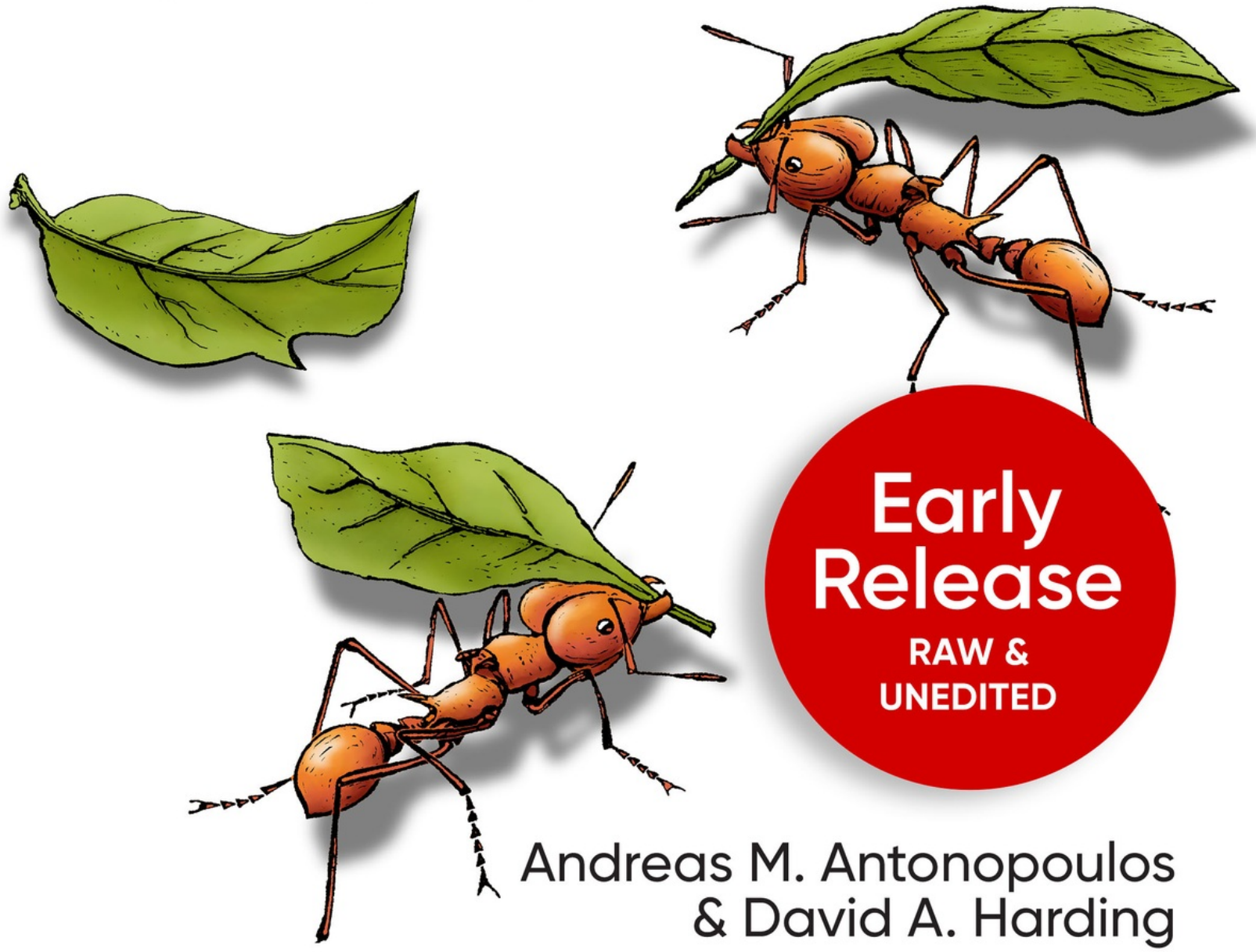
O'REILLY®

Third Edition

Mastering Bitcoin

Programming the Open Blockchain



Early Release

RAW & UNEDITED

Andreas M. Antonopoulos
& David A. Harding

Mastering Bitcoin

THIRD EDITION

Programming the Open Blockchain

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Andreas M. Antonopoulos and David A. Harding

Beijing • Boston • Farnham • Sebastopol • Tokyo

Mastering Bitcoin

by Andreas M. Antonopoulos and David A. Harding

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Acquisitions Editor: Michelle Smith
- Development Editor: Angela Rufino
- Production Editor: Kristen Brown
- Copyeditor:
- Proofreader:
- Indexer:
- Interior Designer: David Futato
- Cover Designer: Randy Comer

- Illustrator:

- December 2014: First Edition
- June 2017: Second Edition
- December 2023: Third Edition

Revision History for the Early Release

- 2023-02-22: First Release
- 2023-03-24: Second Release
- 2023-05-18: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098150099 for release details.

property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-15009-9

[LSI]

Chapter 1. Introduction

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

Bitcoin is a collection of concepts and technologies that form the basis of a digital money ecosystem. Units of currency called bitcoin 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.

TIP

In this book, the unit of currency is called "bitcoin" with a small *b*, and the system is called "Bitcoin", with

a capital *B*.

Users can transfer bitcoin over the network to do just about anything that can be done with conventional currencies, including buying and selling goods, sending money to people or organizations, or extending credit. Bitcoin can be purchased, sold, and exchanged for other currencies at specialized currency exchanges. Bitcoin is arguably the perfect form of money for the internet because it is fast, secure, and borderless.

Unlike traditional currencies, the bitcoin currency is entirely virtual. There are no physical coins or even individual digital coins. The coins are implied in transactions that transfer value from spender to receiver. Users of Bitcoin control keys that allow them to prove ownership of bitcoin in the Bitcoin network. With these keys, they can sign transactions to unlock the value and spend it by transferring it to a new owner. Keys are often stored in a digital wallet on each user's computer or smartphone. Possession of the key that can sign a transaction is the only prerequisite to spending bitcoin, 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. Units of bitcoin are created through a process called "mining," which involves repeatedly performing a competitive computational task that references a list of recent Bitcoin transactions. Any participant in the Bitcoin network may operate as a miner, using their computing devices to help secure transactions. Every 10 minutes, on average, one Bitcoin miner can add

security to past transactions and is rewarded with both brand new bitcoin and the fees paid by recent transactions. Essentially, Bitcoin mining decentralizes the currency-issuance and clearing functions of a central bank and replaces the need for any central bank.

The Bitcoin protocol includes built-in algorithms that regulate the mining function across the network. The difficulty of the computational task that miners must perform is adjusted dynamically so that, on average, someone succeeds every 10 minutes regardless of how many miners (and how much processing) are competing at any moment. The protocol also halves the rate at which new bitcoins are created, limiting the total number of bitcoins that will be created to a fixed total just below 21 million coins. The result is that the number of bitcoins in circulation closely follows an easily predictable curve where half of the remaining coins are added to circulation every four years. By the time the third edition of this book has been published for ten years, 99% of all bitcoins that will ever exist will have been issued. Due to bitcoin's diminishing rate of issuance, over the long term, the Bitcoin currency is deflationary. Furthermore, nobody can force you to accept any bitcoins that were created beyond the expected issuance rate.

Behind the scenes, Bitcoin is also the name of the protocol, a peer-to-peer network, and a distributed computing innovation. Bitcoin builds on decades of research in cryptography and distributed systems and includes at least 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 set of rules for independent transaction validation and currency issuance (consensus rules)
- A mechanism for reaching global decentralized consensus on the valid blockchain (Proof-of-Work algorithm)

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 the 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. Three basic questions for anyone accepting digital money are:

1. Can I trust that the money is authentic and not counterfeit?
2. Can I trust that the digital money can only be spent once (known as the

"double-spend" problem)?
3. Can I be sure that no one else can claim this money belongs to them and not me?

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, 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, decentralized by design, and free of any central authority or point of control that can be attacked or corrupted.

History of Bitcoin

Bitcoin was first described in 2008 with the publication of a paper titled "Bitcoin: A Peer-to-Peer Electronic Cash System,"[1] written under the alias of Satoshi Nakamoto (see XREF HERE). Nakamoto combined several prior inventions such as digital signatures 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. A 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 implementation of the Proof-of-Work algorithm (mining) that provides security and resilience for Bitcoin has increased in power exponentially, and now exceeds the combined number of computing operations of the world's top supercomputers.

Satoshi Nakamoto withdrew from the public in April 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 individual control over the Bitcoin system, which operates based on fully transparent mathematical principles, open source code, and consensus among participants. 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 get multiple participants without a leader 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.

Bitcoin Uses, Users, and Their Stories

Bitcoin is an innovation in the ancient technology of money. At its core, money simply facilitates the exchange of value between people. Therefore, in order to fully understand Bitcoin and its uses, we'll examine it from the perspective of people using it. Each of the people and their stories, as listed here, illustrates one or more specific use cases. We'll be seeing them throughout the book:

North American e-commerce retails
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 her bitcoin to buy a laptop from Bob's online store. 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% 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 a 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.

Web store

Gabriel is an enterprising young teenager in Rio de Janeiro, running a small web store that sells Bitcoin-branded t-shirts, coffee mugs, and stickers. Gabriel is too young to have a bank account, but his parents are encouraging his entrepreneurial spirit.

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 United States 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 bitcoin 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 the real people and real industries currently using Bitcoin to create new markets, new industries, and innovative solutions to global economic issues.

Getting Started

Bitcoin is a protocol that can be accessed using an application that speaks the protocol. A “Bitcoin wallet” is the most common user interface to the Bitcoin system, just like a web browser is the most common user interface for the HTTP protocol. There are many implementations and brands of Bitcoin wallets, just like there are many brands of web browsers (e.g., Chrome, Safari, Firefox, and Internet Explorer). And just like we all have our favorite browsers (Mozilla Firefox, Yay!) and our villains (Internet Explorer, Yuck!), Bitcoin wallets vary in quality, performance, security, privacy, and reliability. There is also a reference implementation of the Bitcoin protocol that includes a wallet, known as “Bitcoin Core,” which is derived from the original implementation written by Satoshi Nakamoto.

Choosing a Bitcoin Wallet

Bitcoin wallets are one of the most actively developed applications in the Bitcoin ecosystem. There is intense competition, and while a new wallet is probably being developed right now, several wallets from last year are no longer actively maintained. Many wallets focus on specific platforms or specific uses and some

are more suitable for beginners while others are filled with features for advanced users. Choosing a wallet is highly subjective and depends on the use and user expertise. Therefore it would be pointless to recommend a specific brand or wallet. However, we can categorize Bitcoin wallets according to their platform and function and provide some clarity about all the different types of wallets that exist. It is worth trying out several different wallets until you find one that fits your needs.

Types of Bitcoin wallets

Bitcoin wallets can be categorized as follows, according to the platform:

Desktop wallet
A desktop wallet was the first type of Bitcoin wallet created as a reference implementation and many users run desktop wallets for the features, autonomy, and control they offer. Running on general-use operating systems such as Windows and Mac OS has certain security disadvantages, however, as these platforms are often insecure and poorly configured.

Mobile wallet
A mobile wallet is the most common type of Bitcoin wallet. Running on smart-phone operating systems such as Apple iOS and Android, these wallets are often a great choice for new users. Many are designed for simplicity and ease-of-use, but there are also fully featured mobile wallets for power users. To avoid downloading and storing large amounts of data, most mobile wallets retrieve information from remote servers, reducing your privacy by disclosing to third parties information about your Bitcoin addresses and balances.

Web wallet

Web wallets 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. Some of these services operate using client-side code running in the user's browser, which keeps control of the Bitcoin keys in the hands of the user, although the user's dependence on the server still compromises their privacy. Most, however, take control of the Bitcoin keys from users in exchange for ease-of-use. It is inadvisable to store large amounts of bitcoin on third-party systems.

Hardware signing devices

Hardware signing devices are devices that can store keys and sign transactions using special-purpose hardware and firmware. They usually connect to a desktop, mobile, or web wallet via USB cable, near-field-communication (NFC), or a camera with QR codes. By handling all Bitcoin-related operations on the specialized hardware, these wallets are less vulnerable to many types of attacks. Hardware signing devices are sometimes called "hardware wallets", but they need to be paired with a full-featured wallet to send and receive transactions, and the security and privacy offered by that paired wallet plays a critical role in how much security and privacy the user obtains when using the hardware signing device.

Full-node vs. Lightweight

Another way to categorize bitcoin wallets is by their degree of autonomy and

how they interact with the Bitcoin network:

Full-node

A full node is a program that validates the entire history of Bitcoin transactions (every transaction by every user, ever). Optionally, full nodes can also store previously validated transactions and serve data to other Bitcoin programs, either on the same computer or over the internet. A full node uses substantial computer resources—about the same as watching an hour-long streaming video for each day of Bitcoin transactions—but the full node offers complete autonomy to its users.

Lightweight client

A lightweight client, also known as a simplified-payment-verification (SPV) client, connects to a full node or other remote server for receiving and sending Bitcoin transaction information, but stores the user wallet locally, partially validates the transactions it receives, and independently creates outgoing transactions.

Third-party API client

A third-party API client is one that interacts with Bitcoin through a third-party system of application programming interfaces (APIs), rather than by connecting to the Bitcoin network directly. The wallet may be stored by the user or by third-party servers, but the client trusts the remote server to provide it with accurate information and protect its privacy.

TIP

Bitcoin is a Peer-to-Peer (P2P) network. Full nodes are the *peers:* each peer individually validates every confirmed transaction and can provide data to its user with complete authority. Lightweight wallets and other software are *clients:* each client depends on one or more peers to provide it with valid data. Bitcoin clients can perform secondary validation on some of the data they receive and make connections to multiple peers to reduce their depedence on the integrity of a single peer, but the security of a client ultimately relies on the integrity of its peers.

Custodial vs. Non-Custodial

A very important additional consideration is *who controls the keys*. As we will see in subsequent chapters, access to bitcoins is controlled by "private keys", which are like very long PIN numbers. If you are the only one to have **custody** and **control** over these private keys, you are in control of your bitcoin. Conversely, if you do not have custody, then your bitcoin is managed by a third-party custodian, who ultimately controls your funds on your behalf. Wallets fall into two important categories based on custody: *non-custodial* wallets where you control the keys and the funds and *custodial* wallets where some third-party controls the keys. To emphasize this point, I (Andreas) coined the phrase:

Your keys, your coins. Not your keys, not your coins.

Combining these categorizations, many Bitcoin wallets fall into a few groups, with the three most common being desktop full node (non-custodial), mobile lightweight wallet (non-custodial), and web third-party wallet (custodial). The lines between different categories are often blurry, as many wallets run on multiple platforms and can interact with the network in different ways.

For the purposes of this book, we will be demonstrating the use of a variety of downloadable Bitcoin clients, from the reference implementation (Bitcoin Core) to mobile and web wallets. Some of the examples will require the use of Bitcoin Core, which, in addition to being a full node, 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 to run Bitcoin Core, or one of the alternative full node implementations.

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 her friend Joe. While at a party, Joe is once again enthusiastically explaining Bitcoin to everyone around him and is offering a demonstration. Intrigued, Alice asks how she can get started with Bitcoin. Joe says that a mobile wallet is best for new users and he recommends a few of his favorite wallets. Alice downloads one of Joe's recommendations and installs it on her phone.

When Alice runs her wallet application for the first time, she chooses the option to create a new Bitcoin wallet. Because the wallet she has chosen is a non-custodial wallet, Alice (and only Alice) will be in control of her keys. Therefore, she bears responsibility for backing them up, since losing the keys means she loses access to her bitcoins. To facilitate this, her wallet produces a *recovery code* that can be used to restore her wallet.

Recovery Codes

Most modern non-custodial Bitcoin wallets will provide a *recovery code* for their user to back up. The recovery code usually consists of numbers, letters, or words selected randomly by the software, and is used as the basis for the keys that are generated by the wallet. See Table 1-1 for examples.

Table 1-1. Sample Recovery Codes

Wallet	Recovery code
BlueWallet	(1) media (2) suspect (3) effort (4) dish (5) album (6) shaft (7) price (8) junk (9) pizza (10) situate (11) oyster (12) rib
Electrum	nephew dog crane clever quantum crazy purse traffic repeat fruit old clutch
Muun	LAFV TZUN V27E NU4D WPF4 BRJ4 ELLP BNFL

TIP

A recovery code is sometimes called a “mnemonic” or “mnemonic phrase”, which implies you should memorize the phrase, but writing the phrase down on paper takes less work and tends to be more reliable than most people’s memories. Another alternative name is “seed phrase” because it provides the input (“seed”) to the function which generates all of a wallet’s keys.

If something happens to Alice’s wallet, she can download a new copy of her wallet software and enter this recovery code to rebuild the wallet database of all

the onchain transactions she's ever sent or received. However, recovering from the recovery code will not by itself restore any additional data Alice entered into her wallet, such as the names she associated with particular addresses or transactions. Although losing access to that metadata isn't as important as losing access to money, it can still be important in its own way. Imagine you need to review an old bank or credit card statement and the name of every entity you paid (or who paid you) has been blanked out. To prevent losing metadata, many wallets provide an additional backup feature beyond recovery codes.

For some wallets, that additional backup feature is even more important today than it used to be. Many Bitcoin payments are now made using *offchain* technology, where not every payment is stored in the public block chain. This reduces users costs and improves privacy, among other benefits, but it means that a mechanism like recovery codes that depends on onchain data can't guarantee recovery of all of a user's bitcoins. For applications with offchain support, it's important to make frequent backups of the wallet database.

Of note, when receiving funds to a new mobile wallet for the first time, many wallets will often re-verify that you have securely backed-up your recovery code. This can range from a simple prompt to requiring the user to manually re-enter the code.

WARNING

Although many legitimate wallets will prompt you to re-enter your recovery code, there are also many malware applications that mimic the design of a wallet, insist you enter your recovery code, and then relay any entered code to the malware developer so they can steal your funds. This is the equivilent of phishing websites that try to trick you into giving them your bank passphrase. For most wallet applications, the only

times they will ask for your recovery code are during the initial set up (before you have received any bitcoins) and during recovery (after you lost access to your original wallet). If the application asks for your recovery code any other time, consult with an expert to ensure you aren't being phished.

Bitcoin addresses

Alice is now ready to start using her new bitcoin wallet. Her wallet application randomly generated a private key (described in more detail in "Private Keys") which will be used to derive Bitcoin addresses that direct to her wallet. At this point, her Bitcoin addresses are not known to the Bitcoin network or "registered" with any part of the Bitcoin system. Her Bitcoin addresses are simply random numbers that correspond to her private key that she can use to control access to the funds. The addresses are generated independently by her wallet without reference or registration with any service.

TIP

There are a variety of Bitcoin addresses and invoice formats. Addresses and invoices can be shared with other bitcoin users who can use them to send bitcoin directly to your wallet. You can share an address or invoice with other people without worrying about the security of your bitcoins. Unlike a bank account number, nobody who learns one of your Bitcoin addresses can withdraw money from your wallet—you must initiate all spends. However, if you give two people the same address, they will be able to see how much bitcoin the other person sent you. If you post your address publicly, everyone will be able to see how much bitcoin other people sent you. To protect your privacy, you should generate a new invoice with a new address each time you request a payment.

Receiving bitcoin

Alice uses the *Receive* button, which displays a QR code along with a Bitcoin address, shown in Figure 1-1.

Figure 1-1. Alice uses the Receive screen on her mobile Bitcoin wallet, and displays her address in a QR code format

The QR code is the square with a pattern of black and white dots, serving as a form of barcode that contains the same information in a format that can be scanned by Joe's smartphone camera. Near the wallet's QR code is the Bitcoin address it encodes, and Alice may choose to manually send her address to Joe by copying it onto her clipboard with a tap.

WARNING

Any funds sent to the addresses in this book will be lost. If you want to test sending bitcoins, please consider donating it to a bitcoin-accepting charity.

Getting Your First Bitcoin

The first task for new users is to acquire some bitcoin.

Bitcoin transactions are irreversible. Most electronic payment networks such as credit cards, debit cards, PayPal, and bank account transfers are reversible. For someone selling bitcoin, this difference introduces a very high risk that the buyer will reverse the electronic payment after they have received bitcoin, in effect defrauding the seller. To mitigate this risk, companies accepting traditional electronic payments in return for bitcoin usually require buyers to undergo identity verification and credit-worthiness checks, which may take several days or weeks. As a new user, this means you cannot buy bitcoin instantly with a credit card. With a bit of patience and creative thinking, however, you won't need to.

Here are some methods for getting bitcoin as a new user:

- Find a friend who has bitcoin and buy some from him or her directly. Many Bitcoin users start this way. This method is the least complicated. One way to meet people with bitcoin is to attend a local Bitcoin meetup listed at Meetup.com.
- Use a classified service such as localbitcoins.com to find a seller in your area

to buy bitcoin for cash in an in-person transaction.

- Earn bitcoin by selling a product or service for bitcoin. If you are a programmer, sell your programming skills. If you're a hairdresser, cut hair for bitcoin.
- Use a bitcoin ATM in your city. A bitcoin ATM is a machine that accepts cash and sends bitcoin to your smartphone bitcoin wallet. Find a bitcoin ATM close to you using an online map from Coin ATM Radar.
- Use a bitcoin currency exchange linked to your bank account. Many countries now have currency exchanges that offer a market for buyers and sellers to swap bitcoin with local currency. Exchange-rate listing services, such as BitcoinAverage, often show a list of bitcoin exchanges for each currency.

TIP

One of the advantages of Bitcoin over other payment systems is that, when used correctly, it affords users much more privacy. Acquiring, holding, and spending bitcoin does not require you to divulge sensitive and personally identifiable information to third parties. However, where bitcoin touches traditional systems, such as currency exchanges, national and international regulations often apply. In order to exchange bitcoin for your national currency, you will often be required to provide proof of identity and banking information. Users should be aware that once a Bitcoin address is attached to an identity, other associated bitcoin transactions may also become easy to identify and track—including transactions made earlier. This is one reason many users choose to maintain dedicated exchange accounts unlinked to their wallets.

Alice was introduced to bitcoin by a friend so she has an easy way to acquire her first bitcoin. Next, we will look at how she buys bitcoin from her friend Joe and how Joe sends the bitcoin to her wallet.

Finding the Current Price of Bitcoin

Before Alice can buy bitcoin from Joe, they have to agree on the *exchange rate* between bitcoin and US dollars. This brings up a common question for those new to bitcoin: “Who sets the bitcoin price?” The short answer is that the price is set by markets.

Bitcoin, like most other currencies, has a *floating exchange rate*. That means that the value of bitcoin fluctuates according to supply and demand in the various markets where it is traded. For example, the “price” of bitcoin in US dollars is calculated in each market based on the most recent trade of bitcoin and US dollars. As such, the price tends to fluctuate minutely several times per second. A pricing service will aggregate the prices from several markets and calculate a volume-weighted average representing the broad market exchange rate of a currency pair (e.g., BTC/USD).

There are hundreds of applications and websites that can provide the current market rate. Here are some of the most popular:

Bitcoin Average

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

CoinCap

A service listing the market capitalization and exchange rates of hundreds of crypto-currencies, including bitcoin.

Chicago Mercantile Exchange Bitcoin Reference Rate

A reference rate that can be used for institutional and contractual reference, provided as part of investment data feeds by the CME.

In addition to these various sites and applications, some bitcoin wallets will automatically convert amounts between bitcoin and other currencies.

Sending and Receiving Bitcoin

Alice has decided to buy 0.001 bitcoin. After she and Joe check the exchange rate, she gives Joe an appropriate amount of cash, opens her mobile wallet application, and selects Receive. This displays a QR code with Alice's first Bitcoin address.

Joe then selects Send on his smartphone wallet and opens the QR code scanner. This allows Joe to scan the barcode with his smartphone camera so that he doesn't have to type in Alice's Bitcoin address, which is quite long and difficult to type.

Joe now has Alice's Bitcoin address set as the recipient. Joe enters the amount as 0.001 bitcoins (BTC), see Figure 1-2. Some wallets may show the amount in a different denomination: 0.001 BTC is 1 millibitcoin (mBTC) or 100,000 satoshis (sats).

Some wallets may also suggest Joe enter a label for this transaction; if so, Joe enters "Alice". Weeks or months from now, this will help Joe remember why he sent these 0.001 bitcoins. Some wallets may also prompt Joe about fees.

Depending on the wallet and how the transaction is being sent, the wallet may ask Joe to either enter a transaction fee rate or prompt him with a suggested feerate. The higher the transaction fee rate, the faster the transaction will be confirmed (see “Confirmations”).

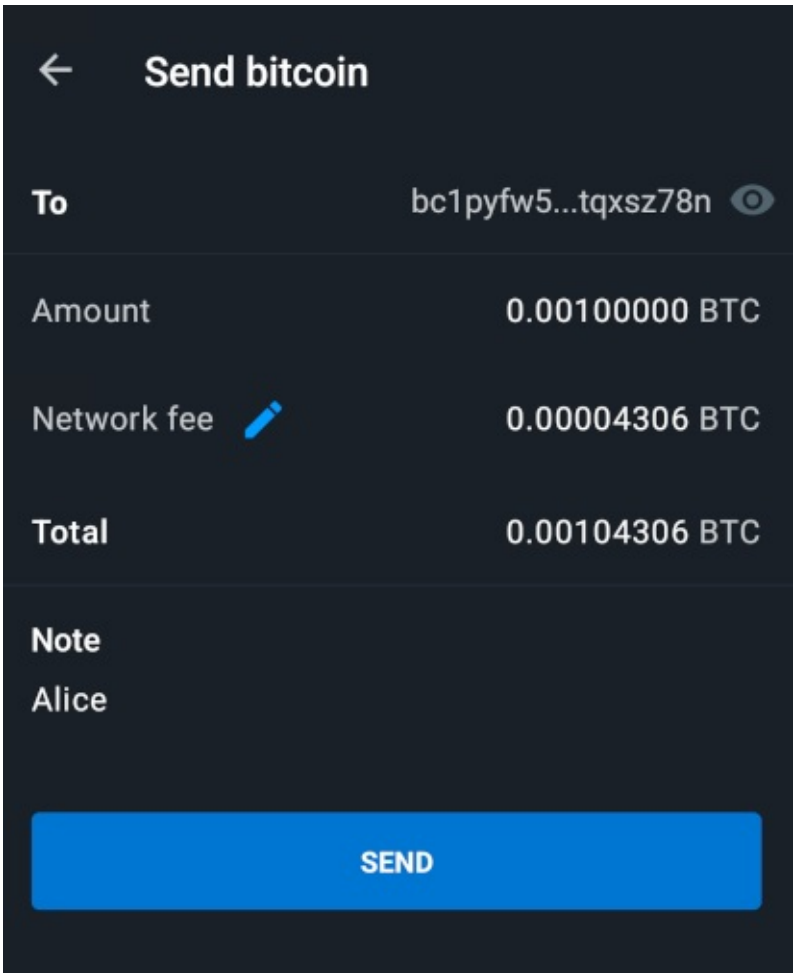


Figure 1-2. Bitcoin wallet send screen

Joe then carefully checks to make sure he has entered the correct amount, because he is about to transmit money and mistakes will soon become irreversible. After double-checking the address and amount, he presses Send to transmit the transaction. Joe’s mobile Bitcoin wallet constructs a transaction that

assigns 0.001 BTC 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 to Alice's new address. As the transaction is transmitted via the peer-to-peer protocol, it quickly propagates across the Bitcoin network. After just a few seconds, most of the well-connected nodes in the network receive the transaction and see Alice's address for the first time.

Meanwhile, Alice's wallet is constantly "listening" for new transactions on the Bitcoin network, looking for any that match the addresses it contains. A few seconds after Joe's wallet transmits the transaction, Alice's wallet will indicate that it is receiving 0.001 BTC.

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 recorded in the bitcoin transaction ledger, known as the blockchain. To be confirmed, a transaction must be included in a block and added to the blockchain, which happens every 10 minutes, on average. In traditional financial terms this is known as *clearing*. For more details on propagation, validation, and clearing (confirmation) of bitcoin transactions, see XREF HERE.

Alice is now the proud owner of 0.001 BTC that she can spend. Over the next few days, Alice buys more bitcoin using an ATM and an exchange. In the next

chapter we will look at her first purchase with bitcoin, and examine the underlying transaction and propagation technologies in more detail.

‘Bitcoin: A Peer-to-Peer Electronic Cash System,” Satoshi Nakamoto (*https://bitcoin.org/bitcoin.pdf*).

Chapter 2. How Bitcoin Works

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

The Bitcoin system, unlike traditional banking and payment systems, does not require trust in third parties. Instead of a central trusted authority, in Bitcoin, each user can use software running on their own computer to verify the correct operation of every aspect of 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 is recorded on the blockchain, the distributed ledger of all transactions. Subsequent chapters will delve into the technology behind transactions, the network, and mining.

Bitcoin Overview

In the overview diagram shown in Figure 2-1, 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.

Each example in this chapter is based on an actual transaction made on the Bitcoin network, simulating the interactions between the users (Joe, Alice, Bob, and Gopesh) by sending funds from one wallet to another. While tracking a transaction through the Bitcoin network to the 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.

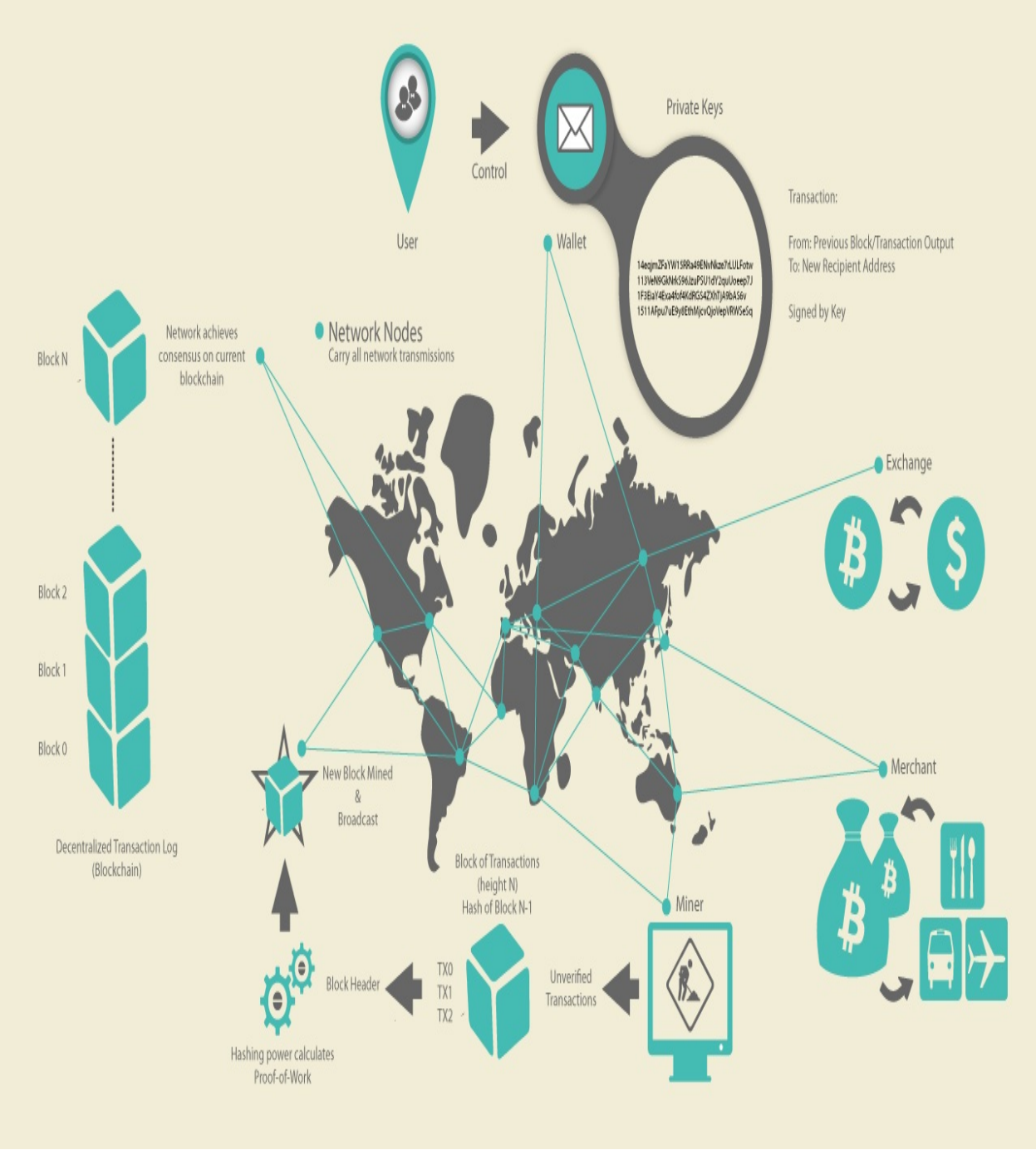


Figure 2-1. Bitcoin overview

Popular blockchain explorers include:

- Blockstream Explorer

- Mempool.Space
- BlockCypher Explorer

Each of these has a search function that can take a Bitcoin address, transaction hash, block number, or block hash and retrieve corresponding information from the Bitcoin network. With each transaction or block example, we will provide a URL so you can look it up yourself and study it in detail.

BLOCK EXPLORER PRIVACY WARNING

Searching information on a block explorer may disclose to its operator that you're interested in that information, allowing them to associate it with your IP address, browser fingerprint, past searches, or other identifiable information. If you look up the transactions in this book, the operator of the block explorer might guess that you're learning about Bitcoin, which shouldn't be a problem. But if you look up your own transactions, the operator may be able to guess how many bitcoins you've received, spent, and currently own.

Buying from an Online Store

Alice, introduced in the previous chapter, is a new user who has just acquired her first bitcoins. In "Getting Your First Bitcoin", Alice met with her friend Joe to exchange some cash for bitcoins. Since then, Alice has bought additional bitcoins. Now Alice will make her first retail transaction, buying access to a premium podcast episode from Bob's online store.

Bob's web store recently started accepting bitcoin payments by adding a bitcoin option to its website. The prices at Bob's store are listed in the local currency

(US dollars), but at checkout, customers have the option of paying in either dollars or bitcoin.

Alice finds the podcast episode she wants to buy and proceeds to the checkout page. At checkout, Alice is offered the option to pay with bitcoin, in addition to the usual options. The checkout cart displays the price in US dollars and also in bitcoin (BTC), at Bitcoin's prevailing exchange rate.

Bob's e-commerce system will automatically create a QR code containing an *invoice* (Figure 2-2).

Unlike a QR code that simply contains a destination Bitcoin address, this invoice is a QR-encoded URI that contains a destination address, a payment amount, and a description. 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.

Figure 2-2. Invoice QR code

TIP

Try to scan this with your wallet to see the address and amount but DO NOT SEND MONEY.

```
bitcoin:bc1qk2g6u8p4qm2s2lh3gts5cpt2mrv5skcuu7u3e4?a
label=Bob%27s%20Store&
message=Purchase%20at%20Bob%27s%20Store

Components of the URI

A Bitcoin address: "bc1qk2g6u8p4qm2s2lh3gts5cpt2mrv5
The payment amount: "0.01577764"
A label for the recipient address: "Bob's Store"
A description for the payment: "Purchase at Bob's St
```

Alice uses her smartphone to scan the barcode on display. Her smartphone shows a payment for the correct amount to `Bob's Store` and she selects Send to authorize the payment. Within a few seconds (about the same amount of time as a credit card authorization), Bob sees the transaction on the register.

NOTE

The Bitcoin network can transact in fractional values, e.g., from millibitcoin (1/1000th of a bitcoin) down to 1/100,000,000th of a bitcoin, which is known as a satoshi. This book uses the same pluralization rules used for dollars and other traditional currencies when talking about amounts greater than one bitcoin and when using decimal notation, such as “10 bitcoins” or “0.001 bitcoins.” The same rules also apply to other bitcoin bookkeeping units, such as millibitcoins and satoshis.

You can examine Alice's transaction to Bob's Store on the blockchain using a block explorer site (Example 2-1):

Example 2-1. View Alice's transaction on Blockstream Explorer

```
https://blockstream.info/tx/674616f1fbc6cc7482136487
```

In the following sections, we will examine this transaction in more detail. We'll 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.

Bitcoin Transactions

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

Transaction Inputs and Outputs

Transactions are like lines in a double-entry bookkeeping ledger. Each transaction contains one or more "inputs," which are like debits against a bitcoin account. On the other side of the transaction, there are one or more "outputs," which are like 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 Figure 2-3.

The transaction also contains proof of ownership for each amount of bitcoin (inputs) whose value is being spent, 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.

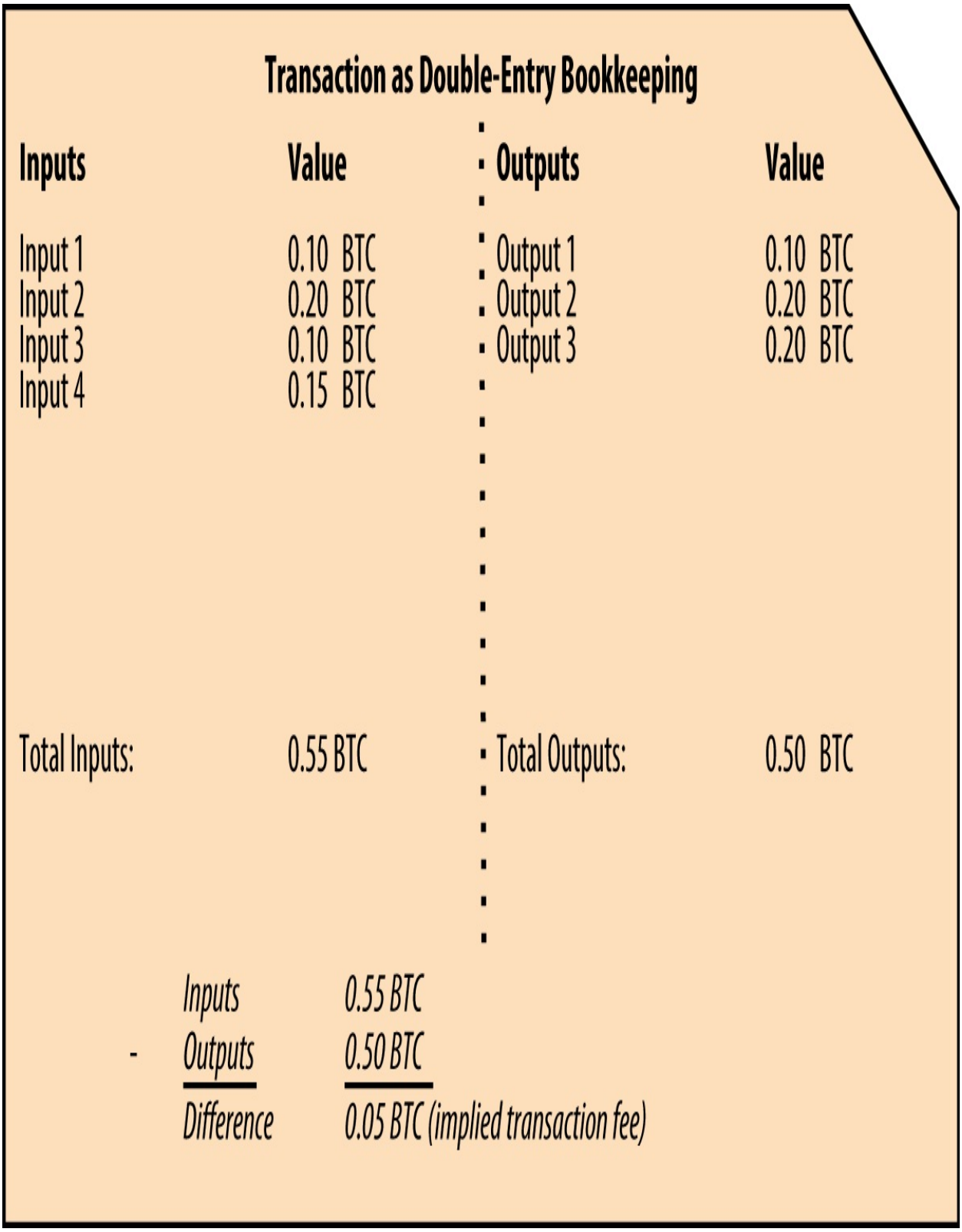


Figure 2-3. Transaction as double-entry bookkeeping

Transaction Chains

Alice's payment to Bob's Store uses a previous transaction's output as its input. In the previous chapter, Alice received bitcoin from her friend Joe in return for cash. We've labeled that as *Transaction 1* (Tx1) in Figure 2-4.

Tx1 sent 0.001 bitcoins (100,000 satoshis) to an output locked by Alice's key. Her new transaction to Bob's Store (Tx2) references the previous output as an input. In the illustration, we show that reference using an arrow and by labeling the input as "Tx1:0". In an actual transaction, the reference is the 32-byte transaction identifier (txid) for the transaction where Alice received the money from Joe. The ":0" indicates the position of the output where Alice received the money; in this case, the first position (position 0).

As shown, actual Bitcoin transactions don't explicitly include the value of their input. To determine the value of an input, software needs to use the input's reference to find the previous transaction output being spent.

Alice's Tx2 contains two new outputs, one paying 75,000 satoshis for the podcast and another paying 20,000 satoshis back to Alice to receive change.

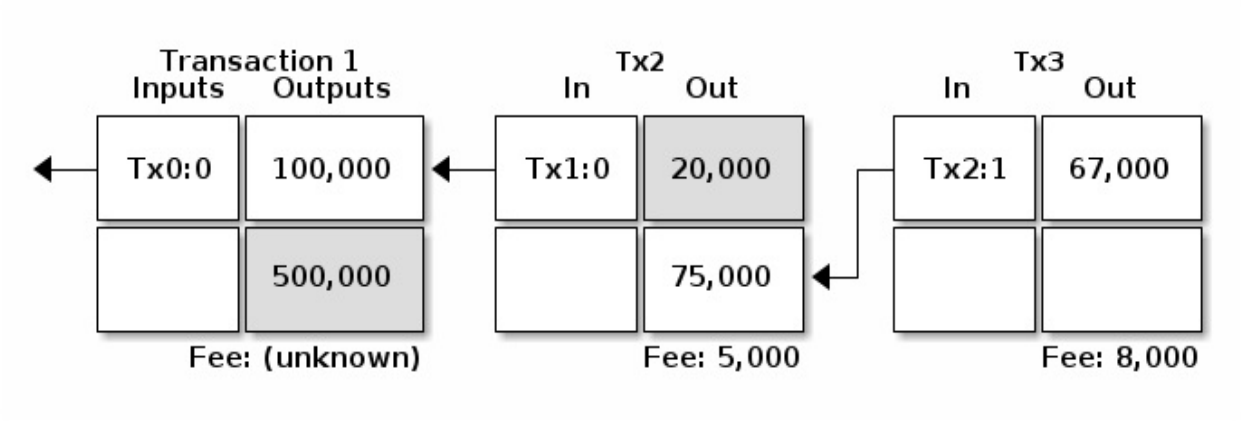


Figure 2-4. A chain of transactions, where the output of one transaction is the input of the next transaction

---

TIP

Serialized Bitcoin transactions---the data format that software uses for sending transactions---encodes the value to transfer using an integer of the smallest defined onchain unit of value. When Bitcoin was first created, this unit didn't have a name and some developers simply called it the *base unit.* Later many users began calling this unit a *satoshi* (sat) in honor of Bitcoin's creator. In Figure 2-4 and some other illustrations in this book, we use satoshi values because that's what the protocol itself uses.

Making Change

In addition to one or more outputs that pay the receiver of bitcoins, many transactions will also include an output that pays the spender of the bitcoins, called a *change* output. This is because transaction inputs, like currency notes, cannot be divided. If you purchase a $5 US dollar item in a store but use a $20 dollar bill to pay for the item, you expect to receive $15 dollars in change. The same concept applies to bitcoin transaction inputs. If you purchased an item that costs 5 bitcoins but only had an input worth 20 bitcoins to use, you would send one output of 5 bitcoins to the store owner and one output of 15 bitcoins back to

yourself as change (not counting your transaction fee).

At the level of the Bitcoin protocol, there is no difference between a change output (and the address it pays, called a *change address*) and a payment output.

Importantly, the change address does not have to be the same address as that of the input and for privacy reasons is often a new address from the owner's wallet. In ideal circumstances, the two different uses of outputs both use never-before-been addresses and otherwise look identical, preventing any third party from determining which outputs are change and which are payments. However, for illustration purposes, we've added shading to the change outputs in Figure 2-4.

Coin selection

Different wallets use different strategies when choosing which inputs to use to a payment, called *coin selection*.

They might aggregate many small inputs, or use one that is equal to or larger than the desired payment. Unless the wallet can aggregate inputs in such a way to exactly match the desired payment plus transaction fees, the wallet will need to generate some change. This is very similar to how people handle cash. If you always use the largest bill in your pocket, you will end up with a pocket full of loose change. If you only use the loose change, you'll always have only big bills. People subconsciously find a balance between these two extremes, and bitcoin wallet developers strive to program this balance.

Common Transaction Forms

A very common form of transaction is a simple payment. This type of transaction has one input and two outputs and is shown in Figure 2-5.

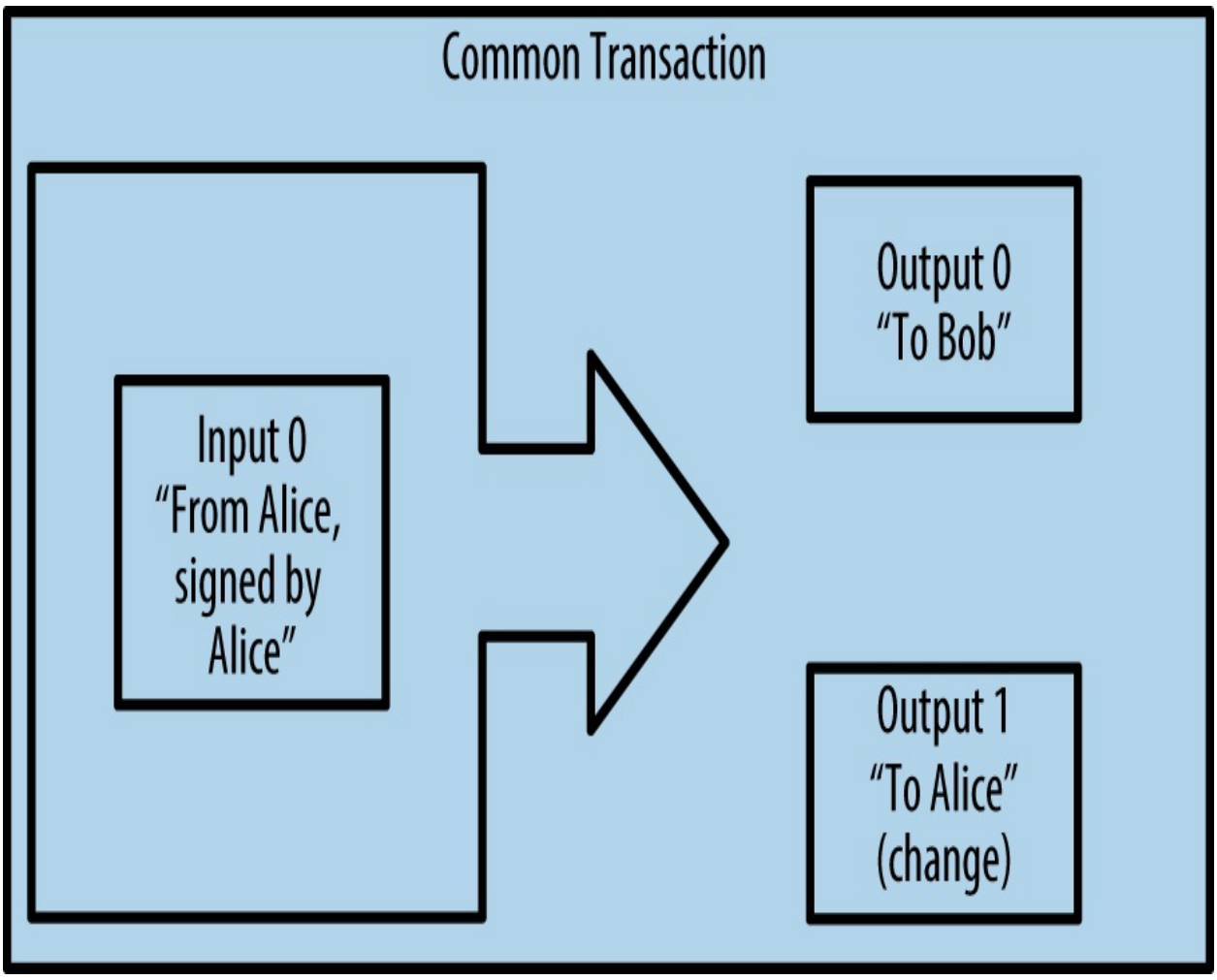


Figure 2-5. Most common transaction

Another common form of transaction is a *consolidation transaction* one that spends several inputs into a single output (Figure 2-6). 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 wallets and

business to clean up lots of smaller amounts.

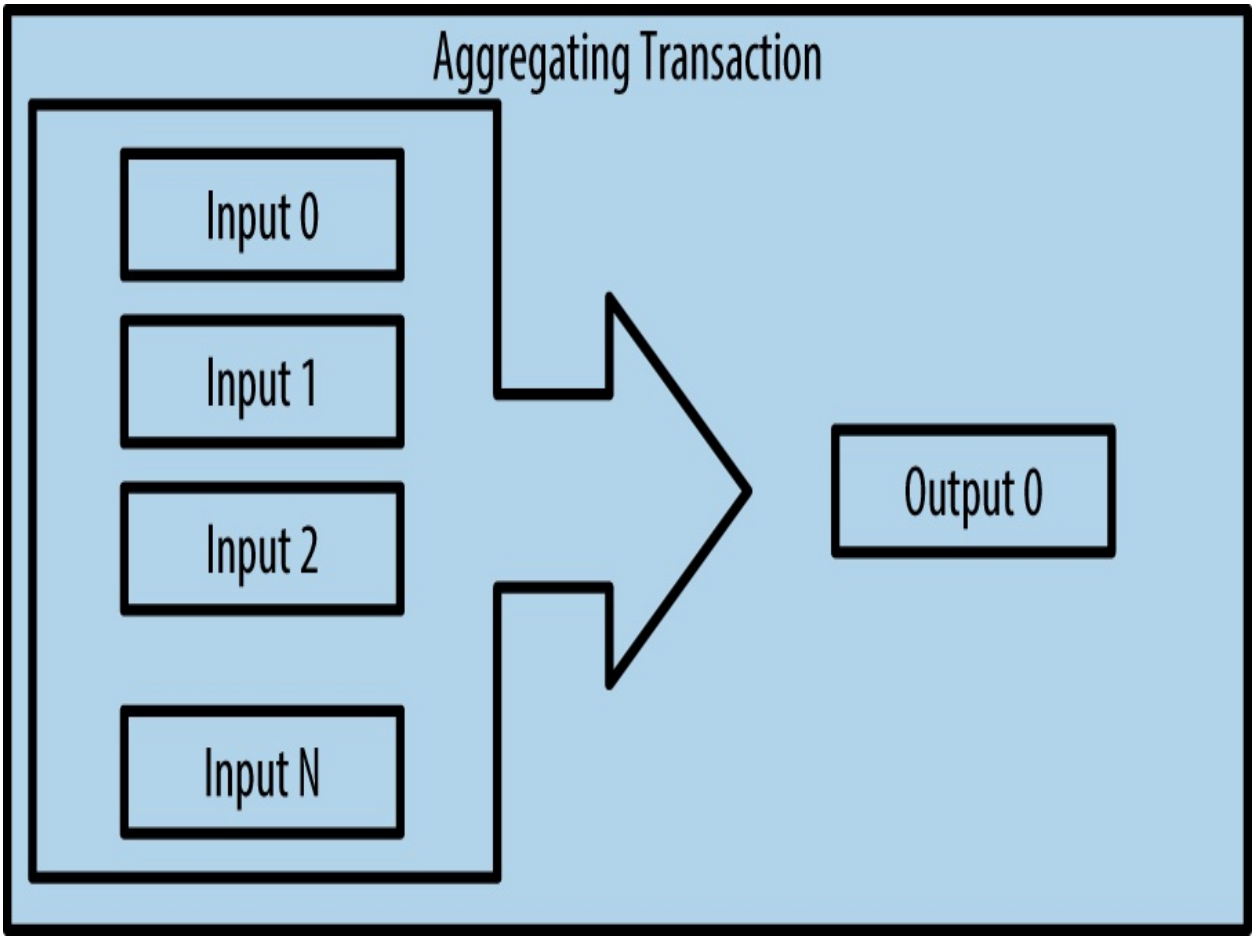


Figure 2-6. Transaction aggregating funds

Finally, another transaction form that is seen often on the bitcoin ledger is *payment batching* that pays to multiple outputs representing multiple recipients (Figure 2-7). This type of transaction is sometimes used by commercial entities to distribute funds, such as when processing payroll payments to multiple employees.

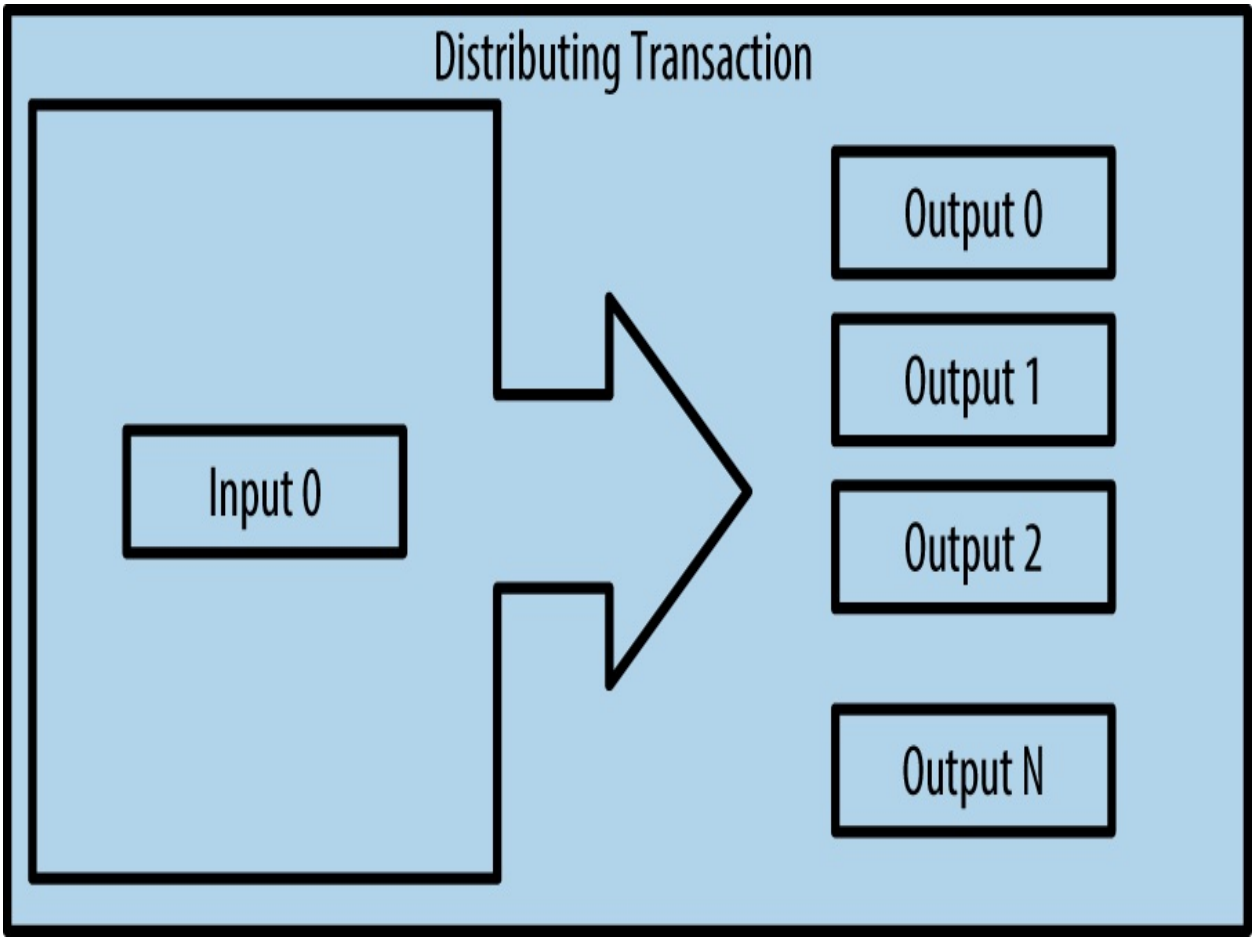


Figure 2-7. Transaction distributing funds

Constructing a Transaction

Alice’s wallet application contains all the logic for selecting inputs and generating outputs to build a transaction to Alice’s specification. Alice only needs to choose a destination, amount, and transaction fee, and the rest happens in the wallet application without her seeing the details. Importantly, if a wallet already knows what inputs it controls, it 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.

Getting the Right Inputs

Alice's wallet application will first have to find inputs that can pay the amount she wants to send to Bob. Most wallets keep track of all the available outputs belonging to addresses in the wallet. 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 Bitcoin"). A bitcoin wallet application that runs on a full node actually contains a copy of every confirmed transaction's unspent outputs, called *Unspent Transaction Outputs* (UTXOs). However, because full nodes use more resources, most user wallets run "lightweight" clients that track only the user's own UTXOs.

If the wallet application does not maintain a copy of all UTXOs, it can query the Bitcoin network to retrieve this information using a variety of APIs available by different providers or by asking a full node using an application programming interface (API) call. Example 2-2 shows an 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. Note that looking up information using a third-party API like this is similar to using a block explorer; see the privacy warning in "Block explorer privacy warning".

Example 2-2. Look up all the unspent outputs for Alice's Bitcoin address

```
$ address=bc1pyfw56zu5vsq0ulu9kytasgw4xwnm3eysll6tfc
$ curl https://blockchain.info/unspent?active=$addre
```

```
{
  "notice": "",
  "unspent_outputs": [
    {
      "tx_hash_big_endian": "4ac5418026798669 35a19d4
      "tx_hash": "eb3ae38f27191aa5f3850dc9cad00492b8
      "tx_output_n": 1,
      "script": "5120225d4d0b946400fe7f85b117d821d53
      "value": 100000,
      "value_hex": "0186a0",
      "confirmations": 111,
      "tx_index": 8276421070086947
    }
  ]
}
```

The response in Example 2-2 shows one unspent output (one that has not been redeemed yet) under the ownership of Alice's address. The response includes the reference to the transaction in which this UTXO is contained (the payment from Joe), the output index number, its value in satoshis, and the script derived from Alice's address. 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.

In this case, this single UTXO is sufficient to pay for the podcast. Had this not been the case, Alice's wallet application might have to combine several smaller UTXOs, like picking coins from a purse until it could find enough to pay for the podcast. 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 contain more money than the cost of the podcast. Alice's change output is created in the very same transaction as the payment to Bob. Essentially, Alice's wallet breaks her funds into two outputs: one to Bob and one back to herself. She can then spend the change output in a subsequent transaction.

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 in value between inputs and outputs. This *transaction fee* is collected by the miner as a fee for validating and including the transaction in a block to be recorded on the blockchain.

TIP

View the transaction from Alice to Bob's Store.

Adding the Transaction to the Ledger

The transaction created by Alice's wallet application 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 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 peer participating by connecting to several other Bitcoin peers. The purpose of the Bitcoin network is

to propagate transactions and blocks to all participants.

How it propagates

Peers in the Bitcoin peer-to-peer network are programs that have both the software logic and the data necessary for them to fully verify the correctness of a new transaction. The connections between peers are often visualized as edges (lines) in a graph, with the peers themselves being the nodes (dots). For that reason, Bitcoin peers are commonly called “full verification nodes”, or *full nodes* for short.

Alice’s wallet application can send the new transaction to any Bitcoin node it is connected to over any type of connection: wired, WiFi, mobile, etc. It can also send the transaction to another program (such as a block explorer) that will relay it to a node. 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 node that receives a valid transaction it has not seen before will immediately forward it to all other nodes to which it is connected, a propagation technique known as *gossiping*. 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 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 an output redeemable by Bob's keys. Bob's wallet application can also independently verify that the transaction is well formed. If Bob is using his own full node, his wallet can further verify Alice's transaction only spends valid UTXOs.

Bitcoin Mining

Alice's transaction is now propagated on the Bitcoin network. It does not become part of the *blockchain* until it is verified and included in a block by a process called *mining*. See XREF HERE for a detailed explanation.

The Bitcoin system of counterfeit protection is based on computation. Transactions are bundled into *blocks*. Blocks have a very small header that must be formed in a very specific way, requiring an enormous amount of computation to get right—but only a small amount of computation to verify as correct. The mining process serves two purposes in bitcoin:

- Miners can only receive honest income from creating blocks that follow all of Bitcoin's *consensus rules*. Therefore, miners are normally incentivized to only include valid transactions in their blocks and the blocks they build upon. This allows users to optionally trust that any transaction in a block is a valid transaction.
- Mining currently creates new bitcoin in each block, almost like a central bank

printing new money. The amount of bitcoin created per block is limited and diminishes with time, following a fixed issuance schedule.

Mining achieves a fine balance between cost and reward. Mining uses electricity to solve a computational problem. A successful miner will collect a *reward* in the form of new bitcoin and transaction fees. However, the reward will only be collected if the miner has correctly validated all the transactions, to the satisfaction of the rules of *consensus*. This delicate balance provides security for bitcoin without a central authority.

Mining is designed to be a decentralized lottery. Each miner can create their own lottery ticket by creating a *block template* that includes the new transactions they want to mine plus some additional data fields. The miner inputs their template into a specially-designed algorithm that scrambles (or “hashes”) the data, producing output that looks nothing like the input data. This *hash* function will always produce the same output for the same input—but nobody can predict what the output will look like for a new input, even if it is only slighly different from a previous input. If the output of hash function matches a template determined by the Bitcoin protocol, the miner wins the lottery and Bitcoin users will accept the block template with its transactions as a valid block. If the output doesn’t match the template, the miner makes a small change to their block template and tries again. As of this writing, the number of block templates miners need to try before finding a winning combination is about 168 billion trillions. That’s also how many times the hash function needs to be run.

However, once a winning combination has been found, anyone can verify the

block is valid by running the hash function just once. That makes a valid block something that requires an incredible amount of work to create but only a trivial amount of work to verify. The simple verification process is able to probabalistically prove the work was done, so the data necessary to generate that proof—in this case, the block—is called Proof-of-Work (PoW).

In "Bitcoin Uses, Users, and Their Stories", we introduced Jing, an entrepreneur in Shanghai. Jing runs a *mining farm*, which is a business that runs thousands of specialized mining computers, competing for the block reward. Jing's mining computers compete against thousands of similar systems in the global lottery to create the next block.

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 Bitcoin protocol automatically increased the difficulty of finding a new block. Soon, Jing and other miners upgraded to more specialized hardware, such as high-end dedicated graphical processing units (GPUs) used in gaming desktops. 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's company also participates in a *mining pool*, which much like a lottery pool allows several participants to share their efforts and rewards. Jing's company now runs a warehouse containing thousands of ASIC miners to mine for bitcoin 24 hours a day. The company pays its electricity costs by selling the bitcoin it is able to generate from mining, creating some income from the profits.

Mining Transactions in Blocks

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 construct a new block, they add unverified transactions from this pool to the new block and then attempt to prove the validity of that new block, with the mining algorithm (Proof-of-Work). The process of mining is explained in detail in XREF HERE.

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 the block reward (currently 12.5 newly created bitcoin) plus the sum of transaction fees from all the transactions included in the 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. Once validated by a full node, it was included in a block template generated by Jing’s mining pool. All the miners participating in that mining pool immediately start trying to generate a Proof-of-Work for the block template. Approximately five minutes after the transaction was first transmitted by Alice’s wallet, one of Jing’s ASIC miners found a solution for the block and announced it to the network. After other miners validated the winning block, they started a new lottery to generate the next block.

Jing’s winning block containing Alice’s transaction became part of the blockchain. The block containing Alice’s transaction is counted as one “confirmation” of that transaction. After the block containing Alice’s transaction has propagated through the network, creating an alternative block with a different version of Alice’s transaction (such as a transaction that doesn’t pay Bob) would require performing the same amount of work as it will take all Bitcoin miners to create an entirely new block. For the entire network to accept an alternative block, an additional new block would need to be mined on top of the alternative.

That means miners have a choice. They can work with Alice on an alternative version of the transaction where she pays Bob, perhaps with Alice paying miners a share of the money she previously paid Bob. This dishonest behavior will require they expend the effort required to create two new blocks. Instead, miners who behave honestly can create a single new block and and receive all of the fees from the transactions they include in it, plus the block reward. Normally, the high cost of dishonestly creating two blocks for a small additional payment is

much less profitable than honestly creating a new block, making it unlikely that a confirmed transaction will be deliberately changed. For Bob, this means that he can begin to believe that the payment from Alice can be relied upon.

TIP

You can see the block that includes Alice's transaction.

Approximately 19 minutes after Jing's block, a new block is mined by another miner. Because this new block is built on top of the block that contained Alice's transaction (giving Alice's transaction two confirmations) Alice's transaction can now only be changed if two alternative blocks are mined—plus a new block built on top of them—for a total of three blocks that would need to be mined for Alice to take back the money she sent Bob. Each block mined on top of the one containing Alice's transaction counts as an additional confirmation. As the blocks pile on top of each other, it becomes harder to reverse the transaction, thereby giving Bob more and more confidence that Alice's payment is secure.

In Figure 2-8, we can the block which contains Alice's transaction. Below it are hundreds of thousands of blocks, 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" of new blocks increases, so does the computation difficulty for the chain as a whole. By convention, any block with more than six confirmations is considered very hard to change, because it would require an immense amount of computation to recalculate six blocks (plus one new block). We will examine the process of mining and the way it builds confidence in more

detail in XREF HERE.

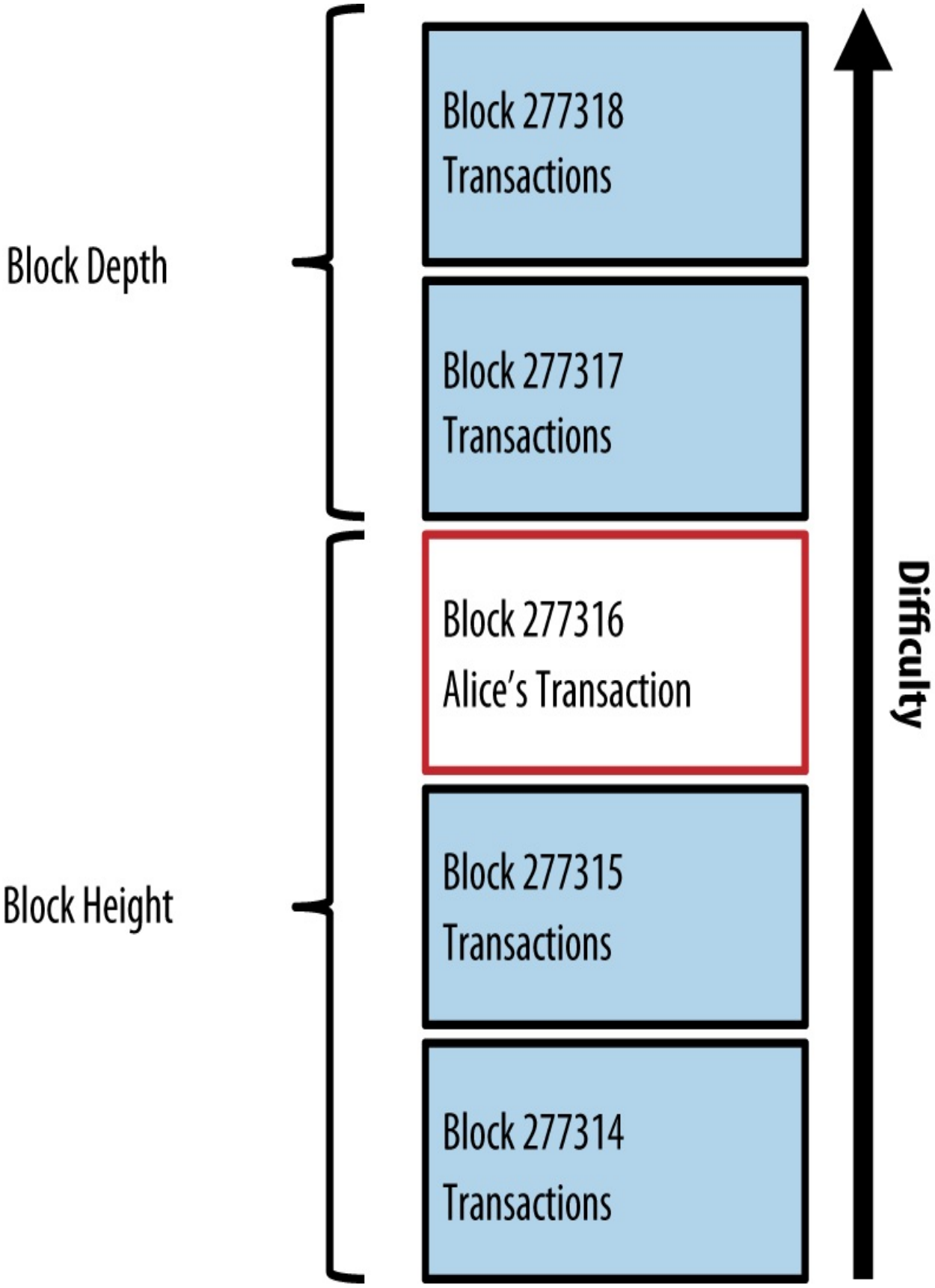

Block Depth
Block Height
Block 277318
Transactions
Block 277317
Transactions
Block 277316
Alice's Transaction
Block 277315
Transactions
Block 277314
Transactions
Difficulty

Figure 2-8. Alice’s transaction included in a block

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 full node can independently verify the transaction as valid and spendable. Full nodes validate every transfer of the funds from the moment the bitcoin were first generated in a block through each subsequent transaction until they reach Bob’s address. Lightweight clients can do what is called a simplified payment verification (see XREF HERE) by confirming that the transaction is in the blockchain and has several blocks mined after it, thus providing assurance that the miners expended significant effort committing to it.

Bob can now spend the output from this and other transactions. For example, Bob can pay a contractor or supplier by transferring value from Alice’s podcast payment to these new owners. Bob’s bitcoin software might consolidate many small payments into a larger payment, perhaps concentrating all the day’s bitcoin revenue into a single transaction. This would consolidate the various payments into a single output (and a single address). For a diagram of a consolidation transaction, see Figure 2-6.

As Bob spends the payments received from Alice and other customers, he extends the chain of transactions. 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 Figure 2-9.

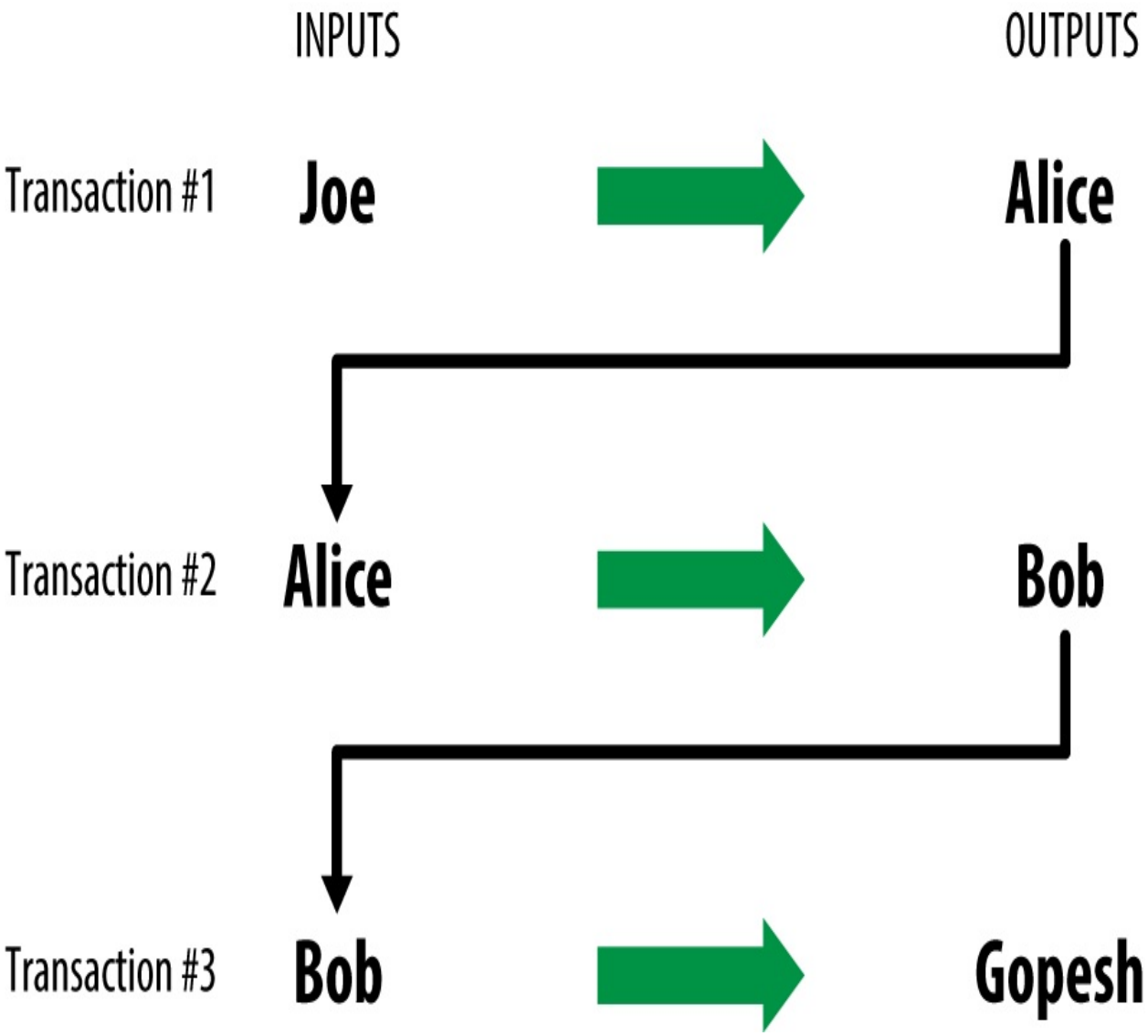


Figure 2-9. Alice’s transaction as part of a transaction chain from Joe to Gopesh

In this chapter, we saw how transactions build a chain that moves value from owner to owner. We also tracked Alice’s transaction, from the moment it was created in her wallet, through the Bitcoin network and to the miners who recorded it on the blockchain. In the rest of this book, we will examine the specific technologies behind wallets, addresses, signatures, transactions, the network, and finally mining.

Chapter 3. Bitcoin Core: The Reference Implementation

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

People only accept money in exchange for their valuable goods and services if they believe that they'll be able to spend that money later. Money that is counterfeit or unexpectedly debased may not be spendable later, so every person accepting bitcoin has a strong incentive to verify the integrity of the bitcoins they receive. The Bitcoin system was designed so that it's possible for software running entirely on your local computer to perfectly prevent counterfeiting, debasement, and several other critical problems. Software which provides that function is called a full verification node because it verifies every confirmed Bitcoin transaction against every rule in the system. Full verification nodes, *full nodes* for short, may also provide tools and data for understanding how Bitcoin

works and what is currently happening in the network.

In this chapter, we'll install Bitcoin Core, the implementation which most full node operators have used since the beginning of the Bitcoin network. We'll then inspect blocks, transactions, and other data from your node, data which is authoritative—not because some powerful entity designated it as such but because your node independently verified it. Throughout the rest of this book, we'll continue using Bitcoin Core to create and examine data related to the blockchain and network.

From Bitcoin to Bitcoin Core

Bitcoin is an *open source* project and the source code is available under an open (MIT) license, free to download and use for any purpose. More than just being open source, Bitcoin is developed by an open community of volunteers. At first, that community consisted of only Satoshi Nakamoto. By 2023, Bitcoin's source code had more than 1,000 contributors with about a dozen developers working on the code almost full-time and several dozen more on a part-time basis. Anyone can contribute to the code—including you!

When Bitcoin was created by Satoshi Nakamoto, the software was mostly completed before the whitepaper reproduced in XREF HERE was published. Satoshi wanted to make sure the implementation worked before publishing a paper about it. That first implementation, then simply known as "Bitcoin", has been heavily modified and improved. It has evolved into what is known as *Bitcoin Core*, to differentiate it from other implementations. Bitcoin Core is the

reference implementation of the Bitcoin system, meaning that it provides a reference for how each part of the technology should be implemented. Bitcoin Core implements all aspects of Bitcoin, including wallets, a transaction and block validation engine, and all modern parts of Bitcoin peer-to-peer communication.

Figure 3-1 shows the architecture of Bitcoin Core.

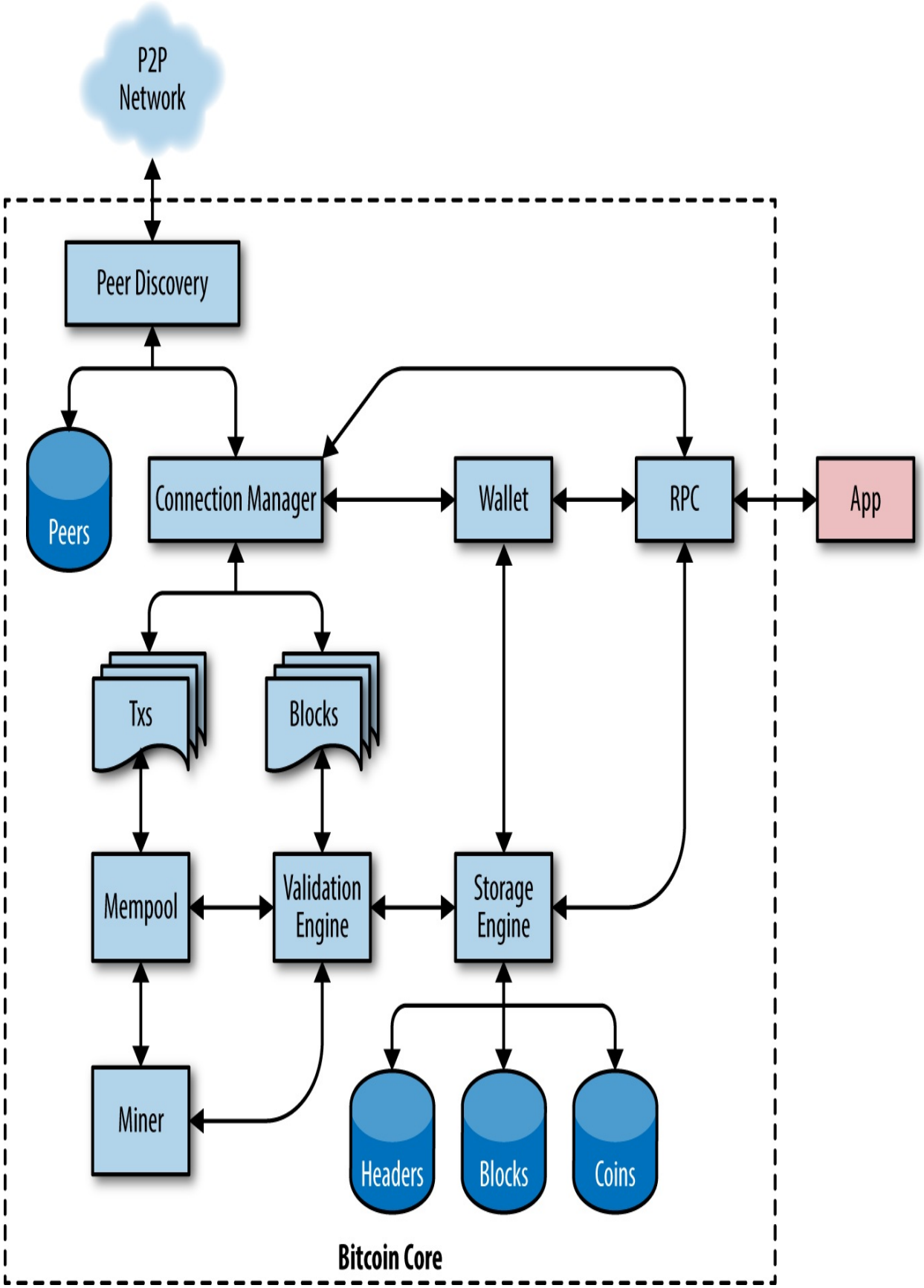

P2P
Network
Peer Discovery
Peers
Connection Manager
Wallet
RPC
App
Txs
Blocks
Mempool
Validation
Engine
Storage
Engine
Miner
Headers
Blocks
Coins
Bitcoin Core

Figure 3-1. Bitcoin Core architecture (Source: Eric Lombrozo)

Bitcoin Development Environment

If you're a developer, you will want to set up a development environment with all the tools, libraries, and support software for writing Bitcoin applications. In this highly technical chapter, we'll walk through that process step-by-step. If the material becomes too dense (and you're not actually setting up a development environment) feel free to skip to the next chapter, which is less technical.

Compiling Bitcoin Core from the Source Code

Bitcoin Core's source code can be downloaded as an archive or by cloning the authoritative source repository from GitHub. On the Bitcoin Core download page, select the most recent version and download the compressed archive of the source code. Alternatively, use the git command line to create a local copy of the source code from the GitHub bitcoin page.

TIP

In many of the examples in this chapter we will be using the operating system's command-line interface (also known as a "shell"), accessed via a "terminal" application. The shell will display a prompt; you type a command; and the shell responds with some text and a new prompt for your next command. The prompt may look different on your system, but in the following examples it is denoted by a `$` symbol. In the examples, when you see text after a `$` symbol, don't type the `$` symbol but type the command

immediately following it, then press Enter to execute the command. In the examples, the lines below each command are the operating system's responses to that command. When you see the next `$` prefix, you'll know it's a new command and you should repeat the process.

In this example, we are using the `git` command to create a local copy ("clone") of the source code:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Enumerating objects: 245912, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 245912 (delta 1), reused 2 (delta 1),
Receiving objects: 100% (245912/245912), 217.74 MiB
Resolving deltas: 100% (175649/175649), done.
```

TIP

Git is the most widely used distributed version control system, an essential part of any software developer's toolkit. You may need to install the `git` command, or a graphical user interface for git, on your operating system if you do not have it already.

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 using the `cd` command:

```
$ cd bitcoin
```

Selecting a Bitcoin Core Release

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
v0.10.0
...
v0.11.2
v0.11.2rc1
v0.12.0rc1
v0.12.0rc2
...
```

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 the time of writing was v24.0.1. To synchronize the local code with this version, use the `git checkout` command:

```
$ git checkout v24.0.1
Note: switching to 'v24.0.1'.

You are in 'detached HEAD' state. You can look arour
changes and commit them, and you can discard any com
state without impacting any branches by switching ba

HEAD is now at b3f866a8d Merge bitcoin/bitcoin#26647
```

You can confirm you have the desired version "checked out" by issuing the command `git status`:

```
HEAD detached at v24.0.1
nothing to commit, working tree clean
```

Configuring the Bitcoin Core Build

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. In this chapter, we will build the Bitcoin Core daemon (server), also known as `bitcoind` on Linux (a Unix-like system). Review the instructions for compiling the `bitcoind` command-line client on your platform by

reading `doc/build-unix.md`. Alternative instructions can be found in the *doc* directory; for example, *build-windows.md* for Windows instructions. As of this writing, instructions are available for Android, FreeBSD, NetBSD, OpenBSD, MacOS (OSX), Unix, and Windows.

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.

```
$ ./autogen.sh
libtoolize: putting auxiliary files in AC_CONFIG_AUX
libtoolize: copying file 'build-aux/ltmain.sh'
libtoolize: putting macros in AC_CONFIG_MACRO_DIRS,
 ...
configure.ac:58: installing 'build-aux/missing'
src/Makefile.am: installing 'build-aux/depcomp'
parallel-tests: installing 'build-aux/test-driver'
```

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. Use the `--help` flag to see the various options:

```
$ ./configure --help
`configure' configures Bitcoin Core 24.0.1 to adapt

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

...
Optional Features:
  --disable-option-checking  ignore unrecognized --e
  --disable-FEATURE       do not include FEATURE (sa
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --enable-silent-rules   less verbose build output
  --disable-silent-rules  verbose build output (undo
...
```

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. If you are in an academic setting, computer lab restrictions may require you to install applications in your home directory (e.g., using `--prefix=$HOME`).

Here are some useful options that override the default behavior of the configure script:

`--prefix=$HOME`

This overrides the default installation location (which is */usr/local/*) for the resulting executable. Use `$HOME` to put everything in your home directory, or a different path.

`--disable-wallet`

This is used to disable the reference wallet implementation.

`--with-incompatible-bdb`

If you are building a wallet, allow the use of an incompatible version of the Berkeley DB library.

`--with-gui=no`

Don’t build the graphical user interface, which requires the Qt library. This builds server and command-line bitcoin only.

Next, run the `configure` script to automatically discover all the necessary libraries and create a customized build script for your system:

```
$ ./configure
checking for pkg-config... /usr/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking build system type... x86_64-pc-linux-gnu
checking host system type... x86_64-pc-linux-gnu
checking for a BSD-compatible install... /usr/bin/ir
...
[many pages of configuration tests follow]
...
```

If all went 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.

Building the Bitcoin Core Executables

Next, you will compile the source code, a process that can take up to an hour to complete, depending on the speed of your CPU and available memory. During the compilation process you should see output every few seconds or every few minutes, or an error if something goes wrong. If an error occurs, or the compilation process is interrupted, it can be resumed any time by typing `make` again. Type `make` to start compiling the executable application:

```
$ make
Making all in src
  CXX      bitcoind-bitcoind.o
  CXX      libbitcoin_node_a-addrdb.o
  CXX      libbitcoin_node_a-addrman.o
  CXX      libbitcoin_node_a-banman.o
  CXX      libbitcoin_node_a-blockencodings.o
  CXX      libbitcoin_node_a-blockfilter.o
[... many more compilation messages follow ...]
```

On a fast system with more than one CPU, you might want to set the number of parallel compile jobs. For instance, `make -j 2` will use two cores if they are available. If all goes well, Bitcoin Core is now compiled. You should run the unit test suite with `make check` to ensure the linked libraries are not broken in obvious ways. The final step is to install the various executables on your system using the `make install` command. You may be prompted for your user password, because this step requires administrative privileges:

```
$ make check && sudo make install
Password:
Making install in src
 ../build-aux/install-sh -c -d '/usr/local/lib'
libtool: install: /usr/bin/install -c bitcoind /usr/
libtool: install: /usr/bin/install -c bitcoin-cli /u
libtool: install: /usr/bin/install -c bitcoin-tx /us
...
```

The default installation of `bitcoind` puts it in */usr/local/bin*. You can confirm that Bitcoin Core is correctly installed by asking the system for the path of the executables, as follows:

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

$ which bitcoin-cli
```

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

Running a Bitcoin Core Node

Bitcoin’s peer-to-peer network is composed of network “nodes,” run mostly by individuals and some of the businesses that provide Bitcoin services. Those running Bitcoin nodes have a direct and authoritative view of the Bitcoin blockchain, with a local copy of all the spendable bitcoins independently validated by their own system. By running a node, you don’t have to rely on any third party to validate a transaction. Additionally, by using a Bitcoin node to fully validate the transactions you receive to your wallet, you contribute to the Bitcoin network and help make it more robust.

Running a node, however, requires downloading and processing over 500 GB of data initially and about 400 MB of Bitcoin transactions per day. These figures are for 2023 and will likely increase over time. If you shut down your node or get disconnected from the internet for several days, your node will need to download the data that it missed. For example, if you close Bitcoin Core for ten days, you will need to download approximately 4 GB the next time you start it.

Depending on whether you choose to index all transactions and keep a full copy of the blockchain, you may also need a lot of disk space---at least 1 TB if you plan to run Bitcoin Core for several years. By default, Bitcoin nodes also transmit transactions and blocks to other nodes (called “peers”), consuming upload internet bandwidth. If your internet connection is limited, has a low data

cap, or is metered (charged by the gigabit), you should probably not run a Bitcoin node on it, or run it in a way that constrains its bandwidth (see Example 3-2). You may connect your node instead to an alternative network, such as a free satellite data provider like Blockstream Satellite.

TIP

Bitcoin Core keeps a full copy of the blockchain by default, with nearly every transaction that has ever been confirmed on the Bitcoin network since its inception in 2009. This dataset is hundreds of gigabytes in size and is downloaded incrementally over several hours or days, depending on the speed of your CPU and internet connection. Bitcoin Core will not be able to process transactions or update account balances until the full blockchain dataset is downloaded. Make sure you have enough disk space, bandwidth, and time to complete the initial synchronization. You can configure Bitcoin Core to reduce the size of the blockchain by discarding old blocks (see Example 3-2), but it will still download the entire dataset.

Despite these resource requirements, thousands of people run Bitcoin nodes. Some are running on systems as simple as a Raspberry Pi (a $35 USD computer the size of a pack of cards).

Why would you want to run a node? Here are some of the most common reasons:

- You do not want to rely on any third party to validate the transactions you receive.
- You do not want to disclose to third parties which transactions belong to your wallet.
- You are developing Bitcoin software and need to rely on a Bitcoin node for programmable (API) access to the network and blockchain.

- You are building applications that must validate transactions according to Bitcoin's consensus rules. Typically, Bitcoin software companies run several nodes.
- You want to support Bitcoin. Running a node that you use to validate the transactions you receive to your wallet makes the network more robust.

If you're reading this book and interested in strong security, superior privacy, or developing Bitcoin software, you should be running your own node.

Configuring the Bitcoin Core Node

Bitcoin Core will look for a configuration file in its data directory on every start. In this section we will examine the various configuration options and set up a configuration file. To locate the configuration file, run `bitcoind -printtoconsole` in your terminal and look for the first couple of lines.

```
$ bitcoind -printtoconsole
2023-01-28T03:21:42Z Bitcoin Core version v24.0.1
2023-01-28T03:21:42Z Using the 'x86_shani(1way,2way)
2023-01-28T03:21:42Z Using RdSeed as an additional e
2023-01-28T03:21:42Z Using RdRand as an additional e
2023-01-28T03:21:42Z Default data directory /home/ha
2023-01-28T03:21:42Z Using data directory /home/hard
2023-01-28T03:21:42Z Config file: /home/harding/.bit
...
[a lot more debug output]
...
```

You can hit Ctrl-C to shut down the node once you determine the location of the config file. Usually the configuration file is inside the *.bitcoin* data directory under your user's home directory. Open the configuration file in your preferred editor.

Bitcoin Core offers more than 100 configuration options that modify the behavior of the network node, the storage of the blockchain, and many other aspects of its operation. To see a listing of these options, run `bitcoind --help`:

```
$ bitcoind --help
Bitcoin Core version v24.0.1

Usage:  bitcoind [options]                     Start

Options:

  -?
       Print this help message and exit

  -alertnotify=<cmd>
       Execute command when an alert is raised (%s i
       message)
...
[many more options]
```

Here are some of the most important options that you can set in the configuration

file, or as command-line parameters to `bitcoind`:

alertnotify
Run a specified command or script to send emergency alerts to the owner of this node.

conf
An alternative location for the configuration file. This only makes sense as a command-line parameter to `bitcoind`, as it can't be inside the configuration file it refers to.

datadir
Select the directory and filesystem in which to put all the blockchain data. By default this is the *.bitcoin* subdirectory of your home directory. Make sure this filesystem has several gigabytes of free space.

prune
Reduce the disk space requirements to this many megabytes, by deleting old blocks. Use this on a resource-constrained node that can't fit the full blockchain.

txindex
Maintain an index of all transactions. This allows you to programmatically retrieve any transaction by its ID provided that the block containing that transaction hasn't been pruned.

dbcache

The size of the UTXO cache. The default is 450 MiB. Increase this size on high-end hardware to read and write from your disk less often, or reduce the size on low-end hardware to save memory at the expense of using your disk more frequently.

blocksonly
Minimize your bandwidth usage by only accepting blocks of confirmed transactions from your peers instead of relaying unconfirmed transactions.

maxmempool
Limit the transaction memory pool to this many megabytes. Use it to reduce memory use on memory-constrained nodes.

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 `getrawtransaction` (see "Exploring and Decoding Transactions"), 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. If you don't set this option at first and later set it to full indexing, you need to wait for it to rebuild the index.

Example 3-1 shows how you might combine the preceding options, with a fully indexed node, running as an API backend for a bitcoin application.

Example 3-1. Sample configuration of a full-index node

```
alertnotify=myemailscript.sh "Alert: %s"
datadir=/lotsofspace/bitcoin
txindex=1
```

Example 3-2 shows a resource-constrained node running on a smaller server.

Example 3-2. Sample configuration of a resource-constrained system

```
alertnotify=myemailscript.sh "Alert: %s"
blocksonly=1
prune=5000
dbcache=150
maxmempool=150
```

Once you've edited the configuration file and set the options that best represent your needs, you can test `bitcoind` with this configuration. Run Bitcoin Core with the option `printtoconsole` to run in the foreground with output to the console:

```
$ bitcoind -printtoconsole
2023-01-28T03:43:39Z Bitcoin Core version v24.0.1
2023-01-28T03:43:39Z Using the 'x86_shani(1way,2way)
2023-01-28T03:43:39Z Using RdSeed as an additional e
2023-01-28T03:43:39Z Using RdRand as an additional e
2023-01-28T03:43:39Z Default data directory /home/ha
```

```
2023-01-28T03:43:39Z Using data directory /lotsofspa
2023-01-28T03:43:39Z Config file: /home/harding/.bit
2023-01-28T03:43:39Z Config file arg: [main] blockfi
2023-01-28T03:43:39Z Config file arg: [main] maxuplo
2023-01-28T03:43:39Z Config file arg: [main] txindex
2023-01-28T03:43:39Z Setting file arg: wallet = ["ms
2023-01-28T03:43:39Z Command-line arg: printtoconsol
2023-01-28T03:43:39Z Using at most 125 automatic cor
2023-01-28T03:43:39Z Using 16 MiB out of 16 MiB requ
2023-01-28T03:43:39Z Using 16 MiB out of 16 MiB requ
2023-01-28T03:43:39Z Script verification uses 3 addi
2023-01-28T03:43:39Z scheduler thread start
2023-01-28T03:43:39Z [http] creating work queue of d
2023-01-28T03:43:39Z Using random cookie authenticat
2023-01-28T03:43:39Z Generated RPC authentication co
2023-01-28T03:43:39Z [http] starting 4 worker thread
2023-01-28T03:43:39Z Using wallet directory /lotsofs
2023-01-28T03:43:39Z init message: Verifying wallet(
2023-01-28T03:43:39Z Using BerkeleyDB version Berkel
2023-01-28T03:43:39Z Using /16 prefix for IP bucketi
2023-01-28T03:43:39Z init message: Loading P2P addre
2023-01-28T03:43:39Z Loaded 63866 addresses from pee
[... more startup messages ...]
```

You can hit Ctrl-C to interrupt the process once you are satisfied that it is loading the correct settings and running as you expect.

To run Bitcoin Core in the background as a process, start it with the `daemon`

option, as `bitcoind -daemon`.

To monitor the progress and runtime status of your Bitcoin node, start it in daemon mode and then use the command `bitcoin-cli getblockchaininfo`:

```
$ bitcoin-cli getblockchaininfo
```


```
{
  "chain": "main",
  "blocks": 0,
  "headers": 83999,
  "bestblockhash": "000000000019d6689c085ae165831e93
  "difficulty": 1,
  "time": 1673379796,
  "mediantime": 1231006505,
  "verificationprogress": 3.783041623201835e-09,
  "initialblockdownload": true,
  "chainwork": "0000000000000000000000000000000000000
  "size_on_disk": 89087,
  "pruned": false,
  "warnings": ""
}
```


This shows a node with a blockchain height of 0 blocks and 83999 headers. The node first fetches the block headers from its peers in order to find the blockchain with the most proof of work and afterward continues to download the full

blocks, validating them as it goes.

Once you are happy with the configuration options you have selected, you should add bitcoin to the startup scripts in your operating system, so that it runs continuously and restarts when the operating system restarts. You will find a number of example startup scripts for various operating systems in bitcoin's source directory under *contrib/init* and a *README.md* file showing which system uses which script.

Bitcoin Core Application Programming Interface (API)

Bitcoin Core 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 Core RPC commands:

```
$ bitcoin-cli help
+== Blockchain ==
getbestblockhash
getblock "blockhash" ( verbosity )
getblockchaininfo
...
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphras
```

```
  walletprocesspsbt "psbt" ( sign "sighashtype" bip32d
```

Each of these commands may take a number of parameters. To get additional help, a detailed description, and information on the parameters, add the command name after `help`. For example, to see help on the `getblockhash` RPC command:

```
  $ bitcoin-cli help getblockhash
  getblockhash height

  Returns hash of block in best-block-chain at height

  Arguments:
  1. height    (numeric, required) The height index

  Result:
  "hex"    (string) The block hash

  Examples:
  > bitcoin-cli getblockhash 1000
  > curl --user myusername --data-binary '{"jsonrpc":
```

At the end of the help information you will see two examples of the RPC command, using the `bitcoin-cli` helper or the HTTP client `curl`. These examples demonstrate how you might call the command. Copy the first example and see the result:

```
$ bitcoin-cli getblockhash 1000
00000000c937983704a73af28acdec37b049d214adbda81d7e2a
```

The result is a block hash, which is described in more detail in the following chapters. But for now, this command should return the same result on your system, demonstrating that your Bitcoin Core node is running, is accepting commands, and has information about block 1000 to return to you.

In the next sections we will demonstrate some very useful RPC commands and their expected output.

Getting Information on Bitcoin Core's Status

Bitcoin Core provides status reports on diffent modules through the JSON-RPC interface. The most important commands include `getblockchaininfo`, `getmempoolinfo`, `getnetworkinfo` and `getwalletinfo`.

Bitcoin's `getblockchaininfo` RPC command was introduced earlier. The `getnetworkinfo` command displays basic information about the status of the Bitcoin network node. Use `bitcoin-cli` to run it:

```
$ bitcoin-cli getnetworkinfo
```

```
{
  "version": 240001,
```

```
    "subversion": "/Satoshi:24.0.1/",
    "protocolversion": 70016,
    "localservices": "0000000000000409",
    "localservicesnames": [
      "NETWORK",
      "WITNESS",
      "NETWORK_LIMITED"
    ],
    "localrelay": true,
    "timeoffset": -1,
    "networkactive": true,
    "connections": 10,
    "connections_in": 0,
    "connections_out": 10,
    "networks": [
      ...
      detailed information about all networks (ipv4, i
      ...
    ],
    "relayfee": 0.00001000,
    "incrementalfee": 0.00001000,
    "localaddresses": [
    ],
    "warnings": ""
  }
```

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 Core software and Bitcoin protocol. We see the current number of connections and various information about the Bitcoin network and the settings related to this node.

TIP

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

Exploring and Decoding Transactions

In “Buying from an Online Store”, Alice made a purchase from Bob’s store. Her transaction was recorded on the blockchain. Let’s use the API to retrieve and examine that transaction by passing the txid as a parameter:

```
$ bitcoin-cli getrawtransaction 4662003086962l5bbc94
38ecdfdfaa2a8029c1f9bcecd1f96177

01000000000101eb3ae38f27191aa5f3850dc9cad00492b88b72
8679268041c54a0100000000ffffffff02204e0000000000022
4c9ace578369740f15e5ec880c28279ee7f51b07dca69c7061e0
00000001600147752c165ea7be772b2c0acb7f4d6047ae6f476
2d8ef13ed0af21d4f4cb82422d6252d70324f6f4576b727b7d91
e739df2f899c49dc267c0ad280aca6dab0d2fa2b42a45182fc83
0000
```

TIP

A transaction ID (txid) is not authoritative. Absence of a txid in the blockchain does not mean the transaction was not processed. This is known as “transaction malleability,” because transactions can be modified prior to confirmation in a block, changing their txids. After a transaction is included in a block, its txid cannot change unless there is a blockchain reorganization where that block is removed from the best blockchain. Reorganizations are rare after a transaction has several confirmations.

The command `getrawtransaction` returns a serialized transaction in hexadecimal notation. To decode that, we use the `decoderawtransaction` command, passing the hex data as a parameter. You can copy the hex returned by `getrawtransaction` and paste it as a parameter to `decoderawtransaction`:

```
$ bitcoin-cli decoderawtransaction 01000000000101eb3
0492b88b72404f9da1356986792680 41c54a0100000000ffffff
03b41daba4c9ace578369740f15e5ec880c28279ee7f51b07dca
00001600147752c165ea7be772b2c0acb7f4d6047ae6f4768e01
4cb82422d6252d70324f6f4576b727b7d918e521c00b51be739c
dab0d2fa2b42a45182fc83e817130100000000
```

```
{
  "txid": "466200308696215bbc949d5141a49a4138ecdfdfa
  "hash": "f7cdbc7cf8b910d35cc69962e791138624e4eae79
  "version": 1,
  "size": 194,
```

```
    "vsize": 143,
    "weight": 569,
    "locktime": 0,
    "vin": [
      {
        "txid": "4ac5418026798669 35a19d4f40728bb89204c
        "vout": 1,
        "scriptSig": {
          "asm": "",
          "hex": ""
        },
        "txinwitness": [
          "cf5efe2d8ef13ed0af21d4f4cb82422d6252d70324f
        ],
        "sequence": 4294967295
      }
    ],
    "vout": [
      {
        "value": 0.00020000,
        "n": 0,
        "scriptPubKey": {
          "asm": "1 3b41daba4c9ace578369740f15e5ec880c
          "desc": "rawtr(3b41daba4c9ace578369740f15e5e
          "hex": "51203b41daba4c9ace578369740f15e5ec88
          "address": "bc1p8dqa4wjvnt890qmfws83te0v3qxz
          "type": "witness_v1_taproot"
        }
      },
      {
```

```
        "value": 0.00075000,
        "n": 1,
        "scriptPubKey": {
          "asm": "0 7752c165ea7be772b2c0acb7f4d6047ae6
          "desc": "addr(bc1qwafvze0200nh9vkq4jmlf4sy0t
          "hex": "00147752c165ea7be772b2c0acb7f4d6047a
          "address": "bc1qwafvze0200nh9vkq4jmlf4sy0tn0
          "type": "witness_v0_keyhash"
        }
      }
    ]
  }
```


The transaction decode shows all the components of this transaction, including the transaction inputs and outputs. In this case we see that the transaction 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`). The two outputs correspond to the payment to Bob and the change back to Alice.

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

Exploring Blocks

Exploring blocks is similar to exploring transactions. However, blocks can be referenced either by the block *height* or by the block *hash*. First, let's find a block by its height. We use the `getblockhash` command, which takes the block height as the parameter and returns the block *header hash* for that block:

```
$ bitcoin-cli getblockhash 123456
000000000002917ed80650c6174aac8dfc46f5fe36480aaef68
```

Now that we know the *header hash* for our chosen block, we can query that block. We use the `getblock` command with the block hash as the parameter:

```
$ bitcoin-cli getblock 000000000002917ed80650c6174a
f6cd83c3ca
```


```
{
  "hash": "000000000002917ed80650c6174aac8dfc46f5fe
  "confirmations": 651742,
  "height": 123456,
  "version": 1,
  "versionHex": "00000001",
  "merkleroot": "0e60651a9934e8f0decd1c5fde39309e48f
  "time": 1305200806,
  "mediantime": 1305197900,
  "nonce": 2436437219,
  "bits": "1a6a93b3",
  "difficulty": 157416.4018436489,
```

```
    "chainwork": "0000000000000000000000000000000000000
    "nTx": 13,
    "previousblockhash": "0000000000000b60bc96a44724fc
    "nextblockhash": "00000000000129f5f02be247070bf73
    "strippedsize": 4179,
    "size": 4179,
    "weight": 16716,
    "tx": [
      "5b75086dafeede555fc8f9a810d8b10df57c46f9f176ccc
      "e3d0425ab346dd5b76f44c222a4bb5d16640a4247050ef8
      "137d247eca8b99dee58e1e9232014183a5c5a9e338001a0
      "5fd167f7b8c417e59106ef5acfe181b09d71b8353a61a55
      "60925f1948b71f429d514ead7ae7391e0edf965bf5a6033
      "d4d5fc1529487527e9873256934dfb1e4cdcb39f4c05095
      "7b29d65e5018c56a33652085dbb13f2df39a1a9942bfe1f
      "0b89e120efd0a4674c127a76ff5f7590ca304e6a064fbc5
      "603f2044da9656084174cfb5812feaf510f862d3addcf70
      "9a4ed892b43a4df916a7a1213b78e83cd83f5695f635d53
      "dda726e3dad9504dce5098dfab5064ecd4a7650bfe854bb
      "e46ea8b4d68719b65ead930f07f1f3804cb3701014f8e6d
      "864a102aeedf53dd9b2baab4eeb898c5083fde6141113e0
    ]
  }
```


The `confirmations` entry tells us the *depth* of this block—how many blocks have been built on top of it, indicating the difficulty of changing any of the transactions in this block. The `height` tells us how many blocks preceeded this block. We see the block's version, the time it was created (according to its

miner), the median time of the 11 blocks that preceed this block (a time measurement that's harder for miners to manipulate), and the size of the block in three different measurements (its legacy stripped size, it's full size, and its size in weight units). We also see some fields used for security and proof-of-work (merkle root, nonce, bits, difficulty, and chainwork); we'll examine those in detail in XREF HERE.

Using Bitcoin Core's Programmatic Interface

The `bitcoin-cli` helper is very useful for exploring the Bitcoin Core API and testing functions. But the whole point of an application programming interface is to access functions programmatically. In this section we will demonstrate accessing Bitcoin Core from another program.

Bitcoin Core's API is a JSON-RPC interface. JSON stands for JavaScript Object Notation and it is a very convenient way to present data that both humans and programs can easily read. RPC stands for Remote Procedure Call, which means that we are calling procedures (functions) that are remote (on the Bitcoin Core node) via a network protocol. In this case, the network protocol is HTTP.

When we used the `bitcoin-cli` command to get help on a command, it showed us an example of using `curl`, the versatile command-line HTTP client to construct one of these JSON-RPC calls:

```
$ curl --user myusername --data-binary '{"jsonrpc":
```

This command shows that `curl` submits an HTTP request to the local host (127.0.0.1), connecting to the default Bitcoin RPC port (8332), and submitting a `jsonrpc` request for the `getblockchaininfo` method using `text/plain` encoding.

You might notice that curl will ask for credentials to be sent along with the request. Bitcoin Core will create a random password on each start and place it in the data directory under the name `.cookie`. The `bitcoin-cli` helper can read this password file given the data directory. Similarly, you can copy the password and pass it to curl (or any higher level Bitcoin Core RPC wrappers). Alternatively, you can create a static password with the helper script provided in *./share/rpcuser/rpcuser.py* in Bitcoin Core's source directory.

If you're implementing a JSON-RPC call in your own program, you can use a generic HTTP library to construct the call, similar to what is shown in the preceding `curl` example.

However, there are libraries in most popular programming languages that "wrap" the Bitcoin Core API in a way that makes this a lot simpler. We will use the `python-bitcoinlib` library to simplify API access. Remember, this requires you to have a running Bitcoin Core instance, which will be used to make JSON-RPC calls.

The Python script in Example 3-3 makes a simple `getblockchaininfo` call and prints the `block` parameter from the data returned by Bitcoin Core.

Example 3-3. Running getblockchaininfo via Bitcoin Core's JSON-RPC API

```
from bitcoin.rpc import RawProxy

# Create a connection to local Bitcoin Core node
p = RawProxy()

# Run the getblockchaininfo command, store the resul
info = p.getblockchaininfo()

# Retrieve the 'blocks' element from the info
print(info['blocks'])
```

Running it gives us the following result:

```
$ python rpc_example.py
773973
```

It tells us how many blocks our local Bitcoin Core node has in its blockchain. Not a spectacular result, but it demonstrates the basic use of the library as a simplified interface to Bitcoin Core's JSON-RPC API.

Next, let's use the `getrawtransaction` and `decodetransaction` calls to retrieve the details of Alice's payment to Bob. In Example 3-4, we retrieve Alice's transaction and list the transaction's outputs. For each output, we show the recipient address and value. As a reminder, Alice's transaction had one output paying Bob and one output for change back to Alice.

Example 3-4. Retrieving a transaction and iterating its outputs

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# Alice's transaction ID
txid = "4662003086962l5bbc949d5141a49a4138ecdfdfaa2a

# First, retrieve the raw transaction in hex
raw_tx = p.getrawtransaction(txid)

# Decode the transaction hex into a JSON object
decoded_tx = p.decoderawtransaction(raw_tx)

# Retrieve each of the outputs from the transaction
for output in decoded_tx['vout']:
    print(output['scriptPubKey']['address'], output[
```

Running this code, we get:

```
$ python rpc_transaction.py
bc1p8dqa4wjvnt890qmfws83te0v3qxzsfu7ul63kp7u56w8qc0c
bc1qwafvze0200nh9vkq4jmlf4sy0tn0ga5w0zpkpg 0.0007500
```

Both of the preceding examples are rather simple. You don't really need a program to run them; you could just as easily use the `bitcoin-cli` helper.

The next example, however, requires several hundred RPC calls and more clearly demonstrates the use of a programmatic interface.

In Example 3-5, we first retrieve block 277316, then retrieve each of the 419 transactions within by reference to each transaction ID. Next, we iterate through each of the transaction's outputs and add up the value.

Example 3-5. Retrieving a block and adding all the transaction outputs

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# The block height where Alice's transaction was rec
blockheight = 277316

# Get the block hash of block with height 277316
blockhash = p.getblockhash(blockheight)

# Retrieve the block by its hash
block = p.getblock(blockhash)

# Element tx contains the list of all transaction ID
transactions = block['tx']

block_value = 0

# Iterate through each transaction ID in the block
for txid in transactions:
```

```
        tx_value = 0
        # Retrieve the raw transaction by ID
        raw_tx = p.getrawtransaction(txid)
        # Decode the transaction
        decoded_tx = p.decoderawtransaction(raw_tx)
        # Iterate through each output in the transactior
        for output in decoded_tx['vout']:
            # Add up the value of each output
            tx_value = tx_value + output['value']

        # Add the value of this transaction to the total
        block_value = block_value + tx_value

    print("Total value in block: ", block_value)
```

Running this code, we get:

```
$ python rpc_block.py

Total value in block:  10322.07722534
```

Our example code calculates that the total value transacted in this block is 10,322.07722534 BTC (including 25 BTC reward and 0.0909 BTC in fees). Compare that to the amount reported by a block explorer site by searching for the block hash or height. Some block explorers report the total value excluding the reward and excluding the fees. See if you can spot the difference.

Alternative Clients, Libraries, and Toolkits

There are many alternative clients, libraries, toolkits, and even full-node implementations in the bitcoin ecosystem. These are implemented in a variety of programming languages, offering programmers native interfaces in their preferred language.

The following sections list some of the best libraries, clients, and toolkits, organized by programming languages.

C/C++

Bitcoin Core

The reference implementation of bitcoin

libbitcoin

Cross-platform C++ development toolkit, node, and consensus library

bitcoin explorer

Libbitcoin's command-line tool

JavaScript

bcoin

A modular and scalable full-node implementation with API

Bitcore

Full node, API, and library by Bitpay

BitcoinJS

A pure JavaScript Bitcoin library for node.js and browsers

Java

bitcoinj

A Java full-node client library

Python

python-bitcoinlib

A Python bitcoin library, consensus library, and node by Peter Todd

pycoin

A Python bitcoin library by Richard Kiss

Go

btcd

A Go language full-node Bitcoin client

Rust

rust-bitcoin

Rust bitcoin library for serialization, parsing, and API calls

C#

NBitcoin

Comprehensive bitcoin library for the .NET framework

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

If you followed the instructions in this chapter, you now have Bitcoin Core running and have begun exploring the network and blockchain using your own full node. From now on you can independently use software you control, on a computer you control, to verify any bitcoins you receive follow every rule in the Bitcoin system without having to trust any outside authority. In the coming chapters, we'll learn more about the rules of the system and how your node and your wallet use them to secure your money, protect your privacy, and make spending and receiving convenient.

Chapter 4. Keys and Addresses

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *arufino@oreilly.com*.

Alice wants to pay Bob, but the the thousands of Bitcoin full nodes who will verify her transaction don't know who Alice or Bob are—and we want to keep it that way to protect their privacy. Alice needs to communicate that Bob should receive some of her bitcoins without tying any aspect of that transaction to Bob's real-world identity or to other Bitcoin payments that Bob receives. The method Alice uses must ensure that only Bob can further spend the bitcoins he receives.

The original Bitcoin paper describes a very simple scheme for achieving those goals, shown in Figure 4-1. A receiver like Bob accepts bitcoins to a public key in a transaction which is signed by the spender (like Alice). The bitcoins which Alice is spending had been previously received to one her public keys, and she

uses the corresponding private key to generate her signature. Full nodes can verify that Alice's signature commits to the output of a hash function that itself commits to Bob's public key and other transaction details.

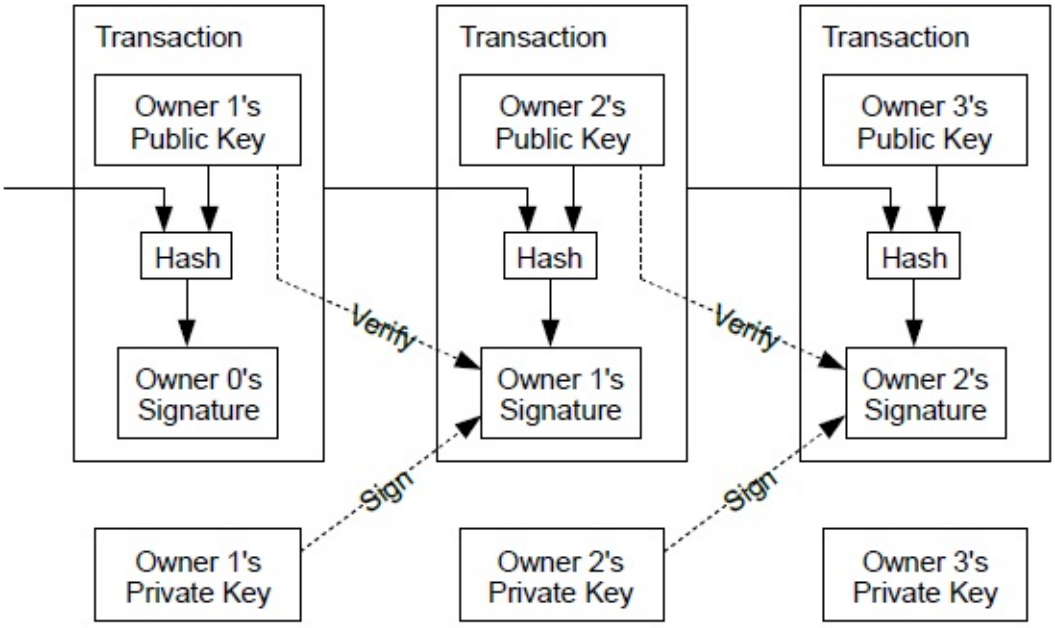


Figure 4-1. Transaction chain from original Bitcoin paper

We'll examine public keys, private keys, signatures, and hash functions in this chapter, and then use all of them together to describe the addresses used by modern Bitcoin software.

Public Key Cryptography

Public key cryptography was invented in the 1970s and is a mathematical foundation for modern 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 easy to calculate in one direction and infeasible to calculate in the opposite direction using the computers and algorithms available today. Based on these mathematical functions, cryptography enables the creation of unforgeable digital signatures. Bitcoin uses elliptic curve addition and multiplication as the basis for its cryptography.

In Bitcoin, we can use public key cryptography to create a key pair that controls access to bitcoin. The key pair consists of a private key and a public key derived from the private key. The public key is used to receive funds, and the private key is used to sign transactions to spend the funds.

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. These signatures can be validated against the public key without revealing the private key.

TIP

In some 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.

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 derived from a number picked at random. From the private key, we use elliptic curve

multiplication, a one-way cryptographic function, to generate a public key (K).

WHY USE ASYMMETRIC CRYPTOGRAPHY (PUBLIC/PRIVATE KEYS)?

Why is asymmetric cryptography used in bitcoin? It's not used to "encrypt" (make secret) the transactions. Rather, the useful property of asymmetric cryptography is the ability to generate *digital signatures*. A private key can be applied to the digital fingerprint of a transaction to produce a numerical signature. This signature can only be produced by someone with knowledge of the private key. However, anyone with access to the public key and the transaction fingerprint can use them to *verify* the signature. This useful property of asymmetric cryptography makes it possible for anyone to verify every signature on every transaction, while ensuring that only the owners of private keys can produce valid signatures.

Private Keys

A private key is simply a number, picked at random. Control over the private key is the root of user control over all funds associated with the corresponding Bitcoin public key. The private key is used to create signatures that are used to spend bitcoin by proving control 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 bitcoin 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

A 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. Be careful, though, as any process that's less than completely random can significantly reduce the security of your private key and the bitcoins it controls.

The first and most important step in generating keys is to find a secure source of randomness (which computer scientists call *entropy*). Creating a Bitcoin key is almost 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 cryptographically-secure random number generators to produce 256 bits of entropy.

More precisely, the private key can be any number between `0` and `n - 1` inclusive, where n is a constant (n = 1.1578 * 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`. 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, which will conveniently produce a 256-bit value that can be interpreted as a number. If the result is less than `n`, we have a suitable private key. Otherwise, we simply try again with another random number.

WARNING

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 pseudorandom 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 bits shown as 64 hexadecimal digits, each 4 bits):

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8
```

TIP

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

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.

Figure 4-2 is an example of an elliptic curve, similar to that used by bitcoin.

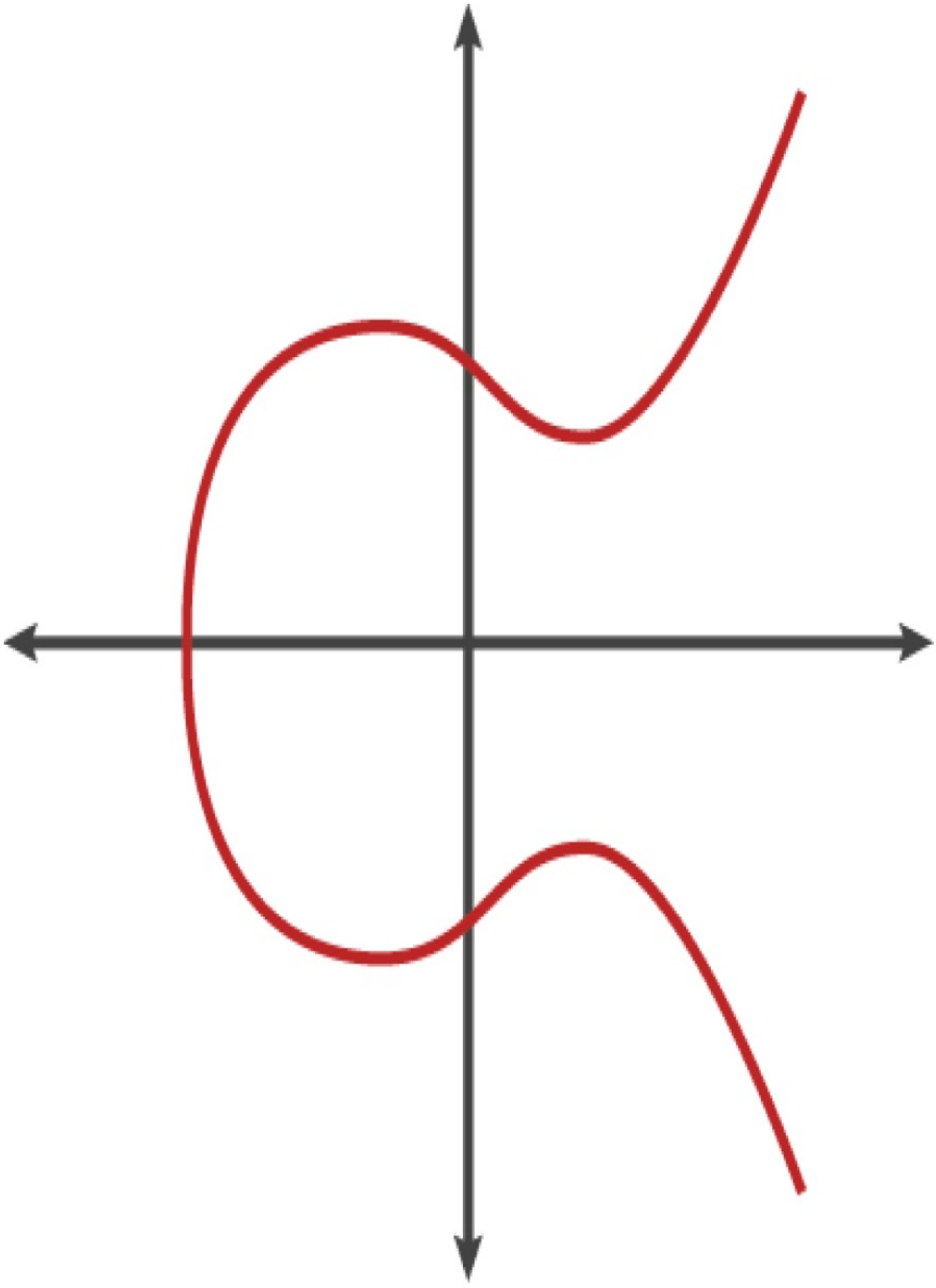


Figure 4-2. An elliptic curve

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:

$$y^2 = (x^3 + 7) \text{ over } (\mathbb{F}_p)$$

or

$$y^2 \mod p = (x^3 + 7) \mod p$$

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 to that of an elliptic curve over real numbers. As an example, Figure 4-3 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.

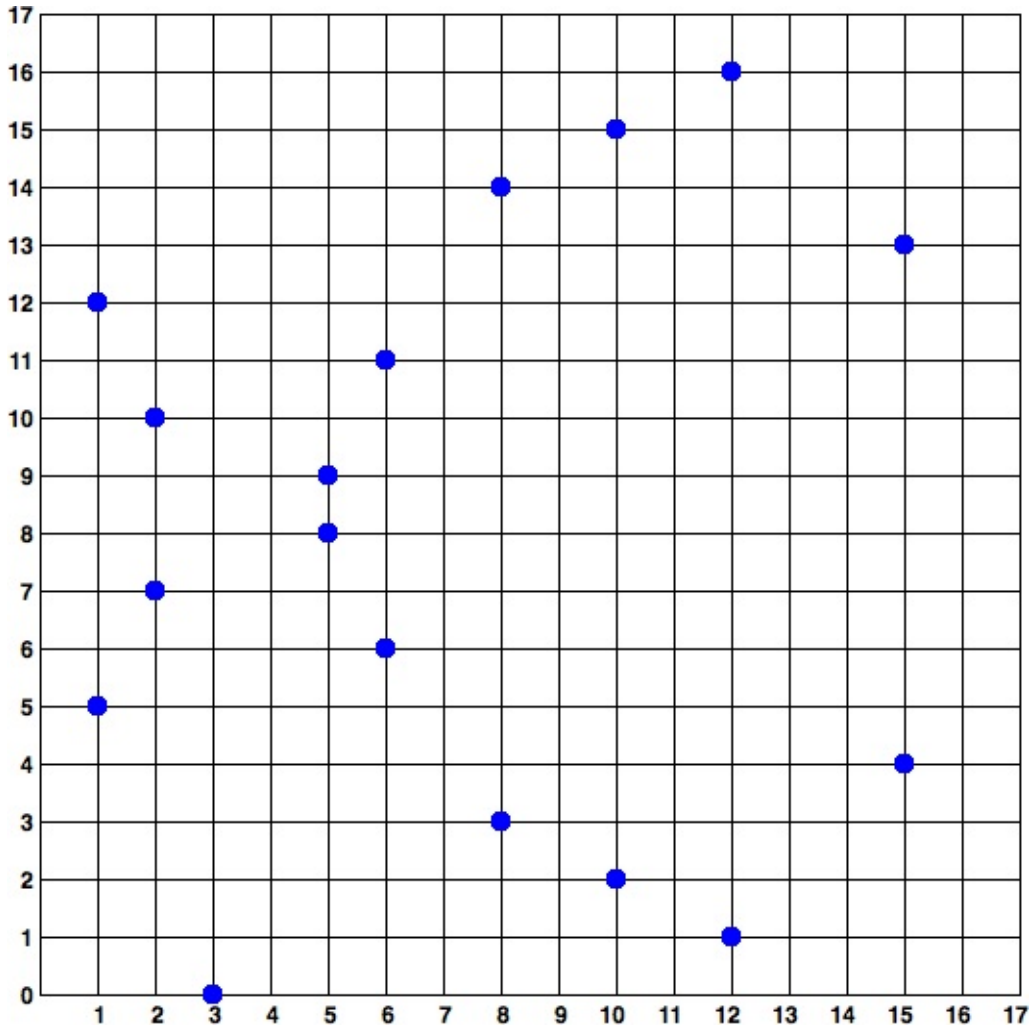


Figure 4-3. Elliptic curve cryptography: visualizing an elliptic curve over F(p), with p=17

So, for example, the following is a point P with coordinates (x,y) that is a point on the `secp256k1` curve:

```
P = (55066263022277343669578718895168534326250603453
```

Example 4-1 shows how you can check this yourself using Python:

Example 4-1. Using Python to confirm that this point is on the elliptic curve

```
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.
Type "help", "copyright", "credits" or "license" for
>>> p = 115792089237316195423570985008687907853269984
>>> x = 55066263022277343669578718895168534326250603
>>> y = 32670510020758816978083085130507043184471273
>>> (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 zero 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 gradeschool 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 P3 = “point at infinity.”

If P_1 is the “point at infinity,” then $P_1 + P_2 = P_2$. 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 zero.

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 and without 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 + … + P (k times). Note that k is sometimes confusingly called an “exponent” in this case.

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.

TIP

Elliptic curve multiplication is a type of function that cryptographers call a "trap door" function: it is easy to do in one direction (multiplication) and impossible to do in the reverse direction (division). Someone with a private key can easily create the public key and then share it with the world knowing that no one can reverse the function and calculate the private key from the public key. This mathematical trick becomes the basis for unforgeable and secure digital signatures that prove control over bitcoin funds.

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:

$$K = k * G$$

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 public key 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 = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDC
```

Public key *K* is defined as a point `K = (x,y)`:

```
K = (x, y)

where,

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328
```

To visualize multiplication of a point with an integer, we will use the simpler elliptic curve over real numbers—remember, the math is the same. Our goal is to find the multiple *kG* of the generator point *G*, which 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.

Figure 4-4 shows the process for deriving *G*, *2G*, *4G*, as a geometric operation

on the curve.

TIP

Many Bitcoin implementations use the libsecp256k1 crytographic library to do the elliptic curve math.

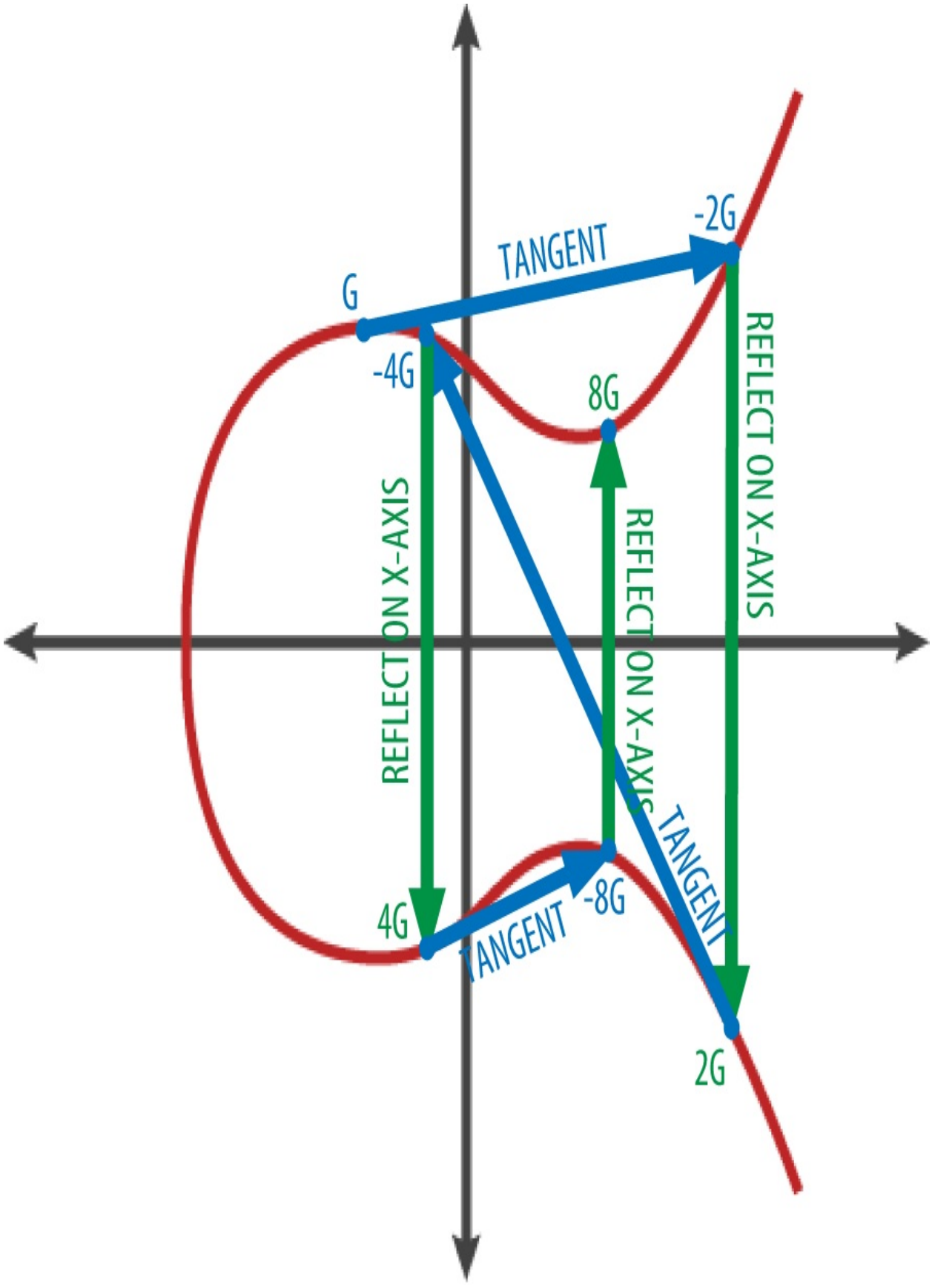

TANGENT
-2G
G
-4G
8G
REFLECT ON X-AXIS
REFLECT ON X-AXIS
REFLECT ON X-AXIS
TANGENT
TANGENT
-8G
4G
2G

Figure 4-4. Elliptic curve cryptography: visualizing the multiplication of a point G by an integer k on an elliptic curve

ScriptPubKey and ScriptSig

Although the illustration from the original Bitcoin paper, Figure 4-1, shows public keys (pubkeys) and signatures (sigs) being used directly, the first version of Bitcoin instead had payments sent to a field called *scriptPubKey* and had them authorized by a field called *scriptSig*. These fields allow additional operations to be performed in addition to (or instead of) verifying that a signature corresponds to a public key. For example, a scriptPubKey can contain two public keys and require two corresponding signatures be placed in the spending scriptSig.

Later, in XREF HERE, we'll learn about scripts in detail. For now, all we need to understand is that bitcoins are received to a scriptPubKey which acts like a public key, and bitcoin spending is authorized by a scriptSig which acts like a signature.

IP Addresses: The Original Address For Bitcoin (P2PK)

We've established that Alice can pay Bob by assigning some of her bitcoins to one of Bob's public keys. But how does Alice get one of Bob's public keys? Bob could just give her a copy, but let's look again at the public key we worked with

in “Public Keys”. Notice that it’s quite long. Imagine Bob trying to read that to Alice over the phone.

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328
```

Instead of direct public key entry, the earliest version of Bitcoin software allowed a spender to enter the receiver’s IP address. This feature was later removed—there are many problems with using IP addresses—but a quick description of it will help us better understand why certain features may have been added to the Bitcoin protocol.

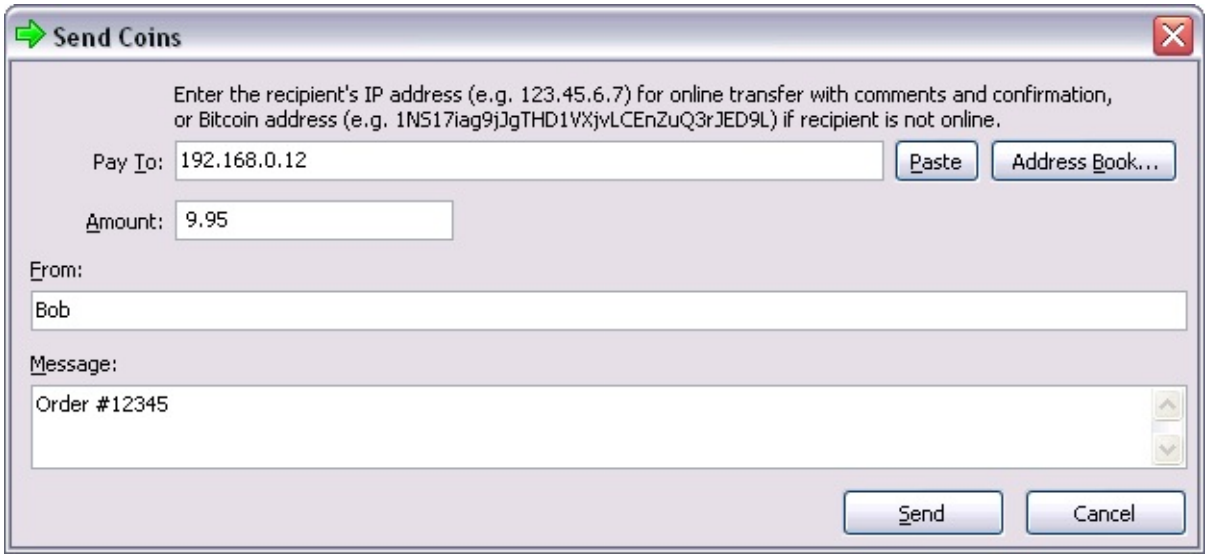


Figure 4-5. Early send screen for Bitcoin via The Internet Archive

If Alice entered Bob’s IP address in Bitcoin 0.1, her full node would establish a connection with his full node and receive a new public key from Bob’s wallet that his node had never previously given anyone. This being a new public key

was important to ensure that different transactions paying Bob couldn’t be connected together by someone looking at the blockchain and noticing that all of the transactions paid the same public key.

Using the public key her node received from Bob’s node, Alice’s wallet would construct a transaction output paying a very simple scriptPubKey:

```
<Bob's public key> OP_CHECKSIG
```

Bob would later be able to spend that output with a scriptSig consisting entirely of his signature:

```
<Bob's signature>
```

To figure out what a scriptPubKey and scriptSig are doing, you can combine them together (scriptSig first) and then note that each piece of data (shown in angle brackets) is placed at the top of a list of items, called a stack. When an operation code (opcode) is encountered, it uses items from the stack, starting with the topmost items. Let’s look at how that works by beginning with the combined script:

```
<Bob's signature> <Bob's public key> OP_CHECKSIG
```

For this script, Bob’s signature is put on the stack, then Bob’s public key is placed on top of it. The `OP_CHECKSIG` operation consumes two elements,

starting with the public key and followed by the signature, removing them from the stack. It verifies the signature corresponds to the public key and also commits to (signs) the various fields in the transaction. If the signature is correct, OP_CHECKSIG replaces itself on the stack with the value 1; if the signature was not correct, it replaces itself with a 0. If there's a non-zero item on top of the stack at the end of evaluation, the script passes. If all scripts in a transaction pass, and all of the other details about the transaction are valid, then full nodes will consider the transaction to be valid.

In short, the script above uses the same public key and signature described in the original paper but adds in the complexity of two script fields and an opcode. That seems like extra work here, but we'll begin to see the benefits when we look at "Legacy Addresses for P2PKH".

This type of output is known today as *Pay-to-Public-Key*, or *P2PK* for short. It was never widely used for payments, and no widely-used program has supported IP address payments for almost a decade.

Legacy Addresses for P2PKH

Entering the IP address of the person you want to pay has a number of advantages, but it also has a number of downsides. One particular downside is that the receiver needs their wallet to be online at their IP address, and it needs to be accessible from the outside world. For a lot of people, that isn't an option. They turn their computers off at night, their laptops go to sleep, they're behind

firewalls, or they're using Network Address Translation (NAT).

This brings us back to the problem of receivers like Bob having to give spenders like Alice a long public key. The shortest version of Bitcoin public keys known to the developers of early Bitcoin were 65 bytes, the equivalent of 130 characters when written in hexadecimal. However, Bitcoin already contained several data structures much larger than 65 bytes which needed to be securely referenced in other parts of Bitcoin using the smallest amount of data that was secure.

Bitcoin accomplishes that with a *hash function*, a function which takes a potentially large amount of data, scrambles it (hashes it), and outputs a fixed amount of data. A cryptographic hash function will always produce the same output when given the same input, and a secure function will also make it impractical for somebody to choose a different input that produces a previously-seen output. That makes the output a *commitment* to the input. It's a promise that, in practice, only input *x* will produce output *X*.

For example, imagine I want to ask you a question and also give you my answer in a form that you can't read immediately. Let's say the question is, "in what year did Satoshi Nakamoto start working on Bitcoin?" I'll give you a commitment to my answer in the form of output from the SHA256 hash function, the function most commonly used in Bitcoin:

```
94d7a772612c8f2f2ec609d41f5bd3d04a5aa1dfe3582f04af51
```

Later, after you tell me your guess to the answer of the question, I can reveal my

answer and prove to you that my answer, as input to the hash function, produces exactly the same output I gave you earlier:

```
$ echo "2007.  He said about a year and a half befor
94d7a772612c8f2f2ec609d41f5bd3d04a5aa1dfe3582f04af51
```

Now imagine that we ask Bob the question, “what is your public key?” Bob can use a hash function to give us a cryptographically secure commitment to his public key. If he later reveals his key, and we verify it produces the same commitment he previously gave us, we can be sure it was the exact same key that was used to create that earlier commitment.

The SHA256 hash function is considered to be very secure and produces 256 bits (32 bytes) of output, less than half the size of original Bitcoin public keys. However, there are other slightly less secure hash functions that produce smaller output, such as the RIPEMD160 hash function whose output is 160 bits (20 bytes). For reasons Satoshi Nakamoto never stated, the original version of Bitcoin made commitments to public keys by first hashing the key with SHA256 and then hashing that output with RIPEMD160; this produced a 20-byte commitment to the public key.

We can look at that algorithmically. 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:

$$A = RIPEMD160(SHA256(K))$$

where *K* is the public key and *A* is the resulting commitment.

Now that we understand how to make a commitment to a public key, we need to figure out how to use it in a transaction. Consider the following scriptPubKey:

```
OP_DUP OP_HASH160 <Bob's commitment> OP_EQUAL OP_CHE
```

And also the following scriptSig:

```
<Bob's signature> <Bob's public key>
```

Together, they form the following script:

```
<sig> <pubkey> OP_DUP OP_HASH160 <commitment> OP_EQU
```

As we did in "IP Addresses: The Original Address For Bitcoin (P2PK)", we start putting items on the stack. Bob's signature goes on first; his public key is then placed on top of the stack. The `OP_DUP` operation duplicates the top item, so the top and second-to-top item on the stack are now both Bob's public key. The `OP_HASH160` operation consumes (removes) the top public key and replaces it with the result of hashing it with `RIPEMD160(SHA256(K))`, so now the top of the stack is a hash of Bob's public key. Next, the commitment to Bob's public key is added to the top of the stack. The `OP_EQUALVERIFY` operation consumes the top two items and verifies that they are equal; that should be the

case if the public key Bob provided in the scriptSig is the same public key used to create the commitment in the scriptPubKey that Alice paid. If `OP_EQUALVERIFY` fails, the whole script fails. Finally, we're left with a stack containing just Bob's signature and his public key; the `OP_CHECKSIG` opcode verifies they correspond with each other and that the signature commits to the transaction.

Although this process of Paying To a Public Key Hash (*P2PKH*) may seem convoluted, it allows Alice's payment to Bob to contain only a 20 byte commitment to his public key instead of the key itself, which would've been 65 bytes in the original version of Bitcoin. That's a lot less data for Bob to have to communicate to Alice.

However, we haven't yet discussed how Bob gets those 20 bytes from his Bitcoin wallet to Alice's wallet. There are commonly used encodings for byte values, such as hexadecimal, but any mistake made in copying a commitment would result in the bitcoins being sent to an unspendable output, causing them to be lost forever. In "Base58Check Encoding", we'll look at compact encoding and reliable checksums.

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

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, base64 representation uses 26 lowercase letters, 26 capital letters, 10 numerals, and 2 more characters such as “+” and “/” to transmit binary data over text-based media such as email.

Base58 is a similar encoding to base64, using 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 lowercase and capital letters and numbers without the four (0, O, l, I) just mentioned. Example 4-2 shows the full base58 alphabet.

Example 4-2. Bitcoin’s base58 alphabet

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrst
```

To add extra security against typos or transcription errors, base58check adds an error-checking code to the base58 alphabet. 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 transcription and typing errors. When presented with 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, an error has been introduced and the base58check data is invalid. 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, the prefix zero (0x00 in hex) indicates that the data should be used as the commitment (hash) in a legacy P2PKH scriptPubKey. A list of common version prefixes is shown in Table 4-1.

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 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. Figure 4-6 illustrates the base58check encoding process.

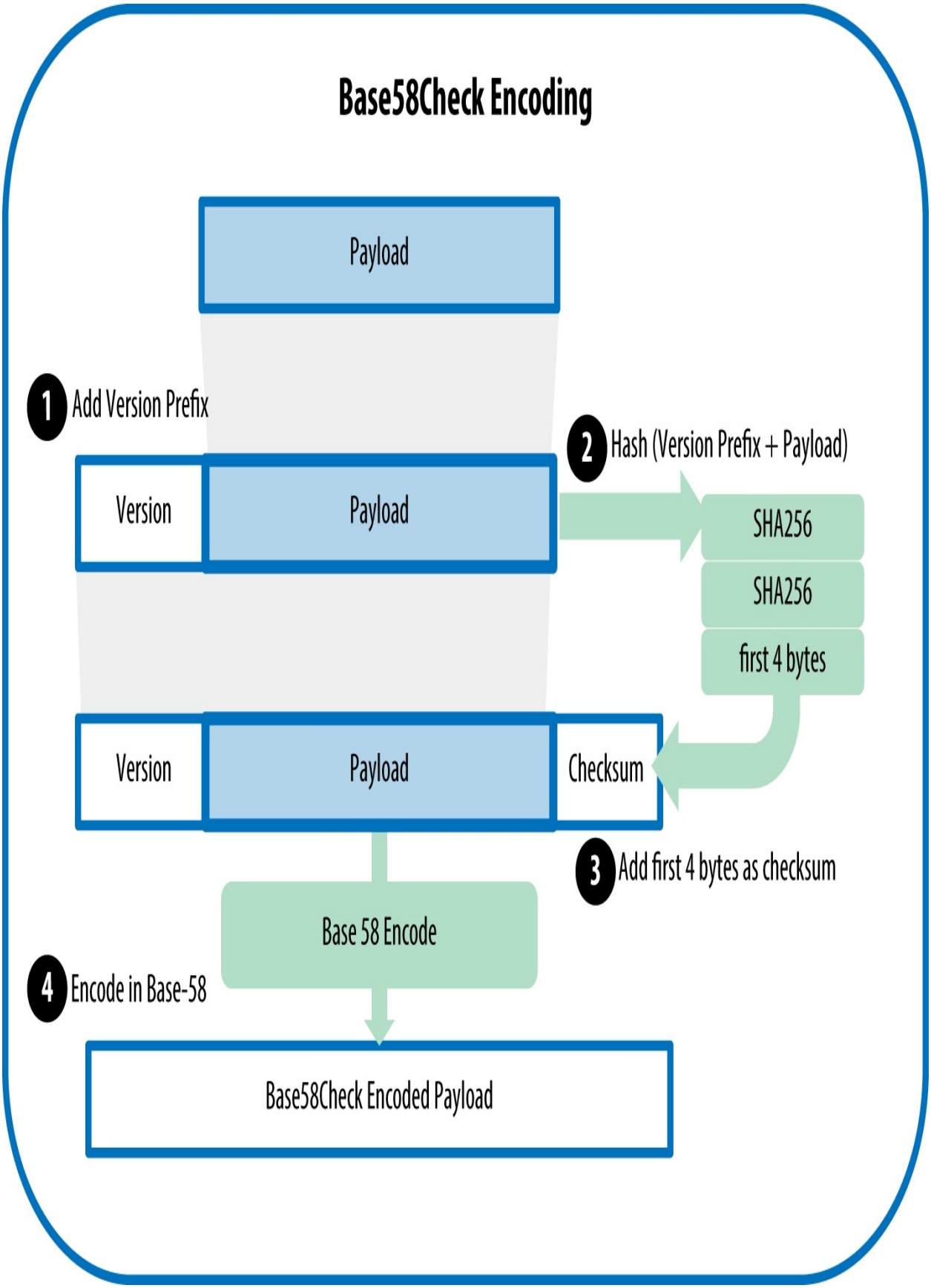

Base58Check Encoding
Payload
1
Add Version Prefix
Version
Payload
2
Hash (Version Prefix + Payload)
SHA256
SHA256
first 4 bytes
Version
Payload
Checksum
3
Add first 4 bytes as checksum
Base 58 Encode
4
Encode in Base-58
Base58Check Encoded Payload

Figure 4-6. Base58Check encoding: a base58, versioned, and checksummed format for unambiguously encoding bitcoin data

In Bitcoin, other data besides public key commitmens are presented to the user in base58check encoding to make that data 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 that starts with a 5. Some example version prefixes and the resulting base58 characters are shown in Table 4-1.

Table 4-1. Base58Check version prefix and encoded result examples

Type	Version prefix (hex)	Base58 result prefix
Address for Pay-to-Public-Key-Hash (P2PKH)	0x00	1
Address for Pay-to-Script-Hash (P2SH)	0x05	3
Testnet Address for P2PKH	0x6F	m or n

Testnet Address for P2SH	0xC4	2
Private Key WIF	0x80	5, K, or L
BIP-32 Extended Public Key	0x0488B21E	xpub

Putting together public keys, hash-based commitments, and base58check encocding, we can see the illustration of the conversion of a public key into a Bitcoin address in Figure 4-7.

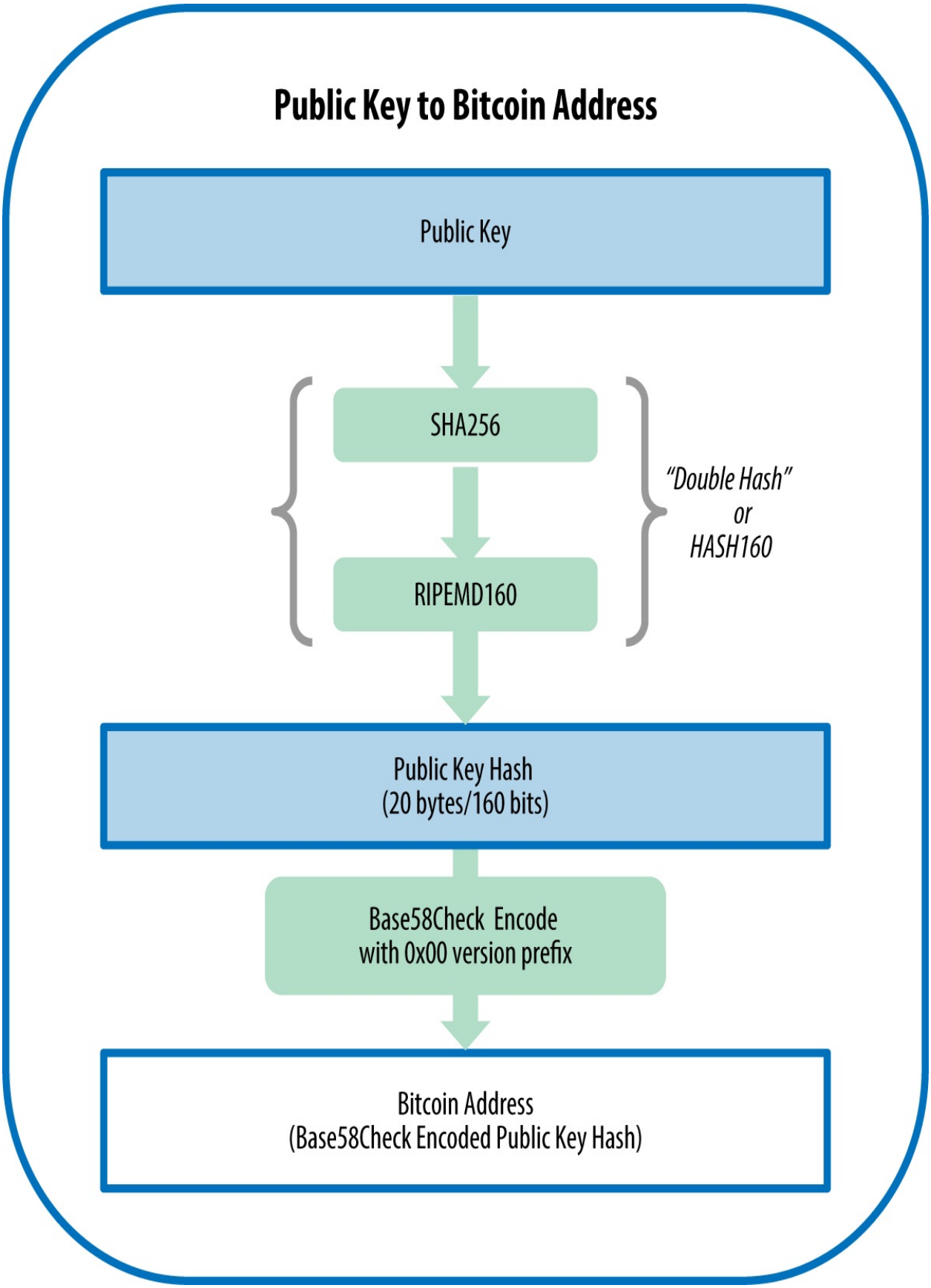

Public Key to Bitcoin Address
Public Key
SHA256
RIPEMD160
"Double Hash"
or
HASH160
Public Key Hash
(20 bytes/160 bits)
Base58Check Encode
with 0x00 version prefix
Bitcoin Address
(Base58Check Encoded Public Key Hash)

Figure 4-7. Public key to Bitcoin address: conversion of a public key into a Bitcoin address

Decode from Base58Check

The Bitcoin Explorer commands (see XREF HERE) 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 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ
wrapper
{
    checksum 4286807748
    payload 1e99423a4ed27608a15a2616a2b0e9e52ced330a
    version 128
}
```

The result contains the key as payload, the WIF version prefix 128, and a checksum.

Notice that the "payload" of the compressed key is appended with the suffix `01`, signaling that the derived public key is to be compressed:

```
$ bx base58check-decode KxFC1jmwwCoACiCAWZ3eXa96mBM6
wrapper
{
    checksum 2339607926
    payload 1e99423a4ed27608a15a2616a2b0e9e52ced330a
    version 128
}
```

Compressed public keys

When Bitcoin was first authored, its developers only knew how to create 65-byte public keys. However, a later developer became aware of an alternative encoding for public keys that used only 33 bytes and which was backwards compatible with all Bitcoin full nodes at the time, so there was no need to change the Bitcoin protocol. Those 33-byte public keys are known as *compressed public keys* and the original 65 byte keys are known as *uncompressed public keys*. Using smaller public keys results in smaller transactions, allowing more payments to be made in the same block.

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!

Here's the public key generated by the private key we created in "Public Keys".

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328
```

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`:

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F1
07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52
```

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, 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 or below the x-axis. As you can see from the graph of the elliptic curve in Figure 4-2, 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 Figure 4-8.

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

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F1
```

This compressed public key corresponds to the same private key, meaning 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 commitment using the HASH160 function (`RIPEMD160(SHA256(K))`) it will produce a *different* commitment than the uncompressed public key, leading to a different 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.

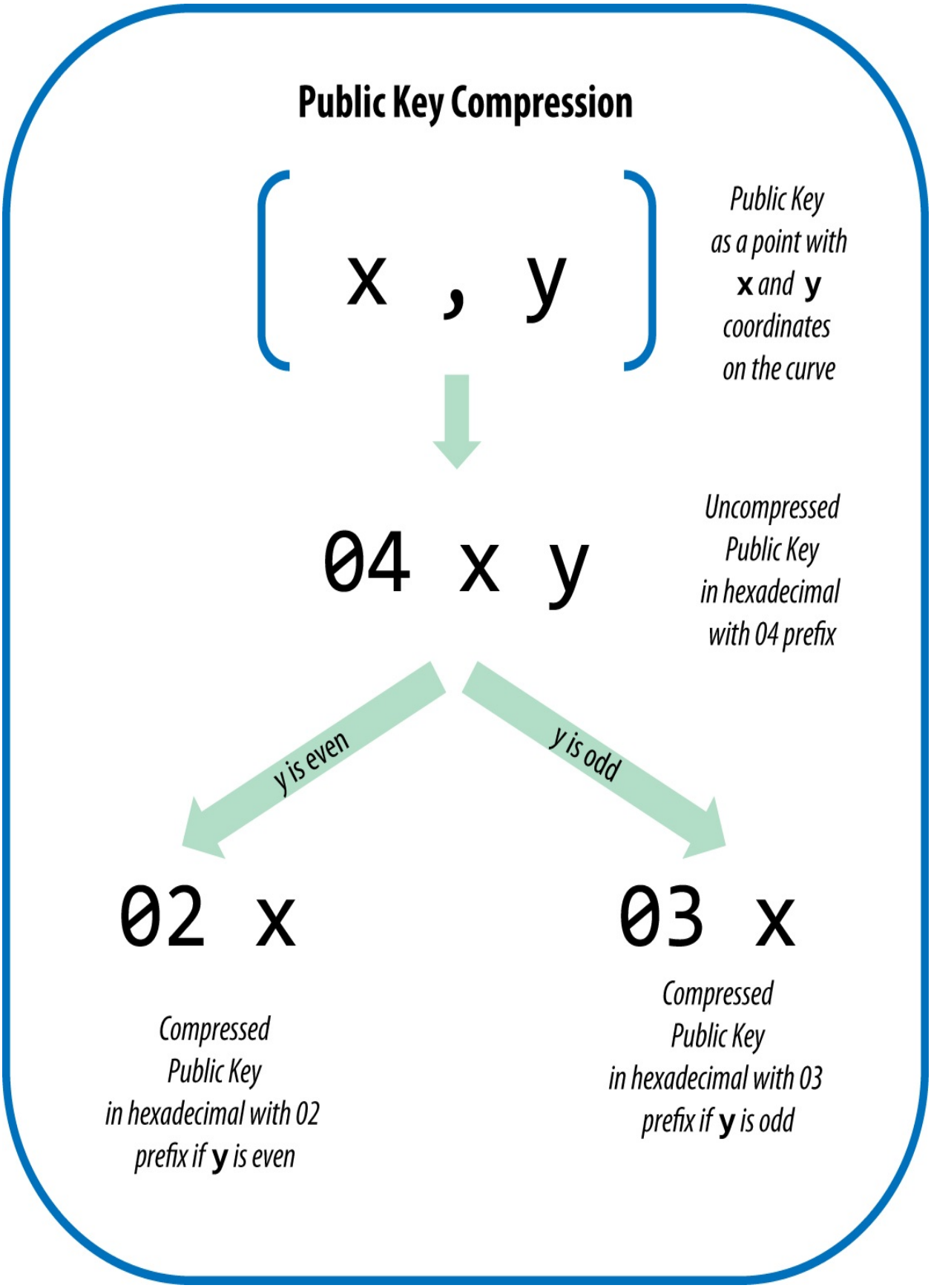

Public Key Compression
x , y
Public Key as a point with x and y coordinates on the curve
04 x y
Uncompressed Public Key in hexadecimal with 04 prefix
y is even
y is odd
02 x
Compressed Public Key in hexadecimal with 02 prefix if y is even
03 x
Compressed Public Key in hexadecimal with 03 prefix if y is odd

Figure 4-8. Public key compression

Compressed public keys are now the default in almost all Bitcoin software, and were made required when using certain new features added in later protocol upgrades.

However, some software still needs to support uncompressed public keys, such as a wallet application importing private keys from an older wallet. When the new wallet scans the block chain for old P2PKH outputs and inputs, it needs to know whether to scan the 65-byte keys (and commitments to those keys) or 33-byte keys (and their commitments). Failure to scan for the correct type can lead to the user not being able to spend their full balance. To resolve this issue, when private keys are exported from a wallet, the Wallet Import Format (WIF) that is used to represent them is implemented slightly differently in newer Bitcoin wallets, to indicate that these private keys have been used to produce compressed public keys.

Legacy Pay-to-Script-Hash (P2SH)

As we've seen in preceding sections, someone receiving Bitcoins (like Bob) can require payments to him contain certain constraints in their scriptPubKeys. Bob will need to fulfill those constraints using a scriptSig when he spends those bitcoins. In "IP Addresses: The Original Address For Bitcoin (P2PK)", the constraint was simply that the scriptSig needed to provide an appropriate signature. In "Legacy Addresses for P2PKH", an appropriate public key also

needed to be provided.

In order for a spender (like Alice) to place the constraints Bob wants in the scriptPubKey she uses to pay him, Bob needs to communicate those constraints to her. This is similar to the problem of Bob needing to communicate his public key to her. Like that problem, where public keys can be fairly large, the constraints Bob uses can also be quite large---potentially thousands of bytes. That's not only thousands of bytes which need to be communicated to Alice, but thousands of bytes for which she needs to pay transaction fees every time she wants to spend money to Bob. However, the solution of using hash functions to create small commitments to large amounts of data also applies here.

The BIP16 upgrade to the Bitcoin protocol in 2013 allows a scriptPubKey to commit to a *redemption script* (*redeemScript*). When Bob spends his bitcoins, his scriptSig need to provide a redeemScript that matches the commitment and also any data necessary to satisfy the redeemScript (such as signatures). Let's start by imagining Bob wants to require two signatures to spend his bitcoins, one signature from his desktop wallet and one from a hardware signing device. He puts those conditions into a redeemScript:

```
  <public key 1> OP_CHECKSIGVERIFY <public key 2> OP_C
```

He then creates a commitment to the redeemScript using the same HASH160 mechanism used for P2PKH commitments, `RIPEMD160(SHA256(script))`. That commitment is placed into the

scriptPubKey using a special template:

```
OP_HASH160 <commitment> OP_EQUAL
```

WARNING

Payments to Script Hashes (P2SH) must use the specific P2SH template with no extra data or conditions in the scriptPubKey. If the scriptPubKey is not exactly `OP_HASH160 <20 bytes> OP_EQUAL`, the redeemScript will not be used and any bitcoins may either be unspendable or spendable by anyone (meaning anyone can take them).

When Bob goes to spend the payment he received to the commitment for his script, he uses a scriptSig that includes the redeemScript, with it serialized as a single data element. He also provides the signatures he needs to satisfy the redeemScript, putting them in the order that they will be consumed by the opcodes:

```
<signature2> <signature1> <redeemScript>
```

When Bitcoin full nodes receive Bob's spend, they'll verify that the serialized redeemScript will hash to the same value as the commitment. Then they'll replace it on the stack with its deserialized value:

```
<signature2> <signature1> <pubkey1> OP_CHECKSIGVERIF
```

The script is executed and, if it passes and all of the other transaction details are correct, the transaction is valid.

Addresses for Pay-to-Script-Hash (P2SH) are also created with base58check. The version prefix is set to 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 XREF HERE) as follows:

```
$ echo \
'DUP HASH160 [89abcdefabbaabbaabbaabbaabbaabbaabbaab
$ bx script-encode < script | bx sha256 | bx ripemd1
| bx base58check-encode --version 5
3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM
```

TIP

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

P2PKH and P2SH are the only two script templates used with base58check encoding. They are now known as legacy addresses and, as of early 2023, are only used in about 10% of transactions. Legacy addresses were supplanted by the bech32 family of addresses.

P2SH COLLISION ATTACKS

All addresses based on hash functions are theoretically vulnerable to an attacker finding two different inputs (e.g. redeemScripts) that produce the same hash function output (commitment). For addresses created entirely by a single party, the chance of an attacker generating a different input for an existing commitment is proportional to the strength of the hash algorithm. For a secure 160-bit algorithm like HASH160, the probability is 1-in-2^{160}. This is a *second pre-image attack*.

However, this changes when an attacker is able to influence the original input value. For example, an attacker participates in the creation of a multisignature script where the attacker doesn't need to submit his public key until after he learns all of the other party's public keys. In that case, the strength of hash algorithm is reduced to its square root. For HASH160, the probability becomes 1-in-2^{80}. This is a *collision attack*.

To put those numbers in context, as of early 2023, all Bitcoin miners combined execute about 2^{80} hash functions every hour. They run a different hash function than HASH160, so their existing hardware can't create collision attacks for it, but the existence of the Bitcoin network proves that collision attacks against 160-bit functions like HASH160 are practical. Bitcoin miners have spent the equivalent of billions of US dollars on special hardware, so creating a collision attack wouldn't be cheap, but there are organizations which expect to receive billions of dollars in bitcoins to addresses generated by processes involving multiple parties, which could make the attack profitable.

There are well established cryptographic protocols for preventing collision attacks but a simple solution which doesn't require any special knowledge on the part of wallet developers is to simply use a stronger hash function. Later upgrades to Bitcoin made that possible and newer Bitcoin addresses provide at least 128 bits of collision resistance. To perform 2^{128} hash operations would require all current Bitcoin miners about 50 billion years to perform.

Although we do not believe there is any immediate threat to anyone creating new P2SH addresses, we recommend all new wallets use newer types of addresses to eliminate address collision attacks as a concern.

Bech32 addresses

In 2017, the Bitcoin protocol was upgraded to prevent transaction identifiers (txids) from being changed without the consent of a spending user (or a quorum of signers when multiple signatures are required). The upgrade, called *segregated witness* (or *segwit* for short), also provided additional capacity for transaction data in blocks and several other benefits. However, users wanting direct access to segwit's benefits had to accept payments to variations on the legacy P2PKH and P2SH scripts.

As mentioned in XREF HERE, one of the advantages of the P2SH output type was that a spender (such as Alice) didn't need to know the details of the script the receiver (such as Bob) used. The segwit upgrade was designed to be compatible with this mechanism, allowing users to immediately begin accessing

many of the new benefits by using a P2SH address. But for Bob to gain access to all of the benefits, he would need Alice's wallet to pay him using a different type of script. That would require Alice's wallet to upgrade to supporting the new scripts.

At first, Bitcoin developers proposed BIP142, which would continue using base58check with a new version byte, similar to the P2SH upgrade. But getting all wallets to upgrade to new scripts with a new base58check version was expected to require almost as much work as getting them to upgrade to an entirely new address format, so several Bitcoin contributors set out to design the best possible address format. They identified several problems with base58check:

- Its mixed case presentation made it inconvenient to read aloud or transcribe. Try reading one of the legacy addresses in this chapter to a friend who you have transcribe it. Notice how you have to prefix every letter with the words "uppercase" and "lowercase". Also note when you review their writing that the uppercase and lowercase versions of some letters can look similar in many people's handwriting.
- It can detect errors, but it can't help users correct those errors. For example, if you accidentally transpose two characters when manually entering an address, your wallet will almost certainly warn that a mistake exists, but it won't help you figure out where the error is located. It might take you several frustrating minutes to eventually discover the mistake.
- A mixed case alphabet also requires extra space to encode in QR code images, which are commonly used to share addresses and invoices between

wallets. That extra space means QR codes need to be larger at the same resolution or they become harder to scan quickly.

- It requires every spender wallet upgrade to support new protocol features like P2SH and segwit. Although the upgrades themselves might not require much code, experience shows that many wallet authors are busy with other work and can sometimes delay upgrading for years. This adversely affects everyone who wants to use the new features.

The developers working on an address format for segwit found solutions for each of these problems in a new address format called bech32 (pronounced with a soft “ch”, as in “besh thirty-two”). The “bech” stands for BCH, the initials of the three individuals who discovered the cyclic code in 1959 and 1960 upon which bech32 is based. The “32” stands for the number of characters in the bech32 alphabet (similar to the 58 in base58check).

- Bech32 uses only numbers and a single case of letters (preferably rendered in lowercase). Despite its alphabet being almost half the size of the base58check alphabet, bech32 addresses are only slightly longer than the longest equivalent P2PKH legacy addresses.
- Bech32 can both detect and help correct errors. In an address of an expected length, it is mathematically guaranteed to detect any error affecting four characters or less; that’s more reliable than base58check. For longer errors, it will fail to detect them less than one time in a billion, which is roughly the same reliability as base58check. Even better, for an address typed with just a few errors, it can tell the user where those errors occurred, allowing them to quickly correct minor transcription mistakes. See Example 4-3 for an example

of an address entered with errors.

Example 4-3. Bech32 typo detection

Address:
bc1p9nh05ha8wrljf7ru236aw**n**4t2x0d5ctkkywm**v**9sclnm4t0av2vgs4k3au7

Detected errors shown in bold. Generated using the bech32 address decoder demo.

- Bech32 is preferably written with only lowercase characters, but those lowercase characters can be replaced with uppercase characters before encoding an address in a QR code. This allows the use of a special QR encoding mode that uses less space. Notice the difference in size and complexity of the two QR codes for the same address in Figure 4-9.

Figure 4-9. The same bech32 address QR encoded in uppercase and lowercase

- Bech32 takes advantage of an upgrade mechanism designed as part of segwit

> to make it possible for spender wallets to be able to pay output types that aren't in use yet. The goal was to allow developers to build a wallet today that allows spending to a bech32 address and have that wallet remain able to spend to bech32 addresses for users of new features added in future protocol upgrades. It was hoped that we might never again need to go through the system-wide upgrade cycles necessary to allow people to fully use P2SH and segwit.

Problems with bech32 addresses

Bech32 addresses would have been a success in every area except for one problem. The mathematical guarantees about their ability to detect errors only apply if the length of the address you enter into a wallet is the same length of the original address. If you add or remove any characters during transcription, the guarantee doesn't apply and your wallet may spend funds to a wrong address. However, even without the guarantee, it was thought that it would be very unlikely that a user adding or removing characters would produce a string with a valid checksum, ensuring users' funds were safe.

Unfortunately, the choice for one of the constants in the bech32 algorithm just happened to make it very easy to add or remove the letter "q" in the penultimate position of an address that ends with the letter "p". In those cases, you can also add or remove the letter "q" multiple times. This will be caught by the checksum some of the time, but it will be missed far more often than the one-in-a-billion expectations for bech32's substitution errors. For an example, see Example 4-4.

Example 4-4. Extending the length of bech32 address without invalidating its checksum

```
Intended bech32 address:
bc1pqqqsq9txsqp

Incorrect addresses with a valid checksum:
bc1pqqqsq9txsqqqqp
bc1pqqqsq9txsqqqqqqp
bc1pqqqsq9txsqqqqqqqqp
bc1pqqqsq9txsqqqqqqqqqp
bc1pqqqsq9txsqqqqqqqqqqp
```

For the initial version of segwit (version 0), this wasn't a practical concern. Only two valid lengths were defined for v0 segwit outputs: 22 bytes and 34 bytes. Those correspond to bech32 addresses 42 characters or 62 characters long, so someone would need to add or remove the letter "q" from the penultimate position of a bech32 address 20 times in order to send money to an invalid address without a wallet being able to detect it. However, it would become a problem for users in the future if a segwit-based upgrade were ever to be implemented.

Bech32m

Although bech32 worked well for segwit v0, developers didn't want to unnecessarily constrain output sizes in later versions of segwit. Without constraints, adding or removing a single "q" in a bech32 address could result in a

user accidentally sending their money to an output that was either unspendable or spendable by anyone (allowing those bitcoins to be taken by anyone). Developers exhaustively analyzed the bech32 problem and found that changing a single constant in their algorithm would eliminate the problem, ensuring that any insertion or deletion of up to five characters will only fail to be detected less often than one time in a billion.

The version of bech32 with a single different constant is known as Bech32 Modified (bech32m). All of the characters in bech32 and bech32m addresses for the same underlying data will be identical except for the last six (the checksum). That means a wallet will need to know which version is in use in order to validate the checksum, but both address types contain an internal version byte that makes determining that easy.

Encoding and Decoding bech32m addresses

In this section, we'll look at the encoding and parsing rules for bech32m Bitcoin addresses since they encompass the ability to parse bech32 addresses and are the current recommended address format for Bitcoin wallets.

Bech32m addresses start with a Human Readable Part (HRP). There are rules in BIP173 for creating your own HRPs, but for Bitcoin you only need to know about the HRPs already chosen, shown in Table 4-2.

Table 4-2. Bech32 HRPs for Bitcoin

bc	Bitcoin mainnet

tb	Bitcoin testnet

The HRP is followed by a separator, the number “1”. Earlier proposals for a protocol separator used a colon but some operating systems and applications which allow a user to double click on a word to highlight it for copy and pasting won’t extend the highlighting to and past a colon. A number ensured double-click highlighting would work with any program that supports bech32m strings in general (which include other numbers). The number “1” was chosen because bech32 strings don’t otherwise use it in order to prevent accidental transliteration between the number “1” and the lowercase letter “l”.

The other part of a bech32m address is called the “data part”. There are three elements to this part:

Witness version
A single byte which encodes as a single character in a bech32m Bitcoin address immediately following the separator. This letter represents the segwit version. The letter “q” is the encoding of “0” for segwit v0, the initial version of segwit where bech32 addresses were introduced. The letter “p” is the encoding of “1” for segwit v1 (also called taproot) where bech32m began to be used. There are seventeen possible versions of segwit and it’s required for Bitcoin that the first byte of a bech32m data part decode to the number 0 through 16 (inclusive).

Witness program
From 2 to 40 bytes. For segwit v0, this witness program must be either 20 or

32 bytes; no other length is valid. For segwit v1, the only defined length as of this writing is 32 bytes but other lengths may be defined later.

Checksum
Exactly 6 characters. This is created using a BCH code, a type of error correction code (although for Bitcoin addresses, we'll see later that it's essential to use the checksum only for error detection—not correction).

Let's illustrate these rules by walking through an example of creating bech32 and bech32m addresses. For all of the following examples, we'll use the bech32m reference code for Python.

Let's start by generating four output scripts, one for each of the different segwit outputs in use at the time of publication, plus one for a future segwit version that doesn't yet have a defined meaning. The scripts are listed in Table 4-3.

Table 4-3. Scripts for different types of segwit outputs

P2WPKH	OP_0 2b626ed108ad00a944bb2922a309844611d25468
P2WSH	OP_0 648a32e50b6fb7c5233b228f60a6a2ca4158400268844c4bc2
P2TR	OP_1 2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1618f
Future	OP_16 0000

Example

For the P2WPKH output, the witness program contains a commitment constructed in exactly the same way as the commitment for a P2PKH output seen in “Legacy Addresses for P2PKH”. A public key is passed into a SHA256 hash function. The resultant 32 byte digest is then passed into a RIPEMD-160 hash function. The digest of that function (the commitment) is placed in the witness program.

For the P2WSH output, we don’t use the P2SH algorithm. Instead we take the script, pass it into a SHA256 hash function, and use the 32-byte digest of that function in the witness program. For P2SH, the SHA256 digest was hashed again with RIPEMD-160, but that may not be secure in some cases; for details, see “P2SH collision attacks”. A result of using SHA256 without RIPEMD160 is that P2WSH commitments are 32 bytes (256 bits) instead 20 bytes (160 bits).

For the Pay-to-Taproot (P2TR) output, the witness program is a point on the secp256k1 curve. It may be a simple public key, but in most cases it should be a public key that commits to some additional data. We’ll learn more about that commitment in XREF HERE.

For the example of a future segwit version, we simply use the highest possible segwit version number (16) and the smallest allowed witness program (2 bytes) with a null value.

Now that we know the version number and the witness program, we can convert

each of them into a bech32 address. Let's use the bech32m reference library for Python to quickly generate those addresses, and then take a deeper look at what's happening:

```
wget https://raw.githubusercontent.com/sipa/bech32/m
2023-01-30 11:59:10 (46.3 MB/s) - 'segwit_addr.py' s

$ python
>>> from segwit_addr import *
>>> from binascii import unhexlify

>>> help(encode)
encode(hrp, witver, witprog)
    Encode a segwit address.

>>> encode('bc', 0, unhexlify('2b626ed108ad00a944bb2
'bc1q9d3xa5gg45q2j39m9y32xzvygcgay4rgc6aaee'
>>> encode('bc', 0, unhexlify('648a32e50b6fb7c5233b2
'bc1qvj9r9egtd7mu2gemy28kpf4zefq4ssqzdzzycj7zjhk4arp
>>> encode('bc', 1, unhexlify('2ceefa5fa770ff24f87c5
'bc1p9nh05ha8wrljf7ru236awm4t2x0d5ctkkywmu9sclnm4t0a
>>> encode('bc', 16, unhexlify('0000'))
'bc1sqqqqkfw08p'
```

If we open the file `segwit_addr.py` and look at what the code is doing, the first thing we will notice is the sole difference between bech32 (used for segwit v0) and bech32m (used for later segwit versions) is the constant.

```
BECH32_CONSTANT = 1
BECH32M_CONSTANT = 0x2bc830a3
```

Next we notice the code produce the checksum. In the final step of the checksum, the appropriate constant is merged into the value using an xor operation. That single value is the only difference between bech32 and bech32m.

With the checksum created, each 5-bit character in the data part (including the witness version, witness program, and checksum) is converted to alphanumeric characters.

For decoding back into a scriptPubKey, we work in reverse. First let’s use the reference library to decode two of our addresses:

```
>>> help(decode)
decode(hrp, addr)
    Decode a segwit address.

>>> _ = decode("bc", "bc1q9d3xa5gg45q2j39m9y32xzvygc
(0, '2b626ed108ad00a944bb2922a309844611d25468')
>>> _ = decode("bc", "bc1p9nh05ha8wrljf7ru236awm4t2x
(1, '2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1
```

We get back both the witness version and the witness program. Those can be inserted into the template for our scriptPubKey:

```
<version> <program>
```

For example:

```
OP_0 2b626ed108ad00a944bb2922a309844611d25468
OP_1 2ceefa5fa770ff24f87c5475d76eab519eda6176b11dbe1
```

WARNING

One possible mistake here to be aware of is that a witness version of `0` is for `OP_0`, which uses the byte 0x00—but a witness version of `1` uses `OP_1`, which is byte 0x51. Witness versions `2` through `16` use 0x52 through 0x60, respectively.

When implementing bech32m encoding or decoding, we very strongly recommend that you use the test vectors provided in BIP350. We also ask that you ensure your code passes the test vectors related to paying future segwit versions that haven't been defined yet. This will help make your software is usable for many years to come even if you aren't able to add support for new Bitcoin features as soon as they become available.

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. Table 4-4 shows several common formats used to represent private keys. Different formats are used in different

circumstances. Hexadecimal and raw binary formats are used internally in software and rarely shown to users. The WIF is used for import/export of keys between wallets and often used in QR code (barcode) representations of private keys.

MODERN RELEVANCY OF PRIVATE KEY FORMATS

Early Bitcoin wallet software generated one or more independent private keys when a new user wallet was initialized. When the initial set of keys had all been used, the wallet might generate additional private keys. Individual private keys could be exported or imported. Any time new private keys were generated or imported, a new backup of the wallet needed to be created.

Later Bitcoin wallets began using deterministic wallets where all private keys are generated from a single seed value. These wallets only ever need to be backed up once for typical onchain use. However, if a user exports a single private key from one of these wallets and an attacker acquires that key plus some non-private data about the wallet, they can potentially derive any private key in the wallet—allowing the attacker to steal all of the wallet funds. Additionally, keys cannot be imported into deterministic wallets. This means almost no modern wallets support the ability to export or import an individual key. The information in this section is mainly of interest to anyone needing compatibility with early Bitcoin wallets.

For more information, see XREF HERE.

Table 4-4. Private key representations (encoding formats)

Type	Prefix	Description
Raw	None	32 bytes
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

Table 4-5 shows the private key generated in several different formats.

Table 4-5. Example: Same key, different formats

Format	Private key
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc3
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpbnkeyh
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6Yw

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. Note that the "raw binary" is not shown in Table 4-5 as any encoding for display here would, by definition, not be raw binary data.

We use the `wif-to-ec` command from Bitcoin Explorer (see XREF HERE) to show that both WIF keys represent the same private key:

```
$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jp
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8

$ bx wif-to-ec KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8
```

Compressed private keys

Ironically, the term "compressed private key" is a misnomer, because when a private key is exported as WIF-compressed it is actually one byte *longer* than an "uncompressed" private key. That is because the private key has an added one-byte suffix (shown as 01 in hex in Table 4-6), which signifies that the private key is from a newer wallet and should only be used to produce compressed public keys. Private keys are not themselves compressed and cannot be compressed. The term "compressed private key" really means "private key from which only compressed public keys should be derived," whereas "uncompressed private

key” really means “private key from which only 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 itself as “compressed” to avoid further confusion

Table 4-6 shows the same key, encoded in WIF and WIF-compressed formats.

Table 4-6. Example: Same key, different formats

Format	Private key
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC53
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2Jpbnkeyhf
Hex-compressed	1E99423A4ED27608A15A2616A2B0E9E52CED330AC53
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6Yw

Notice that the hex-compressed private key format has one extra byte at the end (01 in hex). While the base58 encoding version prefix is the same (0x80) for both WIF and WIF-compressed formats, the addition of one byte on the end of the number causes the first character of the base58 encoding to change from a 5 to either a *K* or *L*. Think of this as the base58 equivalent of the decimal encoding

difference between the number 100 and the number 99. While 100 is one digit longer than 99, it also has a prefix of 1 instead of a prefix of 9. As the length changes, it affects the prefix. In base58, the prefix 5 changes to a *K* or *L* as the length of the number increases by one byte.

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 WIF 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 (uncompressed) keys from older wallets.

TIP

"Compressed private keys" is a misnomer! They are not compressed; rather, WIF-compressed signifies that the keys 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.

Advanced Keys and Addresses

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

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 base58 letters. Vanity addresses require generating and testing billions of candidate private keys, until a Bitcoin address with the desired pattern is found. 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 bitcoin 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 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 Chapter 1, 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 `1Kids111111111111111111111111111` to `1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz`. There are approximately 58^{29} (approximately $1.4 * 10^{51}$) addresses in that range, all starting with "1Kids." Table 4-7 shows the range of addresses that have the prefix 1Kids.

Table 4-7. The range of vanity addresses starting with "1Kids"

From	`1Kids111111111111111111111111111`
	`1Kids111111111111111111111111112`
	`1Kids111111111111111111111111113`
	...

```
To          1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz
```

Let’s look at the pattern “1Kids” as a number and see how frequently we might find this pattern in a Bitcoin address (see Table 4-8). An average desktop computer PC, without any specialized hardware, can search approximately 100,000 keys per second.

Table 4-8. The frequency of a vanity pattern (1KidsCharity) and average search time 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
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" anytime 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 GPUs. 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. A pool is a service that allows those with GPU hardware to earn bitcoin searching for vanity addresses for others. For a fee, 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 security and privacy

Vanity addresses were popular in the early years of Bitcoin but have almost entirely disappeared from use as of 2023. There are two likely causes for this trend:

1. Deterministic wallets: as we saw in "Recovery Codes", it's possible to back up every key in most modern wallets by simply writing down a few words or characters. This is achieved by deriving every key in the wallet from those words or characters using a deterministic algorithm. It's not possible to use vanity addresses with a deterministic wallet unless the user backs up additional data for every vanity address they create. More practically, most wallets using deterministic key generation simply don't allow importing a private key or key tweak from a vanity generator.
2. Avoiding address reuse: using a vanity address to receive multiple payments to the same address creates a link between all of those payments. This might be acceptable to Eugenia if her non-profit needs to report its income and expenditures to a tax authority anyway. However, it also reduces the privacy of people who either pay Eugenia or receive payments from her. For example, Alice may want to donate anonymously and Bob may not want his other customers to know that he gives discount pricing to Eugenia.

Given those problems, we don't expect to see many vanity addresses in the future, although there will probably always be some.

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.

WARNING

Paper wallets are an OBSOLETE technology and are dangerous for most users. There are many subtle pitfalls involved in generating them, not least of which the possibility that the generating code is compromised with a “back door”. Hundreds of bitcoin have been stolen this way. Paper wallets are shown here for informational purposes only and should not be used for storing bitcoin. Use a recovery code to backup your keys, possibly with a hardware signing device to store keys and sign transactions. DO NOT USE PAPER WALLETS.

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. Table 4-9 shows the simplest form of a paper wallet.

Table 4-9. Simplest form of a paper wallet—a printout of the Bitcoin address and private key

Public address	Private key (WIF)
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpU

Paper wallets come in many designs and sizes, with many different features. Figure 4-10 shows a sample paper wallet.

Figure 4-10. An example of a simple paper wallet

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. 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 4-11. An example of a paper wallet with additional copies of the keys on a backup “stub”

From the original public-key focused design of Bitcoin to modern addresses and scripts like bech32m and pay-to-taproot—and even addresses for future Bitcoin upgrades—you’ve learned how the Bitcoin protocol allows spenders to identify the wallets which should receive their payments. But when it’s actually your wallet receiving the payments, you’re going to want the assurance that you’ll still have access to that money even if something happens to your wallet data. In the next chapter, we’ll look at how Bitcoin wallets are designed to protect their funds from a variety of threats.

About the Authors

Andreas M. Antonopoulos is a noted technologist and serial entrepreneur who has become one of the most well-known and well-respected figures in bitcoin. As an engaging public speaker, teacher, and writer, Andreas makes complex subjects accessible and easy to understand. As an advisor, he helps startups recognize, evaluate, and navigate security and business risks.

Andreas grew up with the internet, starting his first company, an early BBS and proto-ISP, as a teenager in his home in Greece. He earned degrees in computer science, data communications, and distributed systems from University College London (UCL)—recently ranked among the world's top 10 universities. After moving to the United States, Andreas cofounded and managed a successful technology research company, and in that role advised dozens of Fortune 500 company executives on networking, security, data centers, and cloud computing. More than 200 of his articles on security, cloud computing, and data centers have been published in print and syndicated worldwide. He holds two patents in networking and security.

In 1990, Andreas started teaching various IT topics in private, professional, and academic environments. He honed his speaking skills in front of audiences ranging in size from five executives in a boardroom to thousands of people in large conferences. With more than 400 speaking engagements under his belt he is considered a world-class and charismatic public speaker and teacher. In 2014, he was appointed as a teaching fellow with the University of Nicosia, the first

university in the world to offer a masters degree in digital currency. In this role, he helped develop the curriculum and cotaught the Introduction to Digital Currencies course, offered as a massive open online course (MOOC) through the university.

As a bitcoin entrepreneur, Andreas has founded a number of bitcoin businesses and launched several community open source projects. He serves as an advisor to several bitcoin and cryptocurrency companies. He is a widely published author of articles and blog posts on bitcoin, a permanent host on the popular Let's Talk Bitcoin podcast, and a frequent speaker at technology and security conferences worldwide.

David A. Harding is a technical writer focused on creating documentation for open source software. He is the co-author of the Bitcoin Optech weekly newsletter (2018-23), 21.co Bitcoin Computer tutorials (2015-17), and Bitcoin.org developer documentation (2014-15). He is also a Brink.dev grant committee member (2022-23) and former board member (2020-22). David previously worked freelance (2007-15).