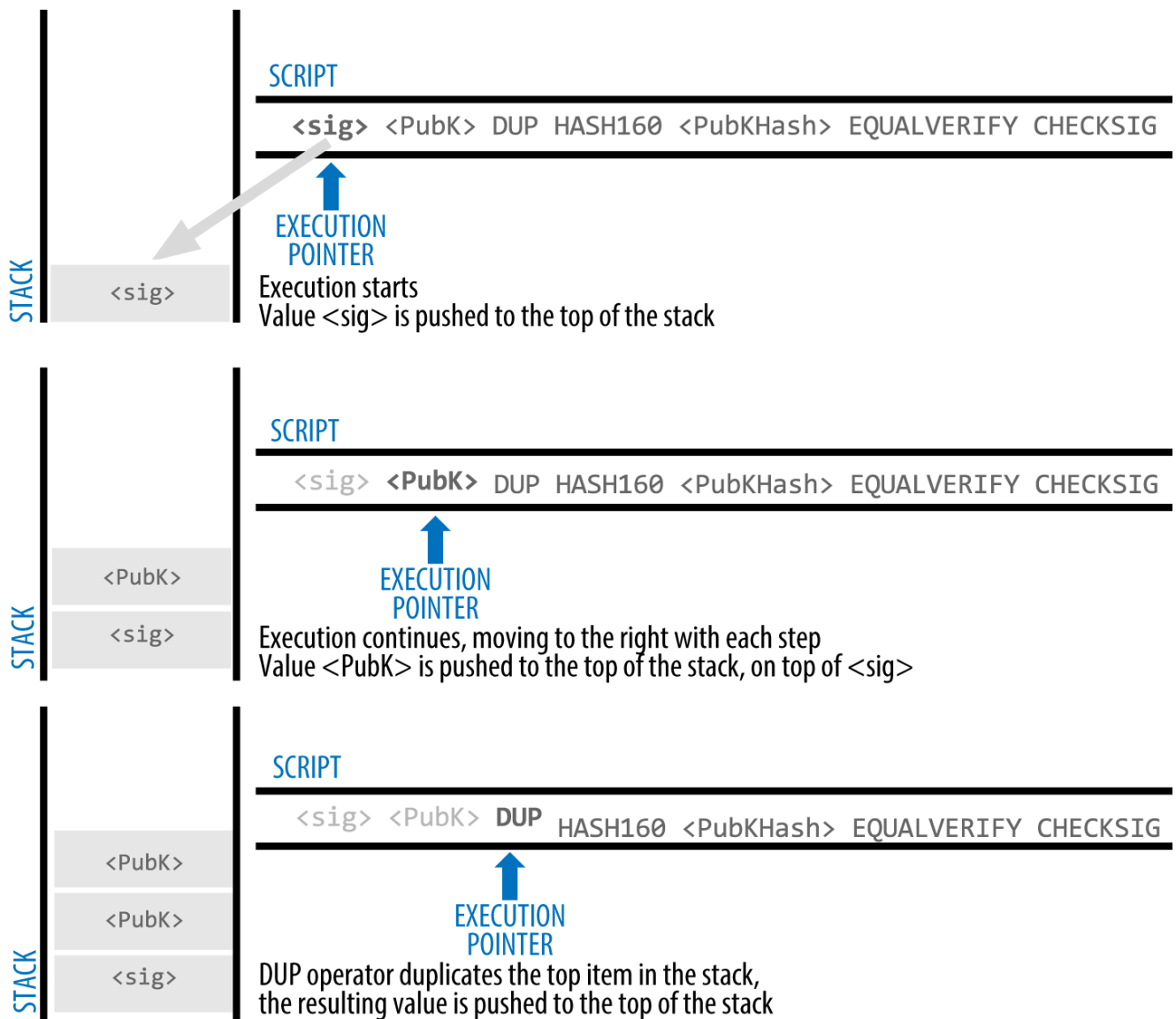


Figure 37. Evaluating a script for a P2PKH transaction (Part 1 of 2)

Pay-to-Public-Key

Pay-to-public-key is a simpler form of a bitcoin payment than pay-to-public-key-hash. With this script form, the public key itself is stored in the locking script, rather than a public-key-hash as with P2PKH earlier, which is much shorter. Pay-to-public-key-hash was invented by Satoshi to make bitcoin addresses shorter, for ease of use. Pay-to-public-key is now most often seen in coinbase transactions, generated by older mining software that has not been updated to use P2PKH.

A pay-to-public-key locking script looks like this:

```
<Public Key A> OP_CHECKSIG
```

The corresponding unlocking script that must be presented to unlock this type of output is a simple signature, like this:

```
<Signature from Private Key A>
```

The combined script, which is validated by the transaction validation software, is:

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

This script is a simple invocation of the **CHECKSIG** operator, which validates the signature as belonging to the correct key and returns TRUE on the stack.

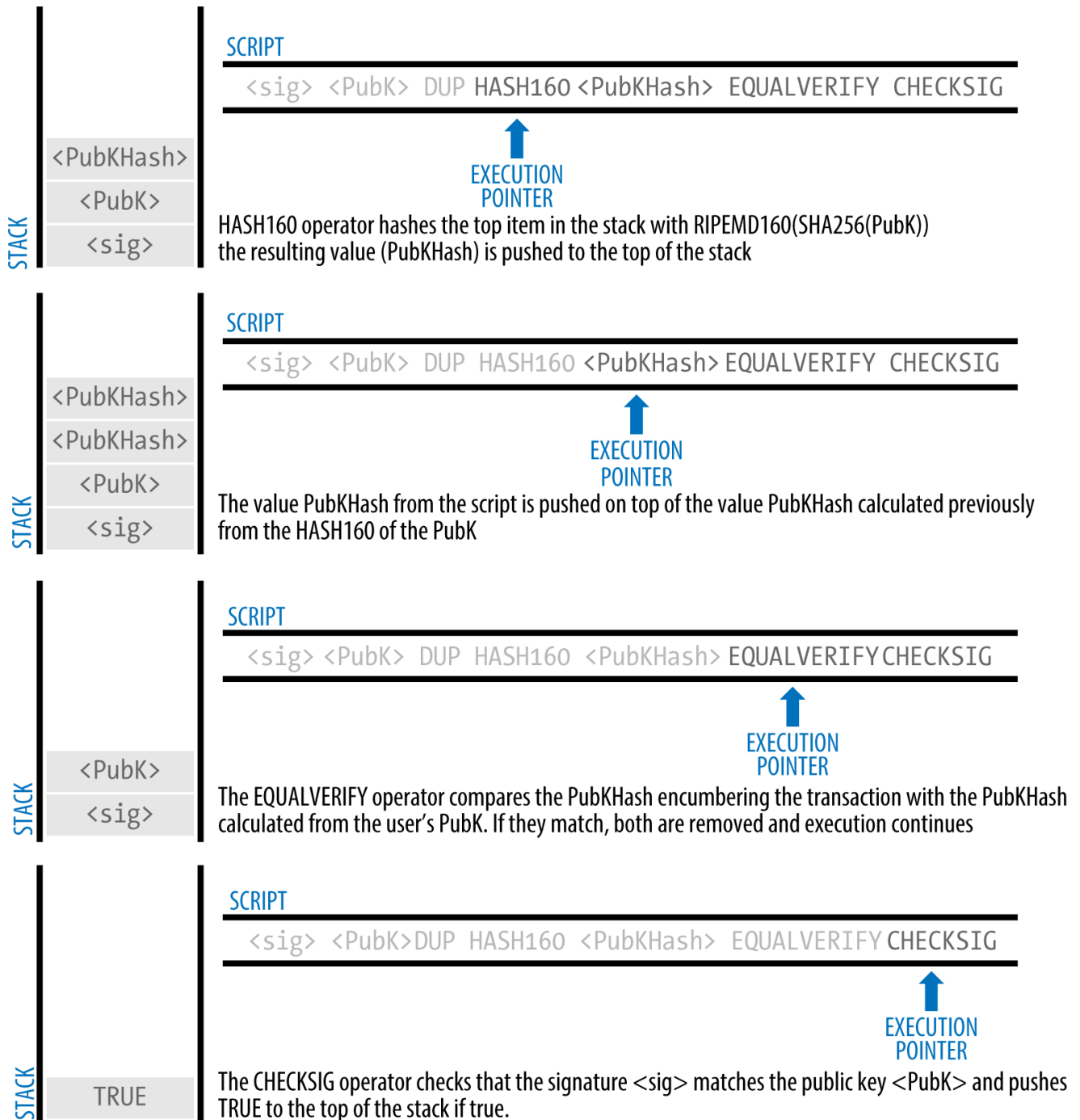


Figure 38. Evaluating a script for a P2PKH transaction (Part 2 of 2)

Multi-Signature

Multi-signature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to release the encumbrance. This is also known as an M-of-N scheme, where N is the total number of keys and M is the threshold of signatures required for

validation. For example, a 2-of-3 multi-signature is one where three public keys are listed as potential signers and at least two of those must be used to create signatures for a valid transaction to spend the funds. At this time, standard multi-signature scripts are limited to at most 15 listed public keys, meaning you can do anything from a 1-of-1 to a 15-of-15 multi-signature or any combination within that range. The limitation to 15 listed keys might be lifted by the time this book is published, so check the `isStandard()` function to see what is currently accepted by the network.

The general form of a locking script setting an M-of-N multi-signature condition is:

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

where N is the total number of listed public keys and M is the threshold of required signatures to spend the output.

A locking script setting a 2-of-3 multi-signature condition looks like this:

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

The preceding locking script can be satisfied with an unlocking script containing pairs of signatures and public keys:

```
OP_0 <Signature B> <Signature C>
```

or any combination of two signatures from the private keys corresponding to the three listed public keys.

NOTE

The prefix `OP_0` is required because of a bug in the original implementation of `CHECKMULTISIG` where one item too many is popped off the stack. It is ignored by `CHECKMULTISIG` and is simply a placeholder.

The two scripts together would form the combined validation script:

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3  
OP_CHECKMULTISIG
```

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In this case, the condition is whether the unlocking script has a valid signature from the two private keys that correspond to two of the three public keys set as an encumbrance.

Data Output (OP_RETURN)

Bitcoin's distributed and timestamped ledger, the blockchain, has potential uses far beyond payments. Many developers have tried to use the transaction scripting language to take advantage of the security and resilience of the system for applications such as digital notary services, stock

certificates, and smart contracts. Early attempts to use bitcoin's script language for these purposes involved creating transaction outputs that recorded data on the blockchain; for example, to record a digital fingerprint of a file in such a way that anyone could establish proof-of-existence of that file on a specific date by reference to that transaction.

The use of bitcoin's blockchain to store data unrelated to bitcoin payments is a controversial subject. Many developers consider such use abusive and want to discourage it. Others view it as a demonstration of the powerful capabilities of blockchain technology and want to encourage such experimentation. Those who object to the inclusion of non-payment data argue that it causes "blockchain bloat," burdening those running full bitcoin nodes with carrying the cost of disk storage for data that the blockchain was not intended to carry. Moreover, such transactions create UTXO that cannot be spent, using the destination bitcoin address as a free-form 20-byte field. Because the address is used for data, it doesn't correspond to a private key and the resulting UTXO can *never* be spent; it's a fake payment. These transactions that can never be spent are therefore never removed from the UTXO set and cause the size of the UTXO database to forever increase, or "bloat."

In version 0.9 of the Bitcoin Core client, a compromise was reached with the introduction of the `OP_RETURN` operator. `OP_RETURN` allows developers to add 80 bytes of nonpayment data to a transaction output. However, unlike the use of "fake" UTXO, the `OP_RETURN` operator creates an explicitly *provably unspendable* output, which does not need to be stored in the UTXO set. `OP_RETURN` outputs are recorded on the blockchain, so they consume disk space and contribute to the increase in the blockchain's size, but they are not stored in the UTXO set and therefore do not bloat the UTXO memory pool and burden full nodes with the cost of more expensive RAM.

`OP_RETURN` scripts look like this:

```
OP_RETURN <data>
```

The data portion is limited to 80 bytes and most often represents a hash, such as the output from the SHA256 algorithm (32 bytes). Many applications put a prefix in front of the data to help identify the application. For example, the [Proof of Existence](#) digital notarization service uses the 8-byte prefix `DOCPROOF`, which is ASCII encoded as `44 4f 43 50 52 4f 4f 46` in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to `OP_RETURN` that could possibly be used to "spend" an `OP_RETURN` output. The whole point of `OP_RETURN` is that you can't spend the money locked in that output, and therefore it does not need to be held in the UTXO set as potentially spendable—`OP_RETURN` is *provably un-spendable*. `OP_RETURN` is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If an `OP_RETURN` is encountered by the script validation software, it results immediately in halting the execution of the validation script and marking the transaction as invalid. Thus, if you accidentally reference an `OP_RETURN` output as an input in a transaction, that transaction is invalid.

A standard transaction (one that conforms to the `isStandard()` checks) can have only one `OP_RETURN` output. However, a single `OP_RETURN` output can be combined in a transaction with outputs of any other type.

Two new command-line options have been added in Bitcoin Core as of version 0.10. The option

`datacarrier` controls relay and mining of OP_RETURN transactions, with the default set to "1" to allow them. The option `datacarriersize` takes a numeric argument specifying the maximum size in bytes of the OP_RETURN data, 40 bytes by default.

NOTE

OP_RETURN was initially proposed with a limit of 80 bytes, but the limit was reduced to 40 bytes when the feature was released. In February 2015, in version 0.10 of Bitcoin Core, the limit was raised back to 80 bytes. Nodes may choose not to relay or mine OP_RETURN, or only relay and mine OP_RETURN containing less than 80 bytes of data.

Pay-to-Script-Hash (P2SH)

Pay-to-script-hash (P2SH) was introduced in 2012 as a powerful new type of transaction that greatly simplifies the use of complex transaction scripts. To explain the need for P2SH, let's look at a practical example.

In [Introduction](#) we introduced Mohammed, an electronics importer based in Dubai. Mohammed's company uses bitcoin's multi-signature feature extensively for its corporate accounts. Multi-signature scripts are one of the most common uses of bitcoin's advanced scripting capabilities and are a very powerful feature. Mohammed's company uses a multi-signature script for all customer payments, known in accounting terms as "accounts receivable," or AR. With the multi-signature scheme, any payments made by customers are locked in such a way that they require at least two signatures to release, from Mohammed and one of his partners or from his attorney who has a backup key. A multi-signature scheme like that offers corporate governance controls and protects against theft, embezzlement, or loss.

The resulting script is quite long and looks like this:

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

Although multi-signature scripts are a powerful feature, they are cumbersome to use. Given the preceding script, Mohammed would have to communicate this script to every customer prior to payment. Each customer would have to use special bitcoin wallet software with the ability to create custom transaction scripts, and each customer would have to understand how to create a transaction using custom scripts. Furthermore, the resulting transaction would be about five times larger than a simple payment transaction, because this script contains very long public keys. The burden of that extra-large transaction would be borne by the customer in the form of fees. Finally, a large transaction script like this would be carried in the UTXO set in RAM in every full node, until it was spent. All of these issues make using complex output scripts difficult in practice.

Pay-to-script-hash (P2SH) was developed to resolve these practical difficulties and to make the use of complex scripts as easy as a payment to a bitcoin address. With P2SH payments, the complex locking script is replaced with its digital fingerprint, a cryptographic hash. When a transaction attempting to spend the UTXO is presented later, it must contain the script that matches the hash, in addition to the unlocking script. In simple terms, P2SH means "pay to a script matching this hash, a script that will be presented later when this output is spent."

In P2SH transactions, the locking script that is replaced by a hash is referred to as the *redeem script* because it is presented to the system at redemption time rather than as a locking script. [Complex script without P2SH](#) shows the script without P2SH and [Complex script as P2SH](#) shows the same script encoded with P2SH.

Table 18. Complex script without P2SH

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 19. Complex script as P2SH

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Locking Script	OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
Unlocking Script	Sig1 Sig2 redeem script

As you can see from the tables, with P2SH the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeem script itself is presented later, as part of the unlocking script when the output is spent. This shifts the burden in fees and complexity from the sender to the recipient (spender) of the transaction.

Let's look at Mohammed's company, the complex multi-signature script, and the resulting P2SH scripts.

First, the multi-signature script that Mohammed's company uses for all incoming payments from customers:

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

If the placeholders are replaced by actual public keys (shown here as 520-bit numbers starting with 04) you can see that this script becomes very long:

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6
984D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E
7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B
49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9
162F0279CFC10F1E8E8F3020DECD8C3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE6
FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD
94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF51
8C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800 5 OP_CHECKMULTISIG
```

This entire script can instead be represented by a 20-byte cryptographic hash, by first applying the SHA256 hashing algorithm and then applying the RIPEMD160 algorithm on the result. The 20-byte hash of the preceding script is:

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

A P2SH transaction locks the output to this hash instead of the longer script, using the locking script:

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

which, as you can see, is much shorter. Instead of "pay to this 5-key multi-signature script," the P2SH equivalent transaction is "pay to a script with this hash." A customer making a payment to Mohammed's company need only include this much shorter locking script in his payment. When Mohammed wants to spend this UTXO, they must present the original redeem script (the one whose hash locked the UTXO) and the signatures necessary to unlock it, like this:

```
<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG>
```

The two scripts are combined in two stages. First, the redeem script is checked against the locking script to make sure the hash matches:

```
<2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG> OP_HASH160 <redeem scriptHash> OP_EQUAL
```

If the redeem script hash matches, the unlocking script is executed on its own, to unlock the redeem script:

```
<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 OP_CHECKMULTISIG
```

Pay-to-script-hash addresses

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP0013. P2SH addresses are Base58Check encodings of the 20-byte hash of a script, just like bitcoin addresses are Base58Check encodings of the 20-byte hash of a public key. P2SH addresses use the version prefix "5", which results in Base58Check-encoded addresses that start with a "3". For example, Mohammed's complex script, hashed and Base58Check-encoded as a P2SH address becomes **39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw**. Now, Mohammed can give this "address" to his customers and they can use almost any bitcoin wallet to make a simple payment, as if it were a bitcoin address. The 3 prefix gives them a hint that this is a special type of address, one corresponding to a script instead of a public key, but otherwise it works in exactly the same way as a payment to a bitcoin address.

P2SH addresses hide all of the complexity, so that the person making a payment does not see the script.

Benefits of pay-to-script-hash

The pay-to-script-hash feature offers the following benefits compared to the direct use of complex scripts in locking outputs:

- Complex scripts are replaced by shorter fingerprints in the transaction output, making the transaction smaller.
- Scripts can be coded as an address, so the sender and the sender's wallet don't need complex engineering to implement P2SH.
- P2SH shifts the burden of constructing the script to the recipient, not the sender.
- P2SH shifts the burden in data storage for the long script from the output (which is in the UTXO set) to the input (stored on the blockchain).
- P2SH shifts the burden in data storage for the long script from the present time (payment) to a future time (when it is spent).
- P2SH shifts the transaction fee cost of a long script from the sender to the recipient, who has to include the long redeem script to spend it.

Redeem script and isStandard validation

Prior to version 0.9.2 of the Bitcoin Core client, pay-to-script-hash was limited to the standard types of bitcoin transaction scripts, by the `isStandard()` function. That means that the redeem script presented in the spending transaction could only be one of the standard types: P2PK, P2PKH, or multi-sig nature, excluding `OP_RETURN` and P2SH itself.

As of version 0.9.2 of the Bitcoin Core client, P2SH transactions can contain any valid script, making the P2SH standard much more flexible and allowing for experimentation with many novel and complex types of transactions.

Note that you are not able to put a P2SH inside a P2SH redeem script, because the P2SH specification is not recursive. You are also still not able to use `OP_RETURN` in a redeem script because `OP_RETURN` cannot be redeemed by definition.

Note that because the redeem script is not presented to the network until you attempt to spend a P2SH output, if you lock an output with the hash of an invalid transaction it will be processed regardless. However, you will not be able to spend it because the spending transaction, which includes the redeem script, will not be accepted because it is an invalid script. This creates a risk, because you can lock bitcoin in a P2SH that cannot be spent later. The network will accept the P2SH encumbrance even if it corresponds to an invalid redeem script, because the script hash gives no indication of the script it represents.

WARNING

P2SH locking scripts contain the hash of a redeem script, which gives no clues as to the content of the redeem script itself. The P2SH transaction will be considered valid and accepted even if the redeem script is invalid. You might accidentally lock bitcoin in such a way that it cannot later be spent.

The Bitcoin Network

Peer-to-Peer Network Architecture

Bitcoin is structured as a peer-to-peer network architecture on top of the Internet. The term peer-to-peer, or P2P, means that the computers that participate in the network are peers to each other, that they are all equal, that there are no "special" nodes, and that all nodes share the burden of providing network services. The network nodes interconnect in a mesh network with a "flat" topology. There is no server, no centralized service, and no hierarchy within the network. Nodes in a peer-to-peer network both provide and consume services at the same time with reciprocity acting as the incentive for participation. Peer-to-peer networks are inherently resilient, decentralized, and open. The preeminent example of a P2P network architecture was the early Internet itself, where nodes on the IP network were equal. Today's Internet architecture is more hierarchical, but the Internet Protocol still retains its flat-topology essence. Beyond bitcoin, the largest and most successful application of P2P technologies is file sharing with Napster as the pioneer and BitTorrent as the most recent evolution of the architecture.

Bitcoin's P2P network architecture is much more than a topology choice. Bitcoin is a peer-to-peer digital cash system by design, and the network architecture is both a reflection and a foundation of that core characteristic. Decentralization of control is a core design principle and that can only be achieved and maintained by a flat, decentralized P2P consensus network.

The term "bitcoin network" refers to the collection of nodes running the bitcoin P2P protocol. In addition to the bitcoin P2P protocol, there are other protocols such as Stratum, which are used for mining and lightweight or mobile wallets. These additional protocols are provided by gateway routing servers that access the bitcoin network using the bitcoin P2P protocol, and then extend that network to nodes running other protocols. For example, Stratum servers connect Stratum mining nodes via the Stratum protocol to the main bitcoin network and bridge the Stratum protocol to the bitcoin P2P protocol. We use the term "extended bitcoin network" to refer to the overall network that includes the bitcoin P2P protocol, pool-mining protocols, the Stratum protocol, and any other related protocols connecting the components of the bitcoin system.

Nodes Types and Roles

Although nodes in the bitcoin P2P network are equal, they may take on different roles depending on the functionality they are supporting. A bitcoin node is a collection of functions: routing, the blockchain database, mining, and wallet services. A full node with all four of these functions is shown in [A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing](#).

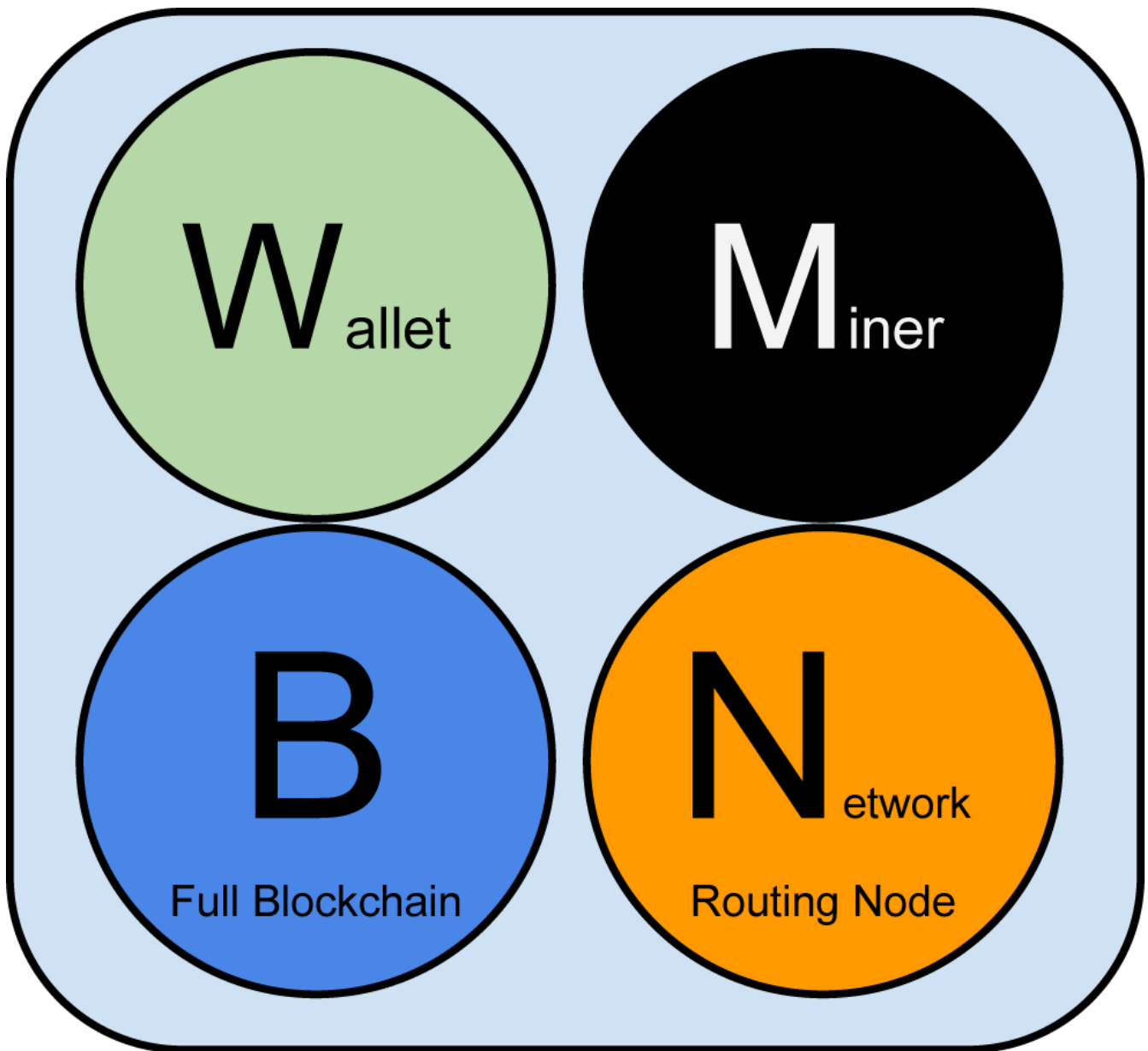


Figure 39. A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing

All nodes include the routing function to participate in the network and might include other functionality. All nodes validate and propagate transactions and blocks, and discover and maintain connections to peers. In the full-node example in [A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing](#), the routing function is indicated by an orange circle named "Network Routing Node."

Some nodes, called full nodes, also maintain a complete and up-to-date copy of the blockchain. Full nodes can autonomously and authoritatively verify any transaction without external reference. Some nodes maintain only a subset of the blockchain and verify transactions using a method called *simplified payment verification*, or SPV. These nodes are known as SPV or lightweight nodes. In the full-node example in the figure, the full-node blockchain database function is indicated by a blue circle named "Full Blockchain." In [The extended bitcoin network showing various node types, gateways, and protocols](#), SPV nodes are drawn without the blue circle, showing that they do not have a full copy of the blockchain.

Mining nodes compete to create new blocks by running specialized hardware to solve the proof-of-

work algorithm. Some mining nodes are also full nodes, maintaining a full copy of the blockchain, while others are lightweight nodes participating in pool mining and depending on a pool server to maintain a full node. The mining function is shown in the full node as a black circle named "Miner."

User wallets might be part of a full node, as is usually the case with desktop bitcoin clients. Increasingly, many user wallets, especially those running on resource-constrained devices such as smartphones, are SPV nodes. The wallet function is shown in [A bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing](#) as a green circle named "Wallet".

In addition to the main node types on the bitcoin P2P protocol, there are servers and nodes running other protocols, such as specialized mining pool protocols and lightweight client-access protocols.

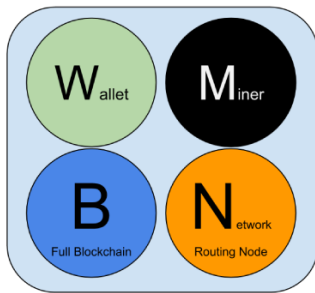
[Different types of nodes on the extended bitcoin network](#) shows the most common node types on the extended bitcoin network.

The Extended Bitcoin Network

The main bitcoin network, running the bitcoin P2P protocol, consists of between 7,000 and 10,000 listening nodes running various versions of the bitcoin reference client (Bitcoin Core) and a few hundred nodes running various other implementations of the bitcoin P2P protocol, such as BitcoinJ, Libbitcoin, and btcd. A small percentage of the nodes on the bitcoin P2P network are also mining nodes, competing in the mining process, validating transactions, and creating new blocks. Various large companies interface with the bitcoin network by running full-node clients based on the Bitcoin Core client, with full copies of the blockchain and a network node, but without mining or wallet functions. These nodes act as network edge routers, allowing various other services (exchanges, wallets, block explorers, merchant payment processing) to be built on top.

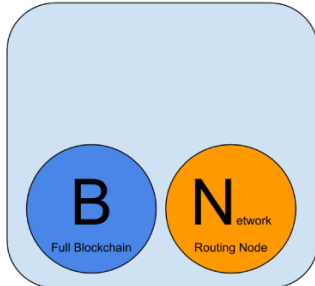
The extended bitcoin network includes the network running the bitcoin P2P protocol, described earlier, as well as nodes running specialized protocols. Attached to the main bitcoin P2P network are a number of pool servers and protocol gateways that connect nodes running other protocols. These other protocol nodes are mostly pool mining nodes (see [Mining and Consensus](#)) and lightweight wallet clients, which do not carry a full copy of the blockchain.

[The extended bitcoin network showing various node types, gateways, and protocols](#) shows the extended bitcoin network with the various types of nodes, gateway servers, edge routers, and wallet clients and the various protocols they use to connect to each other.



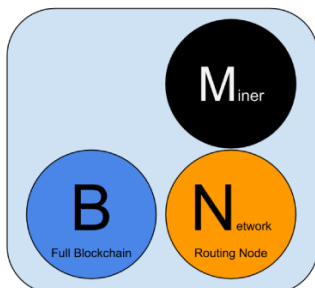
Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



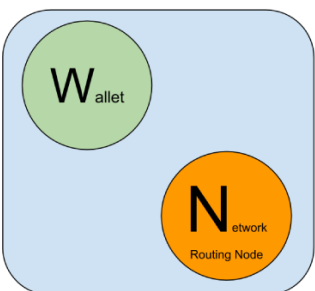
Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



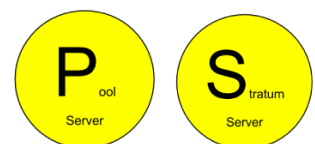
Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.



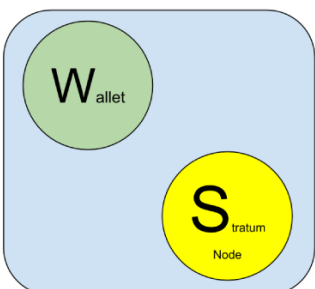
Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.



Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.

Figure 40. Different types of nodes on the extended bitcoin network

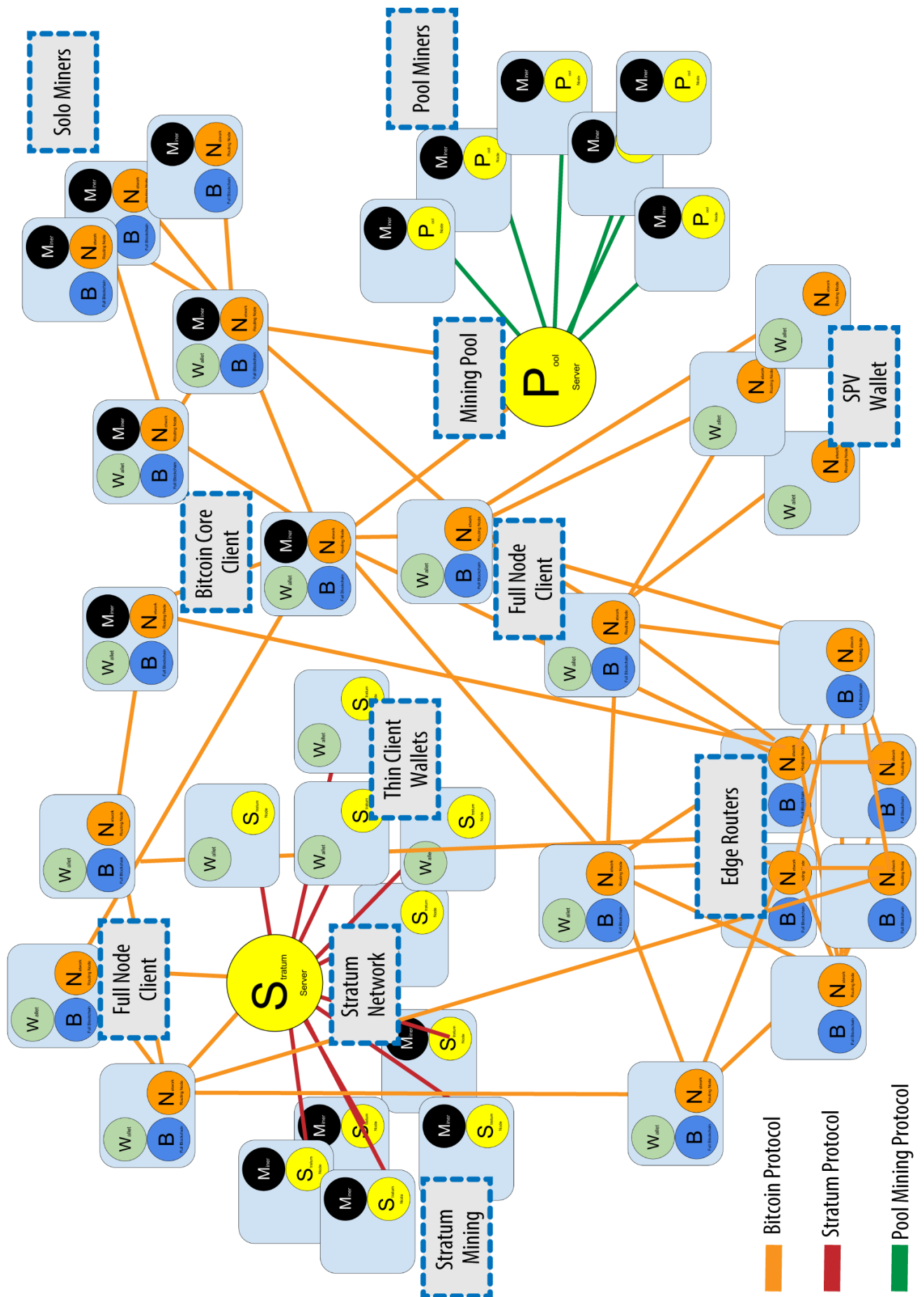


Figure 41. The extended bitcoin network showing various node types, gateways, and protocols

Network Discovery

When a new node boots up, it must discover other bitcoin nodes on the network in order to participate. To start this process, a new node must discover at least one existing node on the network and connect to it. The geographic location of other nodes is irrelevant; the bitcoin network topology is not geographically defined. Therefore, any existing bitcoin nodes can be selected at random.

To connect to a known peer, nodes establish a TCP connection, usually to port 8333 (the port generally known as the one used by bitcoin), or an alternative port if one is provided. Upon establishing a connection, the node will start a "handshake" (see [The initial handshake between peers](#)) by transmitting a `version` message, which contains basic identifying information, including:

`nVersion`

The bitcoin P2P protocol version the client "speaks" (e.g., 70002)

`nLocalServices`

A list of local services supported by the node, currently just `NODE_NETWORK`

`nTime`

The current time

`addrYou`

The IP address of the remote node as seen from this node

`addrMe`

The IP address of the local node, as discovered by the local node

`subver`

A sub-version showing the type of software running on this node (e.g., `"/Satoshi:0.9.2.1/"`)+

`BestHeight`

The block height of this node's blockchain

(See [GitHub](#) for an example of the `version` network message.)

The `version` message is always the first message sent by any peer to another peer. The local peer receiving a `version` message will examine the remote peer's reported `nVersion` and decide if the remote peer is compatible. If the remote peer is compatible, the local peer will acknowledge the `version` message and establish a connection, by sending a `verack`.

How does a new node find peers? The first method is to query DNS using a number of "DNS seeds," which are DNS servers that provide a list of IP addresses of bitcoin nodes. Some of those DNS seeds provide a static list of IP addresses of stable bitcoin listening nodes. Some of the DNS seeds are custom implementations of BIND (Berkeley Internet Name Daemon) that return a random subset from a list of bitcoin node addresses collected by a crawler or a long-running bitcoin node. The Bitcoin Core client contains the names of five different DNS seeds. The diversity of ownership and diversity of implementation of the different DNS seeds offers a high level of reliability for the initial bootstrapping process. In the Bitcoin Core client, the option to use the DNS seeds is controlled by the option switch `-dnsseed` (set to 1 by default, to use the DNS seed).

Alternatively, a bootstrapping node that knows nothing of the network must be given the IP address of at least one bitcoin node, after which it can establish connections through further introductions. The command-line argument `-seednode` can be used to connect to one node just for introductions, using it as a seed. After the initial seed node is used to form introductions, the client will disconnect from it and use the newly discovered peers.

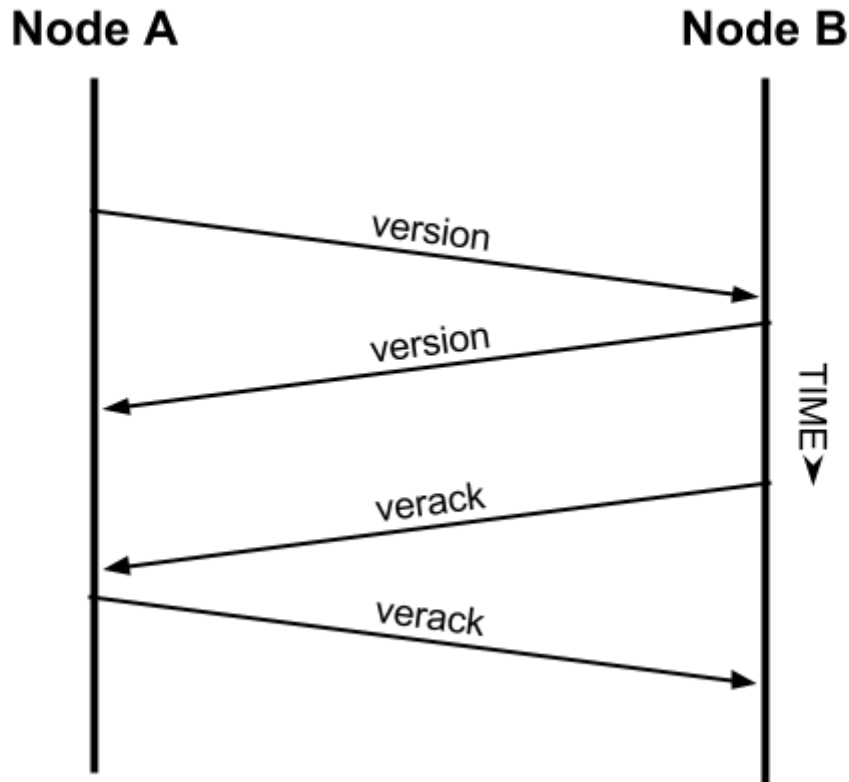


Figure 42. The initial handshake between peers

Once one or more connections are established, the new node will send an `addr` message containing its own IP address to its neighbors. The neighbors will, in turn, forward the `addr` message to their neighbors, ensuring that the newly connected node becomes well known and better connected. Additionally, the newly connected node can send `getaddr` to the neighbors, asking them to return a list of IP addresses of other peers. That way, a node can find peers to connect to and advertise its existence on the network for other nodes to find it. [Address propagation and discovery](#) shows the address discovery protocol.

Node A

Node B

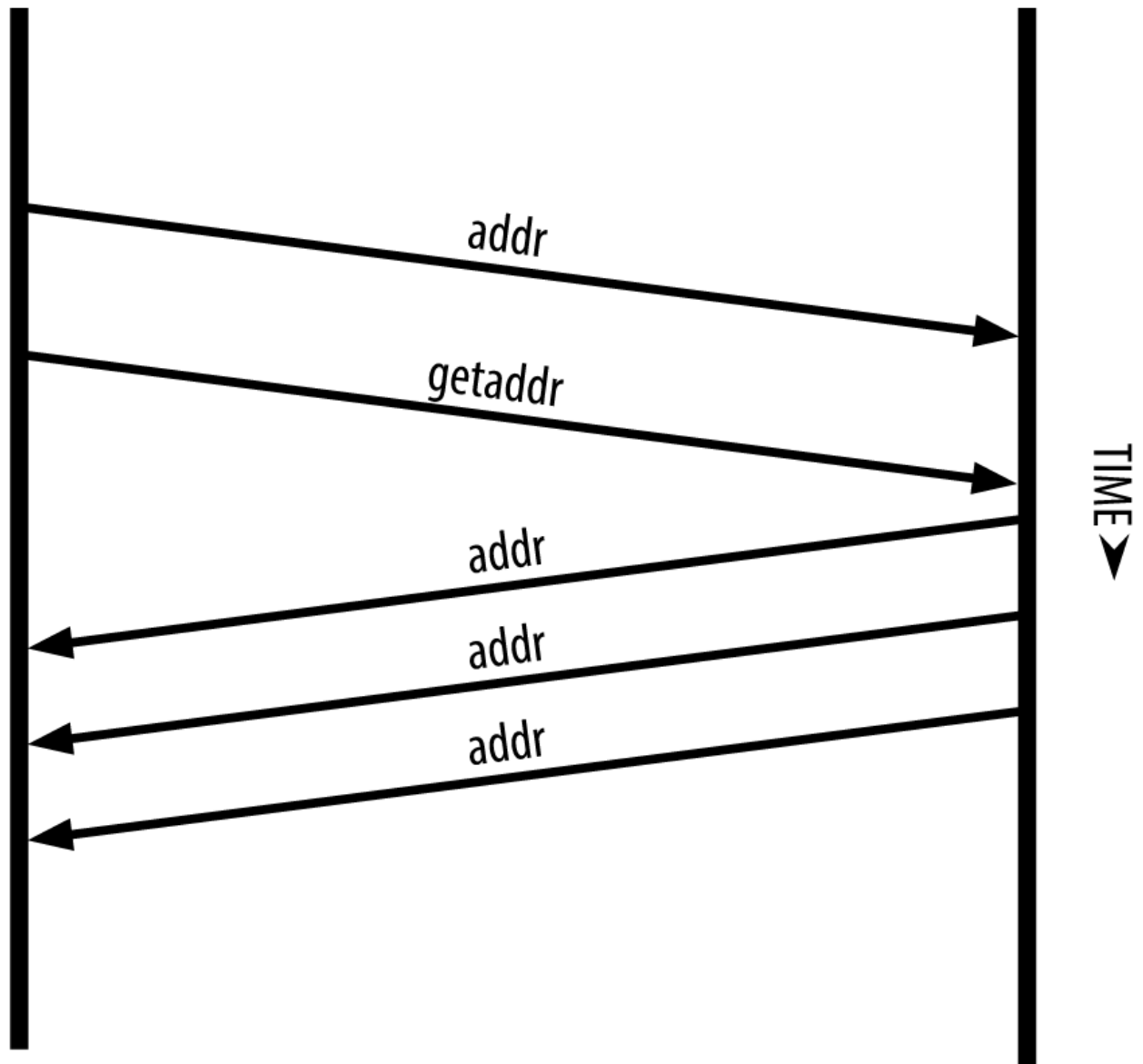


Figure 43. Address propagation and discovery

A node must connect to a few different peers in order to establish diverse paths into the bitcoin network. Paths are not reliable—nodes come and go—and so the node must continue to discover new nodes as it loses old connections as well as assist other nodes when they bootstrap. Only one connection is needed to bootstrap, because the first node can offer introductions to its peer nodes and those peers can offer further introductions. It's also unnecessary and wasteful of network resources to connect to more than a handful of nodes. After bootstrapping, a node will remember its most recent successful peer connections, so that if it is rebooted it can quickly reestablish connections with its former peer network. If none of the former peers respond to its connection request, the node can use the seed nodes to bootstrap again.

On a node running the Bitcoin Core client, you can list the peer connections with the command `getpeerinfo`:

```
$ bitcoin-cli getpeerinfo
```

```
[
  {
    "addr" : "85.213.199.39:8333",
    "services" : "00000001",
    "lastsend" : 1405634126,
    "lastrecv" : 1405634127,
    "bytessent" : 23487651,
    "bytesrecv" : 138679099,
    "conntime" : 1405021768,
    "pingtime" : 0.00000000,
    "version" : 70002,
    "subver" : "/Satoshi:0.9.2.1/",
    "inbound" : false,
    "startingheight" : 310131,
    "banscore" : 0,
    "syncnode" : true
  },
  {
    "addr" : "58.23.244.20:8333",
    "services" : "00000001",
    "lastsend" : 1405634127,
    "lastrecv" : 1405634124,
    "bytessent" : 4460918,
    "bytesrecv" : 8903575,
    "conntime" : 1405559628,
    "pingtime" : 0.00000000,
    "version" : 70001,
    "subver" : "/Satoshi:0.8.6/",
    "inbound" : false,
    "startingheight" : 311074,
    "banscore" : 0,
    "syncnode" : false
  }
]
```

To override the automatic management of peers and to specify a list of IP addresses, users can provide the option `-connect=<IPAddress>` and specify one or more IP addresses. If this option is used, the node will only connect to the selected IP addresses, instead of discovering and maintaining the peer connections automatically.

If there is no traffic on a connection, nodes will periodically send a message to maintain the connection. If a node has not communicated on a connection for more than 90 minutes, it is assumed to be disconnected and a new peer will be sought. Thus, the network dynamically adjusts to transient nodes and network problems, and can organically grow and shrink as needed without any central control.

Full Nodes

Full nodes are nodes that maintain a full blockchain with all transactions. More accurately, they probably should be called "full blockchain nodes." In the early years of bitcoin, all nodes were full nodes and currently the Bitcoin Core client is a full blockchain node. In the past two years, however, new forms of bitcoin clients have been introduced that do not maintain a full blockchain but run as lightweight clients. We'll examine these in more detail in the next section.

Full blockchain nodes maintain a complete and up-to-date copy of the bitcoin blockchain with all the transactions, which they independently build and verify, starting with the very first block (genesis block) and building up to the latest known block in the network. A full blockchain node can independently and authoritatively verify any transaction without recourse or reliance on any other node or source of information. The full blockchain node relies on the network to receive updates about new blocks of transactions, which it then verifies and incorporates into its local copy of the blockchain.

Running a full blockchain node gives you the pure bitcoin experience: independent verification of all transactions without the need to rely on, or trust, any other systems. It's easy to tell if you're running a full node because it requires 20+ gigabytes of persistent storage (disk space) to store the full blockchain. If you need a lot of disk and it takes two to three days to sync to the network, you are running a full node. That is the price of complete independence and freedom from central authority.

There are a few alternative implementations of full blockchain bitcoin clients, built using different programming languages and software architectures. However, the most common implementation is the reference client Bitcoin Core, also known as the Satoshi client. More than 90% of the nodes on the bitcoin network run various versions of Bitcoin Core. It is identified as "Satoshi" in the sub-version string sent in the `version` message and shown by the command `getpeerinfo` as we saw earlier; for example, `/Satoshi:0.8.6/`.

Exchanging "Inventory"

The first thing a full node will do once it connects to peers is try to construct a complete blockchain. If it is a brand-new node and has no blockchain at all, it only knows one block, the genesis block, which is statically embedded in the client software. Starting with block #0 (the genesis block), the new node will have to download hundreds of thousands of blocks to synchronize with the network and re-establish the full blockchain.

The process of syncing the blockchain starts with the `version` message, because that contains `BestHeight`, a node's current blockchain height (number of blocks). A node will see the `version` messages from its peers, know how many blocks they each have, and be able to compare to how many blocks it has in its own blockchain. Peered nodes will exchange a `getblocks` message that contains the hash (fingerprint) of the top block on their local blockchain. One of the peers will be able to identify the received hash as belonging to a block that is not at the top, but rather belongs to an older block, thus deducing that its own local blockchain is longer than its peer's.

The peer that has the longer blockchain has more blocks than the other node and can identify which blocks the other node needs in order to "catch up." It will identify the first 500 blocks to

share and transmit their hashes using an `inv` (inventory) message. The node missing these blocks will then retrieve them, by issuing a series of `getdata` messages requesting the full block data and identifying the requested blocks using the hashes from the `inv` message.

Let's assume, for example, that a node only has the genesis block. It will then receive an `inv` message from its peers containing the hashes of the next 500 blocks in the chain. It will start requesting blocks from all of its connected peers, spreading the load and ensuring that it doesn't overwhelm any peer with requests. The node keeps track of how many blocks are "in transit" per peer connection, meaning blocks that it has requested but not received, checking that it does not exceed a limit (`MAX_BLOCKS_IN_TRANSIT_PER_PEER`). This way, if it needs a lot of blocks, it will only request new ones as previous requests are fulfilled, allowing the peers to control the pace of updates and not overwhelming the network. As each block is received, it is added to the blockchain, as we will see in [The Blockchain](#). As the local blockchain is gradually built up, more blocks are requested and received, and the process continues until the node catches up to the rest of the network.

This process of comparing the local blockchain with the peers and retrieving any missing blocks happens any time a node goes offline for any period of time. Whether a node has been offline for a few minutes and is missing a few blocks, or a month and is missing a few thousand blocks, it starts by sending `getblocks`, gets an `inv` response, and starts downloading the missing blocks. [Node synchronizing the blockchain by retrieving blocks from a peer](#) shows the inventory and block propagation protocol.

Simplified Payment Verification (SPV) Nodes

Not all nodes have the ability to store the full blockchain. Many bitcoin clients are designed to run on space- and power-constrained devices, such as smartphones, tablets, or embedded systems. For such devices, a *simplified payment verification* (SPV) method is used to allow them to operate without storing the full blockchain. These types of clients are called SPV clients or lightweight clients. As bitcoin adoption surges, the SPV node is becoming the most common form of bitcoin node, especially for bitcoin wallets.

SPV nodes download only the block headers and do not download the transactions included in each block. The resulting chain of blocks, without transactions, is 1,000 times smaller than the full blockchain. SPV nodes cannot construct a full picture of all the UTXOs that are available for spending because they do not know about all the transactions on the network. SPV nodes verify transactions using a slightly different methodology that relies on peers to provide partial views of relevant parts of the blockchain on demand.

Node A

Node B

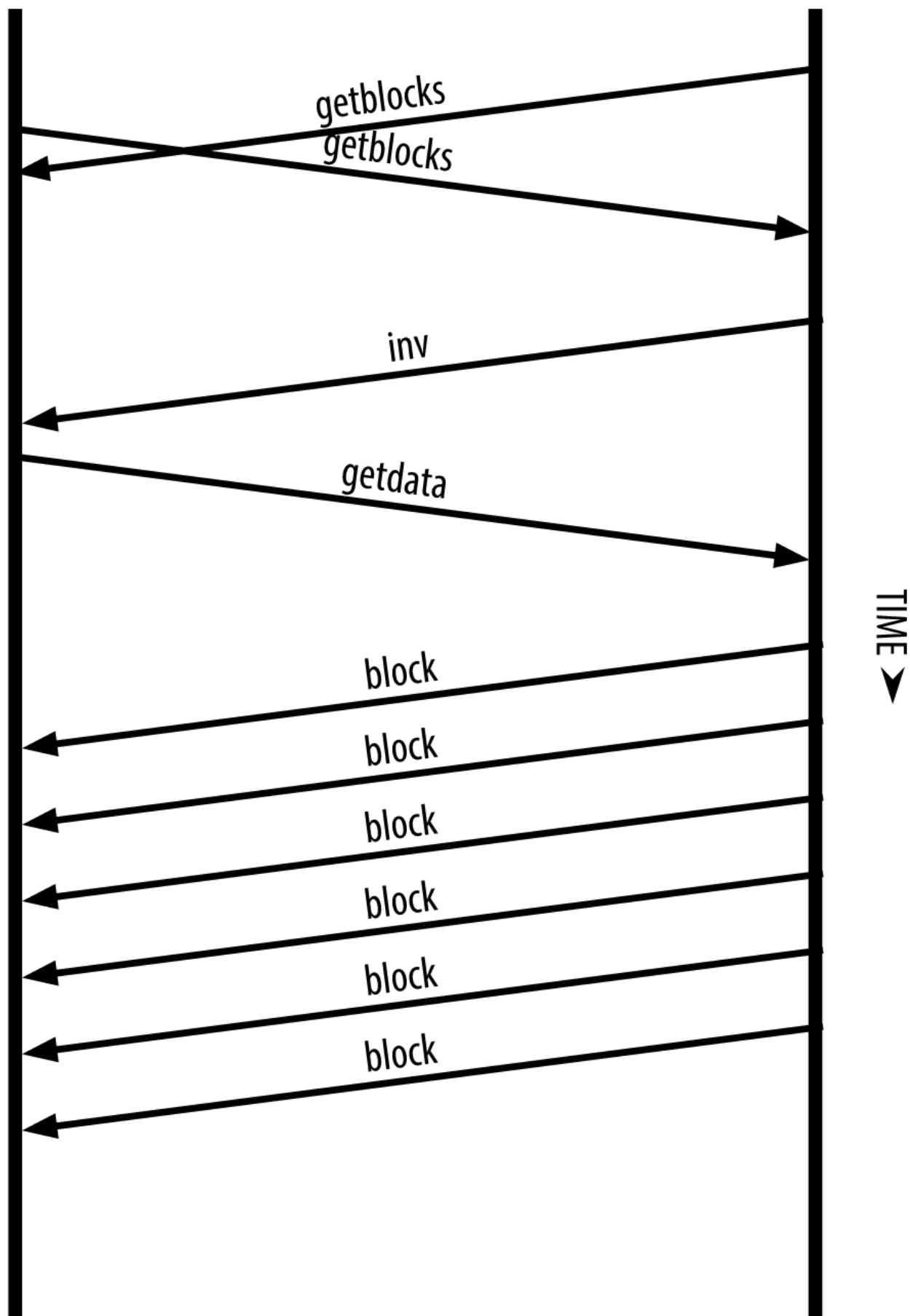


Figure 44. Node synchronizing the blockchain by retrieving blocks from a peer

As an analogy, a full node is like a tourist in a strange city, equipped with a detailed map of every street and every address. By comparison, an SPV node is like a tourist in a strange city asking random strangers for turn-by-turn directions while knowing only one main avenue. Although both tourists can verify the existence of a street by visiting it, the tourist without a map doesn't know what lies down any of the side streets and doesn't know what other streets exist. Positioned in front of 23 Church Street, the tourist without a map cannot know if there are a dozen other "23 Church Street" addresses in the city and whether this is the right one. The mapless tourist's best chance is to ask enough people and hope some of them are not trying to mug him.

Simplified payment verification verifies transactions by reference to their *depth* in the blockchain instead of their *height*. Whereas a full blockchain node will construct a fully verified chain of thousands of blocks and transactions reaching down the blockchain (back in time) all the way to the genesis block, an SPV node will verify the chain of all blocks (but not all transactions) and link that chain to the transaction of interest.

For example, when examining a transaction in block 300,000, a full node links all 300,000 blocks down to the genesis block and builds a full database of UTXO, establishing the validity of the transaction by confirming that the UTXO remains unspent. An SPV node cannot validate whether the UTXO is unspent. Instead, the SPV node will establish a link between the transaction and the block that contains it, using a *merkle path* (see [Merkle Trees](#)). Then, the SPV node waits until it sees the six blocks 300,001 through 300,006 piled on top of the block containing the transaction and verifies it by establishing its depth under blocks 300,006 to 300,001. The fact that other nodes on the network accepted block 300,000 and then did the necessary work to produce six more blocks on top of it is proof, by proxy, that the transaction was not a double-spend.

An SPV node cannot be persuaded that a transaction exists in a block when the transaction does not in fact exist. The SPV node establishes the existence of a transaction in a block by requesting a merkle path proof and by validating the proof of work in the chain of blocks. However, a transaction's existence can be "hidden" from an SPV node. An SPV node can definitely prove that a transaction exists but cannot verify that a transaction, such as a double-spend of the same UTXO, doesn't exist because it doesn't have a record of all transactions. This vulnerability can be used in a denial-of-service attack or for a double-spending attack against SPV nodes. To defend against this, an SPV node needs to connect randomly to several nodes, to increase the probability that it is in contact with at least one honest node. This need to randomly connect means that SPV nodes also are vulnerable to network partitioning attacks or Sybil attacks, where they are connected to fake nodes or fake networks and do not have access to honest nodes or the real bitcoin network.

For most practical purposes, well-connected SPV nodes are secure enough, striking the right balance between resource needs, practicality, and security. For infallible security, however, nothing beats running a full blockchain node.

TIP

A full blockchain node verifies a transaction by checking the entire chain of thousands of blocks below it in order to guarantee that the UTXO is not spent, whereas an SPV node checks how deep the block is buried by a handful of blocks above it.

To get the block headers, SPV nodes use a **getheaders** message instead of **getblocks**. The responding peer will send up to 2,000 block headers using a single **headers** message. The process is otherwise the same as that used by a full node to retrieve full blocks. SPV nodes also set a filter on the connection to peers, to filter the stream of future blocks and transactions sent by the peers. Any

transactions of interest are retrieved using a `getdata` request. The peer generates a `tx` message containing the transactions, in response. [SPV node synchronizing the block headers](#) shows the synchronization of block headers.

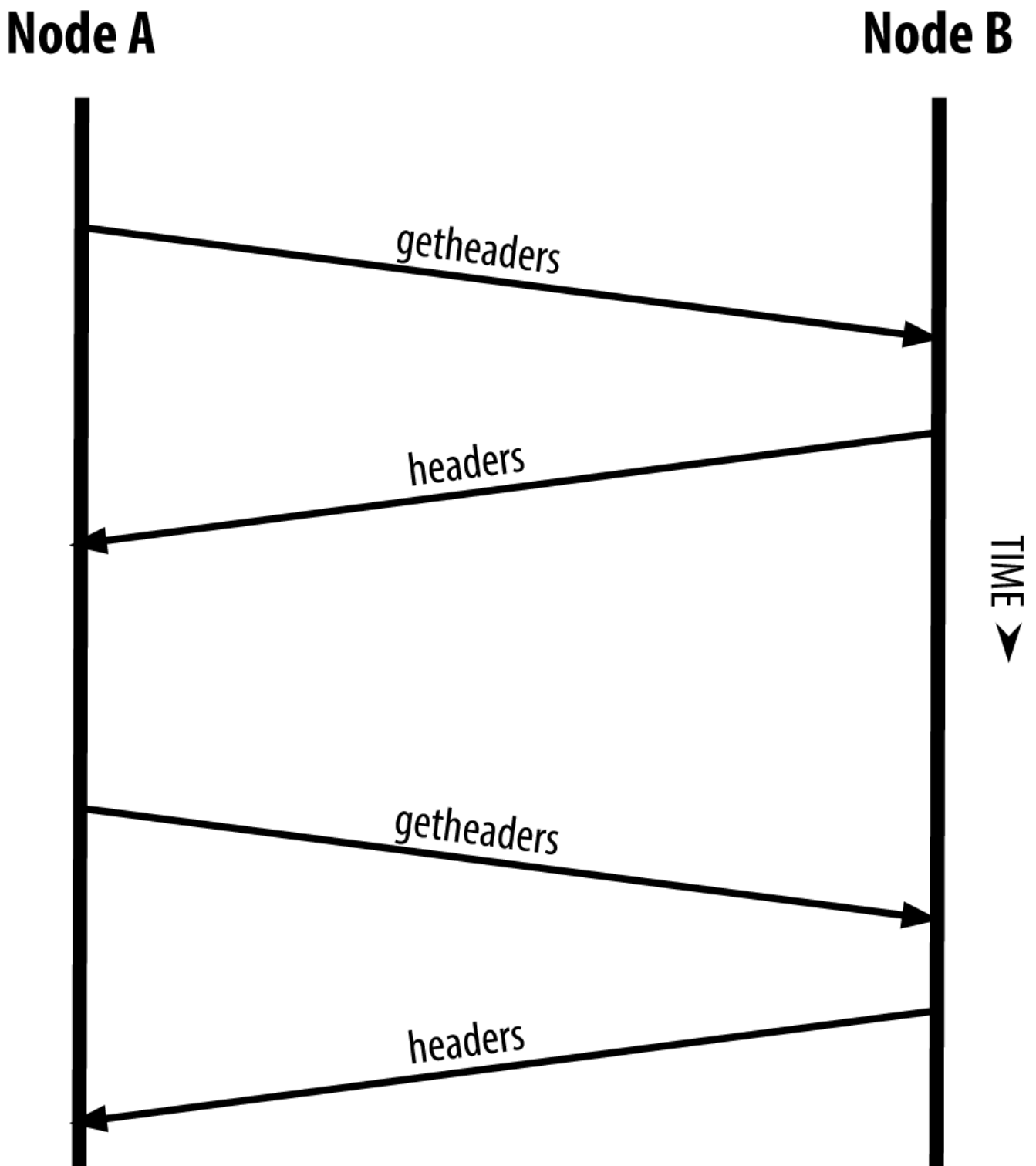


Figure 45. SPV node synchronizing the block headers

Because SPV nodes need to retrieve specific transactions in order to selectively verify them, they also create a privacy risk. Unlike full blockchain nodes, which collect all transactions within each block, the SPV node's requests for specific data can inadvertently reveal the addresses in their wallet. For example, a third party monitoring a network could keep track of all the transactions requested by a wallet on an SPV node and use those to associate bitcoin addresses with the user of that wallet, destroying the user's privacy.

Shortly after the introduction of SPV/lightweight nodes, the bitcoin developers added a feature called *bloom filters* to address the privacy risks of SPV nodes. Bloom filters allow SPV nodes to receive a subset of the transactions without revealing precisely which addresses they are interested in, through a filtering mechanism that uses probabilities rather than fixed patterns.

Bloom Filters

A bloom filter is a probabilistic search filter, a way to describe a desired pattern without specifying it exactly. Bloom filters offer an efficient way to express a search pattern while protecting privacy. They are used by SPV nodes to ask their peers for transactions matching a specific pattern, without revealing exactly which addresses they are searching for.

In our previous analogy, a tourist without a map is asking for directions to a specific address, "23 Church St." If she asks strangers for directions to this street, she inadvertently reveals her destination. A bloom filter is like asking, "Are there any streets in this neighborhood whose name ends in R-C-H?" A question like that reveals slightly less about the desired destination than asking for "23 Church St." Using this technique, a tourist could specify the desired address in more detail as "ending in U-R-C-H" or less detail as "ending in H." By varying the precision of the search, the tourist reveals more or less information, at the expense of getting more or less specific results. If she asks a less specific pattern, she gets a lot more possible addresses and better privacy, but many of the results are irrelevant. If she asks for a very specific pattern, she gets fewer results but loses privacy.

Bloom filters serve this function by allowing an SPV node to specify a search pattern for transactions that can be tuned toward precision or privacy. A more specific bloom filter will produce accurate results, but at the expense of revealing what addresses are used in the user's wallet. A less specific bloom filter will produce more data about more transactions, many irrelevant to the node, but will allow the node to maintain better privacy.

An SPV node will initialize a bloom filter as "empty" and in that state the bloom filter will not match any patterns. The SPV node will then make a list of all the addresses in its wallet and create a search pattern matching the transaction output that corresponds to each address. Usually, the search pattern is a pay-to-public-key-hash script that is the expected locking script that will be present in any transaction paying to the public-key-hash (address). If the SPV node is tracking the balance of a P2SH address, the search pattern will be a pay-to-script-hash script, instead. The SPV node then adds each of the search patterns to the bloom filter, so that the bloom filter can recognize the search pattern if it is present in a transaction. Finally, the bloom filter is sent to the peer and the peer uses it to match transactions for transmission to the SPV node.

Bloom filters are implemented as a variable-size array of N binary digits (a bit field) and a variable number of M hash functions. The hash functions are designed to always produce an output that is between 1 and N , corresponding to the array of binary digits. The hash functions are generated deterministically, so that any node implementing a bloom filter will always use the same hash functions and get the same results for a specific input. By choosing different length (N) bloom filters and a different number (M) of hash functions, the bloom filter can be tuned, varying the level of accuracy and therefore privacy.

In [An example of a simplistic bloom filter, with a 16-bit field and three hash functions](#), we use a very small array of 16 bits and a set of three hash functions to demonstrate how bloom filters work.

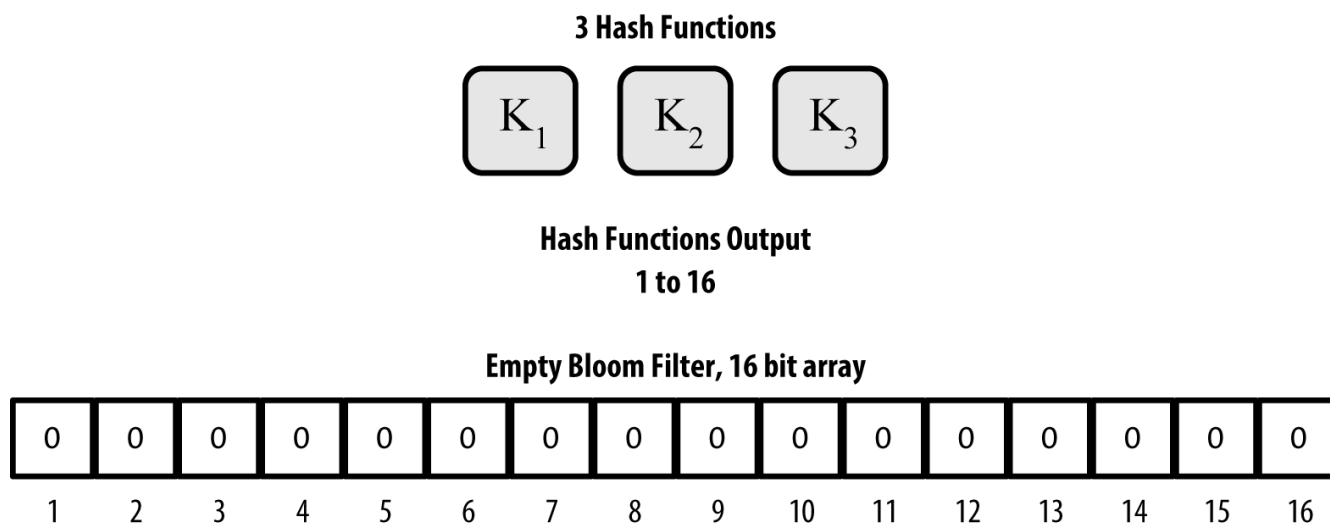


Figure 46. An example of a simplistic bloom filter, with a 16-bit field and three hash functions

The bloom filter is initialized so that the array of bits is all zeros. To add a pattern to the bloom filter, the pattern is hashed by each hash function in turn. Applying the first hash function to the input results in a number between 1 and N. The corresponding bit in the array (indexed from 1 to N) is found and set to **1**, thereby recording the output of the hash function. Then, the next hash function is used to set another bit and so on. Once all M hash functions have been applied, the search pattern will be "recorded" in the bloom filter as M bits that have been changed from 0 to 1.

[Adding a pattern "A" to our simple bloom filter](#) is an example of adding a pattern "A" to the simple bloom filter shown in [An example of a simplistic bloom filter, with a 16-bit field and three hash functions](#).

Adding a second pattern is as simple as repeating this process. The pattern is hashed by each hash function in turn and the result is recorded by setting the bits to **1**. Note that as a bloom filter is filled with more patterns, a hash function result might coincide with a bit that is already set to **1**, in which case the bit is not changed. In essence, as more patterns record on overlapping bits, the bloom filter starts to become saturated with more bits set to **1** and the accuracy of the filter decreases. This is why the filter is a probabilistic data structure—it gets less accurate as more patterns are added. The accuracy depends on the number of patterns added versus the size of the bit array (N) and number of hash functions (M). A larger bit array and more hash functions can record more patterns with higher accuracy. A smaller bit array or fewer hash functions will record fewer patterns and produce less accuracy.

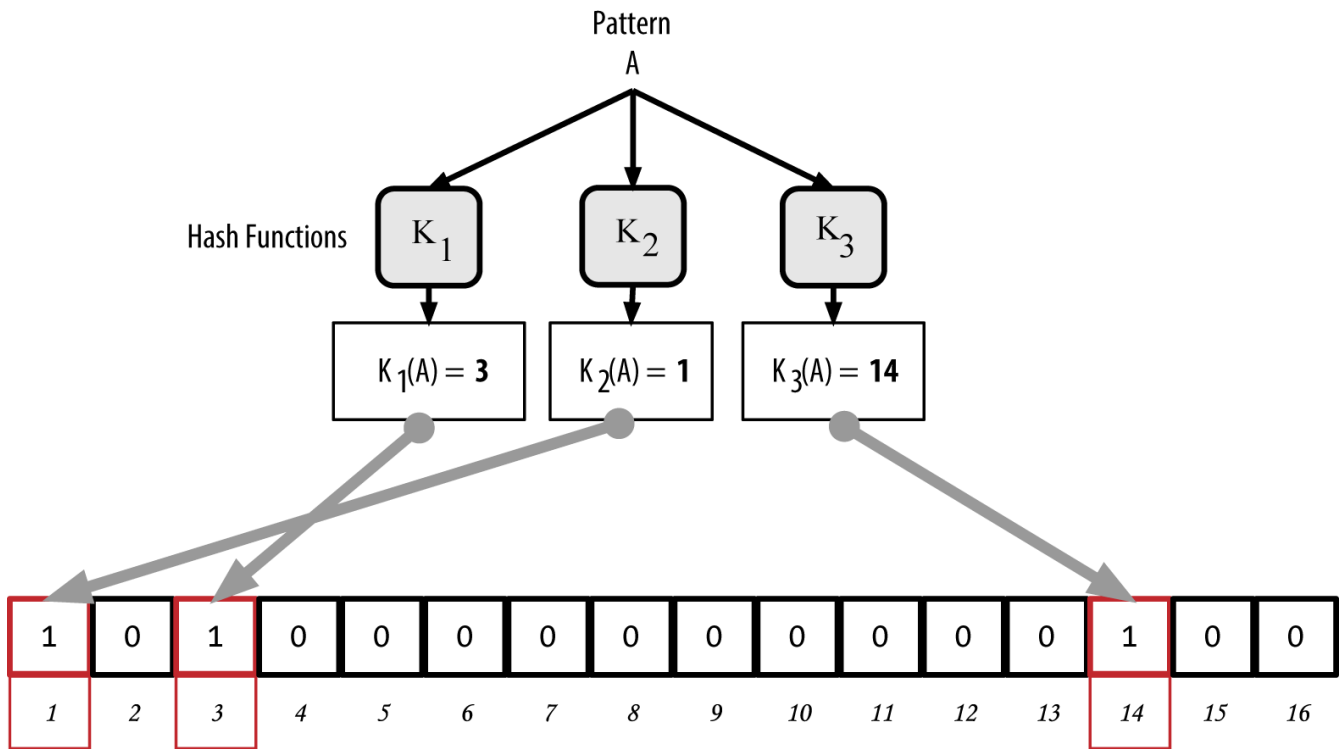


Figure 47. Adding a pattern "A" to our simple bloom filter

Adding a second pattern "B" to our simple bloom filter is an example of adding a second pattern "B" to the simple bloom filter.

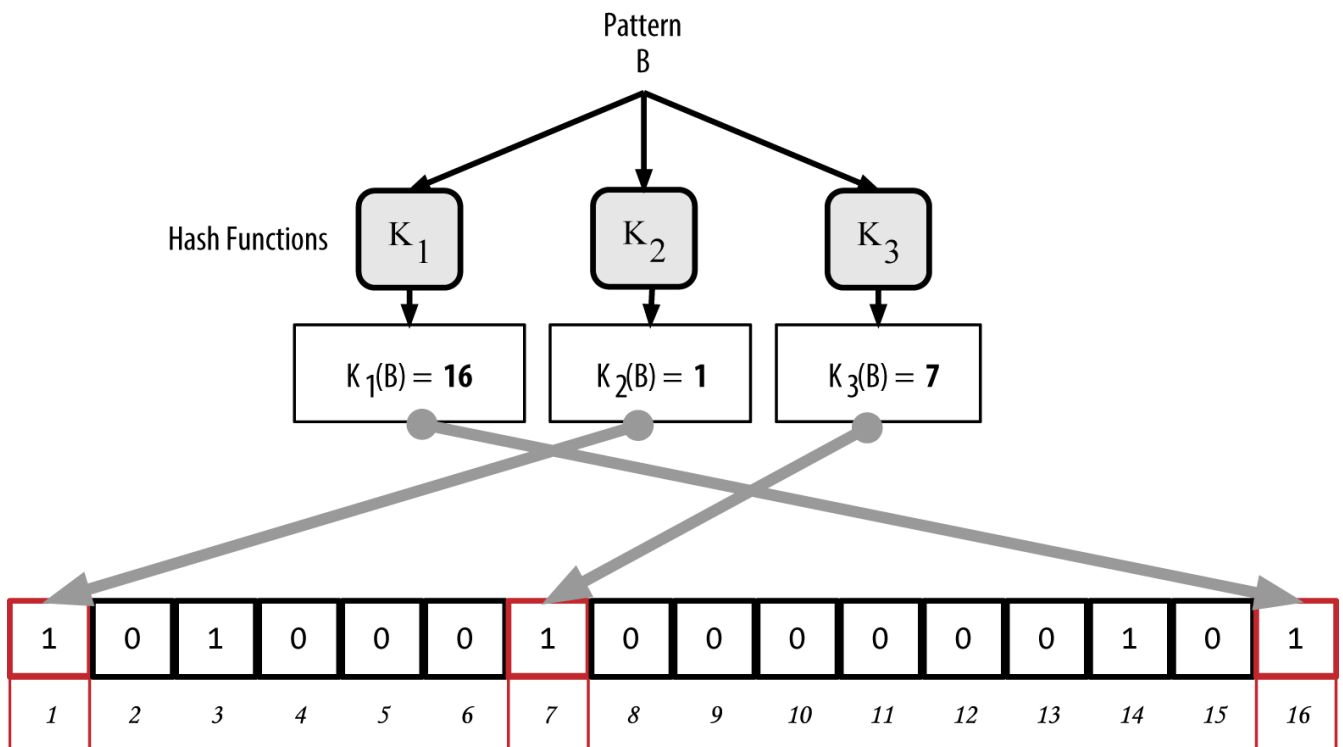


Figure 48. Adding a second pattern "B" to our simple bloom filter

To test if a pattern is part of a bloom filter, the pattern is hashed by each hash function and the resulting bit pattern is tested against the bit array. If all the bits indexed by the hash functions are set to 1, then the pattern is *probably* recorded in the bloom filter. Because the bits may be set because of overlap from multiple patterns, the answer is not certain, but is rather probabilistic. In simple terms, a bloom filter positive match is a "Maybe, Yes."

Testing the existence of pattern "X" in the bloom filter. The result is probabilistic positive match, meaning "Maybe." is an example of testing the existence of pattern "X" in the simple bloom filter. The corresponding bits are set to 1, so the pattern is probably a match.

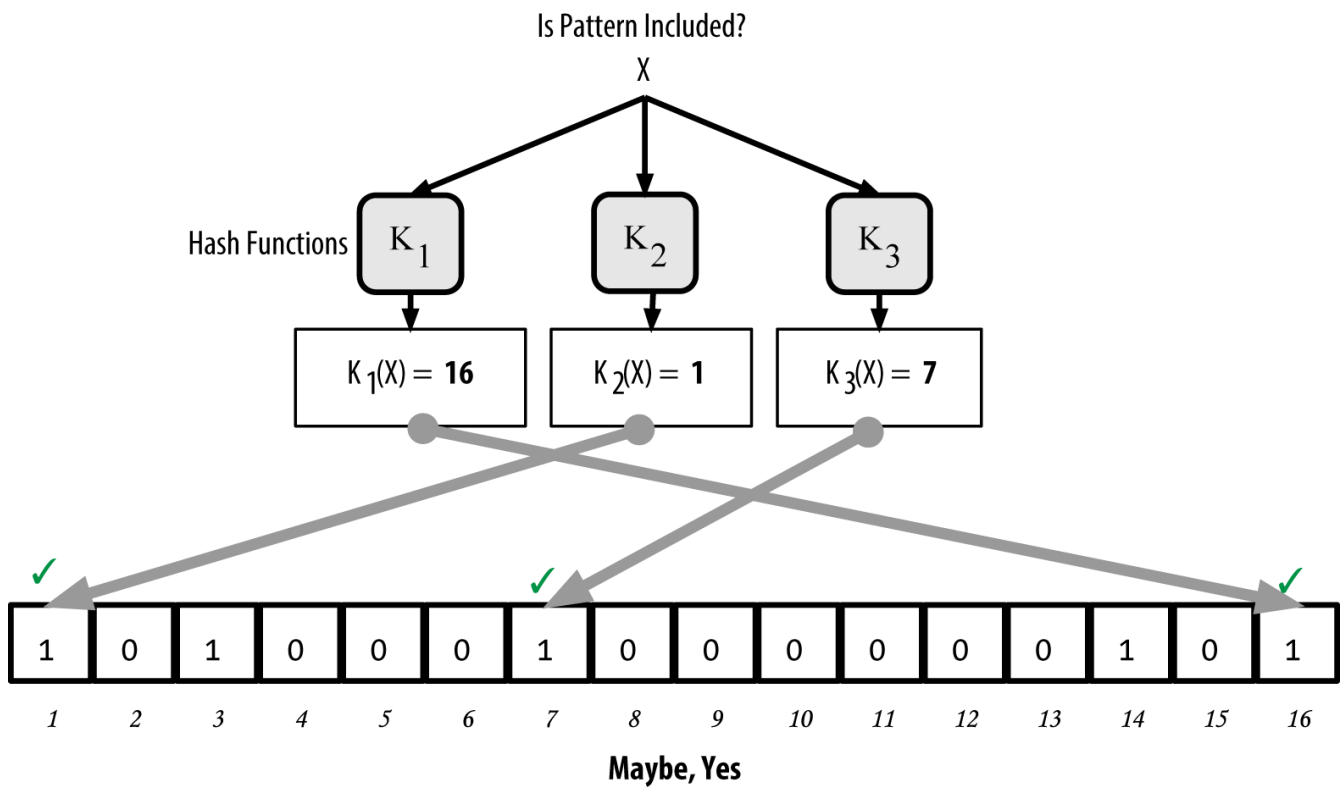


Figure 49. Testing the existence of pattern "X" in the bloom filter. The result is probabilistic positive match, meaning "Maybe."

On the contrary, if a pattern is tested against the bloom filter and any one of the bits is set to 0, this proves that the pattern was not recorded in the bloom filter. A negative result is not a probability, it is a certainty. In simple terms, a negative match on a bloom filter is a "Definitely Not!"

Testing the existence of pattern "Y" in the bloom filter. The result is a definitive negative match, meaning "Definitely Not!" is an example of testing the existence of pattern "Y" in the simple bloom filter. One of the corresponding bits is set to 0, so the pattern is definitely not a match.

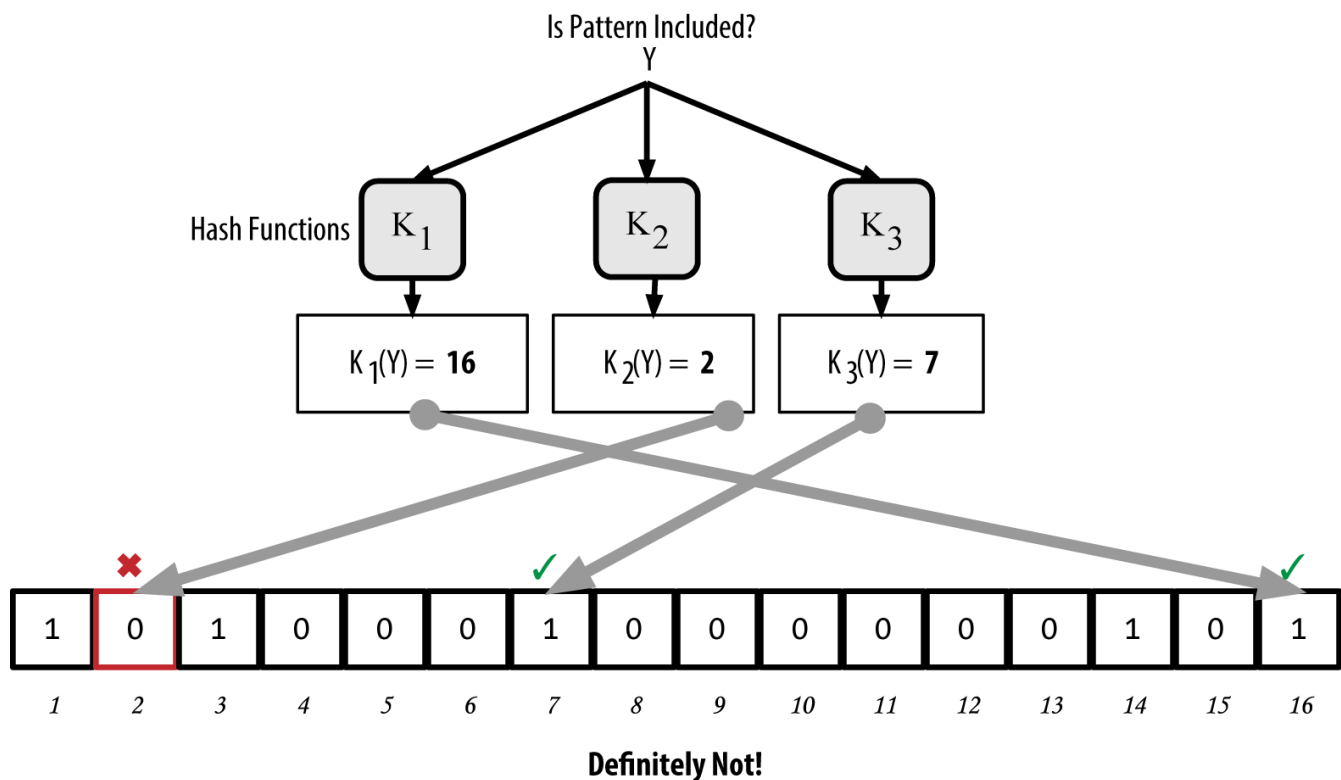


Figure 50. Testing the existence of pattern "Y" in the bloom filter. The result is a definitive negative match, meaning "Definitely Not!"

Bitcoin's implementation of bloom filters is described in Bitcoin Improvement Proposal 37 (BIP0037). See [Bitcoin Improvement Proposals](#) or visit [GitHub](#).

Bloom Filters and Inventory Updates

Bloom filters are used to filter the transactions (and blocks containing them) that an SPV node receives from its peers. SPV nodes will create a filter that matches only the addresses held in the SPV node's wallet. The SPV node will then send a **filterload** message to the peer, containing the bloom filter to use on the connection. After a filter is established, the peer will then test each transaction's outputs against the bloom filter. Only transactions that match the filter are sent to the node.

In response to a **getdata** message from the node, peers will send a **merkleblock** message that contains only block headers for blocks matching the filter and a merkle path (see [Merkle Trees](#)) for each matching transaction. The peer will then also send **tx** messages containing the transactions matched by the filter.

The node setting the bloom filter can interactively add patterns to the filter by sending a **filteradd** message. To clear the bloom filter, the node can send a **filterclear** message. Because it is not possible to remove a pattern from a bloom filter, a node has to clear and resend a new bloom filter if a pattern is no longer desired.

Transaction Pools

Almost every node on the bitcoin network maintains a temporary list of unconfirmed transactions called the *memory pool*, *mempool*, or *transaction pool*. Nodes use this pool to keep track of

transactions that are known to the network but are not yet included in the blockchain. For example, a node that holds a user's wallet will use the transaction pool to track incoming payments to the user's wallet that have been received on the network but are not yet confirmed.

As transactions are received and verified, they are added to the transaction pool and relayed to the neighboring nodes to propagate on the network.

Some node implementations also maintain a separate pool of orphaned transactions. If a transaction's inputs refer to a transaction that is not yet known, such as a missing parent, the orphan transaction will be stored temporarily in the orphan pool until the parent transaction arrives.

When a transaction is added to the transaction pool, the orphan pool is checked for any orphans that reference this transaction's outputs (its children). Any matching orphans are then validated. If valid, they are removed from the orphan pool and added to the transaction pool, completing the chain that started with the parent transaction. In light of the newly added transaction, which is no longer an orphan, the process is repeated recursively looking for any further descendants, until no more descendants are found. Through this process, the arrival of a parent transaction triggers a cascade reconstruction of an entire chain of interdependent transactions by re-uniting the orphans with their parents all the way down the chain.

Both the transaction pool and orphan pool (where implemented) are stored in local memory and are not saved on persistent storage; rather, they are dynamically populated from incoming network messages. When a node starts, both pools are empty and are gradually populated with new transactions received on the network.

Some implementations of the bitcoin client also maintain a UTXO database or UTXO pool, which is the set of all unspent outputs on the blockchain. Although the name "UTXO pool" sounds similar to the transaction pool, it represents a different set of data. Unlike the transaction and orphan pools, the UTXO pool is not initialized empty but instead contains millions of entries of unspent transaction outputs, including some dating back to 2009. The UTXO pool may be housed in local memory or as an indexed database table on persistent storage.

Whereas the transaction and orphan pools represent a single node's local perspective and might vary significantly from node to node depending upon when the node was started or restarted, the UTXO pool represents the emergent consensus of the network and therefore will vary little between nodes. Furthermore, the transaction and orphan pools only contain unconfirmed transactions, while the UTXO pool only contains confirmed outputs.

Alert Messages

Alert messages are a seldom used function, but are nevertheless implemented in most nodes. Alert messages are bitcoin's "emergency broadcast system," a means by which the core bitcoin developers can send an emergency text message to all bitcoin nodes. This feature is implemented to allow the core developer team to notify all bitcoin users of a serious problem in the bitcoin network, such as a critical bug that requires user action. The alert system has only been used a handful of times, most notably in early 2013 when a critical database bug caused a multiblock fork to occur in the bitcoin blockchain.

Alert messages are propagated by the `alert` message. The alert message contains several fields, including:

ID

An alert identified so that duplicate alerts can be detected

Expiration

A time after which the alert expires

RelayUntil

A time after which the alert should not be relayed

MinVer, MaxVer

The range of bitcoin protocol versions that this alert applies to

subVer

The client software version that this alert applies to

Priority

An alert priority level, currently unused

Alerts are cryptographically signed by a public key. The corresponding private key is held by a few select members of the core development team. The digital signature ensures that fake alerts will not be propagated on the network.

Each node receiving this alert message will verify it, check for expiration, and propagate it to all its peers, thus ensuring rapid propagation across the entire network. In addition to propagating the alert, the nodes might implement a user interface function to present the alert to the user.

In the Bitcoin Core client, the alert is configured with the command-line option `-alertnotify`, which specifies a command to run when an alert is received. The alert message is passed as a parameter to the `alertnotify` command. Most commonly, the `alertnotify` command is set to generate an email message to the administrator of the node, containing the alert message. The alert is also displayed as a pop-up dialog in the graphical user interface (bitcoin-Qt) if it is running.

Other implementations of the bitcoin protocol might handle the alert in different ways. Many hardware-embedded bitcoin mining systems do not implement the alert message function because they have no user interface. It is strongly recommended that miners running such mining systems subscribe to alerts via a mining pool operator or by running a lightweight node just for alert purposes.

The Blockchain

Introduction

The blockchain data structure is an ordered, back-linked list of blocks of transactions. The blockchain can be stored as a flat file, or in a simple database. The Bitcoin Core client stores the blockchain metadata using Google's LevelDB database. Blocks are linked "back," each referring to the previous block in the chain. The blockchain is often visualized as a vertical stack, with blocks layered on top of each other and the first block serving as the foundation of the stack. The visualization of blocks stacked on top of each other results in the use of terms such as "height" to refer to the distance from the first block, and "top" or "tip" to refer to the most recently added block.

Each block within the blockchain is identified by a hash, generated using the SHA256 cryptographic hash algorithm on the header of the block. Each block also references a previous block, known as the *parent* block, through the "previous block hash" field in the block header. In other words, each block contains the hash of its parent inside its own header. The sequence of hashes linking each block to its parent creates a chain going back all the way to the first block ever created, known as the *genesis block*.

Although a block has just one parent, it can temporarily have multiple children. Each of the children refers to the same block as its parent and contains the same (parent) hash in the "previous block hash" field. Multiple children arise during a blockchain "fork," a temporary situation that occurs when different blocks are discovered almost simultaneously by different miners (see [Blockchain Forks](#)). Eventually, only one child block becomes part of the blockchain and the "fork" is resolved. Even though a block may have more than one child, each block can have only one parent. This is because a block has one single "previous block hash" field referencing its single parent.

The "previous block hash" field is inside the block header and thereby affects the *current* block's hash. The child's own identity changes if the parent's identity changes. When the parent is modified in any way, the parent's hash changes. The parent's changed hash necessitates a change in the "previous block hash" pointer of the child. This in turn causes the child's hash to change, which requires a change in the pointer of the grandchild, which in turn changes the grandchild, and so on. This cascade effect ensures that once a block has many generations following it, it cannot be changed without forcing a recalculation of all subsequent blocks. Because such a recalculation would require enormous computation, the existence of a long chain of blocks makes the blockchain's deep history immutable, which is a key feature of bitcoin's security.

One way to think about the blockchain is like layers in a geological formation, or glacier core sample. The surface layers might change with the seasons, or even be blown away before they have time to settle. But once you go a few inches deep, geological layers become more and more stable. By the time you look a few hundred feet down, you are looking at a snapshot of the past that has remained undisturbed for millions of years. In the blockchain, the most recent few blocks might be revised if there is a chain recalculation due to a fork. The top six blocks are like a few inches of topsoil. But once you go more deeply into the blockchain, beyond six blocks, blocks are less and less likely to change. After 100 blocks back there is so much stability that the coinbase transaction—the transaction containing newly mined bitcoins—can be spent. A few thousand blocks back (a month) and the blockchain is settled history, for all practical purposes. While the protocol always allows a

chain to be undone by a longer chain and while the possibility of any block being reversed always exists, the probability of such an event decreases as time passes until it becomes infinitesimal.

Structure of a Block

A block is a container data structure that aggregates transactions for inclusion in the public ledger, the blockchain. The block is made of a header, containing metadata, followed by a long list of transactions that make up the bulk of its size. The block header is 80 bytes, whereas the average transaction is at least 250 bytes and the average block contains more than 500 transactions. A complete block, with all transactions, is therefore 1,000 times larger than the block header. [The structure of a block](#) describes the structure of a block.

Table 20. The structure of a block

Size	Field	Description
4 bytes	Block Size	The size of the block, in bytes, following this field
80 bytes	Block Header	Several fields form the block header
1-9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	The transactions recorded in this block

Block Header

The block header consists of three sets of block metadata. First, there is a reference to a previous block hash, which connects this block to the previous block in the blockchain. The second set of metadata, namely the *difficulty*, *timestamp*, and *nonce*, relate to the mining competition, as detailed in [Mining and Consensus](#). The third piece of metadata is the merkle tree root, a data structure used to efficiently summarize all the transactions in the block. [The structure of the block header](#) describes the structure of a block header.

Table 21. The structure of the block header

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)

Size	Field	Description
4 bytes	Difficulty Target	The proof-of-work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the proof-of-work algorithm

The nonce, difficulty target, and timestamp are used in the mining process and will be discussed in more detail in [Mining and Consensus](#).

Block Identifiers: Block Header Hash and Block Height

The primary identifier of a block is its cryptographic hash, a digital fingerprint, made by hashing the block header twice through the SHA256 algorithm. The resulting 32-byte hash is called the *block hash* but is more accurately the *block header hash*, because only the block header is used to compute it. For example, `00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f` is the block hash of the first bitcoin block ever created. The block hash identifies a block uniquely and unambiguously and can be independently derived by any node by simply hashing the block header.

Note that the block hash is not actually included inside the block's data structure, neither when the block is transmitted on the network, nor when it is stored on a node's persistence storage as part of the blockchain. Instead, the block's hash is computed by each node as the block is received from the network. The block hash might be stored in a separate database table as part of the block's metadata, to facilitate indexing and faster retrieval of blocks from disk.

A second way to identify a block is by its position in the blockchain, called the *block height*. The first block ever created is at block height 0 (zero) and is the same block that was previously referenced by the following block hash `00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f`. A block can thus be identified two ways: by referencing the block hash or by referencing the block height. Each subsequent block added "on top" of that first block is one position "higher" in the blockchain, like boxes stacked one on top of the other. The block height on January 1, 2014, was approximately 278,000, meaning there were 278,000 blocks stacked on top of the first block created in January 2009.

Unlike the block hash, the block height is not a unique identifier. Although a single block will always have a specific and invariant block height, the reverse is not true—the block height does not always identify a single block. Two or more blocks might have the same block height, competing for the same position in the blockchain. This scenario is discussed in detail in the section [Blockchain Forks](#). The block height is also not a part of the block's data structure; it is not stored within the block. Each node dynamically identifies a block's position (height) in the blockchain when it is received from the bitcoin network. The block height might also be stored as metadata in an indexed database table for faster retrieval.

TIP

A block's *block hash* always identifies a single block uniquely. A block also always has a specific *block height*. However, it is not always the case that a specific block height can identify a single block. Rather, two or more blocks might compete for a single position in the blockchain.

The Genesis Block

The first block in the blockchain is called the genesis block and was created in 2009. It is the common ancestor of all the blocks in the blockchain, meaning that if you start at any block and follow the chain backward in time, you will eventually arrive at the genesis block.

Every node always starts with a blockchain of at least one block because the genesis block is statically encoded within the bitcoin client software, such that it cannot be altered. Every node always "knows" the genesis block's hash and structure, the fixed time it was created, and even the single transaction within. Thus, every node has the starting point for the blockchain, a secure "root" from which to build a trusted blockchain.

See the statically encoded genesis block inside the Bitcoin Core client, in [chainparams.cpp](#).

The following identifier hash belongs to the genesis block:

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

You can search for that block hash in any block explorer website, such as [blockchain.info](#), and you will find a page describing the contents of this block, with a URL containing that hash:

<https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

<https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

Using the Bitcoin Core reference client on the command line:

```
$ bitcoin-cli getblock  
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

```
{
  "hash" : "00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "confirmations" : 308321,
  "size" : 285,
  "height" : 0,
  "version" : 1,
  "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
  "tx" : [
    "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
  ],
  "time" : 1231006505,
  "nonce" : 2083236893,
  "bits" : "1d00ffff",
  "difficulty" : 1.00000000,
  "nextblockhash" :
    "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

The genesis block contains a hidden message within it. The coinbase transaction input contains the text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks." This message was intended to offer proof of the earliest date this block was created, by referencing the headline of the British newspaper *The Times*. It also serves as a tongue-in-cheek reminder of the importance of an independent monetary system, with bitcoin's launch occurring at the same time as an unprecedented worldwide monetary crisis. The message was embedded in the first block by Satoshi Nakamoto, bitcoin's creator.

Linking Blocks in the Blockchain

Bitcoin full nodes maintain a local copy of the blockchain, starting at the genesis block. The local copy of the blockchain is constantly updated as new blocks are found and used to extend the chain. As a node receives incoming blocks from the network, it will validate these blocks and then link them to the existing blockchain. To establish a link, a node will examine the incoming block header and look for the "previous block hash."

Let's assume, for example, that a node has 277,314 blocks in the local copy of the blockchain. The last block the node knows about is block 277,314, with a block header hash of `0000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249`.

The bitcoin node then receives a new block from the network, which it parses as follows:

```

{
  "size" : 43560,
  "version" : 2,
  "previousblockhash" :
    "0000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
  "merkleroot" :
    "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
  "time" : 1388185038,
  "difficulty" : 1180923195.25802612,
  "nonce" : 4215469401,
  "tx" : [
    "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

    #[... many more transactions omitted ...]

    "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
  ]
}

```

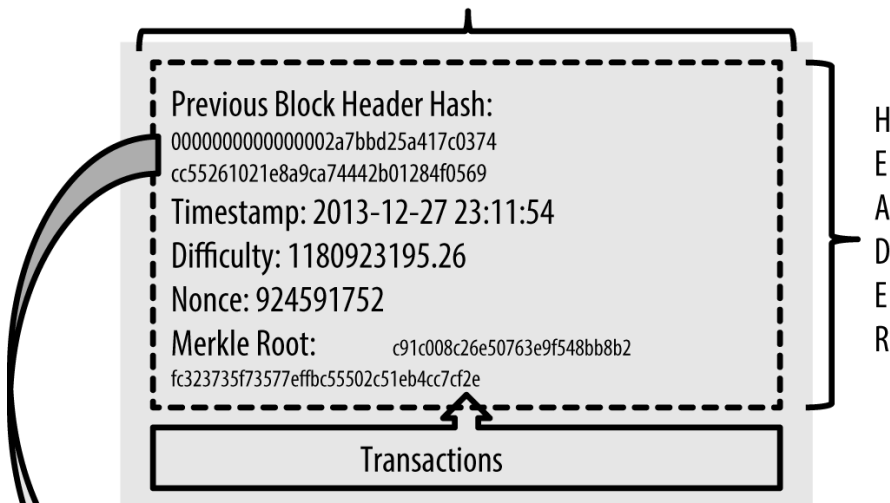
Looking at this new block, the node finds the `previousblockhash` field, which contains the hash of its parent block. It is a hash known to the node, that of the last block on the chain at height 277,314. Therefore, this new block is a child of the last block on the chain and extends the existing blockchain. The node adds this new block to the end of the chain, making the blockchain longer with a new height of 277,315. [Blocks linked in a chain, by reference to the previous block header hash](#) shows the chain of three blocks, linked by references in the `previousblockhash` field.

Merkle Trees

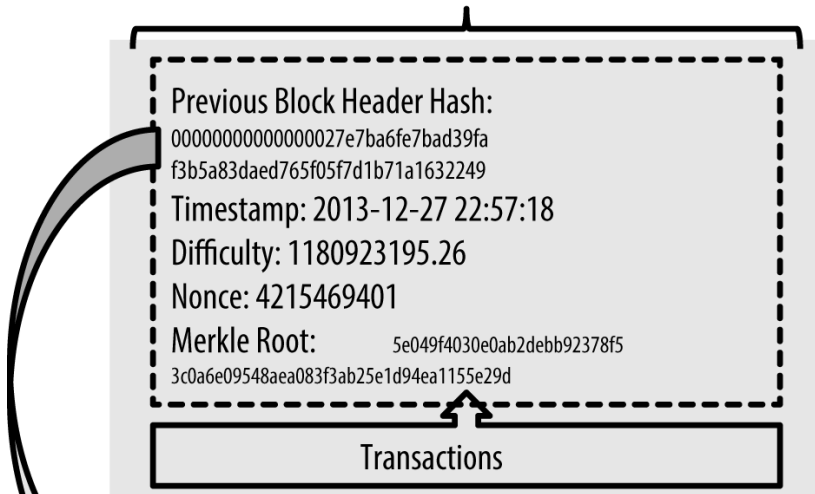
Each block in the bitcoin blockchain contains a summary of all the transactions in the block, using a *merkle tree*.

A *merkle tree*, also known as a *binary hash tree*, is a data structure used for efficiently summarizing and verifying the integrity of large sets of data. Merkle trees are binary trees containing cryptographic hashes. The term "tree" is used in computer science to describe a branching data structure, but these trees are usually displayed upside down with the "root" at the top and the "leaves" at the bottom of a diagram, as you will see in the examples that follow.

Block Height 277316
Header Hash:
000000000000001b6b9a13b095e96db
41c4a928b97ef2d944a9b31b2cc7bdc4



Block Height 277315
Header Hash:
000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569



Block Height 277314
Header Hash:
0000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249

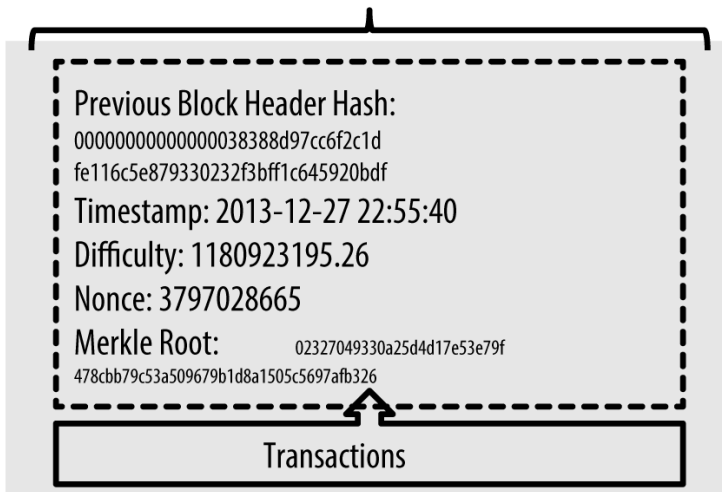


Figure 51. Blocks linked in a chain, by reference to the previous block header hash

Merkle trees are used in bitcoin to summarize all the transactions in a block, producing an overall digital fingerprint of the entire set of transactions, providing a very efficient process to verify whether a transaction is included in a block. A Merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the *root*, or *merkle root*. The cryptographic hash algorithm used in bitcoin's merkle trees is SHA256 applied twice, also known as double-SHA256.

When N data elements are hashed and summarized in a merkle tree, you can check to see if any one data element is included in the tree with at most $2 \cdot \log_2(N)$ calculations, making this a very efficient data structure.

The merkle tree is constructed bottom-up. In the following example, we start with four transactions, A, B, C and D, which form the *leaves* of the Merkle tree, as shown in [Calculating the nodes in a merkle tree](#). The transactions are not stored in the merkle tree; rather, their data is hashed and the resulting hash is stored in each leaf node as H_A , H_B , H_C , and H_D :

$$H_{A\sim} = \text{SHA256}(\text{SHA256}(\text{Transaction A}))$$

Consecutive pairs of leaf nodes are then summarized in a parent node, by concatenating the two hashes and hashing them together. For example, to construct the parent node H_{AB} , the two 32-byte hashes of the children are concatenated to create a 64-byte string. That string is then double-hashed to produce the parent node's hash:

$$H_{AB\sim} = \text{SHA256}(\text{SHA256}(H_{A\sim} + H_{B\sim}))$$

The process continues until there is only one node at the top, the node known as the Merkle root. That 32-byte hash is stored in the block header and summarizes all the data in all four transactions.

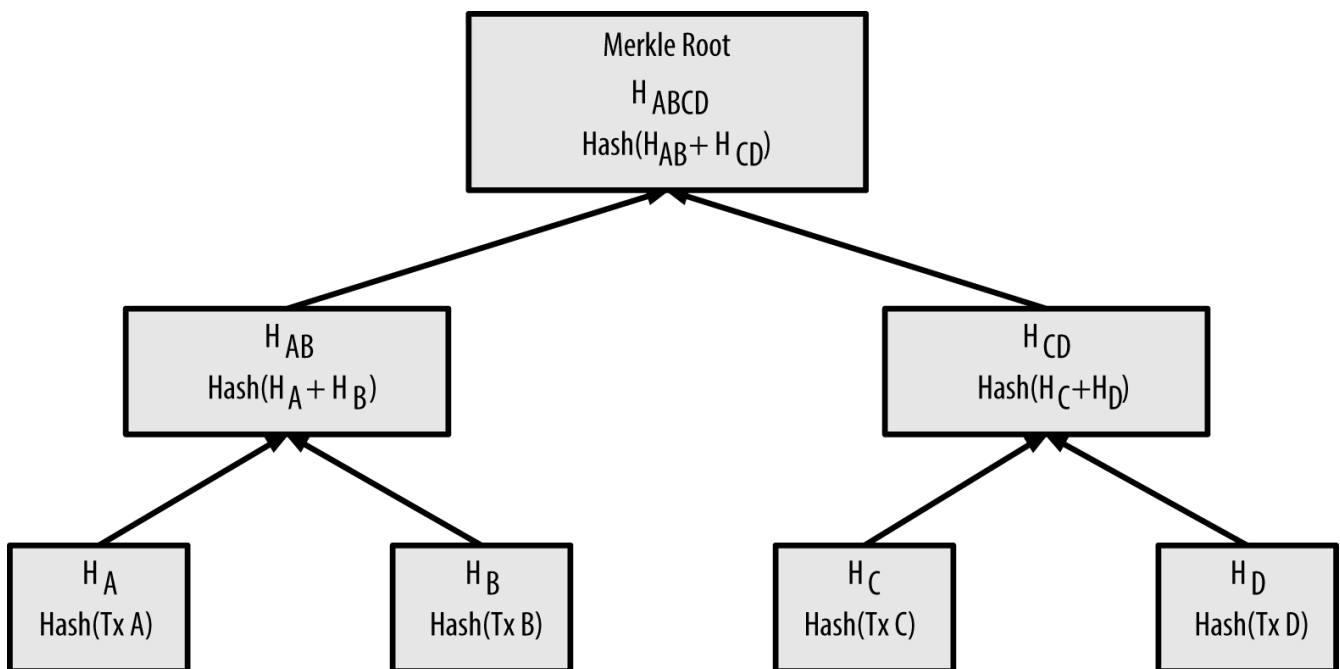


Figure 52. Calculating the nodes in a merkle tree

Because the merkle tree is a binary tree, it needs an even number of leaf nodes. If there is an odd number of transactions to summarize, the last transaction hash will be duplicated to create an even number of leaf nodes, also known as a *balanced tree*. This is shown in [Duplicating one data element achieves an even number of data elements](#), where transaction C is duplicated.

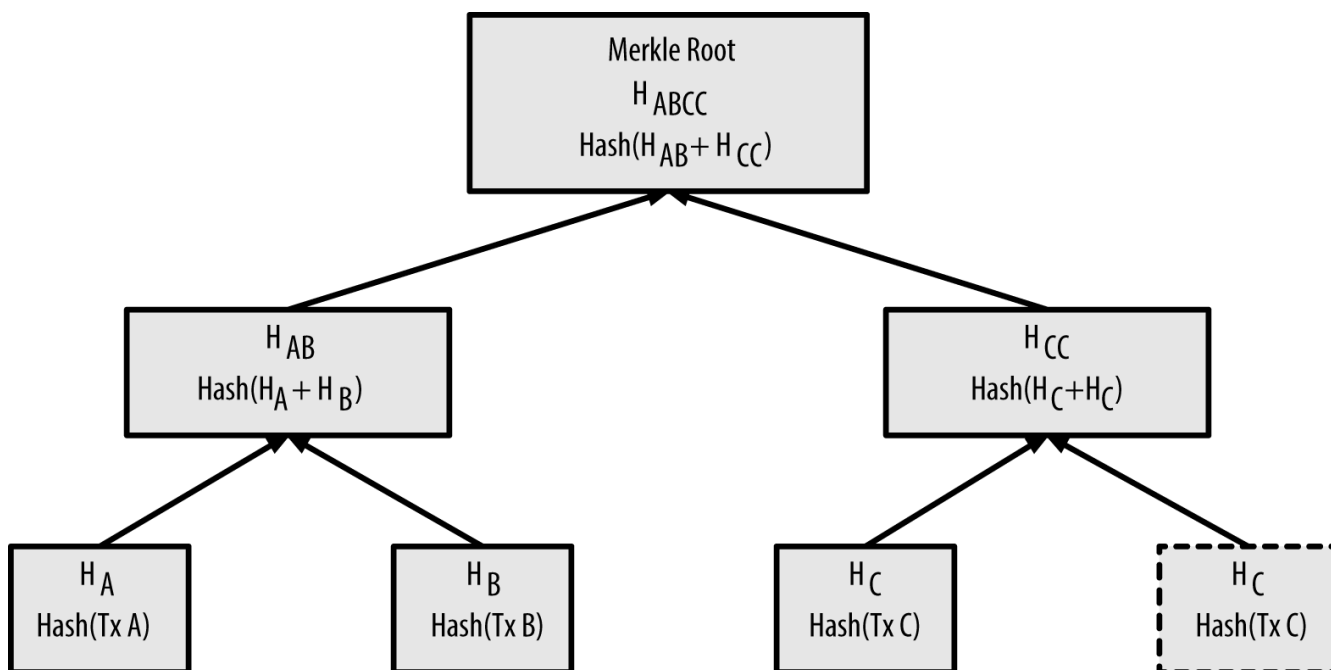


Figure 53. Duplicating one data element achieves an even number of data elements

The same method for constructing a tree from four transactions can be generalized to construct trees of any size. In bitcoin it is common to have several hundred to more than a thousand transactions in a single block, which are summarized in exactly the same way, producing just 32 bytes of data as the single merkle root. In [A merkle tree summarizing many data elements](#), you will see a tree built from 16 transactions. Note that although the root looks bigger than the leaf nodes in the diagram, it is the exact same size, just 32 bytes. Whether there is one transaction or a hundred thousand transactions in the block, the merkle root always summarizes them into 32 bytes.

To prove that a specific transaction is included in a block, a node only needs to produce $\log_2(N)$ 32-byte hashes, constituting an *authentication path* or *merkle path* connecting the specific transaction to the root of the tree. This is especially important as the number of transactions increases, because the base-2 logarithm of the number of transactions increases much more slowly. This allows bitcoin nodes to efficiently produce paths of 10 or 12 hashes (320–384 bytes), which can provide proof of a single transaction out of more than a thousand transactions in a megabyte-size block.

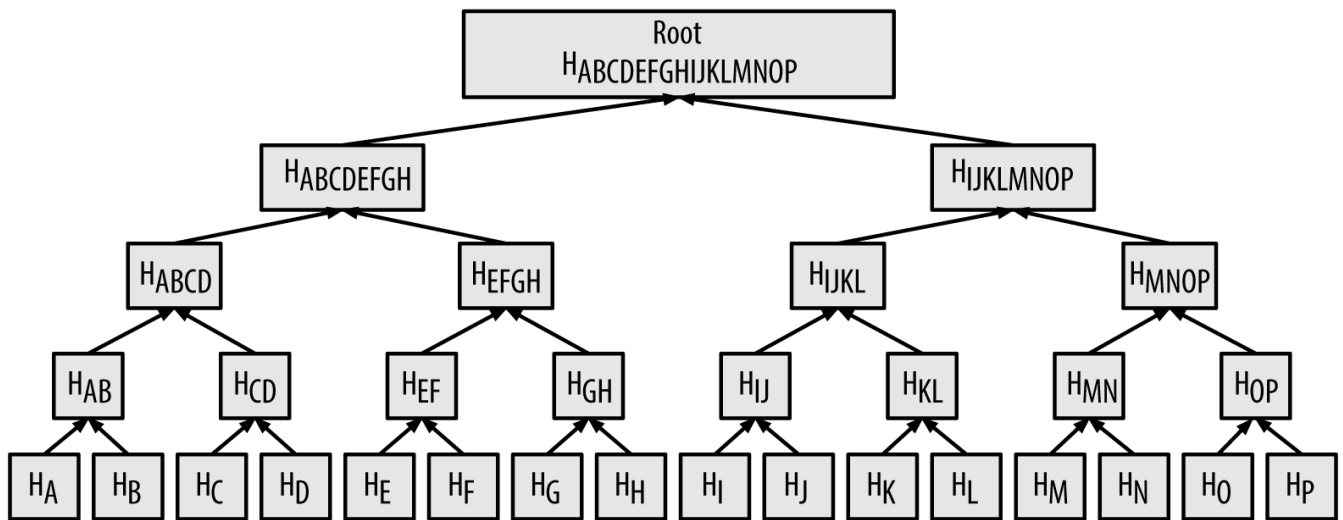


Figure 54. A merkle tree summarizing many data elements

In [A merkle path used to prove inclusion of a data element](#), a node can prove that a transaction K is included in the block by producing a merkle path that is only four 32-byte hashes long (128 bytes total). The path consists of the four hashes (noted in blue in [A merkle path used to prove inclusion of a data element](#)) H_L , H_{IJ} , H_{MNOP} and $H_{ABCDEFGH}$. With those four hashes provided as an authentication path, any node can prove that H_K (noted in green in the diagram) is included in the merkle root by computing four additional pair-wise hashes H_{KL} , H_{IJKL} , $H_{IJKLMNOP}$, and the merkle tree root (outlined in a dotted line in the diagram).

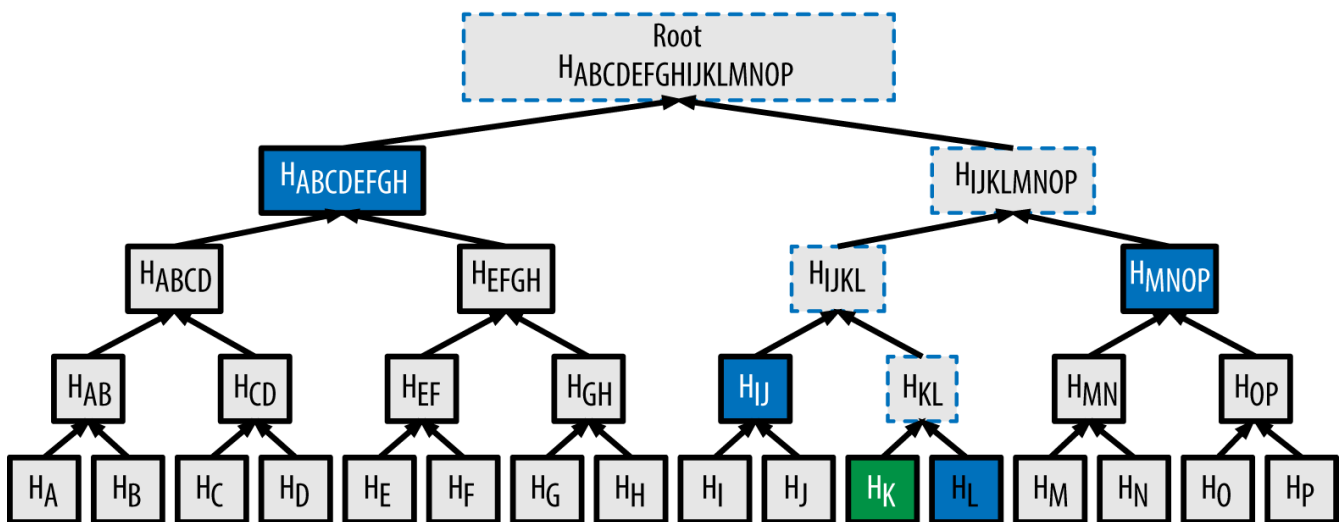


Figure 55. A merkle path used to prove inclusion of a data element

The code in [Building a merkle tree](#) demonstrates the process of creating a merkle tree from the leaf-node hashes up to the root, using the libbitcoin library for some helper functions.

Example 16. Building a merkle tree

```
#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
```



```

bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000011
"),
bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000022
"),
    });
    const bc::hash_digest merkle_root = create_merkle(tx_hashes);
    std::cout << "Result: " << bc::encode_hex(merkle_root) << std::endl;
    return 0;
}

```

Compiling and running the merkle example code shows the result of compiling and running the merkle code.

Example 17. Compiling and running the merkle example code

```

$ # Compile the merkle.cpp code
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the merkle executable
$ ./merkle
Current merkle hash list:
  32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
  30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Current merkle hash list:
  d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

```

The efficiency of merkle trees becomes obvious as the scale increases. [Merkle tree efficiency](#) shows the amount of data that needs to be exchanged as a merkle path to prove that a transaction is part of a block.

Table 22. Merkle tree efficiency

Number of transactions	Approx. size of block	Path size (hashes)	Path size (bytes)
16 transactions	4 kilobytes	4 hashes	128 bytes
512 transactions	128 kilobytes	9 hashes	288 bytes
2048 transactions	512 kilobytes	11 hashes	352 bytes
65,535 transactions	16 megabytes	16 hashes	512 bytes

As you can see from the table, while the block size increases rapidly, from 4 KB with 16 transactions to a block size of 16 MB to fit 65,535 transactions, the merkle path required to prove the inclusion of

a transaction increases much more slowly, from 128 bytes to only 512 bytes. With merkle trees, a node can download just the block headers (80 bytes per block) and still be able to identify a transaction's inclusion in a block by retrieving a small merkle path from a full node, without storing or transmitting the vast majority of the blockchain, which might be several gigabytes in size. Nodes that do not maintain a full blockchain, called simplified payment verification (SPV nodes), use merkle paths to verify transactions without downloading full blocks.

Merkle Trees and Simplified Payment Verification (SPV)

Merkle trees are used extensively by SPV nodes. SPV nodes don't have all transactions and do not download full blocks, just block headers. In order to verify that a transaction is included in a block, without having to download all the transactions in the block, they use an authentication path, or merkle path.

Consider, for example, an SPV node that is interested in incoming payments to an address contained in its wallet. The SPV node will establish a bloom filter on its connections to peers to limit the transactions received to only those containing addresses of interest. When a peer sees a transaction that matches the bloom filter, it will send that block using a **merkleblock** message. The **merkleblock** message contains the block header as well as a merkle path that links the transaction of interest to the merkle root in the block. The SPV node can use this merkle path to connect the transaction to the block and verify that the transaction is included in the block. The SPV node also uses the block header to link the block to the rest of the blockchain. The combination of these two links, between the transaction and block, and between the block and blockchain, proves that the transaction is recorded in the blockchain. All in all, the SPV node will have received less than a kilobyte of data for the block header and merkle path, an amount of data that is more than a thousand times less than a full block (about 1 megabyte currently).

Mining and Consensus

Introduction

Mining is the process by which new bitcoin is added to the money supply. Mining also serves to secure the bitcoin system against fraudulent transactions or transactions spending the same amount of bitcoin more than once, known as a double-spend. Miners provide processing power to the bitcoin network in exchange for the opportunity to be rewarded bitcoin.

Miners validate new transactions and record them on the global ledger. A new block, containing transactions that occurred since the last block, is "mined" every 10 minutes on average, thereby adding those transactions to the blockchain. Transactions that become part of a block and added to the blockchain are considered "confirmed," which allows the new owners of bitcoin to spend the bitcoin they received in those transactions.

Miners receive two types of rewards for mining: new coins created with each new block, and transaction fees from all the transactions included in the block. To earn this reward, the miners compete to solve a difficult mathematical problem based on a cryptographic hash algorithm. The solution to the problem, called the proof of work, is included in the new block and acts as proof that the miner expended significant computing effort. The competition to solve the proof-of-work algorithm to earn reward and the right to record transactions on the blockchain is the basis for bitcoin's security model.

The process of new coin generation is called mining because the reward is designed to simulate diminishing returns, just like mining for precious metals. Bitcoin's money supply is created through mining, similar to how a central bank issues new money by printing bank notes. The amount of newly created bitcoin a miner can add to a block decreases approximately every four years (or precisely every 210,000 blocks). It started at 50 bitcoin per block in January of 2009 and halved to 25 bitcoin per block in November of 2012. It will halve again to 12.5 bitcoin per block sometime in 2016. Based on this formula, bitcoin mining rewards decrease exponentially until approximately the year 2140, when all bitcoin (20.99999998 million) will have been issued. After 2140, no new bitcoins will be issued.

Bitcoin miners also earn fees from transactions. Every transaction may include a transaction fee, in the form of a surplus of bitcoin between the transaction's inputs and outputs. The winning bitcoin miner gets to "keep the change" on the transactions included in the winning block. Today, the fees represent 0.5% or less of a bitcoin miner's income, the vast majority coming from the newly minted bitcoins. However, as the reward decreases over time and the number of transactions per block increases, a greater proportion of bitcoin mining earnings will come from fees. After 2140, all bitcoin miner earnings will be in the form of transaction fees.

The word "mining" is somewhat misleading. By evoking the extraction of precious metals, it focuses our attention on the reward for mining, the new bitcoins in each block. Although mining is incentivized by this reward, the primary purpose of mining is not the reward or the generation of new coins. If you view mining only as the process by which coins are created, you are mistaking the means (incentives) as a goal of the process. Mining is the main process of the decentralized clearinghouse, by which transactions are validated and cleared. Mining secures the bitcoin system and enables the emergence of network-wide consensus without a central authority.

Mining is the invention that makes bitcoin special, a decentralized security mechanism that is the basis for peer-to-peer digital cash. The reward of newly minted coins and transaction fees is an incentive scheme that aligns the actions of miners with the security of the network, while simultaneously implementing the monetary supply.

In this chapter, we will first examine mining as a monetary supply mechanism and then look at the most important function of mining: the decentralized emergent consensus mechanism that underpins bitcoin's security.

Bitcoin Economics and Currency Creation

Bitcoins are "minted" during the creation of each block at a fixed and diminishing rate. Each block, generated on average every 10 minutes, contains entirely new bitcoins, created from nothing. Every 210,000 blocks, or approximately every four years, the currency issuance rate is decreased by 50%. For the first four years of operation of the network, each block contained 50 new bitcoins.

In November 2012, the new bitcoin issuance rate was decreased to 25 bitcoins per block and it will decrease again to 12.5 bitcoins at block 420,000, which will be mined sometime in 2016. The rate of new coins decreases like this exponentially over 64 "halvings" until block 13,230,000 (mined approximately in year 2137), when it reaches the minimum currency unit of 1 satoshi. Finally, after 13.44 million blocks, in approximately 2140, almost 2,099,999,997,690,000 satoshis, or almost 21 million bitcoins, will be issued. Thereafter, blocks will contain no new bitcoins, and miners will be rewarded solely through the transaction fees. [Supply of bitcoin currency over time based on a geometrically decreasing issuance rate](#) shows the total bitcoin in circulation over time, as the issuance of currency decreases.

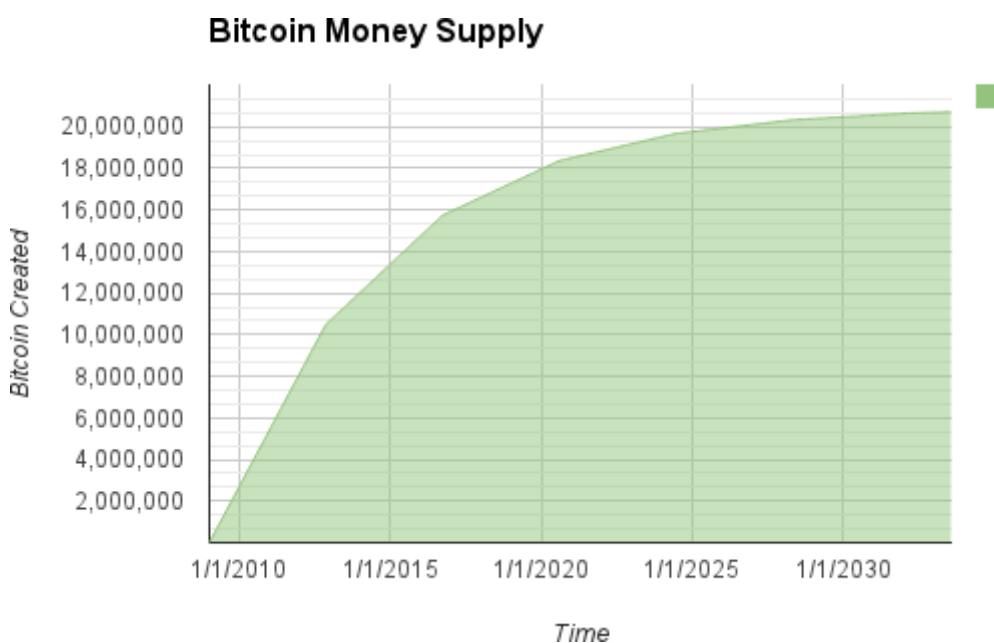


Figure 56. Supply of bitcoin currency over time based on a geometrically decreasing issuance rate

NOTE

The maximum number of coins mined is the *upper limit* of possible mining rewards for bitcoin. In practice, a miner may intentionally mine a block taking less than the full reward. Such blocks have already been mined and more may be mined in the future, resulting in a lower total issuance of the currency.

In the example code in [A script for calculating how much total bitcoin will be issued](#), we calculate the total amount of bitcoin that will be issued.

Example 18. A script for calculating how much total bitcoin will be issued

```
# Original block reward for miners was 50 BTC
start_block_reward = 50
# 210000 is around every 4 years with a 10 minute block interval
reward_interval = 210000

def max_money():
    # 50 BTC = 50 0000 0000 Satoshis
    current_reward = 50 * 10**8
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total

print "Total BTC to ever be created:", max_money(), "Satoshis"
```

Running the [max_money.py script](#) shows the output produced by running this script.

Example 19. Running the max_money.py script

```
$ python max_money.py
Total BTC to ever be created: 2099999997690000 Satoshis
```

The finite and diminishing issuance creates a fixed monetary supply that resists inflation. Unlike a fiat currency, which can be printed in infinite numbers by a central bank, bitcoin can never be inflated by printing.

Deflationary Money

The most important and debated consequence of a fixed and diminishing monetary issuance is that the currency will tend to be inherently *deflationary*. Deflation is the phenomenon of appreciation of value due to a mismatch in supply and demand that drives up the value (and exchange rate) of a currency. The opposite of inflation, price deflation means that the money has more purchasing power over time.

Many economists argue that a deflationary economy is a disaster that should be avoided at all costs. That is because in a period of rapid deflation, people tend to hoard money instead of spending it, hoping that prices will fall. Such a phenomenon unfolded during Japan's "Lost Decade," when a complete collapse of demand pushed the currency into a deflationary spiral.

Bitcoin experts argue that deflation is not bad per se. Rather, deflation is associated with a collapse in demand because that is the only example of deflation we have to study. In a fiat currency with the possibility of unlimited printing, it is very difficult to enter a deflationary spiral unless there is a complete collapse in demand and an unwillingness to print money. Deflation in bitcoin is not caused by a collapse in demand, but by a predictably constrained supply.

In practice, it has become evident that the hoarding instinct caused by a deflationary currency can be overcome by discounting from vendors, until the discount overcomes the hoarding instinct of the buyer. Because the seller is also motivated to hoard, the discount becomes the equilibrium price at which the two hoarding instincts are matched. With discounts of 30% on the bitcoin price, most bitcoin retailers are not experiencing difficulty overcoming the hoarding instinct and generating revenue. It remains to be seen whether the deflationary aspect of the currency is really a problem when it is not driven by rapid economic retraction.

Decentralized Consensus

In the previous chapter we looked at the blockchain, the global public ledger (list) of all transactions, which everyone in the bitcoin network accepts as the authoritative record of ownership.

But how can everyone in the network agree on a single universal "truth" about who owns what, without having to trust anyone? All traditional payment systems depend on a trust model that has a central authority providing a clearinghouse service, basically verifying and clearing all transactions. Bitcoin has no central authority, yet somehow every full node has a complete copy of a public ledger that it can trust as the authoritative record. The blockchain is not created by a central authority, but is assembled independently by every node in the network. Somehow, every node in the network, acting on information transmitted across insecure network connections, can arrive at the same conclusion and assemble a copy of the same public ledger as everyone else. This chapter examines the process by which the bitcoin network achieves global consensus without central authority.

Satoshi Nakamoto's main invention is the decentralized mechanism for *emergent consensus*.

Emergent, because consensus is not achieved explicitly—there is no election or fixed moment when consensus occurs. Instead, consensus is an emergent artifact of the asynchronous interaction of thousands of independent nodes, all following simple rules. All the properties of bitcoin, including currency, transactions, payments, and the security model that does not depend on central authority or trust, derive from this invention.

Bitcoin's decentralized consensus emerges from the interplay of four processes that occur independently on nodes across the network:

- Independent verification of each transaction, by every full node, based on a comprehensive list of criteria
- Independent aggregation of those transactions into new blocks by mining nodes, coupled with demonstrated computation through a proof-of-work algorithm
- Independent verification of the new blocks by every node and assembly into a chain
- Independent selection, by every node, of the chain with the most cumulative computation demonstrated through proof of work

In the next few sections we will examine these processes and how they interact to create the emergent property of network-wide consensus that allows any bitcoin node to assemble its own copy of the authoritative, trusted, public, global ledger.

Independent Verification of Transactions

In [Transactions](#), we saw how wallet software creates transactions by collecting UTXO, providing the appropriate unlocking scripts, and then constructing new outputs assigned to a new owner. The resulting transaction is then sent to the neighboring nodes in the bitcoin network so that it can be propagated across the entire bitcoin network.

However, before forwarding transactions to its neighbors, every bitcoin node that receives a transaction will first verify the transaction. This ensures that only valid transactions are propagated across the network, while invalid transactions are discarded at the first node that encounters them.

Each node verifies every transaction against a long checklist of criteria:

- The transaction's syntax and data structure must be correct.
- Neither lists of inputs or outputs are empty.
- The transaction size in bytes is less than `MAX_BLOCK_SIZE`.
- Each output value, as well as the total, must be within the allowed range of values (less than 21m coins, more than 0).
- None of the inputs have `hash=0`, `N=-1` (coinbase transactions should not be relayed).
- `nLockTime` is less than or equal to `INT_MAX`.
- The transaction size in bytes is greater than or equal to 100.
- The number of signature operations contained in the transaction is less than the signature operation limit.

- The unlocking script (`scriptSig`) can only push numbers on the stack, and the locking script (`scriptPubkey`) must match `isStandard` forms (this rejects "nonstandard" transactions).
- A matching transaction in the pool, or in a block in the main branch, must exist.
- For each input, if the referenced output exists in any other transaction in the pool, the transaction must be rejected.
- For each input, look in the main branch and the transaction pool to find the referenced output transaction. If the output transaction is missing for any input, this will be an orphan transaction. Add to the orphan transactions pool, if a matching transaction is not already in the pool.
- For each input, if the referenced output transaction is a coinbase output, it must have at least `COINBASE_MATURITY` (100) confirmations.
- For each input, the referenced output must exist and cannot already be spent.
- Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in the allowed range of values (less than 21m coins, more than 0).
- Reject if the sum of input values is less than sum of output values.
- Reject if transaction fee would be too low to get into an empty block.
- The unlocking scripts for each input must validate against the corresponding output locking scripts.

These conditions can be seen in detail in the functions `AcceptToMemoryPool`, `CheckTransaction`, and `CheckInputs` in the bitcoin reference client. Note that the conditions change over time, to address new types of denial-of-service attacks or sometimes to relax the rules so as to include more types of transactions.

By independently verifying each transaction as it is received and before propagating it, every node builds a pool of valid (but unconfirmed) transactions known as the *transaction pool*, *memory pool* or *mempool*.

Mining Nodes

Some of the nodes on the bitcoin network are specialized nodes called *miners*. In [Introduction](#) we introduced Jing, a computer engineering student in Shanghai, China, who is a bitcoin miner. Jing earns bitcoin by running a "mining rig," which is a specialized computer-hardware system designed to mine bitcoins. Jing's specialized mining hardware is connected to a server running a full bitcoin node. Unlike Jing, some miners mine without a full node, as we will see in [Mining Pools](#). Like every other full node, Jing's node receives and propagates unconfirmed transactions on the bitcoin network. Jing's node, however, also aggregates these transactions into new blocks.

Jing's node is listening for new blocks, propagated on the bitcoin network, as do all nodes. However, the arrival of a new block has special significance for a mining node. The competition among miners effectively ends with the propagation of a new block that acts as an announcement of a winner. To miners, receiving a new block means someone else won the competition and they lost. However, the end of one round of a competition is also the beginning of the next round. The new block is not just a checkered flag, marking the end of the race; it is also the starting pistol in the race for the next block.

Aggregating Transactions into Blocks

After validating transactions, a bitcoin node will add them to the *memory pool*, or *transaction pool*, where transactions await until they can be included (mined) into a block. Jing's node collects, validates, and relays new transactions just like any other node. Unlike other nodes, however, Jing's node will then aggregate these transactions into a *candidate block*.

Let's follow the blocks that were created during the time Alice bought a cup of coffee from Bob's Cafe (see [Buying a Cup of Coffee](#)). Alice's transaction was included in block 277,316. For the purpose of demonstrating the concepts in this chapter, let's assume that block was mined by Jing's mining system and follow Alice's transaction as it becomes part of this new block.

Jing's mining node maintains a local copy of the blockchain, the list of all blocks created since the beginning of the bitcoin system in 2009. By the time Alice buys the cup of coffee, Jing's node has assembled a chain up to block 277,314. Jing's node is listening for transactions, trying to mine a new block and also listening for blocks discovered by other nodes. As Jing's node is mining, it receives block 277,315 through the bitcoin network. The arrival of this block signifies the end of the competition for block 277,315 and the beginning of the competition to create block 277,316.

During the previous 10 minutes, while Jing's node was searching for a solution to block 277,315, it was also collecting transactions in preparation for the next block. By now it has collected a few hundred transactions in the memory pool. Upon receiving block 277,315 and validating it, Jing's node will also check all the transactions in the memory pool and remove any that were included in block 277,315. Whatever transactions remain in the memory pool are unconfirmed and are waiting to be recorded in a new block.

Jing's node immediately constructs a new empty block, a candidate for block 277,316. This block is called a candidate block because it is not yet a valid block, as it does not contain a valid proof of work. The block becomes valid only if the miner succeeds in finding a solution to the proof-of-work algorithm.

Transaction Age, Fees, and Priority

To construct the candidate block, Jing's bitcoin node selects transactions from the memory pool by applying a priority metric to each transaction and adding the highest priority transactions first. Transactions are prioritized based on the "age" of the UTXO that is being spent in their inputs, allowing for old and high-value inputs to be prioritized over newer and smaller inputs. Prioritized transactions can be sent without any fees, if there is enough space in the block.

The priority of a transaction is calculated as the sum of the value and age of the inputs divided by the total size of the transaction:

$$\text{Priority} = \text{Sum (Value of input * Input Age)} / \text{Transaction Size}$$

In this equation, the value of an input is measured in the base unit, satoshis (1/100m of a bitcoin). The age of a UTXO is the number of blocks that have elapsed since the UTXO was recorded on the blockchain, measuring how many blocks "deep" into the blockchain it is. The size of the transaction is measured in bytes.

For a transaction to be considered "high priority," its priority must be greater than 57,600,000, which corresponds to one bitcoin (100m satoshis), aged one day (144 blocks), in a transaction of 250 bytes total size:

$$\text{High Priority} > 100,000,000 \text{ satoshis} * 144 \text{ blocks} / 250 \text{ bytes} = 57,600,000$$

The first 50 kilobytes of transaction space in a block are set aside for high-priority transactions. Jing's node will fill the first 50 kilobytes, prioritizing the highest priority transactions first, regardless of fee. This allows high-priority transactions to be processed even if they carry zero fees.

Jing's mining node then fills the rest of the block up to the maximum block size (`MAX_BLOCK_SIZE` in the code), with transactions that carry at least the minimum fee, prioritizing those with the highest fee per kilobyte of transaction.

If there is any space remaining in the block, Jing's mining node might choose to fill it with no-fee transactions. Some miners choose to mine transactions without fees on a best-effort basis. Other miners may choose to ignore transactions without fees.

Any transactions left in the memory pool, after the block is filled, will remain in the pool for inclusion in the next block. As transactions remain in the memory pool, their inputs "age," as the UTXO they spend get deeper into the blockchain with new blocks added on top. Because a transaction's priority depends on the age of its inputs, transactions remaining in the pool will age and therefore increase in priority. Eventually a transaction without fees might reach a high enough priority to be included in the block for free.

Bitcoin transactions do not have an expiration time-out. A transaction that is valid now will be valid in perpetuity. However, if a transaction is only propagated across the network once, it will persist only as long as it is held in a mining node memory pool. When a mining node is restarted, its memory pool is wiped clear, because it is a transient non-persistent form of storage. Although a valid transaction might have been propagated across the network, if it is not executed it may eventually not reside in the memory pool of any miner. Wallet software is expected to retransmit such transactions or reconstruct them with higher fees if they are not successfully executed within a reasonable amount of time.

When Jing's node aggregates all the transactions from the memory pool, the new candidate block has 418 transactions with total transaction fees of 0.09094928 bitcoin. You can see this block in the blockchain using the Bitcoin Core client command-line interface, as shown in [Block 277,316](#).

```
$ bitcoin-cli getblockhash 277316
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

$ bitcoin-cli getblock
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

```
{
  "hash" : "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
  "confirmations" : 35561,
  "size" : 218629,
  "height" : 277316,
  "version" : 2,
  "merkleroot" :
"c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
  "tx" : [
    "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",

    ... 417 more transactions ...

  ],
  "time" : 1388185914,
  "nonce" : 924591752,
  "bits" : "1903a30c",
  "difficulty" : 1180923195.25802612,
  "chainwork" :
"00000000000000000000000000000000000000000000000000000000934695e92aaf53afa1a",
  "previousblockhash" :
"0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569",
  "nextblockhash" :
"00000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
```

The Generation Transaction

The first transaction added to the block is a special transaction, called a *generation transaction* or *coinbase transaction*. This transaction is constructed by Jing's node and is his reward for the mining effort. Jing's node creates the generation transaction as a payment to his own wallet: "Pay Jing's address 25.09094928 bitcoin." The total amount of reward that Jing collects for mining a block is the sum of the coinbase reward (25 new bitcoins) and the transaction fees (0.09094928) from all the transactions included in the block as shown in [Generation transaction](#):

```
$ bitcoin-cli getrawtransaction
d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1
```

Example 21. Generation transaction

```
{
    "hex" :
"010000000100000000000000000000000000000000000000000000000000000000000000000000000000000fffff
f0f03443b0403858402062f503253482fffffffff0110c08d9500000000232102aa970c592640d19de
03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",
    "txid" : "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "coinbase" : "03443b0403858402062f503253482f",
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 25.09094928,
            "n" : 0,
            "scriptPubKey" : {
                "asm" :
"02aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21OP_CHECKSIG",
                "hex" :
"2102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac",
                "reqSigs" : 1,
                "type" : "pubkey",
                "addresses" : [
                    "1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N"
                ]
            }
        }
    ],
    "blockhash" :
"0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35566,
    "time" : 1388185914,
    "blocktime" : 1388185914
}
```

Unlike regular transactions, the generation transaction does not consume (spend) UTXO as inputs. Instead, it has only one input, called the *coinbase*, which creates bitcoin from nothing. The generation transaction has one output, payable to the miner's own bitcoin address. The output of the generation transaction sends the value of 25.09094928 bitcoins to the miner's bitcoin address, in this case **1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N**.

Coinbase Reward and Fees

To construct the generation transaction, Jing's node first calculates the total amount of transaction fees by adding all the inputs and outputs of the 418 transactions that were added to the block. The fees are calculated as:

$$\text{Total Fees} = \text{Sum}(\text{Inputs}) - \text{Sum}(\text{Outputs})$$

In block 277,316, the total transaction fees are 0.09094928 bitcoins.

Next, Jing's node calculates the correct reward for the new block. The reward is calculated based on the block height, starting at 50 bitcoins per block and reduced by half every 210,000 blocks. Because this block is at height 277,316, the correct reward is 25 bitcoins.

The calculation can be seen in function `GetBlockSubsidy` in the Bitcoin Core client, as shown in [Calculating the block reward — Function GetBlockSubsidy, Bitcoin Core Client, main.cpp](#).

Example 22. Calculating the block reward — Function GetBlockSubsidy, Bitcoin Core Client, main.cpp

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)
{
    int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
    // Force block reward to zero when right shift is undefined.
    if (halvings >= 64)
        return 0;

    CAmount nSubsidy = 50 * COIN;
    // Subsidy is cut in half every 210,000 blocks which will occur approximately
    every 4 years.
    nSubsidy >>= halvings;
    return nSubsidy;
}
```

The initial subsidy is calculated in satoshis by multiplying 50 with the `COIN` constant (100,000,000 satoshis). This sets the initial reward (`nSubsidy`) at 5 billion satoshis.

Next, the function calculates the number of `halvings` that have occurred by dividing the current block height by the halving interval (`SubsidyHalvingInterval`). In the case of block 277,316, with a halving interval every 210,000 blocks, the result is 1 halving.

The maximum number of halvings allowed is 64, so the code imposes a zero reward (return only the fees) if the 64 halvings is exceeded.

Next, the function uses the binary-right-shift operator to divide the reward (`nSubsidy`) by two for each round of halving. In the case of block 277,316, this would binary-right-shift the reward of 5 billion satoshis once (one halving) and result in 2.5 billion satoshis, or 25 bitcoins. The binary-right-shift operator is used because it is more efficient for division by two than integer or floating-point

division.

Finally, the coinbase reward (**nSubsidy**) is added to the transaction fees (**nFees**), and the sum is returned.

Structure of the Generation Transaction

With these calculations, Jing's node then constructs the generation transaction to pay himself 25.09094928 bitcoin.

As you can see in [Generation transaction](#), the generation transaction has a special format. Instead of a transaction input specifying a previous UTXO to spend, it has a "coinbase" input. We examined transaction inputs in [The structure of a transaction input](#). Let's compare a regular transaction input with a generation transaction input. [The structure of a "normal" transaction input](#) shows the structure of a regular transaction, while [The structure of a generation transaction input](#) shows the structure of the generation transaction's input.

Table 23. The structure of a "normal" transaction input

Size	Field	Description
32 bytes	Transaction Hash	Pointer to the transaction containing the UTXO to be spent
4 bytes	Output Index	The index number of the UTXO to be spent, first one is 0
1-9 bytes (VarInt)	Unlocking-Script Size	Unlocking-Script length in bytes, to follow
Variable	Unlocking-Script	A script that fulfills the conditions of the UTXO locking script.
4 bytes	Sequence Number	Currently disabled Tx-replacement feature, set to 0xFFFFFFFF

Table 24. The structure of a generation transaction input

Size	Field	Description
32 bytes	Transaction Hash	All bits are zero: Not a transaction hash reference
4 bytes	Output Index	All bits are ones: 0xFFFFFFFF
1-9 bytes (VarInt)	Coinbase Data Size	Length of the coinbase data, from 2 to 100 bytes
Variable	Coinbase Data	Arbitrary data used for extra nonce and mining tags in v2 blocks, must begin with block height
4 bytes	Sequence Number	Set to 0xFFFFFFFF

In a generation transaction, the first two fields are set to values that do not represent a UTXO reference. Instead of a "Transaction Hash," the first field is filled with 32 bytes all set to zero. The "Output Index" is filled with 4 bytes all set to 0xFF (255 decimal). The "Unlocking Script" is replaced by coinbase data, an arbitrary data field used by the miners.

Coinbase Data

Generation transactions do not have an unlocking script (a.k.a., `scriptSig`) field. Instead, this field is replaced by coinbase data, which must be between 2 and 100 bytes. Except for the first few bytes, the rest of the coinbase data can be used by miners in any way they want; it is arbitrary data.

In the genesis block, for example, Satoshi Nakamoto added the text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" in the coinbase data, using it as a proof of the date and to convey a message. Currently, miners use the coinbase data to include extra nonce values and strings identifying the mining pool, as we will see in the following sections.

The first few bytes of the coinbase used to be arbitrary, but that is no longer the case. As per Bitcoin Improvement Proposal 34 (BIP0034), version-2 blocks (blocks with the version field set to 2) must contain the block height index as a script "push" operation in the beginning of the coinbase field.

In block 277,316 we see that the coinbase (see [Generation transaction](#)), which is in the "Unlocking Script" or `scriptSig` field of the transaction input, contains the hexadecimal value `03443b0403858402062f503253482f`. Let's decode this value.

The first byte, `03`, instructs the script execution engine to push the next three bytes onto the script stack (see [Push value onto stack](#)). The next three bytes, `0x443b04`, are the block height encoded in little-endian format (backward, least significant byte first). Reverse the order of the bytes and the result is `0x043b44`, which is 277,316 in decimal.

The next few hexadecimal digits (`03858402062`) are used to encode an extra *nonce* (see [The Extra Nonce Solution](#)), or random value, used to find a suitable proof of work solution.

The final part of the coinbase data (`2f503253482f`) is the ASCII-encoded string `/P2SH/`, which indicates that the mining node that mined this block supports the pay-to-script-hash (P2SH) improvement defined in BIP0016. The introduction of the P2SH capability required a "vote" by miners to endorse either BIP0016 or BIP0017. Those endorsing the BIP0016 implementation were to include `/P2SH/` in their coinbase data. Those endorsing the BIP0017 implementation of P2SH were to include the string `p2sh/CHV` in their coinbase data. The BIP0016 was elected as the winner, and many miners continued including the string `/P2SH/` in their coinbase to indicate support for this feature.

[Extract the coinbase data from the genesis block](#) uses the libbitcoin library introduced in [Alternative Clients, Libraries, and Toolkits](#) to extract the coinbase data from the genesis block, displaying Satoshi's message. Note that the libbitcoin library contains a static copy of the genesis block, so the example code can retrieve the genesis block directly from the library.

Example 23. Extract the coinbase data from the genesis block

```
/*
    Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::block_type block = bc::genesis_block();
    // Genesis block contains a single coinbase transaction.
    assert(block.transactions.size() == 1);
    // Get first transaction in block (coinbase).
    const bc::transaction_type& coinbase_tx = block.transactions[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs.size() == 1);
    const bc::transaction_input_type& coinbase_input = coinbase_tx.inputs[0];
    // Convert the input script to its raw format.
    const bc::data_chunk& raw_message = save_script(coinbase_input.script);
    // Convert this to an std::string.
    std::string message;
    message.resize(raw_message.size());
    std::copy(raw_message.begin(), raw_message.end(), message.begin());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}
```

We compile the code with the GNU C++ compiler and run the resulting executable, as shown in [Compiling and running the satoshi-words example code](#).

Example 24. Compiling and running the satoshi-words example code

```
$ # Compile the code
$ g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Run the executable
$ ./satoshi-words
^D    <GS>^A^DThe Times 03/Jan/2009 Chancellor on brink of second bailout for
banks
```

Constructing the Block Header

To construct the block header, the mining node needs to fill in six fields, as listed in [The structure of the block header](#).

Table 25. The structure of the block header

Size	Field	Description
4 bytes	Version	A version number to track software/protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of the previous (parent) block in the chain
32 bytes	Merkle Root	A hash of the root of the merkle tree of this block's transactions
4 bytes	Timestamp	The approximate creation time of this block (seconds from Unix Epoch)
4 bytes	Difficulty Target	The proof-of-work algorithm difficulty target for this block
4 bytes	Nonce	A counter used for the proof-of-work algorithm

At the time that block 277,316 was mined, the version number describing the block structure is version 2, which is encoded in little-endian format in 4 bytes as `0x02000000`.

Next, the mining node needs to add the "Previous Block Hash." That is the hash of the block header of block 277,315, the previous block received from the network, which Jing's node has accepted and selected as the parent of the candidate block 277,316. The block header hash for block 277,315 is:

```
00000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

The next step is to summarize all the transactions with a merkle tree, in order to add the merkle root to the block header. The generation transaction is listed as the first transaction in the block. Then, 418 more transactions are added after it, for a total of 419 transactions in the block. As we saw in the [Merkle Trees](#), there must be an even number of "leaf" nodes in the tree, so the last transaction is duplicated, creating 420 nodes, each containing the hash of one transaction. The transaction hashes are then combined, in pairs, creating each level of the tree, until all the transactions are summarized into one node at the "root" of the tree. The root of the merkle tree summarizes all the transactions into a single 32-byte value, which you can see listed as "merkle root" in [Block 277,316](#), and here:

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

The mining node will then add a 4-byte timestamp, encoded as a Unix "Epoch" timestamp, which is based on the number of seconds elapsed from January 1, 1970, midnight UTC/GMT. The time `1388185914` is equal to Friday, 27 Dec 2013, 23:11:54 UTC/GMT.

The node then fills in the difficulty target, which defines the required proof-of-work difficulty to make this a valid block. The difficulty is stored in the block as a "difficulty bits" metric, which is a

mantissa-exponent encoding of the target. The encoding has a 1-byte exponent, followed by a 3-byte mantissa (coefficient). In block 277,316, for example, the difficulty bits value is `0x1903a30c`. The first part `0x19` is a hexadecimal exponent, while the next part, `0x03a30c`, is the coefficient. The concept of a difficulty target is explained in [Difficulty Target and Retargeting](#) and the "difficulty bits" representation is explained in [Difficulty Representation](#).

The final field is the nonce, which is initialized to zero.

With all the other fields filled, the block header is now complete and the process of mining can begin. The goal is now to find a value for the nonce that results in a block header hash that is less than the difficulty target. The mining node will need to test billions or trillions of nonce values before a nonce is found that satisfies the requirement.

Mining the Block

Now that a candidate block has been constructed by Jing's node, it is time for Jing's hardware mining rig to "mine" the block, to find a solution to the proof-of-work algorithm that makes the block valid. Throughout this book we have studied cryptographic hash functions as used in various aspects of the bitcoin system. The hash function SHA256 is the function used in bitcoin's mining process.

In the simplest terms, mining is the process of hashing the block header repeatedly, changing one parameter, until the resulting hash matches a specific target. The hash function's result cannot be determined in advance, nor can a pattern be created that will produce a specific hash value. This feature of hash functions means that the only way to produce a hash result matching a specific target is to try again and again, randomly modifying the input until the desired hash result appears by chance.

Proof-Of-Work Algorithm

A hash algorithm takes an arbitrary-length data input and produces a fixed-length deterministic result, a digital fingerprint of the input. For any specific input, the resulting hash will always be the same and can be easily calculated and verified by anyone implementing the same hash algorithm. The key characteristic of a cryptographic hash algorithm is that it is virtually impossible to find two different inputs that produce the same fingerprint. As a corollary, it is also virtually impossible to select an input in such a way as to produce a desired fingerprint, other than trying random inputs.

With SHA256, the output is always 256 bits long, regardless of the size of the input. In [SHA256 example](#), we will use the Python interpreter to calculate the SHA256 hash of the phrase, "I am Satoshi Nakamoto."

Example 25. SHA256 example

```
$ python
```

```
Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

[SHA256 example](#) shows the result of calculating the hash of "I am Satoshi Nakamoto": `5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e`. This 256-bit number is the *hash* or *digest* of the phrase and depends on every part of the phrase. Adding a single letter, punctuation mark, or any other character will produce a different hash.

Now, if we change the phrase, we should expect to see completely different hashes. Let's try that by adding a number to the end of our phrase, using the simple Python scripting in [SHA256 A script for generating many hashes by iterating on a nonce](#).

Example 26. SHA256 A script for generating many hashes by iterating on a nonce

```
# example of iterating a nonce in a hashing algorithm's input

import hashlib

text = "I am Satoshi Nakamoto"

# iterate nonce from 0 to 19
for nonce in xrange(20):

    # add the nonce to the end of the text
    input = text + str(nonce)

    # calculate the SHA-256 hash of the input (text+nonce)
    hash = hashlib.sha256(input).hexdigest()

    # show the input and hash result
    print input, '=>', hash
```

Running this will produce the hashes of several phrases, made different by adding a number at the end of the text. By incrementing the number, we can get different hashes, as shown in [SHA256 output of a script for generating many hashes by iterating on a nonce](#).

Example 27. SHA256 output of a script for generating many hashes by iterating on a nonce

```
$ python hash_example.py
```

```
I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

Each phrase produces a completely different hash result. They seem completely random, but you can reproduce the exact results in this example on any computer with Python and see the same exact hashes.

The number used as a variable in such a scenario is called a *nonce*. The nonce is used to vary the output of a cryptographic function, in this case to vary the SHA256 fingerprint of the phrase.

[illegible]

To give a simple analogy, imagine a game where players throw a pair of dice repeatedly, trying to throw less than a specified target. In the first round, the target is 12. Unless you throw double-six,

you win. In the next round the target is 11. Players must throw 10 or less to win, again an easy task. Let's say a few rounds later the target is down to 5. Now, more than half the dice throws will add up to more than 5 and therefore be invalid. It takes exponentially more dice throws to win, the lower the target gets. Eventually, when the target is 2 (the minimum possible), only one throw out of every 36, or 2% of them, will produce a winning result.

In [SHA256 output of a script for generating many hashes by iterating on a nonce](#), the winning "nonce" is 13 and this result can be confirmed by anyone independently. Anyone can add the number 13 as a suffix to the phrase "I am Satoshi Nakamoto" and compute the hash, verifying that it is less than the target. The successful result is also proof of work, because it proves we did the work to find that nonce. While it only takes one hash computation to verify, it took us 13 hash computations to find a nonce that worked. If we had a lower target (higher difficulty) it would take many more hash computations to find a suitable nonce, but only one hash computation for anyone to verify. Furthermore, by knowing the target, anyone can estimate the difficulty using statistics and therefore know how much work was needed to find such a nonce.

Bitcoin's proof of work is very similar to the challenge shown in [SHA256 output of a script for generating many hashes by iterating on a nonce](#). The miner constructs a candidate block filled with transactions. Next, the miner calculates the hash of this block's header and sees if it is smaller than the current *target*. If the hash is not less than the target, the miner will modify the nonce (usually just incrementing it by one) and try again. At the current difficulty in the bitcoin network, miners have to try quadrillions of times before finding a nonce that results in a low enough block header hash.

A very simplified proof-of-work algorithm is implemented in Python in [Simplified proof-of-work implementation](#).

Example 28. Simplified proof-of-work implementation

```
#!/usr/bin/env python
# example of proof-of-work algorithm

import hashlib
import time

max_nonce = 2 ** 32 # 4 billion

def proof_of_work(header, difficulty_bits):

    # calculate the difficulty target
    target = 2 ** (256-difficulty_bits)

    for nonce in xrange(max_nonce):
        hash_result = hashlib.sha256(str(header)+str(nonce)).hexdigest()

        # check if this is a valid result, below the target
        if long(hash_result, 16) < target:
            print "Success with nonce %d" % nonce
            print "Hash is %s" % hash_result
```

```

        return (hash_result, nonce)

    print "Failed after %d (max_nonce) tries" % nonce
    return nonce

if __name__ == '__main__':

    nonce = 0
    hash_result = ''

    # difficulty from 0 to 31 bits
    for difficulty_bits in xrange(32):

        difficulty = 2 ** difficulty_bits
        print "Difficulty: %ld (%d bits)" % (difficulty, difficulty_bits)

        print "Starting search..."

        # checkpoint the current time
        start_time = time.time()

        # make a new block which includes the hash from the previous block
        # we fake a block of transactions - just a string
        new_block = 'test block with transactions' + hash_result

        # find a valid nonce for the new block
        (hash_result, nonce) = proof_of_work(new_block, difficulty_bits)

        # checkpoint how long it took to find a result
        end_time = time.time()

        elapsed_time = end_time - start_time
        print "Elapsed Time: %.4f seconds" % elapsed_time

        if elapsed_time > 0:

            # estimate the hashes per second
            hash_power = float(long(nonce)/elapsed_time)
            print "Hashing Power: %ld hashes per second" % hash_power

```

Running this code, you can set the desired difficulty (in bits, how many of the leading bits must be zero) and see how long it takes for your computer to find a solution. In [Running the proof of work example for various difficulties](#), you can see how it works on an average laptop.

Example 29. Running the proof of work example for various difficulties

```
$ python proof-of-work-example.py*
```

Difficulty: 1 (0 bits)

[...]

Difficulty: 8 (3 bits)

Starting search...

Success with nonce 9

Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1

Elapsed Time: 0.0004 seconds

Hashing Power: 25065 hashes per second

Difficulty: 16 (4 bits)

Starting search...

Success with nonce 25

Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148

Elapsed Time: 0.0005 seconds

Hashing Power: 52507 hashes per second

Difficulty: 32 (5 bits)

Starting search...

Success with nonce 36

Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903

Elapsed Time: 0.0006 seconds

Hashing Power: 58164 hashes per second

[...]

Difficulty: 4194304 (22 bits)

Starting search...

Success with nonce 1759164

Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cefc3

Elapsed Time: 13.3201 seconds

Hashing Power: 132068 hashes per second

Difficulty: 8388608 (23 bits)

Starting search...

Success with nonce 14214729

Hash is 000001408cf12dbd20fcba6372a223e098d58786c6ff93488a9f74f5df4df0a3

Elapsed Time: 110.1507 seconds

Hashing Power: 129048 hashes per second

Difficulty: 16777216 (24 bits)

Starting search...

Success with nonce 24586379

Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95

Elapsed Time: 195.2991 seconds

Hashing Power: 125890 hashes per second

[...]

Difficulty: 67108864 (26 bits)

Starting search...

Success with nonce 84561291

Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a

Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second

As you can see, increasing the difficulty by 1 bit causes an exponential increase in the time it takes to find a solution. If you think of the entire 256-bit number space, each time you constrain one more bit to zero, you decrease the search space by half. In [Running the proof of work example for various difficulties](#), it takes 84 million hash attempts to find a nonce that produces a hash with 26 leading bits as zero. Even at a speed of more than 120,000 hashes per second, it still requires 10 minutes on a consumer laptop to find this solution.

At the time of writing, the network is attempting to find a block whose header hash is less than `000000000000004c296e6376db3a241271f43fd3f5de7ba18986e517a243baa7`. As you can see, there are a lot of zeros at the beginning of that hash, meaning that the acceptable range of hashes is much smaller, hence it's more difficult to find a valid hash. It will take on average more than 150 quadrillion hash calculations per second for the network to discover the next block. That seems like an impossible task, but fortunately the network is bringing 100 petahashes per second (PH/sec) of processing power to bear, which will be able to find a block in about 10 minutes on average.

Difficulty Representation

In [Block 277,316](#), we saw that the block contains the difficulty target, in a notation called "difficulty bits" or just "bits," which in block 277,316 has the value of `0x1903a30c`. This notation expresses the difficulty target as a coefficient/exponent format, with the first two hexadecimal digits for the exponent and the next six hex digits as the coefficient. In this block, therefore, the exponent is `0x19` and the coefficient is `0x03a30c`.

The formula to calculate the difficulty target from this representation is:

$$\text{target} = \text{coefficient} * 2^{(8 * (\text{exponent} - 3))}$$

Using that formula, and the difficulty bits value `0x1903a30c`, we get:

$$\begin{aligned}\text{target} &= 0x03a30c * 2^{(0x08 * (0x19 - 0x03))} \\ \Rightarrow \text{target} &= 0x03a30c * 2^{(0x08 * 0x16)} \\ \Rightarrow \text{target} &= 0x03a30c * 2^{0xB0}\end{aligned}$$

which in decimal is:

$$\begin{aligned}\Rightarrow \text{target} &= 238,348 * 2^{176} \\ \Rightarrow \text{target} &= \\ &22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328\end{aligned}$$

switching back to hexadecimal:

```
=> target = 0x00000000000000003A30C0000000000000000000000000000000000000000000
```

This means that a valid block for height 277,316 is one that has a block header hash that is less than the target. In binary that number would have more than the first 60 bits set to zero. With this level of difficulty, a single miner processing 1 trillion hashes per second (1 tera-hash per second or 1 TH/sec) would only find a solution once every 8,496 blocks or once every 59 days, on average.

Difficulty Target and Retargeting

As we saw, the target determines the difficulty and therefore affects how long it takes to find a solution to the proof-of-work algorithm. This leads to the obvious questions: Why is the difficulty adjustable, who adjusts it, and how?

Bitcoin's blocks are generated every 10 minutes, on average. This is bitcoin's heartbeat and underpins the frequency of currency issuance and the speed of transaction settlement. It has to remain constant not just over the short term, but over a period of many decades. Over this time, it is expected that computer power will continue to increase at a rapid pace. Furthermore, the number of participants in mining and the computers they use will also constantly change. To keep the block generation time at 10 minutes, the difficulty of mining must be adjusted to account for these changes. In fact, difficulty is a dynamic parameter that will be periodically adjusted to meet a 10-minute block target. In simple terms, the difficulty target is set to whatever mining power will result in a 10-minute block interval.

How, then, is such an adjustment made in a completely decentralized network? Difficulty retargeting occurs automatically and on every full node independently. Every 2,016 blocks, all nodes retarget the proof-of-work difficulty. The equation for retargeting difficulty measures the time it took to find the last 2,016 blocks and compares that to the expected time of 20,160 minutes (two weeks based upon a desired 10-minute block time). The ratio between the actual timespan and desired timespan is calculated and a corresponding adjustment (up or down) is made to the difficulty. In simple terms: If the network is finding blocks faster than every 10 minutes, the difficulty increases. If block discovery is slower than expected, the difficulty decreases.

The equation can be summarized as:

```
New Difficulty = Old Difficulty * (Actual Time of Last 2016 Blocks / 20160 minutes)
```

[Retargeting the proof-of-work difficulty](#) — `CalculateNextWorkRequired()` in `pow.cpp` shows the code used in the Bitcoin Core client.

```
// Limit adjustment step
int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
LogPrintf("  nActualTimespan = %d before bounds\n", nActualTimespan);
if (nActualTimespan < params.nPowTargetTimespan/4)
    nActualTimespan = params.nPowTargetTimespan/4;
if (nActualTimespan > params.nPowTargetTimespan*4)
    nActualTimespan = params.nPowTargetTimespan*4;

// Retarget
const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
arith_uint256 bnNew;
arith_uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= params.nPowTargetTimespan;

if (bnNew > bnPowLimit)
    bnNew = bnPowLimit;
```

NOTE

While the difficulty calibration happens every 2,016 blocks, because of an off-by-one error in the original Bitcoin Core client it is based on the total time of the previous 2,015 blocks (not 2,016 as it should be), resulting in a retargeting bias towards higher difficulty by 0.05%.

The parameters Interval (2,016 blocks) and TargetTimespan (two weeks as 1,209,600 seconds) are defined in *chainparams.cpp*.

To avoid extreme volatility in the difficulty, the retargeting adjustment must be less than a factor of four (4) per cycle. If the required difficulty adjustment is greater than a factor of four, it will be adjusted by the maximum and not more. Any further adjustment will be accomplished in the next retargeting period because the imbalance will persist through the next 2,016 blocks. Therefore, large discrepancies between hashing power and difficulty might take several 2,016 block cycles to balance out.

TIP

The difficulty of finding a bitcoin block is approximately *10 minutes of processing* for the entire network, based on the time it took to find the previous 2,016 blocks, adjusted every 2,016 blocks.

Note that the target difficulty is independent of the number of transactions or the value of transactions. This means that the amount of hashing power and therefore electricity expended to secure bitcoin is also entirely independent of the number of transactions. Bitcoin can scale up, achieve broader adoption, and remain secure without any increase in hashing power from today's level. The increase in hashing power represents market forces as new miners enter the market to compete for the reward. As long as enough hashing power is under the control of miners acting

honestly in pursuit of the reward, it is enough to prevent "takeover" attacks and, therefore, it is enough to secure bitcoin.

The target difficulty is closely related to the cost of electricity and the exchange rate of bitcoin vis-à-vis the currency used to pay for electricity. High-performance mining systems are about as efficient as possible with the current generation of silicon fabrication, converting electricity into hashing computation at the highest rate possible. The primary influence on the mining market is the price of one kilowatt-hour in bitcoin, because that determines the profitability of mining and therefore the incentives to enter or exit the mining market.

Successfully Mining the Block

As we saw earlier, Jing's node has constructed a candidate block and prepared it for mining. Jing has several hardware mining rigs with application-specific integrated circuits, where hundreds of thousands of integrated circuits run the SHA256 algorithm in parallel at incredible speeds. These specialized machines are connected to his mining node over USB. Next, the mining node running on Jing's desktop transmits the block header to his mining hardware, which starts testing trillions of nonces per second.

Almost 11 minutes after starting to mine block 277,316, one of the hardware mining machines finds a solution and sends it back to the mining node. When inserted into the block header, the nonce 4,215,469,401 produces a block hash of:

00000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569

which is less than the target:

000000000000000000003A30C00

Immediately, Jing's mining node transmits the block to all its peers. They receive, validate, and then propagate the new block. As the block ripples out across the network, each node adds it to its own copy of the blockchain, extending it to a new height of 277,316 blocks. As mining nodes receive and validate the block, they abandon their efforts to find a block at the same height and immediately start computing the next block in the chain.

In the next section, we'll look at the process each node uses to validate a block and select the longest chain, creating the consensus that forms the decentralized blockchain.

Validating a New Block

The third step in bitcoin's consensus mechanism is independent validation of each new block by every node on the network. As the newly solved block moves across the network, each node performs a series of tests to validate it before propagating it to its peers. This ensures that only valid blocks are propagated on the network. The independent validation also ensures that miners who act honestly get their blocks incorporated in the blockchain, thus earning the reward. Those miners who act dishonestly have their blocks rejected and not only lose the reward, but also waste

the effort expended to find a proof-of-work solution, thus incurring the cost of electricity without compensation.

When a node receives a new block, it will validate the block by checking it against a long list of criteria that must all be met; otherwise, the block is rejected. These criteria can be seen in the Bitcoin Core client in the functions `CheckBlock` and `CheckBlockHeader` and include:

- The block data structure is syntactically valid
- The block header hash is less than the target difficulty (enforces the proof of work)
- The block timestamp is less than two hours in the future (allowing for time errors)
- The block size is within acceptable limits
- The first transaction (and only the first) is a coinbase generation transaction
- All transactions within the block are valid using the transaction checklist discussed in [Independent Verification of Transactions](#)

The independent validation of each new block by every node on the network ensures that the miners can't cheat. In previous sections we saw how the miners get to write a transaction that awards them the new bitcoins created within the block and claim the transaction fees. Why don't miners write themselves a transaction for a thousand bitcoin instead of the correct reward? Because every node validates blocks according to the same rules. An invalid coinbase transaction would make the entire block invalid, which would result in the block being rejected and, therefore, that transaction would never become part of the ledger. The miners have to construct a perfect block, based on the shared rules that all nodes follow, and mine it with a correct solution to the proof of work. To do so, they expend a lot of electricity in mining, and if they cheat, all the electricity and effort is wasted. This is why independent validation is a key component of decentralized consensus.

Assembling and Selecting Chains of Blocks

The final step in bitcoin's decentralized consensus mechanism is the assembly of blocks into chains and the selection of the chain with the most proof of work. Once a node has validated a new block, it will then attempt to assemble a chain by connecting the block to the existing blockchain.

Nodes maintain three sets of blocks: those connected to the main blockchain, those that form branches off the main blockchain (secondary chains), and finally, blocks that do not have a known parent in the known chains (orphans). Invalid blocks are rejected as soon as any one of the validation criteria fails and are therefore not included in any chain.

The "main chain" at any time is whichever chain of blocks has the most cumulative difficulty associated with it. Under most circumstances this is also the chain with the most blocks in it, unless there are two equal-length chains and one has more proof of work. The main chain will also have branches with blocks that are "siblings" to the blocks on the main chain. These blocks are valid but not part of the main chain. They are kept for future reference, in case one of those chains is extended to exceed the main chain in difficulty. In the next section ([Blockchain Forks](#)), we will see how secondary chains occur as a result of an almost simultaneous mining of blocks at the same height.

When a new block is received, a node will try to slot it into the existing blockchain. The node will look at the block's "previous block hash" field, which is the reference to the new block's parent. Then, the node will attempt to find that parent in the existing blockchain. Most of the time, the parent will be the "tip" of the main chain, meaning this new block extends the main chain. For example, the new block 277,316 has a reference to the hash of its parent block 277,315. Most nodes that receive 277,316 will already have block 277,315 as the tip of their main chain and will therefore link the new block and extend that chain.

Sometimes, as we will see in [Blockchain Forks](#), the new block extends a chain that is not the main chain. In that case, the node will attach the new block to the secondary chain it extends and then compare the difficulty of the secondary chain to the main chain. If the secondary chain has more cumulative difficulty than the main chain, the node will *reconverge* on the secondary chain, meaning it will select the secondary chain as its new main chain, making the old main chain a secondary chain. If the node is a miner, it will now construct a block extending this new, longer, chain.

If a valid block is received and no parent is found in the existing chains, that block is considered an "orphan." Orphan blocks are saved in the orphan block pool where they will stay until their parent is received. Once the parent is received and linked into the existing chains, the orphan can be pulled out of the orphan pool and linked to the parent, making it part of a chain. Orphan blocks usually occur when two blocks that were mined within a short time of each other are received in reverse order (child before parent).

By selecting the greatest-difficulty chain, all nodes eventually achieve network-wide consensus. Temporary discrepancies between chains are resolved eventually as more proof of work is added, extending one of the possible chains. Mining nodes "vote" with their mining power by choosing which chain to extend by mining the next block. When they mine a new block and extend the chain, the new block itself represents their vote.

In the next section we will look at how discrepancies between competing chains (forks) are resolved by the independent selection of the longest difficulty chain.

Blockchain Forks

Because the blockchain is a decentralized data structure, different copies of it are not always consistent. Blocks might arrive at different nodes at different times, causing the nodes to have different perspectives of the blockchain. To resolve this, each node always selects and attempts to extend the chain of blocks that represents the most proof of work, also known as the longest chain or greatest cumulative difficulty chain. By summing the difficulty recorded in each block in a chain, a node can calculate the total amount of proof of work that has been expended to create that chain. As long as all nodes select the longest cumulative difficulty chain, the global bitcoin network eventually converges to a consistent state. Forks occur as temporary inconsistencies between versions of the blockchain, which are resolved by eventual reconvergence as more blocks are added to one of the forks.

In the next few diagrams, we follow the progress of a "fork" event across the network. The diagram is a simplified representation of bitcoin as a global network. In reality, the bitcoin network's topology is not organized geographically. Rather, it forms a mesh network of interconnected nodes, which might be located very far from each other geographically. The representation of a geographic

topology is a simplification used for the purposes of illustrating a fork. In the real bitcoin network, the "distance" between nodes is measured in "hops" from node to node, not on their physical location. For illustration purposes, different blocks are shown as different colors, spreading across the network and coloring the connections they traverse.

In the first diagram ([Visualization of a blockchain fork event—before the fork](#)), the network has a unified perspective of the blockchain, with the blue block as the tip of the main chain.



Figure 57. Visualization of a blockchain fork event—before the fork

A "fork" occurs whenever there are two candidate blocks competing to form the longest blockchain. This occurs under normal conditions whenever two miners solve the proof-of-work algorithm within a short period of time from each other. As both miners discover a solution for their respective candidate blocks, they immediately broadcast their own "winning" block to their immediate neighbors who begin propagating the block across the network. Each node that receives a valid block will incorporate it into its blockchain, extending the blockchain by one block. If that node later sees another candidate block extending the same parent, it connects the second candidate on a secondary chain. As a result, some nodes will "see" one candidate block first, while other nodes will see the other candidate block and two competing versions of the blockchain will emerge.

In [Visualization of a blockchain fork event: two blocks found simultaneously](#), we see two miners who mine two different blocks almost simultaneously. Both of these blocks are children of the blue block, meant to extend the chain by building on top of the blue block. To help us track it, one is visualized as a red block originating from Canada, and the other is marked as a green block originating from Australia.

Let's assume, for example, that a miner in Canada finds a proof-of-work solution for a block "red" that extends the blockchain, building on top of the parent block "blue." Almost simultaneously, an Australian miner who was also extending block "blue" finds a solution for block "green," his candidate block. Now, there are two possible blocks, one we call "red," originating in Canada, and one we call "green," originating in Australia. Both blocks are valid, both blocks contain a valid solution to the proof of work, and both blocks extend the same parent. Both blocks likely contain

most of the same transactions, with only perhaps a few differences in the order of transactions.

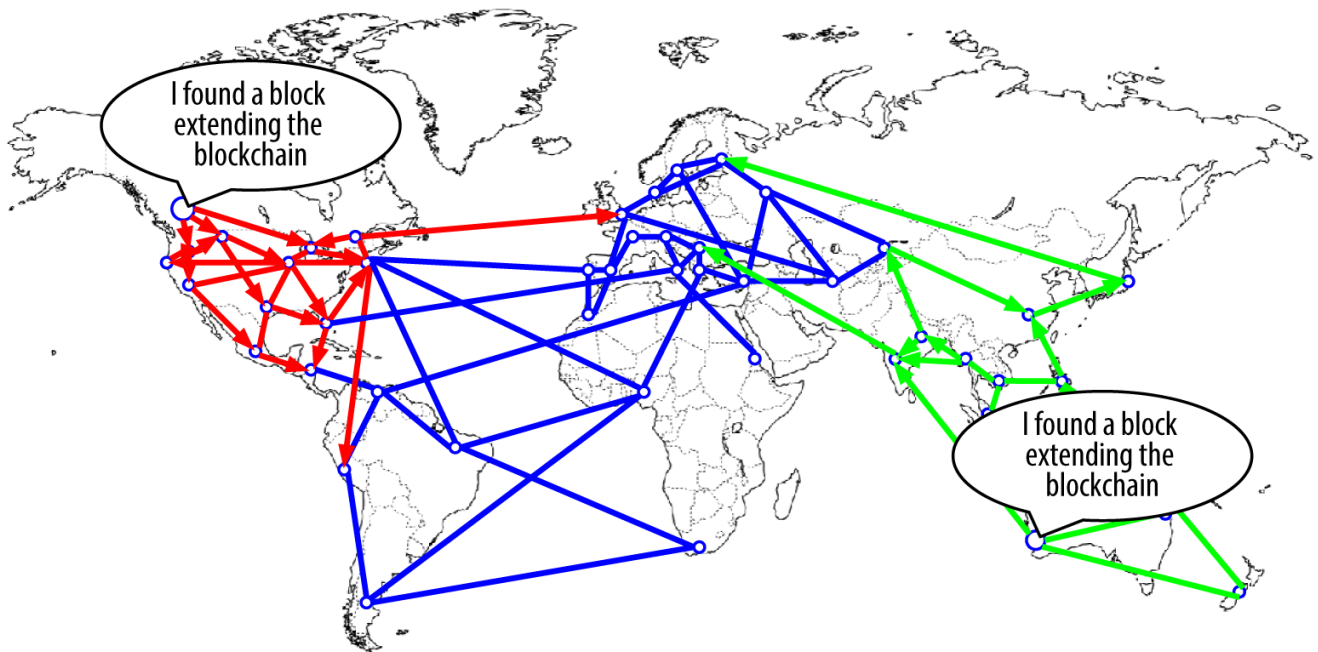


Figure 58. Visualization of a blockchain fork event: two blocks found simultaneously

As the two blocks propagate, some nodes receive block "red" first and some receive block "green" first. As shown in [Visualization of a blockchain fork event: two blocks propagate, splitting the network](#), the network splits into two different perspectives of the blockchain, one side topped with a red block, the other with a green block.

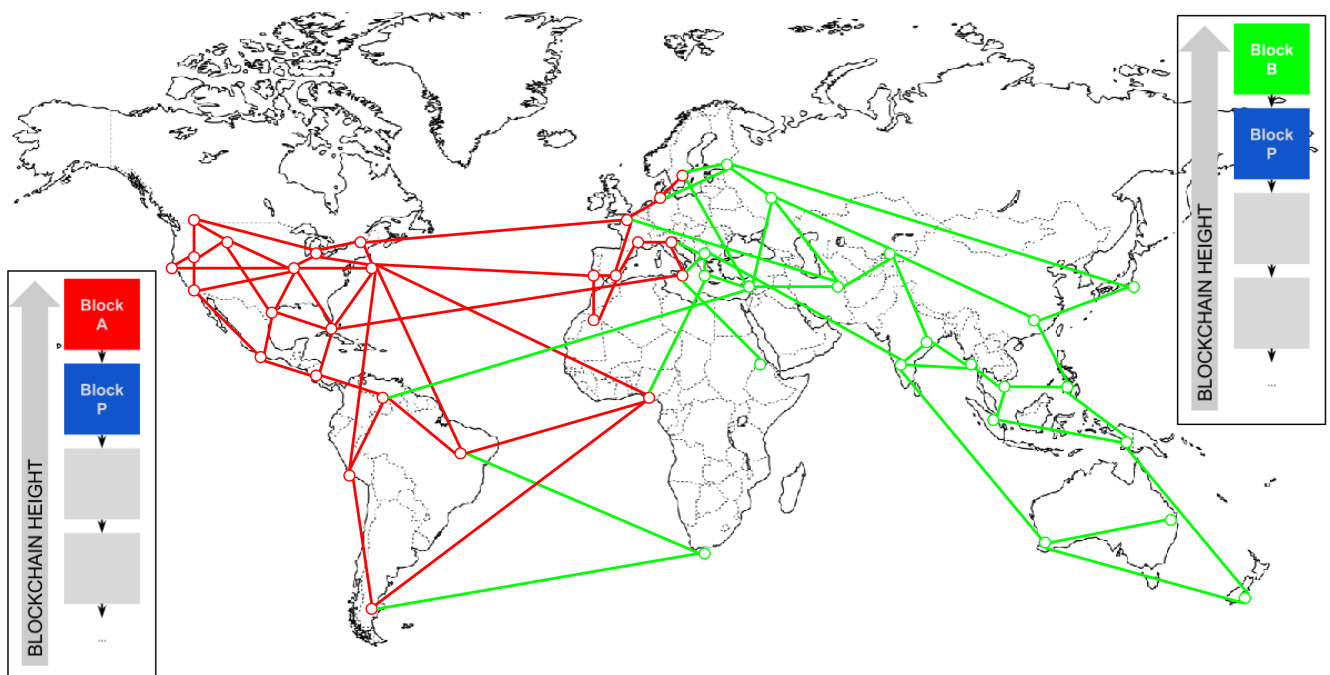


Figure 59. Visualization of a blockchain fork event: two blocks propagate, splitting the network

From that moment, the bitcoin network nodes closest (topologically, not geographically) to the Canadian node will hear about block "red" first and will create a new greatest-cumulative-difficulty blockchain with "red" as the last block in the chain (e.g., blue-red), ignoring the candidate block "green" that arrives a bit later. Meanwhile, nodes closer to the Australian node will take that block as the winner and extend the blockchain with "green" as the last block (e.g., blue-green), ignoring

"red" when it arrives a few seconds later. Any miners that saw "red" first will immediately build candidate blocks that reference "red" as the parent and start trying to solve the proof of work for these candidate blocks. The miners that accepted "green" instead will start building on top of "green" and extending that chain.

Forks are almost always resolved within one block. As part of the network's hashing power is dedicated to building on top of "red" as the parent, another part of the hashing power is focused on building on top of "green." Even if the hashing power is almost evenly split, it is likely that one set of miners will find a solution and propagate it before the other set of miners have found any solutions. Let's say, for example, that the miners building on top of "green" find a new block "pink" that extends the chain (e.g., blue-green-pink). They immediately propagate this new block and the entire network sees it as a valid solution as shown in [Visualization of a blockchain fork event: a new block extends one fork](#).

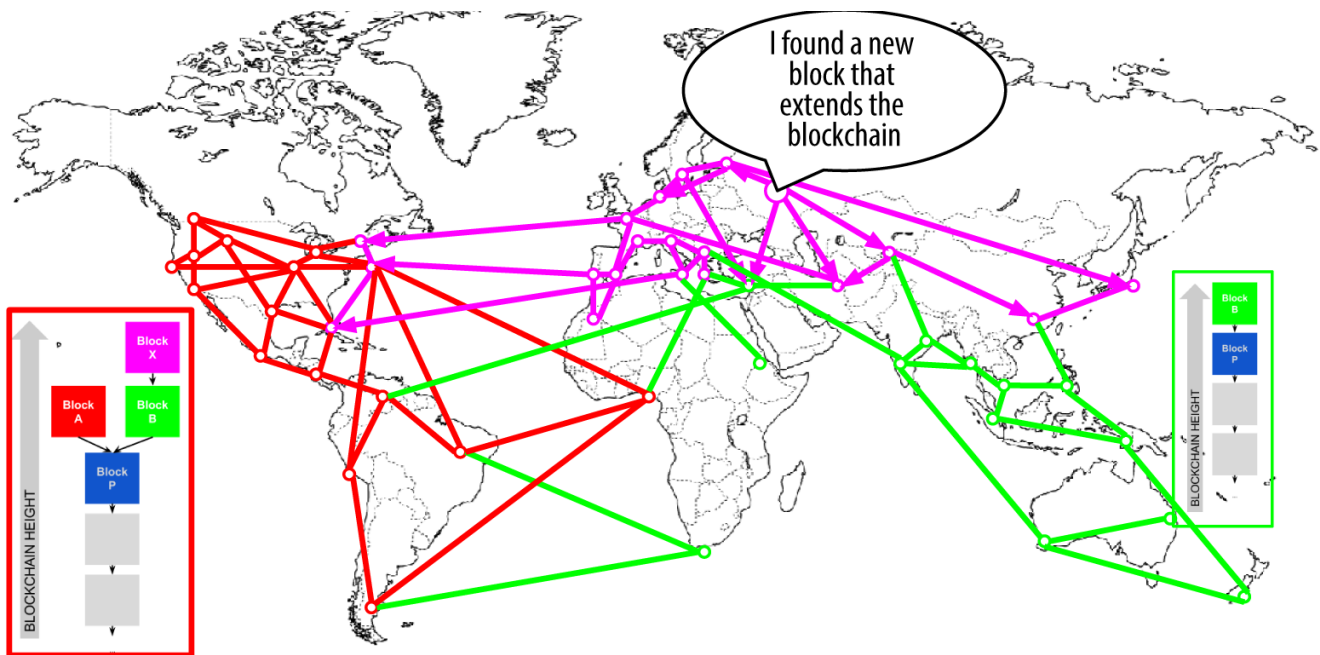


Figure 60. Visualization of a blockchain fork event: a new block extends one fork

All nodes that had chosen "green" as the winner in the previous round will simply extend the chain one more block. The nodes that chose "red" as the winner, however, will now see two chains: blue-green-pink and blue-red. The chain blue-green-pink is now longer (more cumulative difficulty) than the chain blue-red. As a result, those nodes will set the chain blue-green-pink as main chain and change the blue-red chain to being a secondary chain, as shown in [Visualization of a blockchain fork event: the network reconverges on a new longest chain](#). This is a chain reconvergence, because those nodes are forced to revise their view of the blockchain to incorporate the new evidence of a longer chain. Any miners working on extending the chain blue-red will now stop that work because their candidate block is an "orphan," as its parent "red" is no longer on the longest chain. The transactions within "red" are queued up again for processing in the next block, because that block is no longer in the main chain. The entire network re-converges on a single blockchain blue-green-pink, with "pink" as the last block in the chain. All miners immediately start working on candidate blocks that reference "pink" as their parent to extend the blue-green-pink chain.

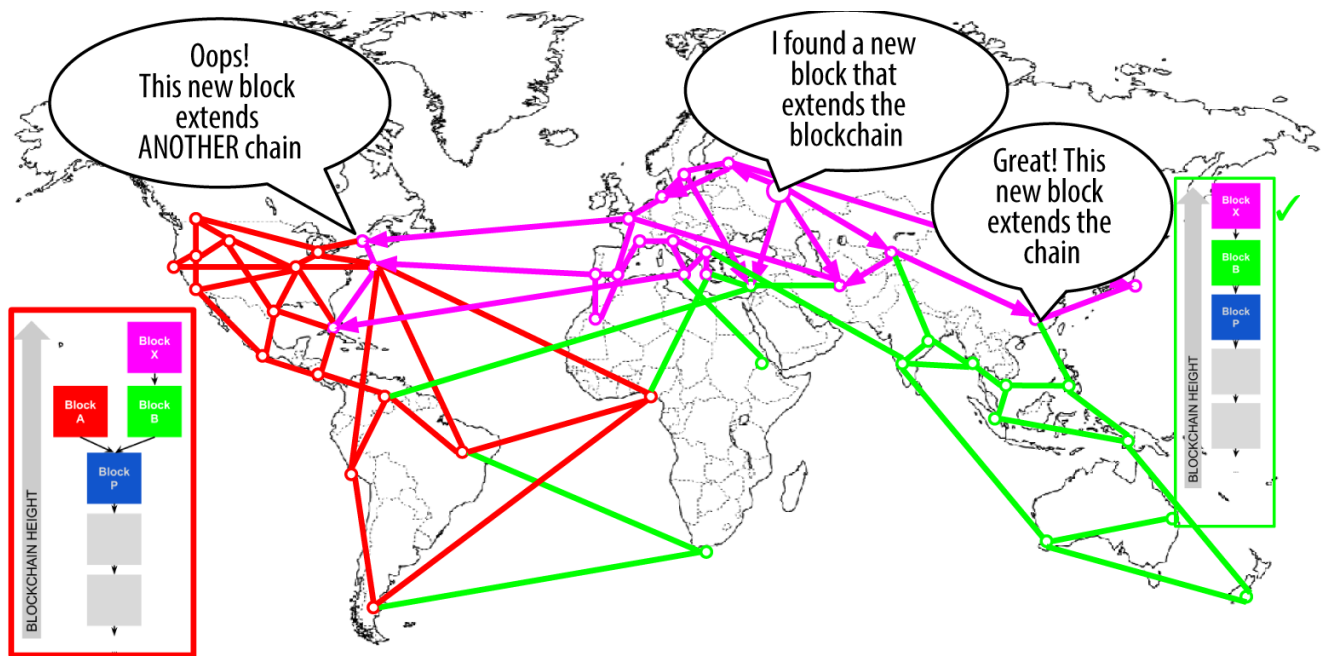


Figure 61. Visualization of a blockchain fork event: the network reconverges on a new longest chain

It is theoretically possible for a fork to extend to two blocks, if two blocks are found almost simultaneously by miners on opposite "sides" of a previous fork. However, the chance of that happening is very low. Whereas a one-block fork might occur every week, a two-block fork is exceedingly rare.

Bitcoin's block interval of 10 minutes is a design compromise between fast confirmation times (settlement of transactions) and the probability of a fork. A faster block time would make transactions clear faster but lead to more frequent blockchain forks, whereas a slower block time would decrease the number of forks but make settlement slower.

Mining and the Hashing Race

Bitcoin mining is an extremely competitive industry. The hashing power has increased exponentially every year of bitcoin's existence. Some years the growth has reflected a complete change of technology, such as in 2010 and 2011 when many miners switched from using CPU mining to GPU mining and field programmable gate array (FPGA) mining. In 2013 the introduction of ASIC mining lead to another giant leap in mining power, by placing the SHA256 function directly on silicon chips specialized for the purpose of mining. The first such chips could deliver more mining power in a single box than the entire bitcoin network in 2010.

The following list shows the total hashing power of the bitcoin network, over the first five years of operation:

2009

0.5 MH/sec–8 MH/sec (16% growth)

2010

8 MH/sec–116 GH/sec (14,500% growth)

2011

16 GH/sec–9 TH/sec (562% growth)

2012

9 TH/sec–23 TH/sec (2.5% growth)

2013

23 TH/sec–10 PH/sec (450% growth)

2014

10 PH/sec–150 PH/sec in August (15% growth)

In the chart in [Total hashing power, gigahashes per second, over two years](#), we see the bitcoin network's hashing power increase over the past two years. As you can see, the competition between miners and the growth of bitcoin has resulted in an exponential increase in the hashing power (total hashes per second across the network).



Figure 62. Total hashing power, gigahashes per second, over two years

As the amount of hashing power applied to mining bitcoin has exploded, the difficulty has risen to match it. The difficulty metric in the chart shown in [Bitcoin's mining difficulty metric, over two years](#) is measured as a ratio of current difficulty over minimum difficulty (the difficulty of the first block).

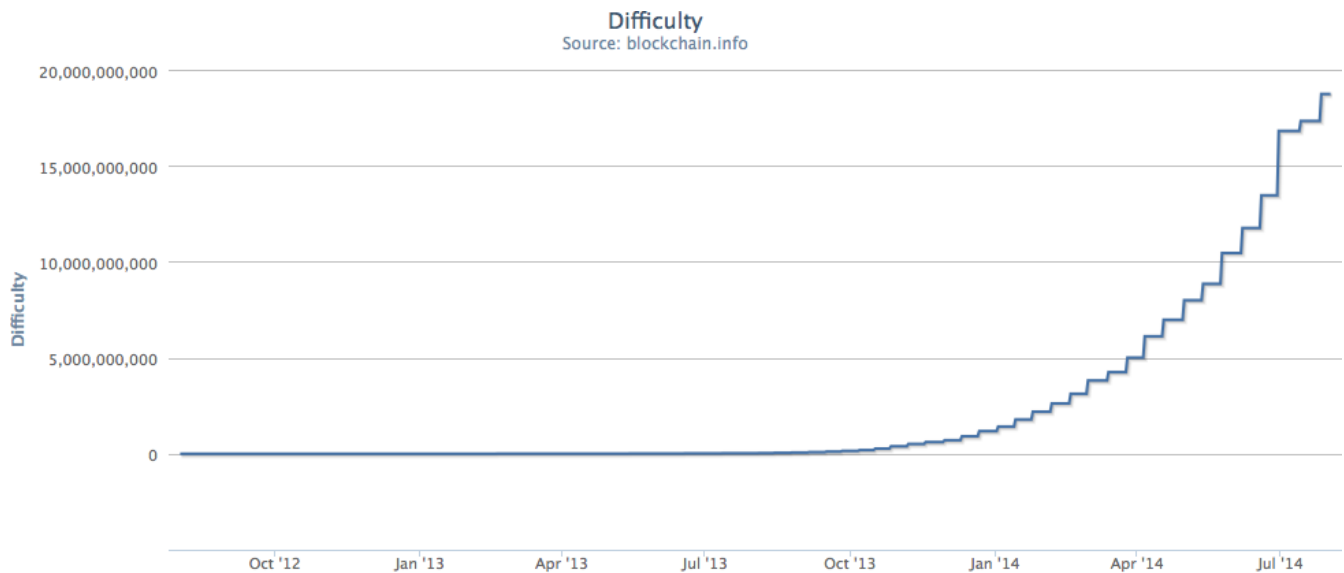


Figure 63. Bitcoin's mining difficulty metric, over two years

In the last two years, the ASIC mining chips have become increasingly denser, approaching the cutting edge of silicon fabrication with a feature size (resolution) of 22 nanometers (nm). Currently, ASIC manufacturers are aiming to overtake general-purpose CPU chip manufacturers, designing chips with a feature size of 16nm, because the profitability of mining is driving this industry even faster than general computing. There are no more giant leaps left in bitcoin mining, because the industry has reached the forefront of Moore's Law, which stipulates that computing density will double approximately every 18 months. Still, the mining power of the network continues to advance at an exponential pace as the race for higher density chips is matched with a race for higher density data centers where thousands of these chips can be deployed. It's no longer about how much mining can be done with one chip, but how many chips can be squeezed into a building, while still dissipating the heat and providing adequate power.

The Extra Nonce Solution

Since 2012, bitcoin mining has evolved to resolve a fundamental limitation in the structure of the block header. In the early days of bitcoin, a miner could find a block by iterating through the nonce until the resulting hash was below the target. As difficulty increased, miners often cycled through all 4 billion values of the nonce without finding a block. However, this was easily resolved by updating the block timestamp to account for the elapsed time. Because the timestamp is part of the header, the change would allow miners to iterate through the values of the nonce again with different results. Once mining hardware exceeded 4 GH/sec, however, this approach became increasingly difficult because the nonce values were exhausted in less than a second. As ASIC mining equipment started pushing and then exceeding the TH/sec hash rate, the mining software needed more space for nonce values in order to find valid blocks. The timestamp could be stretched a bit, but moving it too far into the future would cause the block to become invalid. A new source of "change" was needed in the block header. The solution was to use the coinbase transaction as a source of extra nonce values. Because the coinbase script can store between 2 and 100 bytes of data, miners started using that space as extra nonce space, allowing them to explore a much larger range of block header values to find valid blocks. The coinbase transaction is included in the merkle tree, which means that any change in the coinbase script causes the merkle root to change. Eight bytes of extra nonce, plus the 4 bytes of "standard" nonce allow miners to explore a total 2^{96} (8 followed by 28 zeros) possibilities *per second* without having to modify the timestamp. If, in the future, miners

could run through all these possibilities, they could then modify the timestamp. There is also more space in the coinbase script for future expansion of the extra nonce space.

Mining Pools

In this highly competitive environment, individual miners working alone (also known as solo miners) don't stand a chance. The likelihood of them finding a block to offset their electricity and hardware costs is so low that it represents a gamble, like playing the lottery. Even the fastest consumer ASIC mining system cannot keep up with commercial systems that stack tens of thousands of these chips in giant warehouses near hydro-electric power stations. Miners now collaborate to form mining pools, pooling their hashing power and sharing the reward among thousands of participants. By participating in a pool, miners get a smaller share of the overall reward, but typically get rewarded every day, reducing uncertainty.

Let's look at a specific example. Assume a miner has purchased mining hardware with a combined hashing rate of 6,000 gigahashes per second (GH/s), or 6 TH/s. In August of 2014 this equipment costs approximately \$10,000. The hardware consumes 3 kilowatts (kW) of electricity when running, 72 kW-hours a day, at a cost of \$7 or \$8 per day on average. At current bitcoin difficulty, the miner will be able to solo mine a block approximately once every 155 days, or every 5 months. If the miner does find a single block in that timeframe, the payout of 25 bitcoins, at approximately \$600 per bitcoin, will result in a single payout of \$15,000, which will cover the entire cost of the hardware and the electricity consumed over the time period, leaving a net profit of approximately \$3,000. However, the chance of finding a block in a five-month period depends on the miner's luck. He might find two blocks in five months and make a very large profit. Or he might not find a block for 10 months and suffer a financial loss. Even worse, the difficulty of the bitcoin proof-of-work algorithm is likely to go up significantly over that period, at the current rate of growth of hashing power, meaning the miner has, at most, six months to break even before the hardware is effectively obsolete and must be replaced by more powerful mining hardware. If this miner participates in a mining pool, instead of waiting for a once-in-five-months \$15,000 windfall, he will be able to earn approximately \$500 to \$750 per week. The regular payouts from a mining pool will help him amortize the cost of hardware and electricity over time without taking an enormous risk. The hardware will still be obsolete in six to nine months and the risk is still high, but the revenue is at least regular and reliable over that period.

Mining pools coordinate many hundreds or thousands of miners, over specialized pool-mining protocols. The individual miners configure their mining equipment to connect to a pool server, after creating an account with the pool. Their mining hardware remains connected to the pool server while mining, synchronizing their efforts with the other miners. Thus, the pool miners share the effort to mine a block and then share in the rewards.

Successful blocks pay the reward to a pool bitcoin address, rather than individual miners. The pool server will periodically make payments to the miners' bitcoin addresses, once their share of the rewards has reached a certain threshold. Typically, the pool server charges a percentage fee of the rewards for providing the pool-mining service.

Miners participating in a pool split the work of searching for a solution to a candidate block, earning "shares" for their mining contribution. The mining pool sets a lower difficulty target for earning a share, typically more than 1,000 times easier than the bitcoin network's difficulty. When someone in the pool successfully mines a block, the reward is earned by the pool and then shared

with all miners in proportion to the number of shares they contributed to the effort.

Pools are open to any miner, big or small, professional or amateur. A pool will therefore have some participants with a single small mining machine, and others with a garage full of high-end mining hardware. Some will be mining with a few tens of a kilowatt of electricity, others will be running a data center consuming a megawatt of power. How does a mining pool measure the individual contributions, so as to fairly distribute the rewards, without the possibility of cheating? The answer is to use bitcoin's proof-of-work algorithm to measure each pool miner's contribution, but set at a lower difficulty so that even the smallest pool miners win a share frequently enough to make it worthwhile to contribute to the pool. By setting a lower difficulty for earning shares, the pool measures the amount of work done by each miner. Each time a pool miner finds a block header hash that is less than the pool difficulty, she proves she has done the hashing work to find that result. More importantly, the work to find shares contributes, in a statistically measurable way, to the overall effort to find a hash lower than the bitcoin network's target. Thousands of miners trying to find low-value hashes will eventually find one low enough to satisfy the bitcoin network target.

Let's return to the analogy of a dice game. If the dice players are throwing dice with a goal of throwing less than four (the overall network difficulty), a pool would set an easier target, counting how many times the pool players managed to throw less than eight. When pool players throw less than eight (the pool share target), they earn shares, but they don't win the game because they don't achieve the game target (less than four). The pool players will achieve the easier pool target much more often, earning them shares very regularly, even when they don't achieve the harder target of winning the game. Every now and then, one of the pool players will throw a combined dice throw of less than four and the pool wins. Then, the earnings can be distributed to the pool players based on the shares they earned. Even though the target of eight-or-less wasn't winning, it was a fair way to measure dice throws for the players, and it occasionally produces a less-than-four throw.

Similarly, a mining pool will set a pool difficulty that will ensure that an individual pool miner can find block header hashes that are less than the pool difficulty quite often, earning shares. Every now and then, one of these attempts will produce a block header hash that is less than the bitcoin network target, making it a valid block and the whole pool wins.

Managed pools

Most mining pools are "managed," meaning that there is a company or individual running a pool server. The owner of the pool server is called the *pool operator*, and he charges pool miners a percentage fee of the earnings.

The pool server runs specialized software and a pool-mining protocol that coordinates the activities of the pool miners. The pool server is also connected to one or more full bitcoin nodes and has direct access to a full copy of the blockchain database. This allows the pool server to validate blocks and transactions on behalf of the pool miners, relieving them of the burden of running a full node. For pool miners, this is an important consideration, because a full node requires a dedicated computer with at least 15 to 20 GB of persistent storage (disk) and at least 2 GB of memory (RAM). Furthermore, the bitcoin software running on the full node needs to be monitored, maintained, and upgraded frequently. Any downtime caused by a lack of maintenance or lack of resources will hurt the miner's profitability. For many miners, the ability to mine without running a full node is another big benefit of joining a managed pool.

Pool miners connect to the pool server using a mining protocol such as Stratum (STM) or GetBlockTemplate (GBT). An older standard called GetWork (GWK) has been mostly obsolete since late 2012, because it does not easily support mining at hash rates above 4 GH/s. Both the STM and GBT protocols create block *templates* that contain a template of a candidate block header. The pool server constructs a candidate block by aggregating transactions, adding a coinbase transaction (with extra nonce space), calculating the merkle root, and linking to the previous block hash. The header of the candidate block is then sent to each of the pool miners as a template. Each pool miner then mines using the block template, at a lower difficulty than the bitcoin network difficulty, and sends any successful results back to the pool server to earn shares.

P2Pool

Managed pools create the possibility of cheating by the pool operator, who might direct the pool effort to double-spend transactions or invalidate blocks (see [Consensus Attacks](#)). Furthermore, centralized pool servers represent a single-point-of-failure. If the pool server is down or is slowed by a denial-of-service attack, the pool miners cannot mine. In 2011, to resolve these issues of centralization, a new pool mining method was proposed and implemented: P2Pool is a peer-to-peer mining pool, without a central operator.

P2Pool works by decentralizing the functions of the pool server, implementing a parallel blockchain-like system called a *share chain*. A share chain is a blockchain running at a lower difficulty than the bitcoin blockchain. The share chain allows pool miners to collaborate in a decentralized pool, by mining shares on the share chain at a rate of one share block every 30 seconds. Each of the blocks on the share chain records a proportionate share reward for the pool miners who contribute work, carrying the shares forward from the previous share block. When one of the share blocks also achieves the difficulty target of the bitcoin network, it is propagated and included on the bitcoin blockchain, rewarding all the pool miners who contributed to all the shares that preceded the winning share block. Essentially, instead of a pool server keeping track of pool miner shares and rewards, the share chain allows all pool miners to keep track of all shares using a decentralized consensus mechanism like bitcoin's blockchain consensus mechanism.

P2Pool mining is more complex than pool mining because it requires that the pool miners run a dedicated computer with enough disk space, memory, and Internet bandwidth to support a full bitcoin node and the P2Pool node software. P2Pool miners connect their mining hardware to their local P2Pool node, which simulates the functions of a pool server by sending block templates to the mining hardware. On P2Pool, individual pool miners construct their own candidate blocks, aggregating transactions much like solo miners, but then mine collaboratively on the share chain. P2Pool is a hybrid approach that has the advantage of much more granular payouts than solo mining, but without giving too much control to a pool operator like managed pools.

Recently, participation in P2Pool has increased significantly as mining concentration in mining pools has approached levels that create concerns of a 51% attack (see [Consensus Attacks](#)). Further development of the P2Pool protocol continues with the expectation of removing the need for running a full node and therefore making decentralized mining even easier to use.

Even though P2Pool reduces the concentration of power by mining pool operators, it is conceivably vulnerable to 51% attacks against the share chain itself. A much broader adoption of P2Pool does not solve the 51% attack problem for bitcoin itself. Rather, P2Pool makes bitcoin more robust overall, as part of a diversified mining ecosystem.

Consensus Attacks

Bitcoin's consensus mechanism is, at least theoretically, vulnerable to attack by miners (or pools) that attempt to use their hashing power to dishonest or destructive ends. As we saw, the consensus mechanism depends on having a majority of the miners acting honestly out of self-interest. However, if a miner or group of miners can achieve a significant share of the mining power, they can attack the consensus mechanism so as to disrupt the security and availability of the bitcoin network.

It is important to note that consensus attacks can only affect future consensus, or at best the most recent past (tens of blocks). Bitcoin's ledger becomes more and more immutable as time passes. While in theory, a fork can be achieved at any depth, in practice, the computing power needed to force a very deep fork is immense, making old blocks practically immutable. Consensus attacks also do not affect the security of the private keys and signing algorithm (ECDSA). A consensus attack cannot steal bitcoins, spend bitcoins without signatures, redirect bitcoins, or otherwise change past transactions or ownership records. Consensus attacks can only affect the most recent blocks and cause denial-of-service disruptions on the creation of future blocks.

One attack scenario against the consensus mechanism is called the "51% attack." In this scenario a group of miners, controlling a majority (51%) of the total network's hashing power, collude to attack bitcoin. With the ability to mine the majority of the blocks, the attacking miners can cause deliberate "forks" in the blockchain and double-spend transactions or execute denial-of-service attacks against specific transactions or addresses. A fork/double-spend attack is one where the attacker causes previously confirmed blocks to be invalidated by forking below them and re-converging on an alternate chain. With sufficient power, an attacker can invalidate six or more blocks in a row, causing transactions that were considered immutable (six confirmations) to be invalidated. Note that a double-spend can only be done on the attacker's own transactions, for which the attacker can produce a valid signature. Double-spending one's own transactions is profitable if by invalidating a transaction the attacker can get a nonreversible exchange payment or product without paying for it.

Let's examine a practical example of a 51% attack. In the first chapter, we looked at a transaction between Alice and Bob for a cup of coffee. Bob, the cafe owner, is willing to accept payment for cups of coffee without waiting for confirmation (mining in a block), because the risk of a double-spend on a cup of coffee is low in comparison to the convenience of rapid customer service. This is similar to the practice of coffee shops that accept credit card payments without a signature for amounts below \$25, because the risk of a credit-card chargeback is low while the cost of delaying the transaction to obtain a signature is comparatively larger. In contrast, selling a more expensive item for bitcoin runs the risk of a double-spend attack, where the buyer broadcasts a competing transaction that spends the same inputs (UTXO) and cancels the payment to the merchant. A double-spend attack can happen in two ways: either before a transaction is confirmed, or if the attacker takes advantage of a blockchain fork to undo several blocks. A 51% attack allows attackers to double-spend their own transactions in the new chain, thus undoing the corresponding transaction in the old chain.

In our example, malicious attacker Mallory goes to Carol's gallery and purchases a beautiful triptych painting depicting Satoshi Nakamoto as Prometheus. Carol sells "The Great Fire" paintings for \$250,000 in bitcoin, to Mallory. Instead of waiting for six or more confirmations on the

transaction, Carol wraps and hands the paintings to Mallory after only one confirmation. Mallory works with an accomplice, Paul, who operates a large mining pool, and the accomplice launches a 51% attack as soon as Mallory's transaction is included in a block. Paul directs the mining pool to re-mine the same block height as the block containing Mallory's transaction, replacing Mallory's payment to Carol with a transaction that double-spends the same input as Mallory's payment. The double-spend transaction consumes the same UTXO and pays it back to Mallory's wallet, instead of paying it to Carol, essentially allowing Mallory to keep the bitcoin. Paul then directs the mining pool to mine an additional block, so as to make the chain containing the double-spend transaction longer than the original chain (causing a fork below the block containing Mallory's transaction). When the blockchain fork resolves in favor of the new (longer) chain, the double-spent transaction replaces the original payment to Carol. Carol is now missing the three paintings and also has no bitcoin payment. Throughout all this activity, Paul's mining pool participants might remain blissfully unaware of the double-spend attempt, because they mine with automated miners and cannot monitor every transaction or block.

To protect against this kind of attack, a merchant selling large-value items must wait at least six confirmations before giving the product to the buyer. Alternatively, the merchant should use an escrow multi-signature account, again waiting for several confirmations after the escrow account is funded. The more confirmations elapse, the harder it becomes to invalidate a transaction with a 51% attack. For high-value items, payment by bitcoin will still be convenient and efficient even if the buyer has to wait 24 hours for delivery, which would correspond to approximately 144 confirmations.

In addition to a double-spend attack, the other scenario for a consensus attack is to deny service to specific bitcoin participants (specific bitcoin addresses). An attacker with a majority of the mining power can simply ignore specific transactions. If they are included in a block mined by another miner, the attacker can deliberately fork and re-mine that block, again excluding the specific transactions. This type of attack can result in a sustained denial of service against a specific address or set of addresses for as long as the attacker controls the majority of the mining power.

Despite its name, the 51% attack scenario doesn't actually require 51% of the hashing power. In fact, such an attack can be attempted with a smaller percentage of the hashing power. The 51% threshold is simply the level at which such an attack is almost guaranteed to succeed. A consensus attack is essentially a tug-of-war for the next block and the "stronger" group is more likely to win. With less hashing power, the probability of success is reduced, because other miners control the generation of some blocks with their "honest" mining power. One way to look at it is that the more hashing power an attacker has, the longer the fork he can deliberately create, the more blocks in the recent past he can invalidate, or the more blocks in the future he can control. Security research groups have used statistical modeling to claim that various types of consensus attacks are possible with as little as 30% of the hashing power.

The massive increase of total hashing power has arguably made bitcoin impervious to attacks by a single miner. There is no possible way for a solo miner to control more than a small percentage of the total mining power. However, the centralization of control caused by mining pools has introduced the risk of for-profit attacks by a mining pool operator. The pool operator in a managed pool controls the construction of candidate blocks and also controls which transactions are included. This gives the pool operator the power to exclude transactions or introduce double-spend transactions. If such abuse of power is done in a limited and subtle way, a pool operator could conceivably profit from a consensus attack without being noticed.

Not all attackers will be motivated by profit, however. One potential attack scenario is where an attacker intends to disrupt the bitcoin network without the possibility of profiting from such disruption. A malicious attack aimed at crippling bitcoin would require enormous investment and covert planning, but could conceivably be launched by a well-funded, most likely state-sponsored, attacker. Alternatively, a well-funded attacker could attack bitcoin's consensus by simultaneously amassing mining hardware, compromising pool operators and attacking other pools with denial-of-service. All of these scenarios are theoretically possible, but increasingly impractical as the bitcoin network's overall hashing power continues to grow exponentially.

Undoubtedly, a serious consensus attack would erode confidence in bitcoin in the short term, possibly causing a significant price decline. However, the bitcoin network and software are constantly evolving, so consensus attacks would be met with immediate countermeasures by the bitcoin community, making bitcoin hardier, stealthier, and more robust than ever.

Alternative Chains, Currencies, and Applications

Bitcoin was the result of 20 years of research in distributed systems and currencies and brought a revolutionary new technology into the space: the decentralized consensus mechanism based on proof of work. This invention at the heart of bitcoin has ushered a wave of innovation in currencies, financial services, economics, distributed systems, voting systems, corporate governance, and contracts.

In this chapter we'll examine the many offshoots of the bitcoin and blockchain inventions: the alternative chains, currencies, and applications built since the introduction of this technology in 2009. Mostly, we will look at alternative coins, or *alt coins*, which are digital currencies implemented using the same design pattern as bitcoin, but with a completely separate blockchain and network.

For every alt coin mentioned in this chapter, 50 or more will go unmentioned, eliciting howls of anger from their creators and fans. The purpose of this chapter is not to evaluate or qualify alt coins, or even to mention the most significant ones based on some subjective assessment. Instead, we will highlight a few examples that show the breadth and variety of the ecosystem, noting the first-of-a-kind for each innovation or significant differentiation. Some of the most interesting examples of alt coins are in fact complete failures from a monetary perspective. That perhaps makes them even more interesting for study and highlights the fact that this chapter is not to be used as an investment guide.

With new coins introduced every day, it would be impossible not to miss some important coin, perhaps the one that changes history. The rate of innovation is what makes this space so exciting and guarantees this chapter will be incomplete and out-of-date as soon as it is published.

A Taxonomy of Alternative Currencies and Chains

Bitcoin is an open source project, and its code has been used as the basis for many other software projects. The most common form of software spawned from bitcoin's source code are alternative decentralized currencies, or *alt coins*, which use the same basic building blocks to implement digital currencies.

There are a number of protocol layers implemented on top of bitcoin's blockchain. These *meta coins*, *meta chains*, or *blockchain apps* use the blockchain as an application platform or extend the bitcoin protocol by adding protocol layers. Examples include Colored Coins, Mastercoin, NXT, and Counterparty.

In the next section we will examine a few notable alt coins, such as Litecoin, Dogecoin, Freicoin, Primecoin, Peercoin, Darkcoin, and Zerocoin. These alt coins are notable for historical reasons or because they are good examples for a specific type of alt coin innovation, not because they are the most valuable or "best" alt coins.

In addition to the alt coins, there are also a number of alternative blockchain implementations that are not really "coins," which I call *alt chains*. These alt chains implement a consensus algorithm and

distributed ledger as a platform for contracts, name registration, or other applications. Alt chains use the same basic building blocks and sometimes also use a currency or token as a payment mechanism, but their primary purpose is not currency. We will look at Namecoin and Ethereum as examples of alt chains.

Finally, there are a number of bitcoin contenders that offer digital currency or digital payment networks, but without using a decentralized ledger or consensus mechanism based on proof of work, such as Ripple and others. These non-blockchain technologies are outside the scope of this book and will not be covered in this chapter.

Meta Coin Platforms

Meta coins and meta chains are software layers implemented on top of bitcoin, either implementing a currency-inside-a-currency, or a platform/protocol overlay inside the bitcoin system. These function layers extend the core bitcoin protocol and add features and capabilities by encoding additional data inside bitcoin transactions and bitcoin addresses. The first implementations of meta coins used various hacks to add metadata to the bitcoin blockchain, such as using bitcoin addresses to encode data or using unused transaction fields (e.g., the transaction sequence field) to encode metadata about the added protocol layer. Since the introduction of the `OP_RETURN` transaction scripting opcode, the meta coins have been able to record metadata more directly in the blockchain, and most are migrating to using that instead.

Colored Coins

Colored coins is a meta protocol that overlays information on small amounts of bitcoin. A "colored" coin is an amount of bitcoin repurposed to express another asset. Imagine, for example, taking a \$1 note and putting a stamp on it that said, "This is a 1 share certificate of Acme Inc." Now the \$1 serves two purposes: it is a currency note and also a share certificate. Because it is more valuable as a share, you would not want to use it to buy candy, so effectively it is no longer useful as currency. Colored coins work in the same way by converting a specific, very small amount of bitcoin into a traded certificate that represents another asset. The term "color" refers to the idea of giving special meaning through the addition of an attribute such as a color—it is a metaphor, not an actual color association. There are no colors in colored coins.

Colored coins are managed by specialized wallets that record and interpret the metadata attached to the colored bitcoins. Using such a wallet, the user will convert an amount of bitcoins from uncolored currency into colored coins by adding a label that has a special meaning. For example, a label could represent stock certificates, coupons, real property, commodities, or collectible tokens. It is entirely up to the users of colored coins to assign and interpret the meaning of the "color" associated with specific coins. To color the coins, the user defines the associated metadata, such as the type of issuance, whether it can be subdivided into smaller units, a symbol and description, and other related information. Once colored, these coins can be bought and sold, subdivided, and aggregated, and receive dividend payments. The colored coins can also be "uncolored" by removing the special association and redeemed for their face value in bitcoin.

To demonstrate the use of colored coins, we have created a set of 20 colored coins with symbol "MasterBTC" that represent coupons for a free copy of this book shown in [The metadata profile of the colored coins recorded as a coupon for a free copy of the book](#). Each unit of MasterBTC,

represented by these colored coins, can now be sold or given to any bitcoin user with a colored-coin-capable wallet, who can then transfer them to others or redeem them with the issuer for a free copy of the book. This example of colored coins can be seen [here](#).

Example 31. The metadata profile of the colored coins recorded as a coupon for a free copy of the book

```
{
  "source_addresses": [
    "3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD"
  ],
  "contract_url":
  "https://www.coinprism.info/asset/3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD",
  "name_short": "MasterBTC",
  "name": "Free copy of \"Mastering Bitcoin\"",
  "issuer": "Andreas M. Antonopoulos",
  "description": "This token is redeemable for a free copy of the book \"Mastering Bitcoin\"",
  "description_mime": "text/x-markdown; charset=UTF-8",
  "type": "Other",
  "divisibility": 0,
  "link_to_website": false,
  "icon_url": null,
  "image_url": null,
  "version": "1.0"
}
```

Mastercoin

Mastercoin is a protocol layer on top of bitcoin that supports a platform for various applications extending the bitcoin system. Mastercoin uses the currency MST as a token for conducting Mastercoin transactions but it is not primarily a currency. Rather, it is a platform for building other things, such as user currencies, smart property tokens, de-centralized asset exchanges, and contracts. Think of Mastercoin as an application-layer protocol on top of bitcoin's financial transaction transport layer, just like HTTP runs on top of TCP.

Mastercoin operates primarily through transactions sent to and from a special bitcoin address called the "exodus" address (`1EXoDusjGwvnjZUyKkxZ4UHEf77z6A5S4P`), just like HTTP uses a specific TCP port (port 80) to differentiate its traffic from the rest of the TCP traffic. The Mastercoin protocol is gradually transitioning from using the specialized exodus address and multi-signatures to using the `OP_RETURN` bitcoin operator to encode transaction metadata.

Counterparty

Counterparty is another protocol layer implemented on top of bitcoin. Counterparty enables user currencies, tradable tokens, financial instruments, decentralized asset exchanges, and other features. Counterparty is implemented primarily using the `OP_RETURN` operator in bitcoin's scripting language to record metadata that enhances bitcoin transactions with additional meaning.

Counterparty uses the currency XCP as a token for conducting Counterparty transactions.

Alt Coins

The vast majority of alt coins are derived from bitcoin's source code, also known as "forks." Some are implemented "from scratch" based on the blockchain model but without using any of bitcoin's source code. Alt coins and alt chains (in the next section) are both separate implementations of blockchain technology and both forms use their own blockchain. The difference in the terms is to indicate that alt coins are primarily used as currency, whereas alt chains are used for other purposes, not primarily currency.

Strictly speaking, the first major "alt" fork of bitcoin's code was not an alt coin but the alt chain *Namecoin*, which we will discuss in the next section.

Based on the date of announcement, the first alt coin that was a fork of bitcoin appeared in August 2011; it was called *IXCoin*. IXCoin modified a few of the bitcoin parameters, specifically accelerating the creation of currency by increasing the reward to 96 coins per block.

In September 2011, *Tenebrix* was launched. Tenebrix was the first cryptocurrency to implement an alternative proof-of-work algorithm, namely *scrypt*, an algorithm originally designed for password stretching (brute-force resistance). The stated goal of Tenebrix was to make a coin that was resistant to mining with GPUs and ASICs, by using a memory-intensive algorithm. Tenebrix did not succeed as a currency, but it was the basis for Litecoin, which has enjoyed great success and has spawned hundreds of clones.

Litecoin, in addition to using *scrypt* as the proof-of-work algorithm, also implemented a faster block-generation time, targeted at 2.5 minutes instead of bitcoin's 10 minutes. The resulting currency is touted as "silver to bitcoin's gold" and is intended as a light-weight alternative currency. Due to the faster confirmation time and the 84 million total currency limit, many adherents of Litecoin believe it is better suited for retail transactions than bitcoin.

Alt coins continued to proliferate in 2011 and 2012, either based on bitcoin or on Litecoin. By 2013, there were 20 alt coins vying for position in the market. By the end of 2013, this number had exploded to 200, with 2013 quickly becoming the "year of the alt coins." The growth of alt coins continued in 2014, with more than 500 alt coins in existence at the time of writing. More than half the alt coins today are clones of Litecoin.

Creating an alt coin is easy, which is why there are now more than 500 of them. Most of the alt coins differ very slightly from bitcoin and do not offer anything worth studying. Many are in fact just attempts to enrich their creators. Among the copycats and pump-and-dump schemes, there are, however, some notable exceptions and very important innovations. These alt coins take radically different approaches or add significant innovation to bitcoin's design pattern. There are three primary areas where these alt coins differentiate from bitcoin:

- Different monetary policy
- Different proof of work or consensus mechanism
- Specific features, such as strong anonymity

For more information, see this [graphical timeline of alt coins and alt chains](#).

Evaluating an Alt Coin

With so many alt coins out there, how does one decide which ones are worthy of attention? Some alt coins attempt to achieve broad distribution and use as currencies. Others are laboratories for experimenting on different features and monetary models. Many are just get-rich-quick schemes by their creators. To evaluate alt coins, I look at their defining characteristics and their market metrics.

Here are some questions to ask about how well an alt coin differentiates from bitcoin:

- Does the alt coin introduce a significant innovation?
- Is the difference compelling enough to attract users away from bitcoin?
- Does the alt coin address an interesting niche market or application?
- Can the alt coin attract enough miners to be secured against consensus attacks?

Here are some of the key financial and market metrics to consider:

- What is the total market capitalization of alt coin?
- How many estimated users/wallets does the alt coin have?
- How many merchants accept the alt coin?
- How many daily transactions (volume) are executed on the alt coin?
- How much value is transacted daily?

In this chapter, we will concentrate primarily on the technical characteristics and innovation potential of alt coins represented by the first set of questions.

Monetary Parameter Alternatives: Litecoin, Dogecoin, Freicoin

Bitcoin has a few monetary parameters that give it distinctive characteristics of a deflationary fixed-issuance currency. It is limited to 21 million major currency units (or 21 quadrillion minor units), it has a geometrically declining issuance rate, and it has a 10-minute block "heartbeat," which controls the speed of transaction confirmation and currency generation. Many alt coins have tweaked the primary parameters to achieve different monetary policies. Among the hundreds of alt coins, some of the most notable examples include the following.

Litecoin

One of the first alt coins, released in 2011, Litecoin is the second most successful digital currency after bitcoin. Its primary innovations were the use of *script* as the proof-of-work algorithm (inherited from Tenebrix) and its faster/lighter currency parameters.

- Block generation time: 2.5 minutes
- Total currency: 84 million coins by 2140
- Consensus algorithm: Script proof of work
- Market capitalization: \$160 million in mid-2014

Dogecoin

Dogecoin was released in December 2013, based on a fork of Litecoin. Dogecoin is notable because it has a monetary policy of rapid issuance and a very high currency cap, to encourage spending and tipping. Dogecoin is also notable because it was started as a joke but became quite popular, with a large and active community, before declining rapidly in 2014.

- Block generation time: 60 seconds
- Total currency: 100,000,000,000 (100 billion) Doge by 2015
- Consensus algorithm: Scrypt proof of work
- Market capitalization: \$12 million in mid-2014

Freicoin

Freicoin was introduced in July 2012. It is a *demurrage currency*, meaning it has a negative interest rate for stored value. Value stored in Freicoin is assessed a 4.5% APR fee, to encourage consumption and discourage hoarding of money. Freicoin is notable in that it implements a monetary policy that is the exact opposite of Bitcoin's deflationary policy. Freicoin has not seen success as a currency, but it is an interesting example of the variety of monetary policies that can be expressed by alt coins.

- Block generation: 10 minutes
- Total currency: 100 million coins by 2140
- Consensus algorithm: SHA256 proof of work
- Market capitalization: \$130,000 in mid-2014

Consensus Innovation: Peercoin, Myriad, Blackcoin, Vericoin, NXT

Bitcoin's consensus mechanism is based on proof of work using the SHA256 algorithm. The first alt coins introduced scrypt as an alternative proof-of-work algorithm, as a way to make mining more CPU-friendly and less susceptible to centralization with ASICs. Since then, innovation in the consensus mechanism has continued at a frenetic pace. Several alt coins adopted a variety of algorithms such as scrypt, scrypt-N, Skein, Groestl, SHA3, X11, Blake, and others. Some alt coins combined multiple algorithms for proof of work. In 2013, we saw the invention of an alternative to proof of work, called *proof of stake*, which forms the basis of many modern alt coins.

Proof of stake is a system by which existing owners of a currency can "stake" currency as interest-bearing collateral. Somewhat like a certificate of deposit (CD), participants can reserve a portion of their currency holdings, while earning an investment return in the form of new currency (issued as interest payments) and transaction fees.

Peercoin

Peercoin was introduced in August 2012 and is the first alt coin to use a hybrid proof-of-work and proof-of-stake algorithm to issue new currency.

- Block generation: 10 minutes
- Total currency: No limit

- Consensus algorithm: (Hybrid) proof-of-stake with initial proof-of-work
- Market capitalization: \$14 million in mid-2014

Myriad

Myriad was introduced in February 2014 and is notable because it uses five different proof-of-work algorithms (SHA256d, Scrypt, Qubit, Skein, or Myriad-Groestl) simultaneously, with difficulty varying for each algorithm depending on miner participation. The intent is to make Myriad immune to ASIC specialization and centralization as well as much more resistant to consensus attacks, because multiple mining algorithms would have to be attacked simultaneously.

- Block generation: 30-second average (2.5 minutes target per mining algorithm)
- Total currency: 2 billion by 2024
- Consensus algorithm: Multi-algorithm proof-of-work
- Market capitalization: \$120,000 in mid-2014

Blackcoin

Blackcoin was introduced in February 2014 and uses a proof-of-stake consensus algorithm. It is also notable for introducing "multipools," a type of mining pool that can switch between different alt coins automatically, depending on profitability.

- Block generation: 1 minute
- Total currency: No limit
- Consensus algorithm: Proof-of-stake
- Market capitalization: \$3.7 million in mid-2014

VeriCoin

VeriCoin was launched in May 2014. It uses a proof-of-stake consensus algorithm with a variable interest rate that dynamically adjusts based on market forces of supply and demand. It also is the first alt coin featuring auto-exchange to bitcoin for payment in bitcoin from the wallet.

- Block generation: 1 minute
- Total currency: No limit
- Consensus algorithm: Proof-of-stake
- Market capitalization: \$1.1 million in mid-2014

NXT

NXT (pronounced "Next") is a "pure" proof-of-stake alt coin, in that it does not use proof-of-work mining. NXT is a from-scratch implementation of a cryptocurrency, not a fork of bitcoin or any other alt coins. NXT implements many advanced features, including a name registry (similar to Namecoin), a decentralized asset exchange (similar to Colored Coins), integrated decentralized and secure messaging (similar to Bitmessage), and stake delegation (to delegate proof-of-stake to others). NXT adherents call it a "next-generation" or 2.0 cryptocurrency.

- Block generation: 1 minute
- Total currency: No limit
- Consensus algorithm: Proof-of-stake
- Market capitalization: \$30 million in mid-2014

Dual-Purpose Mining Innovation: Primecoin, Curecoin, Gridcoin

Bitcoin's proof-of-work algorithm has just one purpose: securing the bitcoin network. Compared to traditional payment system security, the cost of mining is not very high. However, it has been criticized by many as being "wasteful." The next generation of alt coins attempt to address this concern. Dual-purpose proof-of-work algorithms solve a specific "useful" problem, while producing proof of work to secure the network. The risk of adding an external use to the currency's security is that it also adds external influence to the supply/demand curve.

Primecoin

Primecoin was announced in July 2013. Its proof-of-work algorithm searches for prime numbers, computing Cunningham and bi-twin prime chains. Prime numbers are useful in a variety of scientific disciplines. The Primecoin blockchain contains the discovered prime numbers, thereby producing a public record of scientific discovery in parallel to the public ledger of transactions.

- Block generation: 1 minute
- Total currency: No limit
- Consensus algorithm: Proof of work with prime number chain discovery
- Market capitalization: \$1.3 million in mid-2014

Curecoin

Curecoin was announced in May 2013. It combines a SHA256 proof-of-work algorithm with protein-folding research through the Folding@Home project. Protein folding is a computationally intensive simulation of biochemical interactions of proteins, used to discover new drug targets for curing diseases.

- Block generation: 10 minutes
- Total currency: No limit
- Consensus algorithm: Proof of work with protein-folding research
- Market capitalization: \$58,000 in mid-2014

Gridcoin

Gridcoin was introduced in October 2013. It supplements script-based proof of work with subsidies for participation in BOINC open grid computing. BOINC—Berkeley Open Infrastructure for Network Computing—is an open protocol for scientific research grid computing, which allows participants to share their spare computing cycles for a broad range of academic research computing. Gridcoin uses BOINC as a general-purpose computing platform, rather than to solve specific science problems such as prime numbers or protein folding.

- Block generation: 150 seconds
- Total currency: No limit
- Consensus algorithm: Proof-of-work with BOINC grid computing subsidy
- Market capitalization: \$122,000 in mid-2014

Anonymity-Focused Alt Coins: CryptoNote, Bytecoin, Monero, Zerocash/Zerocoin, Darkcoin

Bitcoin is often mistakenly characterized as "anonymous" currency. In fact, it is relatively easy to connect identities to bitcoin addresses and, using big-data analytics, connect addresses to each other to form a comprehensive picture of someone's bitcoin spending habits. Several alt coins aim to address this issue directly by focusing on strong anonymity. The first such attempt is most likely *Zerocoin*, a meta-coin protocol for preserving anonymity on top of bitcoin, introduced with a paper at the 2013 IEEE Symposium on Security and Privacy. Zerocoin will be implemented as a completely separate alt coin called Zerocash, in development at time of writing. An alternative approach to anonymity was launched with *CryptoNote* in a paper published in October 2013. CryptoNote is a foundational technology that is implemented by a number of alt coin forks discussed next. In addition to Zerocash and CryptoNotes, there are several other independent anonymous coins, such as Darkcoin, that use stealth addresses or transaction re-mixing to deliver anonymity.

Zerocoin/Zerocash

Zerocoin is a theoretical approach to digital currency anonymity introduced in 2013 by researchers at Johns Hopkins. Zerocash is an alt-coin implementation of Zerocoin that is in development and not yet released.

CryptoNote

CryptoNote is a reference implementation alt coin that provides the basis for anonymous digital cash. It was introduced in October 2013. It is designed to be forked into different implementations and has a built-in periodic reset mechanism that makes it unusable as a currency itself. Several alt coins have been spawned from CryptoNote, including Bytecoin (BCN), Aeon (AEON), Boolberry (BBR), duckNote (DUCK), Fantomcoin (FCN), Monero (XMR), MonetaVerde (MCN), and Quazarcoin (QCN). CryptoNote is also notable for being a complete ground-up implementation of a crypto-currency, not a fork of bitcoin.

Bytecoin

Bytecoin was the first implementation spawned from CryptoNote, offering a viable anonymous currency based on the CryptoNote technology. Bytecoin was launched in July 2012. Note that there was a previous alt coin named Bytecoin with currency symbol BTE, whereas the CryptoNote-derived Bytecoin has the currency symbol BCN. Bytecoin uses the Cryptonight proof-of-work algorithm, which requires access to at least 2 MB of RAM per instance, making it unsuitable for GPU or ASIC mining. Bytecoin inherits ring signatures, unlinkable transactions, and blockchain analysis-resistant anonymity from CryptoNote.

- Block generation: 2 minutes

- Total currency: 184 billion BCN
- Consensus algorithm: Cryptonight proof of work
- Market capitalization: \$3 million in mid-2014

Monero

Monero is another implementation of CryptoNote. It has a slightly flatter issuance curve than Bytecoin, issuing 80% of the currency in the first four years. It offers the same anonymity features inherited from CryptoNote.

- Block generation: 1 minute
- Total currency: 18.4 million XMR
- Consensus algorithm: Cryptonight proof of work
- Market capitalization: \$5 million in mid-2014

Darkcoin

Darkcoin was launched in January 2014. Darkcoin implements anonymous currency using a re-mixing protocol for all transactions called DarkSend. Darkcoin is also notable for using 11 rounds of different hash functions (blake, bmw, groestl, jh, keccak, skein, luffa, cubehash, shavite, simd, echo) for the proof-of-work algorithm.

- Block generation: 2.5 minutes
- Total currency: Maximum 22 million DRK
- Consensus algorithm: Multi-algorithm multi-round proof of work
- Market capitalization: \$19 million in mid-2014

Noncurrency Alt Chains

Alt chains are alternative implementations of the blockchain design pattern, which are not primarily used as currency. Many include a currency, but the currency is used as a token for allocating something else, such as a resource or a contract. The currency, in other words, is not the main point of the platform; it is a secondary feature.

Namecoin

Namecoin was the first fork of the bitcoin code. Namecoin is a decentralized key-value registration and transfer platform using a blockchain. It supports a global domain-name registry similar to the domain-name registration system on the Internet. Namecoin is currently used as an alternative domain name service (DNS) for the root-level domain **.bit**. Namecoin also can be used to register names and key-value pairs in other namespaces; for storing things like email addresses, encryption keys, SSL certificates, file signatures, voting systems, stock certificates; and a myriad of other applications.

The Namecoin system includes the Namecoin currency (symbol NMC), which is used to pay transaction fees for registration and transfer of names. At current prices, the fee to register a name

is 0.01 NMC or approximately 1 US cent. As in bitcoin, the fees are collected by namecoin miners.

Namecoin's basic parameters are the same as bitcoin's:

- Block generation: 10 minutes
- Total currency: 21 million NMC by 2140
- Consensus algorithm: SHA256 proof of work
- Market capitalization: \$10 million in mid-2014

Namecoin's namespaces are not restricted, and anyone can use any namespace in any way. However, certain namespaces have an agreed-upon specification so that when it is read from the blockchain, application-level software knows how to read and proceed from there. If it is malformed, then whatever software you used to read from the specific namespace will throw an error. Some of the popular namespaces are:

- **d/** is the domain-name namespace for **.bit** domains
- **id/** is the namespace for storing person identifiers such as email addresses, PGP keys, and so on
- **u/** is an additional, more structured specification to store identities (based on openspecs)

The Namecoin client is very similar to Bitcoin Core, because it is derived from the same source code. Upon installation, the client will download a full copy of the Namecoin blockchain and then will be ready to query and register names. There are three main commands:

name_new

Query or preregister a name

name_firstupdate

Register a name and make the registration public

name_update

Change the details or refresh a name registration

For example, to register the domain **mastering-bitcoin.bit**, we use the command **name_new** as follows:

```
$ namecoind name_new d/mastering-bitcoin
```

```
[  
  "21cbab5b1241c6d1a6ad70a2416b3124eb883ac38e423e5ff591d1968eb6664a",  
  "a05555e0fc56c023"  
]
```

The **name_new** command registers a claim on the name, by creating a hash of the name with a random key. The two strings returned by **name_new** are the hash and the random key (**a05555e0fc56c023** in the preceding example) that can be used to make the name registration public. Once that claim has been recorded on the Namecoin blockchain it can be converted to a public

registration with the `name_firstupdate` command, by supplying the random key:

```
$ namecoind name_firstupdate d/mastering-bitcoin a05555e0fc56c023 '{"map": {"www": {"ip": "1.2.3.4"}}}'  
b7a2e59c0a26e5e2664948946ebeca1260985c2f616ba579e6bc7f35ec234b01
```

This example will map the domain name `www.mastering-bitcoin.bit` to IP address 1.2.3.4. The hash returned is the transaction ID that can be used to track this registration. You can see what names are registered to you by running the `name_list` command:

```
$ namecoind name_list
```

```
[  
  {  
    "name" : "d/mastering-bitcoin",  
    "value" : "{map: {www: {ip:1.2.3.4}}}",  
    "address" : "NCccBXrRUahAGrisBA1BLPWQfSrups8Geh",  
    "expires_in" : 35929  
  }  
]
```

Namecoin registrations need to be updated every 36,000 blocks (approximately 200 to 250 days). The `name_update` command has no fee and therefore renewing domains in Namecoin is free. Third-party providers can handle registration, automatic renewal, and updating via a web interface, for a small fee. With a third-party provider you avoid the need to run a Namecoin client, but you lose the independent control of a decentralized name registry offered by Namecoin.

Ethereum

Ethereum is a Turing-complete contract processing and execution platform based on a blockchain ledger. It is not a clone of Bitcoin, but a completely independent design and implementation. Ethereum has a built-in currency, called *ether*, which is required in order to pay for contract execution. Ethereum's blockchain records *contracts*, which are expressed in a low-level, byte code-like, Turing-complete language. Essentially, a contract is a program that runs on every node in the Ethereum system. Ethereum contracts can store data, send and receive ether payments, store ether, and execute an infinite range (hence Turing-complete) of computable actions, acting as decentralized autonomous software agents.

Ethereum can implement quite complex systems that are otherwise implemented as alt chains themselves. For example, the following is a Namecoin-like name registration contract written in Ethereum (or more accurately, written in a high-level language that can be compiled to Ethereum code):

```
if !contract.storage[msg.data[0]]: # Is the key not yet taken?
    # Then take it!
    contract.storage[msg.data[0]] = msg.data[1]
    return(1)
else:

    return(0) // Otherwise do nothing
```

Future of Currencies

The future of cryptographic currencies overall is even brighter than the future of bitcoin. Bitcoin introduced a completely new form of decentralized organization and consensus that has spawned hundreds of incredible innovations. These inventions will likely affect broad sectors of the economy, from distributed systems science to finance, economics, currencies, central banking, and corporate governance. Many human activities that previously required centralized institutions or organizations to function as authoritative or trusted points of control can now be decentralized. The invention of the blockchain and consensus system will significantly reduce the cost of organization and coordination on large-scale systems, while removing opportunities for concentration of power, corruption, and regulatory capture.

Bitcoin Security

Securing bitcoin is challenging because bitcoin is not an abstract reference to value, like a balance in a bank account. Bitcoin is very much like digital cash or gold. You've probably heard the expression, "Possession is nine-tenths of the law." Well, in bitcoin, possession is ten-tenths of the law. Possession of the keys to unlock the bitcoin is equivalent to possession of cash or a chunk of precious metal. You can lose it, misplace it, have it stolen, or accidentally give the wrong amount to someone. In every one of these cases, users have no recourse, just as if they dropped cash on a public sidewalk.

However, bitcoin has capabilities that cash, gold, and bank accounts do not. A bitcoin wallet, containing your keys, can be backed up like any file. It can be stored in multiple copies, even printed on paper for hard-copy backup. You can't "back up" cash, gold, or bank accounts. Bitcoin is different enough from anything that has come before that we need to think about bitcoin security in a novel way too.

Security Principles

The core principle in bitcoin is decentralization and it has important implications for security. A centralized model, such as a traditional bank or payment network, depends on access control and vetting to keep bad actors out of the system. By comparison, a decentralized system like bitcoin pushes the responsibility and control to the users. Because security of the network is based on proof of work, not access control, the network can be open and no encryption is required for bitcoin traffic.

On a traditional payment network, such as a credit card system, the payment is open-ended because it contains the user's private identifier (the credit card number). After the initial charge, anyone with access to the identifier can "pull" funds and charge the owner again and again. Thus, the payment network has to be secured end-to-end with encryption and must ensure that no eavesdroppers or intermediaries can compromise the payment traffic, in transit or when it is stored (at rest). If a bad actor gains access to the system, he can compromise current transactions *and* payment tokens that can be used to create new transactions. Worse, when customer data is compromised, the customers are exposed to identity theft and must take action to prevent fraudulent use of the compromised accounts.

Bitcoin is dramatically different. A bitcoin transaction authorizes only a specific value to a specific recipient and cannot be forged or modified. It does not reveal any private information, such as the identities of the parties, and cannot be used to authorize additional payments. Therefore, a bitcoin payment network does not need to be encrypted or protected from eavesdropping. In fact, you can broadcast bitcoin transactions over an open public channel, such as unsecured WiFi or Bluetooth, with no loss of security.

Bitcoin's decentralized security model puts a lot of power in the hands of the users. With that power comes responsibility for maintaining the secrecy of the keys. For most users that is not easy to do, especially on general-purpose computing devices such as Internet-connected smartphones or laptops. Although bitcoin's decentralized model prevents the type of mass compromise seen with credit cards, many users are not able to adequately secure their keys and get hacked, one by one.

Developing Bitcoin Systems Securely

The most important principle for bitcoin developers is decentralization. Most developers will be familiar with centralized security models and might be tempted to apply these models to their bitcoin applications, with disastrous results.

Bitcoin's security relies on decentralized control over keys and on independent transaction validation by miners. If you want to leverage Bitcoin's security, you need to ensure that you remain within the Bitcoin security model. In simple terms: don't take control of keys away from users and don't take transactions off the blockchain.

For example, many early bitcoin exchanges concentrated all user funds in a single "hot" wallet with keys stored on a single server. Such a design removes control from users and centralizes control over keys in a single system. Many such systems have been hacked, with disastrous consequences for their customers.

Another common mistake is to take transactions "off blockchain" in a misguided effort to reduce transaction fees or accelerate transaction processing. An "off blockchain" system will record transactions on an internal, centralized ledger and only occasionally synchronize them to the bitcoin blockchain. This practice, again, substitutes decentralized bitcoin security with a proprietary and centralized approach. When transactions are off blockchain, improperly secured centralized ledgers can be falsified, diverting funds and depleting reserves, unnoticed.

Unless you are prepared to invest heavily in operational security, multiple layers of access control, and audits (as the traditional banks do) you should think very carefully before taking funds outside of Bitcoin's decentralized security context. Even if you have the funds and discipline to implement a robust security model, such a design merely replicates the fragile model of traditional financial networks, plagued by identity theft, corruption, and embezzlement. To take advantage of Bitcoin's unique decentralized security model, you have to avoid the temptation of centralized architectures that might feel familiar but ultimately subvert Bitcoin's security.

The Root of Trust

Traditional security architecture is based upon a concept called the *root of trust*, which is a trusted core used as the foundation for the security of the overall system or application. Security architecture is developed around the root of trust as a series of concentric circles, like layers in an onion, extending trust outward from the center. Each layer builds upon the more-trusted inner layer using access controls, digital signatures, encryption, and other security primitives. As software systems become more complex, they are more likely to contain bugs, which make them vulnerable to security compromise. As a result, the more complex a software system becomes, the harder it is to secure. The root of trust concept ensures that most of the trust is placed within the least complex part of the system, and therefore least vulnerable, parts of the system, while more complex software is layered around it. This security architecture is repeated at different scales, first establishing a root of trust within the hardware of a single system, then extending that root of trust through the operating system to higher-level system services, and finally across many servers layered in concentric circles of diminishing trust.

Bitcoin security architecture is different. In Bitcoin, the consensus system creates a trusted public ledger that is completely decentralized. A correctly validated blockchain uses the genesis block as

the root of trust, building a chain of trust up to the current block. Bitcoin systems can and should use the blockchain as their root of trust. When designing a complex bitcoin application that consists of services on many different systems, you should carefully examine the security architecture in order to ascertain where trust is being placed. Ultimately, the only thing that should be explicitly trusted is a fully validated blockchain. If your application explicitly or implicitly vests trust in anything but the blockchain, that should be a source of concern because it introduces vulnerability. A good method to evaluate the security architecture of your application is to consider each individual component and evaluate a hypothetical scenario where that component is completely compromised and under the control of a malicious actor. Take each component of your application, in turn, and assess the impacts on the overall security if that component is compromised. If your application is no longer secure when components are compromised, that shows you have misplaced trust in those components. A bitcoin application without vulnerabilities should be vulnerable only to a compromise of the bitcoin consensus mechanism, meaning that its root of trust is based on the strongest part of the bitcoin security architecture.

The numerous examples of hacked bitcoin exchanges serve to underscore this point because their security architecture and design fails even under the most casual scrutiny. These centralized implementations had invested trust explicitly in numerous components outside the bitcoin blockchain, such as hot wallets, centralized ledger databases, vulnerable encryption keys, and similar schemes.

User Security Best Practices

Humans have used physical security controls for thousands of years. By comparison, our experience with digital security is less than 50 years old. Modern general-purpose operating systems are not very secure and not particularly suited to storing digital money. Our computers are constantly exposed to external threats via always-on Internet connections. They run thousands of software components from hundreds of authors, often with unconstrained access to the user's files. A single piece of rogue software, among the many thousands installed on your computer, can compromise your keyboard and files, stealing any bitcoin stored in wallet applications. The level of computer maintenance required to keep a computer virus-free and trojan-free is beyond the skill level of all but a tiny minority of computer users.

Despite decades of research and advancements in information security, digital assets are still woefully vulnerable to a determined adversary. Even the most highly protected and restricted systems, in financial services companies, intelligence agencies, and defense contractors, are frequently breached. Bitcoin creates digital assets that have intrinsic value and can be stolen and diverted to new owners instantly and irrevocably. This creates a massive incentive for hackers. Until now, hackers had to convert identity information or account tokens—such as credit cards, and bank accounts—into value after compromising them. Despite the difficulty of fencing and laundering financial information, we have seen ever-escalating thefts. Bitcoin escalates this problem because it doesn't need to be fenced or laundered; it is intrinsic value within a digital asset.

Fortunately, bitcoin also creates the incentives to improve computer security. Whereas previously the risk of computer compromise was vague and indirect, bitcoin makes these risks clear and obvious. Holding bitcoin on a computer serves to focus the user's mind on the need for improved computer security. As a direct result of the proliferation and increased adoption of bitcoin and

other digital currencies, we have seen an escalation in both hacking techniques and security solutions. In simple terms, hackers now have a very juicy target and users have a clear incentive to defend themselves.

Over the past three years, as a direct result of bitcoin adoption, we have seen tremendous innovation in the realm of information security in the form of hardware encryption, key storage and hardware wallets, multi-signature technology, and digital escrow. In the following sections we will examine various best practices for practical user security.

Physical Bitcoin Storage

Because most users are far more comfortable with physical security than information security, a very effective method for protecting bitcoins is to convert them into physical form. Bitcoin keys are nothing more than long numbers. This means that they can be stored in a physical form, such as printed on paper or etched on a metal coin. Securing the keys then becomes as simple as physically securing the printed copy of the bitcoin keys. A set of bitcoin keys that is printed on paper is called a "paper wallet," and there are many free tools that can be used to create them. I personally keep the vast majority of my bitcoins (99% or more) stored on paper wallets, encrypted with BIP0038, with multiple copies locked in safes. Keeping bitcoin offline is called *cold storage* and it is one of the most effective security techniques. A cold storage system is one where the keys are generated on an offline system (one never connected to the Internet) and stored offline either on paper or on digital media, such as a USB memory stick.

Hardware Wallets

In the long term, bitcoin security increasingly will take the form of hardware tamper-proof wallets. Unlike a smartphone or desktop computer, a bitcoin hardware wallet has just one purpose: to hold bitcoins securely. Without general-purpose software to compromise and with limited interfaces, hardware wallets can deliver an almost foolproof level of security to nonexpert users. I expect to see hardware wallets become the predominant method of bitcoin storage. For an example of such a hardware wallet, see the [Trezor](#).

Balancing Risk

Although most users are rightly concerned about bitcoin theft, there is an even bigger risk. Data files get lost all the time. If they contain bitcoin, the loss is much more painful. In the effort to secure their bitcoin wallets, users must be very careful not to go too far and end up losing the bitcoin. In July of 2011, a well-known bitcoin awareness and education project lost almost 7,000 bitcoins. In their effort to prevent theft, the owners had implemented a complex series of encrypted backups. In the end they accidentally lost the encryption keys, making the backups worthless and losing a fortune. Like hiding money by burying it in the desert, if you secure your bitcoin too well you might not be able to find it again.

Diversifying Risk

Would you carry your entire net worth in cash in your wallet? Most people would consider that reckless, yet bitcoin users often keep all their bitcoin in a single wallet. Instead, users should spread the risk among multiple and diverse bitcoin wallets. Prudent users will keep only a small fraction,

perhaps less than 5%, of their bitcoins in an online or mobile wallet as "pocket change." The rest should be split between a few different storage mechanisms, such as a desktop wallet and offline (cold storage).

Multi-sig and Governance

Whenever a company or individual stores large amounts of bitcoin, they should consider using a multi-signature bitcoin address. Multi-signature addresses secure funds by requiring more than one signature to make a payment. The signing keys should be stored in a number of different locations and under the control of different people. In a corporate environment, for example, the keys should be generated independently and held by several company executives, to ensure no single person can compromise the funds. Multi-signature addresses can also offer redundancy, where a single person holds several keys that are stored in different locations.

Survivability

One important security consideration that is often overlooked is availability, especially in the context of incapacity or death of the key holder. Bitcoin users are told to use complex passwords and keep their keys secure and private, not sharing them with anyone. Unfortunately, that practice makes it almost impossible for the user's family to recover any funds if the user is not available to unlock them. In most cases, in fact, the families of bitcoin users might be completely unaware of the existence of the bitcoin funds.

If you have a lot of bitcoin, you should consider sharing access details with a trusted relative or lawyer. A more complex survivability scheme can be set up with multi-signature access and estate planning through a lawyer specialized as a "digital asset executor."

Conclusion

Bitcoin is a completely new, unprecedented, and complex technology. Over time we will develop better security tools and practices that are easier to use by nonexperts. For now, bitcoin users can use many of the tips discussed here to enjoy a secure and trouble-free bitcoin experience.

Appendix A: Transaction Script Language Operators, Constants, and Symbols

[Push value onto stack](#) shows operators for pushing values onto the stack.

Table 26. Push value onto stack

Symbol	Value (hex)	Description
OP_0 or OP_FALSE	0x00	An empty array is pushed onto the stack
1-75	0x01-0x4b	Push the next N bytes onto the stack, where N is 1 to 75 bytes
OP_PUSHDATA1	0x4c	The next script byte contains N, push the following N bytes onto the stack
OP_PUSHDATA2	0x4d	The next two script bytes contain N, push the following N bytes onto the stack
OP_PUSHDATA4	0x4e	The next four script bytes contain N, push the following N bytes onto the stack
OP_1NEGATE	0x4f	Push the value "-1" onto the stack
OP_RESERVED	0x50	Halt - Invalid transaction unless found in an unexecuted OP_IF clause
OP_1 or OP_TRUE	0x51	Push the value "1" onto the stack
OP_2 to OP_16	0x52 to 0x60	For OP_N, push the value "N" onto the stack. E.g., OP_2 pushes "2"

[Conditional flow control](#) shows conditional flow control operators.

Table 27. Conditional flow control

Symbol	Value (hex)	Description
OP_NOP	0x61	Do nothing
OP_VER	0x62	Halt - Invalid transaction unless found in an unexecuted OP_IF clause
OP_IF	0x63	Execute the statements following if top of stack is not 0

Symbol	Value (hex)	Description
OP_NOTIF	0x64	Execute the statements following if top of stack is 0
OP_VERIF	0x65	Halt - Invalid transaction
OP_VERNOTIF	0x66	Halt - Invalid transaction
OP_ELSE	0x67	Execute only if the previous statements were not executed
OP_ENDIF	0x68	End the OP_IF, OP_NOTIF, OP_ELSE block
OP_VERIFY	0x69	Check the top of the stack, halt and invalidate transaction if not TRUE
OP_RETURN	0x6a	Halt and invalidate transaction

[Stack operations](#) shows operators used to manipulate the stack.

Table 28. Stack operations

Symbol	Value (hex)	Description
OP_TOALTSTACK	0x6b	Pop top item from stack and push to alternative stack
OP_FROMALTSTACK	0x6c	Pop top item from alternative stack and push to stack
OP_2DROP	0x6d	Pop top two stack items
OP_2DUP	0x6e	Duplicate top two stack items
OP_3DUP	0x6f	Duplicate top three stack items
OP_2OVER	0x70	Copy the third and fourth items in the stack to the top
OP_2ROT	0x71	Move the fifth and sixth items in the stack to the top
OP_2SWAP	0x72	Swap the two top pairs of items in the stack
OP_IFDUP	0x73	Duplicate the top item in the stack if it is not 0
OP_DEPTH	0x74	Count the items on the stack and push the resulting count
OP_DROP	0x75	Pop the top item in the stack
OP_DUP	0x76	Duplicate the top item in the stack
OP_NIP	0x77	Pop the second item in the stack

Symbol	Value (hex)	Description
OP_OVER	0x78	Copy the second item in the stack and push it onto the top
OP_PICK	0x79	Pop value N from top, then copy the Nth item to the top of the stack
OP_ROLL	0x7a	Pop value N from top, then move the Nth item to the top of the stack
OP_ROT	0x7b	Rotate the top three items in the stack
OP_SWAP	0x7c	Swap the top three items in the stack
OP_TUCK	0x7d	Copy the top item and insert it between the top and second item.

[String splice operations](#) shows string operators.

Table 29. String splice operations

Symbol	Value (hex)	Description
OP_CAT	0x7e	Disabled (concatenates top two items)
OP_SUBSTR	0x7f	Disabled (returns substring)
OP_LEFT	0x80	Disabled (returns left substring)
OP_RIGHT	0x81	Disabled (returns right substring)
OP_SIZE	0x82	Calculate string length of top item and push the result

[Binary arithmetic and conditionals](#) shows binary arithmetic and boolean logic operators.

Table 30. Binary arithmetic and conditionals

Symbol	Value (hex)	Description
OP_INVERT	0x83	Disabled (Flip the bits of the top item)
OP_AND	0x84	Disabled (Boolean AND of two top items)
OP_OR	0x85	Disabled (Boolean OR of two top items)

Symbol	Value (hex)	Description
<i>OP_XOR</i>	0x86	Disabled (Boolean XOR of two top items)
<i>OP_EQUAL</i>	0x87	Push TRUE (1) if top two items are exactly equal, push FALSE (0) otherwise
<i>OP_EQUALVERIFY</i>	0x88	Same as <i>OP_EQUAL</i> , but run <i>OP_VERIFY</i> after to halt if not TRUE
<i>OP_RESERVED1</i>	0x89	Halt - Invalid transaction unless found in an unexecuted <i>OP_IF</i> clause
<i>OP_RESERVED2</i>	0x8a	Halt - Invalid transaction unless found in an unexecuted <i>OP_IF</i> clause

[Numeric operators](#) shows numeric (arithmetic) operators.

Table 31. Numeric operators

Symbol	Value (hex)	Description
<i>OP_1ADD</i>	0x8b	Add 1 to the top item
<i>OP_1SUB</i>	0x8c	Subtract 1 from the top item
<i>OP_2MUL</i>	0x8d	Disabled (multiply top item by 2)
<i>OP_2DIV</i>	0x8e	Disabled (divide top item by 2)
<i>OP_NEGATE</i>	0x8f	Flip the sign of top item
<i>OP_ABS</i>	0x90	Change the sign of the top item to positive
<i>OP_NOT</i>	0x91	If top item is 0 or 1 Boolean flip it, otherwise return 0
<i>OP_0NOTEQUAL</i>	0x92	If top item is 0 return 0, otherwise return 1
<i>OP_ADD</i>	0x93	Pop top two items, add them and push result
<i>OP_SUB</i>	0x94	Pop top two items, subtract first from second, push result
<i>OP_MUL</i>	0x95	Disabled (multiply top two items)
<i>OP_DIV</i>	0x96	Disabled (divide second item by first item)

Symbol	Value (hex)	Description
OP_MOD	0x97	Disabled (remainder divide second item by first item)
OP_LSHIFT	0x98	Disabled (shift second item left by first item number of bits)
OP_RSHIFT	0x99	Disabled (shift second item right by first item number of bits)
OP_BOOLAND	0x9a	Boolean AND of top two items
OP_BOOLOR	0x9b	Boolean OR of top two items
OP_NUMEQUAL	0x9c	Return TRUE if top two items are equal numbers
OP_NUMEQUALVERIFY	0x9d	Same as NUMEQUAL, then OP_VERIFY to halt if not TRUE
OP_NUMNOTEQUAL	0x9e	Return TRUE if top two items are not equal numbers
OP_LESSTHAN	0x9f	Return TRUE if second item is less than top item
OP_GREATERTHAN	0xa0	Return TRUE if second item is greater than top item
OP_LESSTHANOREQUAL	0xa1	Return TRUE if second item is less than or equal to top item
OP_GREATERTHANOREQUAL	0xa2	Return TRUE if second item is great than or equal to top item
OP_MIN	0xa3	Return the smaller of the two top items
OP_MAX	0xa4	Return the larger of the two top items
OP_WITHIN	0xa5	Return TRUE if the third item is between the second item (or equal) and first item

[Cryptographic and hashing operations](#) shows cryptographic function operators.

Table 32. Cryptographic and hashing operations

Symbol	Value (hex)	Description
OP_RIPEMD160	0xa6	Return RIPEMD160 hash of top item
OP_SHA1	0xa7	Return SHA1 hash of top item
OP_SHA256	0xa8	Return SHA256 hash of top item

Symbol	Value (hex)	Description
OP_HASH160	0xa9	Return RIPEMD160(SHA256(x)) hash of top item
OP_HASH256	0xaa	Return SHA256(SHA256(x)) hash of top item
OP_CODESEPARATOR	0xab	Mark the beginning of signature-checked data
OP_CHECKSIG	0xac	Pop a public key and signature and validate the signature for the transaction's hashed data, return TRUE if matching
OP_CHECKSIGVERIFY	0xad	Same as CHECKSIG, then OP_VERIFY to halt if not TRUE
OP_CHECKMULTISIG	0xae	Run CHECKSIG for each pair of signature and public key provided. All must match. Bug in implementation pops an extra value, prefix with OP_NOP as workaround
OP_CHECKMULTISIGVERIFY	0xaf	Same as CHECKMULTISIG, then OP_VERIFY to halt if not TRUE

[Non-operators](#) shows nonoperator symbols

Table 33. Non-operators

Symbol	Value (hex)	Description
OP_NOP1-OP_NOP10	0xb0-0xb9	Does nothing, ignored

[Reserved OP codes for internal use by the parser](#) shows operator codes reserved for use by the internal script parser.

Table 34. Reserved OP codes for internal use by the parser

Symbol	Value (hex)	Description
OP_SMALLDATA	0xf9	Represents small data field
OP_SMALLINTEGER	0xfa	Represents small integer data field
OP_PUBKEYS	0xfb	Represents public key fields
OP_PUBKEYHASH	0xfd	Represents a public key hash field
OP_PUBKEY	0xfe	Represents a public key field

Symbol	Value (hex)	Description
OP_INVALIDOPCODE	0xff	Represents any OP code not currently assigned

Appendix B: Bitcoin Improvement Proposals

Bitcoin improvement proposals are design documents providing information to the bitcoin community, or describing a new feature for bitcoin or its processes or environment.

As per BIP0001 *BIP Purpose and Guidelines*, there are three kinds of BIP:

Standard BIP

Describes any change that affects most or all bitcoin implementations, such as a change to the network protocol, a change in block or transaction validity rules, or any change or addition that affects the interoperability of applications using bitcoin.

Informational BIP

Describes a bitcoin design issue, or provides general guidelines or information to the bitcoin community, but does not propose a new feature. Informational BIPs do not necessarily represent a bitcoin community consensus or recommendation, so users and implementors may ignore informational BIPs or follow their advice.

Process BIP

Describes a bitcoin process, or proposes a change to (or an event in) a process. Process BIPs are like standard BIPs but apply to areas other than the bitcoin protocol itself. They might propose an implementation, but not to bitcoin's codebase; they often require community consensus; and unlike informational BIPs, they are more than recommendations, and users are typically not free to ignore them. Examples include procedures, guidelines, changes to the decision-making process, and changes to the tools or environment used in Bitcoin development. Any meta-BIP is also considered a process BIP.

Bitcoin improvement proposals are recorded in a versioned repository on [GitHub](#). [Snapshot of BIPs](#) shows a snapshot of BIPs in the Fall of 2014. Consult the authoritative repository for up-to-date information on existing BIPs and their contents.

Table 35. Snapshot of BIPs

BIP#	Link	Title	Owner	Type	Status
1	https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki	BIP Purpose and Guidelines	Amir Taaki	Standard	Active
10	https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki	Multi-Sig Transaction Distribution	Alan Reiner	Informational	Draft

BIP#	Link	Title	Owner	Type	Status
11	https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki	M-of-N Standard Transactions	Gavin Andresen	Standard	Accepted
12	https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki	OP_EVAL	Gavin Andresen	Standard	Withdrawn
13	https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki	Address Format for pay-to-script-hash	Gavin Andresen	Standard	Final
14	https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki	Protocol Version and User Agent	Amir Taaki, Patrick Strateman	Standard	Accepted
15	https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki	Aliases	Amir Taaki	Standard	Withdrawn
16	https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki	Pay To Script Hash	Gavin Andresen	Standard	Accepted
17	https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki	OP_CHECKHASHVERIFY (CHV)	Luke Dashjr	Withdrawn	Draft

BIP#	Link	Title	Owner	Type	Status
18	https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki	hashScriptCheck	Luke Dashjr	Standard	Draft
19	https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki	M-of-N Standard Transactions (Low SigOp)	Luke Dashjr	Standard	Draft
20	https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki	URI Scheme	Luke Dashjr	Standard	Replaced
21	https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki	URI Scheme	Nils Schneider, Matt Corallo	Standard	Accepted
22	https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki	getblocktemplate - Fundamentals	Luke Dashjr	Standard	Accepted
23	https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki	getblocktemplate - Pooled Mining	Luke Dashjr	Standard	Accepted
30	https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki	Duplicate transactions	Pieter Wuille	Standard	Accepted

BIP#	Link	Title	Owner	Type	Status
31	https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki	Pong message	Mike Hearn	Standard	Accepted
32	https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki	Hierarchical Deterministic Wallets	Pieter Wuille	Informational	Accepted
33	https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki	Stratized Nodes	Amir Taaki	Standard	Draft
34	https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki	Block v2, Height in coinbase	Gavin Andresen	Standard	Accepted
35	https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki	mempool message	Jeff Garzik	Standard	Accepted
36	https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki	Custom Services	Stefan Thomas	Standard	Draft
37	https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki	Bloom filtering	Mike Hearn and Matt Corallo	Standard	Accepted

BIP#	Link	Title	Owner	Type	Status
38	https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki	Passphrase-protected private key	Mike Caldwell	Standard	Draft
39	https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki	Mnemonic code for generating deterministic keys	Slush	Standard	Draft
40		Stratum wire protocol	Slush	Standard	BIP number allocated
41		Stratum mining protocol	Slush	Standard	BIP number allocated
42	https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki	A finite monetary supply for bitcoin	Pieter Wuille	Standard	Draft
43	https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki	Purpose Field for Deterministic Wallets	Slush	Standard	Draft
44	https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki	Multi-Account Hierarchy for Deterministic Wallets	Slush	Standard	Draft
50	https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki	March 2013 Chain Fork Post-Mortem	Gavin Andresen	Informational	Draft

BIP#	Link	Title	Owner	Type	Status
60	https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki	Fixed Length "version" Message (Relay-Transactions Field)	Amir Taaki	Standard	Draft
61	https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki	"reject" P2P message	Gavin Andresen	Standard	Draft
62	https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki	Dealing with malleability	Pieter Wuille	Standard	Draft
63		Stealth Addresses	Peter Todd	Standard	BIP number allocated
64	https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki	getutxos message	Mike Hearn	Standard	Draft
70	https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki	Payment protocol	Gavin Andresen	Standard	Draft
71	https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki	Payment protocol MIME types	Gavin Andresen	Standard	Draft
72	https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki	Payment protocol URIs	Gavin Andresen	Standard	Draft

BIP#	Link	Title	Owner	Type	Status
73	https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki	Use "Accept" header with Payment Request URLs	Stephen Pair	Standard	Draft

Appendix C: pycoin, ku, and tx

The Python library **pycoin**, originally written and maintained by Richard Kiss, is a Python-based library that supports manipulation of bitcoin keys and transactions, even supporting the scripting language enough to properly deal with nonstandard transactions.

The pycoin library supports both Python 2 (2.7.x) and Python 3 (after 3.3), and comes with some handy command-line utilities, **ku** and **tx**.

Key Utility (KU)

The command-line utility **ku** ("key utility") is a Swiss Army knife for manipulating keys. It supports BIP32 keys, WIF, and addresses (bitcoin and alt coins). Following are some examples.

Create a BIP32 key using the default entropy sources of GPG and */dev/random*:

```

$ ku create

input          : create
network        : Bitcoin
wallet key     :
xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJBsY7VqXUG7hipgdWaU
                m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version :
xpub661MyMwAqRbcFpYYiuvZpKjKhJJDZYAkWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
                DGcpfT56AMFeo8M8KPkFMfLUtvwjwb6WPv8rY65L2q8Hz
tree depth    : 0
fingerprint   : 9d9c6092
parent f'print : 00000000
child index    : 0
chain code     : 80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key    : yes
secret exponent :
112471538590155650688604752840386134637231974546906847202389294096567806844862
  hex          : f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbc
  wif          : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
  uncompressed : 5KhoEavGNNH4GHKoy2PtU4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x  :
76460638240546478364843397478278468101877117767873462127021560368290114016034
public pair y  :
59807879657469774102040120298272207730921291736633247737077406753676825777701
  x as hex     : a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
  y as hex     : 843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
  y parity     : odd
key pair as sec :
03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
  uncompressed :
04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322

843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160        : 9d9c609247174ae323acfc96c852753fe3c8819d
  uncompressed : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
  uncompressed  : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM

```

Create a BIP32 key from a passphrase:

WARNING

The passphrase in this example is way too easy to guess.

```
$ ku P:foo
```

```
input          : P:foo
network        : Bitcoin
wallet key     : xprv9s21ZrQH143K31AgNK5pyVvW23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
                ZoY5eSJMj2Vbyvi2hbmQnCuHBujZ2WXGTux1X2k9Krdtq
public version : xpub661MyMwAqRbcFVF9ULcQLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtS
                VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
tree depth    : 0
fingerprint   : 5d353a2e
parent f'print : 00000000
child index   : 0
chain code    : 5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key   : yes
secret exponent :
65825730547097305716057160437970790220123864299761908948746835886007793998275
  hex         : 91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif          : L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
  uncompressed : 5JvNzA5vXDoKYJdw8SwwLHxUxaWvn9mDea6k1vRPCX7KLUVWa7W
public pair x :
81821982719381104061777349269130419024493616650993589394553404347774393168191
public pair y :
58994218069605424278320703250689780154785099509277691723126325051200459038290
  x as hex    : b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
  y as hex    : 826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
  y parity    : even
key pair as sec : 02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
  uncompressed : 04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
                826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160        : 5d353a2ecdb262477172852d57a3f11de0c19286
  uncompressed : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
  uncompressed : 1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT
```

Get info as JSON:

```
$ ku P:foo -P -j
```

```
{
  "y_parity": "even",
  "public_pair_y_hex":
"826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
  "private_key": "no",
  "parent_fingerprint": "00000000",
  "tree_depth": "0",
  "network": "Bitcoin",
  "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT",
  "key_pair_as_sec_uncompressed":
"04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff
4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
  "public_pair_x_hex":
"b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f",
  "wallet_key":
"xpub661MyMwAqRbcFVF9ULcqlDsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjo
UFidhjFj82pVShWu9curWmb2zy",
  "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
  "child_index": "0",
  "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
  "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
  "fingerprint": "5d353a2e",
  "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
  "input": "P:foo",
  "public_pair_x":
"81821982719381104061777349269130419024493616650993589394553404347774393168191",
  "public_pair_y":
"58994218069605424278320703250689780154785099509277691723126325051200459038290",
  "key_pair_as_sec":
"02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}
```

Public BIP32 key:

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcqlDsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoU
FidhjFj82pVShWu9curWmb2zy
```

Generate a subkey:

```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmy
g6iogWKxx6mefEw4MEroLgKj
```

Hardened subkey:

```
$ ku -w -s3/2H P:foo  
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8  
iDkBN45k67UKsJUXM1JfRCdn1
```

WIF:

```
$ ku -W P:foo  
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

Address:

```
$ ku -a P:foo  
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

Generate a bunch of subkeys:

```
$ ku P:foo -s 0/0-5 -w  
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3  
bhXxKWB89Ggn2dzLcoJsuEdRK  
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVeJ5QHckc5Adtwxa28ACmANi9XhCrRvtFqQcUx  
t8rUgFz3souMiDdWxJDZnQxzx  
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxa6srXUPBtj3PTxQFkZJAiwoUpmvtrx  
KZu4zfsnr3pqyy2vthpkwuoVq  
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8Esk  
pzKL1Y8Gk9aX6QbryA5raK73p  
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtfXdiEY8UsRNJfqK6  
DAyZXoMvtaLHyWQx3FS4A9zw  
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKd84Mm7Wtc2xmzFuKg7iYf7XFHKKsSaYKWKJ  
bR54bnyAD9GzjUYbAYTtN4ruo
```

Generate the corresponding addresses:

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkjHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqs5YfvuzA1Lb
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDURL
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

Generate the corresponding WIFs:

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQ6xLDee2H9v4LvWg14cLJW7QwWPzCtKHdWMAQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnMTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuVFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

Check that it works by choosing a BIP32 string (the one corresponding to subkey 0/3):

```
$ ku -W
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8Esk
pzKL1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnMTDMrqemY8UF
$ ku -a
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8Esk
pzKL1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

Yep, looks familiar.

From secret exponent:

```
$ ku 1
```

```
input          : 1
network        : Bitcoin
secret exponent : 1
hex            : 1
wif            : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn
uncompressed   : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf
public pair x   :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y   :
32670510020758816978083085130507043184471273380659243275938904335757337482424
x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity        : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                  483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
uncompressed    : 1EHNa6Q4Jz2uvNEuL497mE43ikXhwF6kZm
```

Litecoin version:

```
$ ku -nL 1
```

```
input          : 1
network        : Litecoin
secret exponent : 1
hex            : 1
wif            : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdWUwyfRDeGZm76aUjV
uncompressed   : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x   :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y   :
32670510020758816978083085130507043184471273380659243275938904335757337482424
x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity        : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                  483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6sUkLkoqDEVUnJ
uncompressed    : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

Dogecoin WIF:

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qr ioRbQmjxac5TVoTtZuot
```

From public pair (on Testnet):

```
$ ku -nT
55066263022277343669578718895168534326250603453777594175500187360389116729240,even

input          :
550662630222773436695787188951685343262506034537775941755001873603
                        89116729240,even
network        : Bitcoin testnet
public pair x  :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y  :
32670510020758816978083085130507043184471273380659243275938904335757337482424
x as hex       :
79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
y as hex       :
483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity       : even
key pair as sec :
0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
uncompressed   :
0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798

483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
uncompressed   : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCybB6J1vRfbwM5hemdJz73FwDBC8r
uncompressed   : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

From hash160:

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6

input          : 751e76e8199196d454941c45d1b3a323f1433bd6
network        : Bitcoin
hash160        : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

As a Dogecoin address:

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6

input           : 751e76e8199196d454941c45d1b3a323f1433bd6
network         : Dogecoin
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

Transaction Utility (TX)

The command-line utility **tx** will display transactions in human-readable form, fetch base transactions from pycoin's transaction cache or from web services (blockchain.info, blockr.io, and biteasy.com are currently supported), merge transactions, add or delete inputs or outputs, and sign transactions.

Following are some examples.

View the famous "pizza" transaction [PIZZA]:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache to
cache transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment
variable PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
        [-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
        [--remove-tx-in tx_in_index_to_delete]
        [--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
        [-b BITCOIND_URL] [-o path-to-output-file]
        argument [argument ...]
tx: error: can't find Tx with id
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
```

Oops! We don't have web services set up. Let's do that now:

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

It's not done automatically so a command-line tool won't leak potentially private information about what transactions you're interested in to a third-party website. If you don't care, you could put

these lines into your *.profile*.

Let's try again:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Version: 1 tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: (unknown) from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
including unspents in hex dump since transaction not fully signed
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e00000000
4a493046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d780221
00a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901fffffffff010010
a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

** can't validate transaction as source transactions missing
```

The final line appears because to validate the transactions' signatures, you technically need the source transactions. So let's add **-a** to augment the transactions with source information:

```

$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may be
incorrect
warning: transaction fee lower than (casually calculated) expected value of 0.1
mBTC, transaction might not propagate
Version: 1 tx hash
49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (valid anytime)
Input:
  0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk from
1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0 10000000.00000
mBTC sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input 10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees      0.00000 mBTC

010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e00000000
4a493046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d780221
00a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901fffffffff010010
a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

all incoming transaction values validated

```

Now, let's look at unspent outputs for a specific address (UTXO). In block #1, we see a coinbase transaction to **12c6DSiU4Rq3P4ZxziKxZrL5LmMBrzjrJX**. Let's use **fetch_unspent** to find all coins in this address:

```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxZrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/0/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/31/76a914119b098e
2e980a229e139a9ed01a469e518e6f2688ac/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/86/76a914119b098e
2e980a229e139a9ed01a469e518e6f2688ac/10000
a66ddddd42f9f2491d3c336ce5527d45cc5c2163aaed3158f81dc054447f447a2/0/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/10000
ffd901679de65d4398de90cefe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcf4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/5/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/1/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/100000
fd87f9adebb17f4ebb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/1
dfdf0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098e2
e980a229e139a9ed01a469e518e6f2688ac/2000000

0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0/410496b538e8535
19c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141781e
62294721166bf621e73a82cbf2342c858eeac/5000000000
```

Appendix D: Bitcoin Explorer (bx) Commands

Usage: bx COMMAND [--help]

Info: The bx commands are:

address-decode
address-embed
address-encode
address-validate
base16-decode
base16-encode
base58-decode
base58-encode
base58check-decode
base58check-encode
base64-decode
base64-encode
bitcoin160
bitcoin256
btc-to-satoshi
ec-add
ec-add-secrets
ec-multiply
ec-multiply-secrets
ec-new
ec-to-address
ec-to-public
ec-to-wif
fetch-balance
fetch-header
fetch-height
fetch-history
fetch-stealth
fetch-tx
fetch-tx-index
hd-new
hd-private
hd-public
hd-to-address
hd-to-ec
hd-to-public
hd-to-wif
help
input-set
input-sign
input-validate

```
message-sign
message-validate
mnemonic-decode
mnemonic-encode
ripemd160
satoshi-to-btc
script-decode
script-encode
script-to-address
seed
send-tx
send-tx-node
send-tx-p2p
settings
sha160
sha256
sha512
stealth-decode
stealth-encode
stealth-public
stealth-secret
stealth-shared
tx-decode
tx-encode
uri-decode
uri-encode
validate-tx
watch-address
wif-to-ec
wif-to-public
wrap-decode
wrap-encode
```

For more information, see the [Bitcoin Explorer home page](#) and [Bitcoin Explorer user documentation](#).

Examples of bx command use

Let's look at some examples of using Bitcoin Explorer commands to experiment with keys and addresses:

Generate a random "seed" value using the `seed` command, which uses the operating system's random number generator. Pass the seed to the `ec-new` command to generate a new private key. We save the standard output into the file `private_key`:

```
$ bx seed | bx ec-new > private_key
$ cat private_key
73096ed11ab9f1db6135857958ece7d73ea7c30862145bcc4bbc7649075de474
```

Now, generate the public key from that private key using the `ec-to-public` command. We pass the `private_key` file into the standard input and save the standard output of the command into a new file `public_key`:

```
$ bx ec-to-public < private_key > public_key
$ cat public_key
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

We can reformat the `public_key` as an address using the `ec-to-address` command. We pass the `public_key` into standard input:

```
$ bx ec-to-address < public_key
17re1S4Q8ZHYP8Kw7xQad1Lr6XUzWUnkG
```

Keys generated in this manner produce a type-0 nondeterministic wallet. That means that each key is generated from an independent seed. Bitcoin Explorer commands can also generate keys deterministically, in accordance with BIP0032. In this case, a "master" key is created from a seed and then extended deterministically to produce a tree of subkeys, resulting in a type-2 deterministic wallet.

First, we use the `seed` and `hd-new` commands to generate a master key that will be used as the basis to derive a hierarchy of keys.

```
$ bx seed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1

$ bx hd-new < seed > master
$ cat master
xprv9s21ZrQH143K2BEhMYpNQoUvAgIEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBq
kYoWWMCRwzfuJk8RwF1SVEpAQ
```

We now use the `hd-private` command to generate a hardened "account" key and a sequence of two private keys within the account.

```
$ bx hd-private --hard < master > account
$ cat account
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwcgS8P
YbgoR2NWHeLyvg4DhoEE68A1n

$ bx hd-private --index 0 < account
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVESuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F
3DWPkX7Tp1V1u97HKG1FJwUpU

$ bx hd-private --index 1 < account
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdsuMVERadbgDbziCRJru9n6tz
EWrASVpEdrZrFidt1RDfn4yA3
```

Next we use the `hd-public` command to generate the corresponding sequence of two public keys.

```
$ bx hd-public --index 0 < account
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3ve
zHz5wzaSW4FiGrseNDR4LKqTy

$ bx hd-public --index 1 < account
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWb
GQDhohEcDFTEYMvYzwRD7Juf8
```

The public keys can also be derived from their corresponding private keys using the `hd-to-public` command.

```
$ bx hd-private --index 0 < account | bx hd-to-public
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3ve
zHz5wzaSW4FiGrseNDR4LKqTy

$ bx hd-private --index 1 < account | bx hd-to-public
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWb
GQDhohEcDFTEYMvYzwRD7Juf8
```

We can generate a practically limitless number of keys in a deterministic chain, all derived from a single seed. This technique is used in many wallet applications to generate keys that can be backed up and restored with a single seed value. This is easier than having to back up the wallet with all its randomly generated keys every time a new key is created.

The seed can be encoded using the `mnemonic-encode` command.

```
$ bx hd-mnemonic < seed > words
adore repeat vision worst especially veil inch woman cast recall dwell appreciate
```

The seed can then be decoded using the `mnemonic-decode` command.

```
$ bx mnemonic-decode < words  
eb68ee9f3df6bd4441a9feadec179ff1
```

Mnemonic encoding can make the seed easier to record and even remember.