CHAPTER 7

The Blockchain

The blockchain is arguably the most important innovation introduced by Bitcoin. It is the missing link that makes distributed peer-to-peer digital currencies possible. The blockchain is in essence a distributed database holding all the Bitcoin transactions since the beginning (January 3, 2009) and a method to secure this database.

The blockchain keeps a secure list of all the transactions. However, there are relevant questions, such as whether a particular transaction output is spendable, that the blockchain does not answer directly. Software that uses the blockchain, like mining nodes or wallets, has to parse the blockchain to extract the relevant information. This information extracted from the blockchain is usually fed to a database. For instance, the Bitcoin node software uses LevelDB, a key-value store to keep a copy of the unspent transaction outputs (UTXO). To populate this database, the whole blockchain has to be parsed as explained at the beginning of Chapter 6 (see Figure 6.2).

The blockchain uses **proof-of-work** to secure the distributed database. This means the blockchain is secured against tamper attempts by the computational power that has been applied to create it. An attacker wishing to change the blockchain would have to apply a computational power equivalent to all the computational power spent from that point in time to the present. Furthermore, the attacker would have to outrun the legitimate Bitcoin network, which keeps adding entries to the distributed database. In other words, it would have to catch up, computationally speaking, with the legitimate network to change the information in the database. Attacks to the blockchain will be covered in section 7.5.

Section 7.1 introduces hash functions, the cryptographic primitive used in the blockchain. Section 7.2 presents time-stamping, a way to secure information at a certain point in time. Section 7.3 introduces the proof-of-work concept. Section 7.4 puts together the previous sections, explaining how the Bitcoin blockchain works. The last sections cover particular details of the blockchain, such as double-spend attacks and how to protect against them (section 7.5), how to update blocks quickly (section 7.6), and scalability with an increasing rate of transactions (section 7.7).

7.1 HASH FUNCTIONS

A **hash function** is an algorithm that takes data of arbitrary length as an input and outputs a bit-string of fixed length, named the **hash value**. The hash value is always the same for the same input data. A hash function is a map from the set of input data (of arbitrary length) to the set of hash values, with the particularity that small differences in the input data produce large differences in the result. A common requirement is that hash computation is fast. Hash functions are widely used in computing, for instance to quickly locate data records using hash tables. The size of the hash values is usually smaller than the size of possible input data. Therefore many input data points will share a single hash value.

A good hash function should distribute input values to hash values proportionally, so that every hash value is linked to roughly the same number of possible input values. One way to achieve this proportionality is making the hash value "behave" as randomly as possible. Note that although the hash value "behaves" as though it was random, it is still deterministic: given an input, its hash value will always be the same[1].

Bitcoin uses **cryptographic hash functions** to perform proof-of-work. Cryptographic hash functions (sometimes called secure hash functions) impose additional requirements over regular hash functions:

- **One-wayness (preimage resistance)**. Given the hash value, it must be computationally infeasible to find out the input data. As will become clear later, this is a key property for the application of proof-of-work.
- **Weak collision resistance**. Given an input it is computationally infeasible to find another input with the same hash value.
- **Strong collision resistance**. It is computationally infeasible to find two input data points that result in the same hash value.

Bitcoin uses SHA256^2 as its proof-of-work function. SHA256^2 is the application of the SHA256 hashing function twice. SHA256 is part of a set (SHA-2) of hash functions designed by the NSA and published by NIST in 2001. SHA256 is a hash function of this family whose output is 256 bits long.

Figure 7.1 shows an example of the SHA256^2 hash function. The input value at the top of the figure is the string "This is the message #1". The hash value in hexadecimal

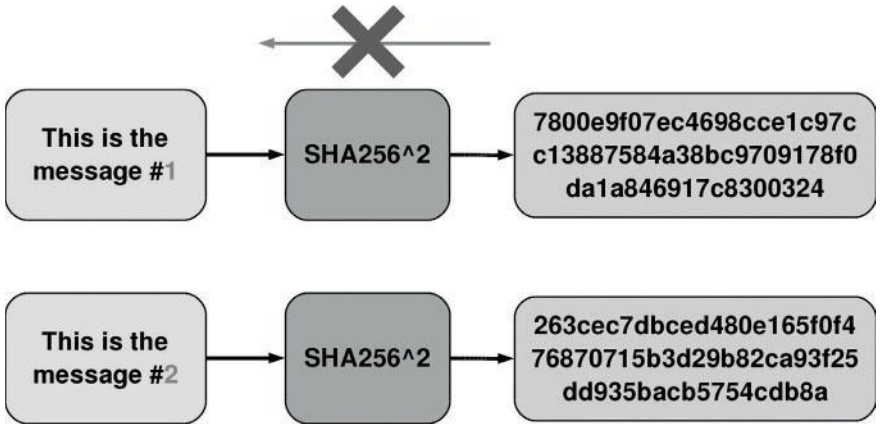

This is the message #1
SHA256^2
7800e9f07ec4698cce1c97c c13887584a38bc9709178f0 da1a846917c8300324
This is the message #2
SHA256^2
263cec7dbced480e165f0f4 76870715b3d29b82ca93f25 dd935bacb5754cdb8a

FIGURE 7.1 Hash functions

[1] Sometimes hash functions are compared to meat grinders. The analogy works in the sense that it is not possible to revert from the hamburger to the piece of meat. However, the analogy is not so good in stressing that, given the same exact piece of meat, the hamburger must be exactly the same.

is “7800e9f07...”. In the lower part of the figure the input value is the string “This is the message #2” and the hash value is “263cec7db...”. Note how a change of just one digit in the message leads to a completely different hash value. A good hash function behaves like a random mapping from the input value to the hash value.

SHA256 meets the preimage resistance requirement: given the hash value it is computationally infeasible to recover the message that generated it. Computationally infeasible means that there is no known algorithm that can recover the message within a time that is related polynomially to the size of the input. What this means in practice is that the best known algorithms to break a hashing function, i.e. recover the message given the hash, are brute-force algorithms that take an impractical (exponential) amount of time.

Many cryptographic hash functions, SHA256 among them, are constructed from a simpler primitive called a **compression function**. A compression function operates on an input of a fixed length and produces an output of the same length. The goal of a compression function is to scramble the bits of the input in a deterministic but complicated way to arrive at the output. This is achieved putting the original message through a series of data-shifting and mixing operations with random-like constants[2].

The **Merkle-Damgård construction** is a recipe to build cryptographic hash functions that accept input data of arbitrary length using a compression function as a building block. Merkle and Damgård demonstrated that, if the compression function is collision-resistant, then the whole construction is also collision-resistant. Figure 7.2 shows the general construction. In the case of SHA256, the compression function (labeled *f* in the figure) operates on 256 bits of data. The compression function accepts two inputs: an intermediate hash value[3] and a block of input data. To compute the SHA256 hash, the input data (also called the message) is first broken into blocks of 256 bits length. The end of the message is padded with zeros and the length of the message (the figure is a simplification, see National Institute of Standards and Technology (2001) for details). The intermediate hash value is initialized to *IHV* (Initial Hash Value). Then the compression function is applied to each message block, using the hash value from the last step as the

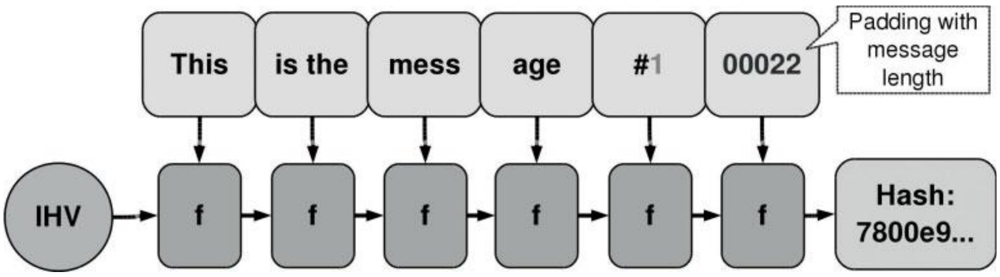


FIGURE 7.2 Merkle-Damgård construction

[2] The compression function itself is usually created from a building block which is iterated several times, 64 times in the case of SHA256. The interested reader can find the details in National Institute of Standards and Technology (2001).

[3] The intermediate hash value is used to initialize the internal registers at the beginning of the first round (National Institute of Standards and Technology, 2001).

intermediate hash value into the next step. The hash value yielded by the last step is the SHA256 hash of the whole message.

SHA256, as with most cryptographic hash functions, was designed to be fast on general purpose hardware. Also, because the round of the compression function uses only bitwise operations and additions with 32-bit registers, it is suitable for efficient hardware implementation. This has given rise to the ASIC mining industry, which will be covered in Chapter 9.

PASSWORD HASHING

Many websites face the problem of authenticating users. It is common practice to issue or allow users to choose a user-name and password that will be subsequently used for authentication. Websites could retain this information in clear-text. This is easy to implement: each time a user logs on, the password sent along is just compared with the stored clear-text password. However, this is considered a very bad security practice, because if the website were to be compromised the passwords of all its users would automatically get leaked.

Thus it is recommended that websites do not keep a copy of the clear-text password, but of the hashed password. When the user logs on, the password sent along is hashed, and this hash is compared with the stored hash. If the website gets hacked, then only hashed passwords get leaked. To obtain the clear-text passwords from the hashed passwords, the attacker would have to brute-force the hashes. Brute-forcing is done by trying many possible clear-text passwords to see if their hashes match the desired hash. This is a time-consuming process, but there are some shortcuts. One of these shortcuts is to construct **rainbow tables**. If an attacker wants to crack many passwords at the same time, she could store a list of passwords along with their hashes in a database (called a rainbow table). Once a rainbow table is available, some passwords can be cracked just by searching their hashes in the table.

To protect against rainbow tables it is advisable to salt passwords. A nonce (a random number used only once), called the salt, is stored along with the hash of each password. The **salt** is prefixed to the clear-text password before hashing. This defeats the purpose of rainbow tables. It is still possible to brute-force passwords, but it has to be done one password at a time, worsening the economies of scale of password cracking.

Many standardized hash functions, including SHA256, are fast. This can be a problem for password hashing, because fast hash functions make password cracking easier. One possible solution is to apply the hash function multiple times. For instance, the Linux operating system allows specifying a number of rounds (for instance 50,000) that the hashing function will be applied to a (salted) password prior to storing it in the /etc/shadow file. Using many rounds slows password cracking attempts.

Another approach to protect against brute-force password cracking is to use a key-derivation function. A **key-derivation function** is a function that intentionally takes a large amount of time to evaluate. As legitimate users only need to perform this computation once when authenticating, its difficulty can be set such that it takes a slightly noticeable time, say half a second, on regular hardware. An attacker trying to brute-force the hash would have to spend half a second for each trial, slowing her down considerably. Examples of key-derivation functions are slow hash functions such as in **PBKDF2** (Kaliski, 2000) or **bcrypt** (Provos and Mazières, 1999) and **memory-hard hash functions** such as **scrypt** (Percival, 2012). Scrypt will be covered in section 11.1 when introducing Litecoin.

The Merkle-Damgård construction is vulnerable to **length extension attacks**. Suppose an attacker does not know the message m, but knows its hash $H(m)$. She could compute the hash of a message $m \| m'$ (where $\|$ denotes concatenation) by using $H(m)$ as the IHV to the hash algorithm and then feeding it m', thus obtaining $H(m \| m')$ as a result[4]. One solution to the length extension problem is proposed in Ferguson et al., (2010): hash twice. That is, apply the hash function again to the hash of the message[5].

It is believed that Satoshi chose SHA256^2 instead of plain SHA256 as a protection against length-extension attacks. It is not clear how to exploit length extension against the proof-of-work, but it seems that Satoshi decided to play it safe. Besides, the main downside of double hashing, namely an increase in computation time, is irrelevant in Bitcoin, because the whole point of the proof-of-work is to make finding the nonce computationally expensive (section 7.3).

7.2 TIME-STAMP

A digital **time-stamp** is analogous to a physical time-stamp, such as a postmark on a letter or a rubber stamp from an official organism. A digital time-stamp proves that certain information—say a digital document—existed at a particular time. It has many applications, such as documenting that a contract between two parties occurred, that a transaction in a website materialized, or indeed that a group of transactions in a digital currency took place.

The information included in a digital time-stamp is usually the hash of the data to secure. Using a hash has several advantages. First, the information to be time-stamped can be kept private and separate from the medium used to secure the time-stamp. Second, a hash is usually significantly smaller than the information that generated it, reducing storage costs. Third, digital signatures usually work best on data of a predetermined size.

[4] If the hash of m is padded and the length has been appended to it, the attacker might not be able to compute the hash of an extended message $m \| m'$ but of $m \| padding + length(m) \| m'$. This might still be acceptable for some attacks.

[5] Bitcoin applies the double hashing differently than in Ferguson et al. (2010). Bitcoin computes $H(H(m))$, whereas Ferguson et al. (2010) recommend computing $H(H(m) \| m)$.

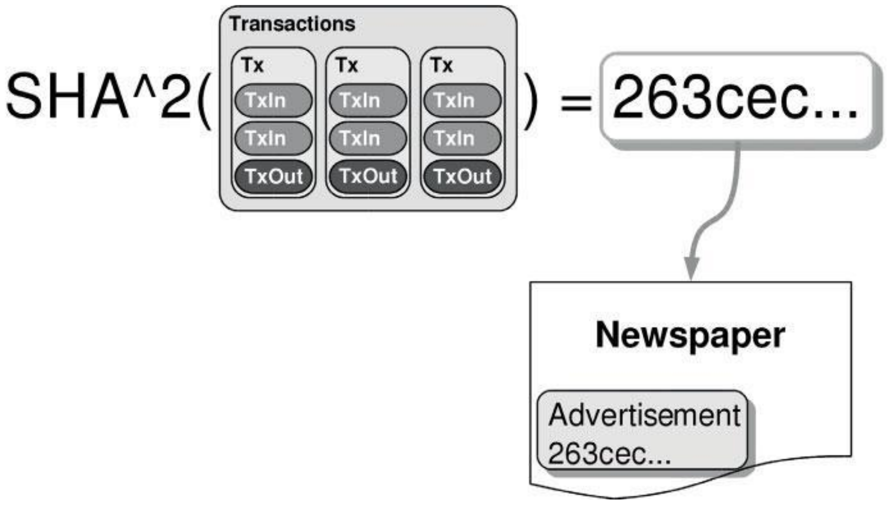


FIGURE 7.3 Time-stamping a group of transactions by publishing their hash in a newspaper

There are several ways to secure a digital time-stamp. A naive method would send a trusted party a copy of the information, and the trusted party would store in a (hopefully) safe place the data received, together with the time of reception. This method is susceptible to the trusted third party losing the database or being compromised. A second method would rely on a trusted third party, called a **Time-Stamping Authority** (TSA) (also called a **digital notary**) to sign with its private key a combination of the data to be secured together with the time when this data was communicated to the authority. This signature would be sent back to the original owner of the data. The security of this set-up rests on the integrity of the TSA. If the TSA decided to collaborate with a cheater, it could time-stamp a document retroactively[6].

A third method to secure a digital time-stamp would be to publish the hash of the data in a public place, such as a newspaper. This is shown in Figure 7.3, where a group of transactions is hashed, and the hash is published in an advertisement in a public newspaper. An attacker wishing to change any of the transactions would have to find a collision with the published hash[7], or alternatively subvert all the papers with a copy of the published hash.

A time-stamping authority could use this idea, publishing for instance a hash of the signature of a time-stamp. It would, however, be costly to publish a hash for every time-stamp. A TSA could achieve economies of scale by "mixing" the hashes of multiple time-stamps before publishing. Combining the hashes of multiple data to be time-stamped can be done efficiently using Merkle trees, covered in section 7.6.

[6] The time stamping authority method can be refined to account for a non-trusted TSA (Haber and Stornetta, 1991; Massias et al., 1999).

[7] The attacker would have to find a change in any transaction such that when all the transactions are hashed again, the original hash is recovered. If the hash function is collision-resistant, this attack is computationally infeasible.

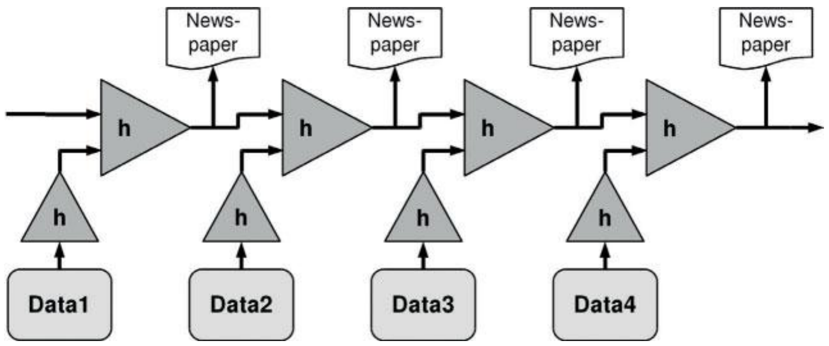


FIGURE 7.4 Linked time-stamping

Figure 7.4 introduces another refinement to further secure digital time-stamps: **linked time-stamps**. The main insight is that if the hash to be published is linked to the previously published hash, this will help further secure the older hash. An attacker who wishes to change some data from the older hash would have to find a *double* collision in the hash function: one collision for the old hash and another for the new published hash. After another step in the chain, the attacker would have to find a *triple* collision. Thus the hash chain increases exponentially the security of older hashes. Although in Figure 7.4 the hash is published at each point in time, hashes could be published with a longer periodicity to save costs.

In summary, a Time-Stamping Authority could work in the following way. First, data to be time-stamped from clients is gathered during a certain period. When the collection period finishes, the data is hashed together using, for instance, a Merkle tree (section 7.6). The resulting hash (root of the Merkle tree) is hashed together with the final hash of the previous time period, and this final hash is published.

A digital currency could use this method to secure transactions. But this requires a central counterparty to gather the transactions and publish them. It would also need a public medium, such as newspaper advertisements, to publish the hashes. The central counterparty would presumably bear the cost of publishing the hashes. This central counterparty running the currency constitutes a central point of trust in the system. However, the main design principle of Bitcoin is distributed trust, and the proof-of-work is the last piece of the puzzle to achieve it.

7.3 PROOF-OF-WORK

Digital services are subject to several types of attacks and abuses against them. One example, denial of service (DoS), occurs when a server is flooded with bogus requests for its services. If the number of bogus requests is greater than the number of requests the server can cope with, legitimate users will be unable to get served, or experience delays. A denial of service attack thus disrupts the regular flow of a server and should be avoided. Another such attack is email spam, where an email account is filled with unsolicited email that usually contains advertising or malware.

DENIAL OF SERVICE (DOS) ATTACKS

Denial of Service attacks are attempts to make a machine unavailable. DoS attacks were featured in mainstream news when the Anonymous collective attacked several websites.

A DoS attack usually enlists many participants in disparate locations. Sometimes an attacker can enlist a botnet[a] under her control to perform the attack. These are called Distributed Denial of Service (DDoS) attacks. DDoS attacks are harder to counter, because the bogus requests come from many IP addresses, making it harder to distinguish attackers from legitimate users.

An account of underground groups on the internet practising DoS for profit can be found in Menn (2010). An account of the adventures of Anonymous, including several of their DoS attacks, can be found in Olson (2013).

[a] A botnet is a group of computers under the control of a single (malicious) entity. Computers connected to the internet are usually enlisted to be part of the botnet by infecting them with malware, turning them into zombie computers. The entity controlling the botnet can command them to perform a task, such as reporting login or financial information, participating in a DoS attack, clicking on advertisements on a website (click fraud), serving email spam, or even mining Bitcoins for their controller. This last application was popular for a brief period, but given the advances in mining technology it is not economical any more (Chapter 9).

One possible defense against these attacks is to require the client requesting the service prove that some work has been done, i.e. **proof-of-work**. The proof-of-work could be the solution of a computationally-hard problem, a memory-hard problem, a problem requiring user intervention (such as a CAPTCHA)[8], and so on.

The problem must be moderately *hard to solve but easy (computationally fast) to verify*. A service provider can then pose the problem to anyone who requests the service, and only grant access to those users that pass the challenge. This allows the service provider to identify the users that are willing to pay the price to enter the service. If the proof-of-work is well designed this price will be a minor inconvenience (like a short delay) for legitimate users but an economic deterrent to attackers of the service. Proof-of-work systems can be implemented following two protocols:

- **Challenge-Response**. This protocol assumes an ongoing communication between the client and the server. First the client requests the service, then the server chooses a proof-of-work and challenges the client. The client then has to solve the proof-of-work and send the response to the server. Finally the server verifies that the proof-of-work has been correctly done and then grants the client access to the service. This is

[8] CAPTCHA or "Completely Automated Public Turing test to tell Computers and Humans Apart" is a proof-of-work test where the user is presented with hard to read letters that she must type back. It is used to tell apart humans from machines, given that humans are still somewhat better than artificial intelligence algorithms at recognizing characters. However, artificial intelligence is progressing fast, as is the difficulty of the CAPTCHAs.

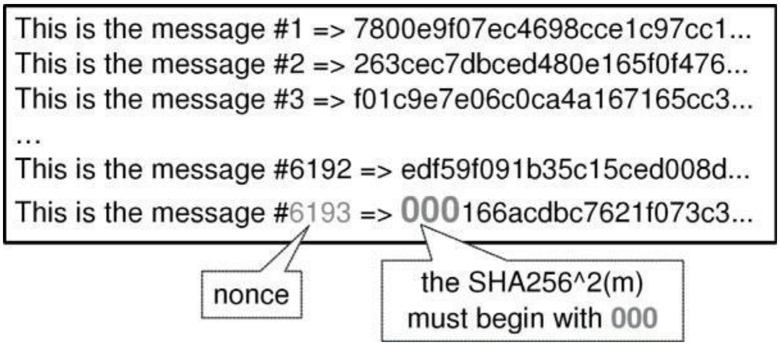
the model used for example in CAPTCHAs. The advantage of this protocol is that the server can adapt the difficulty of the proof-of-work to the conditions, such as server load.

- **Solution-Verification**. This protocol is asynchronous: solution and verification can be done at different times. Ongoing communication between the server and the client is not required. First, the client creates a proof-of-work problem and solves it: this problem should be different every time and chosen by an algorithm: for instance, the client could use the result of a hash function to generate it. Then the client sends the solution to the server, who verifies it and proceeds accordingly.

To secure the blockchain—the distributed transaction database—Bitcoin requires proof-of-work to be performed on blocks of transactions following the Solution-Verification protocol. Bitcoin uses **partial hash inversion** as the proof-of-work function. Partial hash inversion requires that the hash of a block of transactions matches a certain pattern. The pattern to match is that the hash starts with at least a certain number of 0 bits. This is called hash inversion because the proof-of-work must invert (i.e. match) a certain pattern in part of the hash. It is important that the hash function is preimage-resistant. Otherwise it would not be computationally hard to find a partial hash inversion, defeating the purpose of the proof-of-work.

Figure 7.5 shows partial hash inversion applied to the message "This is the message #". The hash function used in the figure is SHA256^2, Bitcoin's hash function. The partial hash inversion in this example requires that the first 12 bits (3 hexadecimal characters in the figure) are zero. The free variable to solve the partial hash is a nonce appended to the message. To solve the partial hash inversion, a nonce must be found such that the hash of the message and the nonce matches the partial hash, i.e. the hash starts with at least three zero characters. Figure 7.5 increments the nonce until it finds a solution. The solution is found in the nonce value 6193. Although this example starts with a nonce value of 1 and increments it at each step, nothing prevents one from trying nonces at random. For well behaved hash functions, the computational cost of both methods is equivalent.

It is computationally costly to solve the partial hash inversion, because many nonces (6,193 in this case) had to be tried before finding a solution. However, it is computationally cheap to verify that the work has been done, it only requires one hash



```
This is the message #1 => 7800e9f07ec4698cce1c97cc1...
This is the message #2 => 263cec7dbced480e165f0f476...
This is the message #3 => f01c9e7e06c0ca4a167165cc3...
...
This is the message #6192 => edf59f091b35c15ced008d...
This is the message #6193 => 000166acdbc7621f073c3...
```

FIGURE 7.5 Partial hash inversion proof-of-work

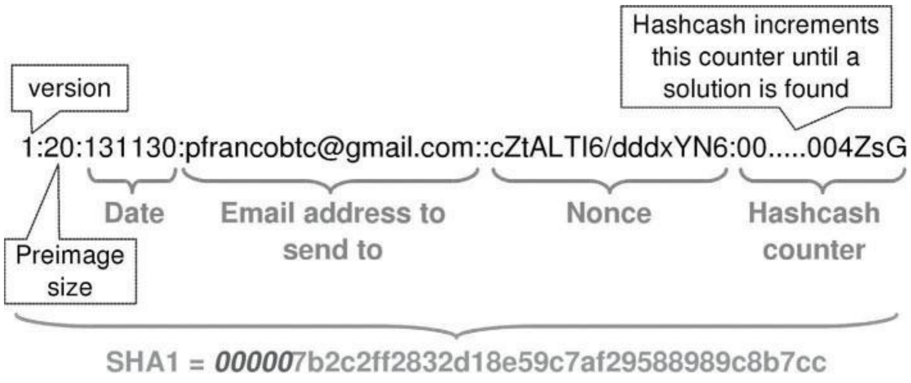

Hashcash increments this counter until a solution is found
version
1:20:131130:pfrancobtc@gmail.com::cZtALTI6/dddxYN6:00.....004ZsG
Date
Email address to send to
Nonce
Hashcash counter
Preimage size
SHA1 = 00000 7b2c2ff2832d18e59c7af29588989c8b7cc

FIGURE 7.6 Hashcash

evaluation[9]. One of the advantages of using partial hash inversion is that the difficulty of the proof-of-work can be easily adjusted by changing the number of 0 bits that the solution hash must begin with.

Bitcoin borrowed the idea of using partial hash inversion as a proof-of-work function from **Hashcash**. Hashcash was introduced by Adam Back in 1997 as a mechanism to put a curb on spam email (Back, 2002). Email is cheap to send, and this has led to the proliferation of spammers. Spammers send millions of unsolicited emails with commercials (or malware). Even though the probability of any single email to have its desired effect on the recipient (i.e. clicking on it) is very small, it is economically profitable for spammers because of the very low cost per email sent. Hashcash proposes to add a token to the header of an email. This token includes a proof-of-work that some computational time has been spent to generate it. The proof-of-work function used in Hashcash is partial hash inversion, using SHA-1 as the hashing function.

Figure 7.6 presents an example of a Hashcash header. The header includes:

- The **version** of the Hashcash protocol, 1.
- The **preimage** size, 20 bits or 5 initial zero characters in hexadecimal. The preimage size indicates the number of initial zero bits for the partial hash inversion.
- The **date** when the email is sent, 131130 or November 30, 2013. The hashcash token is only valid for a certain period.
- The **recipient email address**, pfrancobtc@gmail.com.
- A **nonce** (cZtALTI6/dddxYN6) used only for one email. The logic behind this nonce is to avoid the same hashcash header being used to send many emails. The receiving email server can store the nonces in a cache and if an email arrives whose nonce is found in the cache, the email is dropped. This avoids a botnet master from reusing the same hashcash header in multiple emails.
- The **hashcash counter** (00…004ZsG). Hashcash increments this counter until it finds a value such that the hash of the whole header matches the preimage requirement.

[9] The function that checks that the proof-of-work test is correctly passed in the Bitcoin Core source code is called CheckProofOfWork and can be found in `main.cpp`.

The SHA-1 hash of the hashcash header is shown at the bottom of Figure 7.6. The first 20 bits (5 hexadecimal characters) of this hash are zeros, and therefore the header is valid. The amount of work to solve the partial hash collision problem is exponential in the number of initial zeros, doubling with each additional zero bit. On the other hand, confirming the validity of the hashcash header always takes the same amount of work.

7.4 THE BLOCKCHAIN

Bitcoin combines the ideas behind linked time-stamping and Hashcash style proof-of-work to arrive at a way to secure the distributed database. This is the main innovation introduced by Bitcoin.

The **blockchain** is an *ever-growing chain of blocks*. Each block contains a group of new transactions and a link to the previous block in the chain. New transactions in the network are collected into a block which is appended to the blockchain. Note that old transactions are still in the blockchain: old blocks are never removed from the blockchain, thus the blockchain can only increase in length.

Each block is secured with a partial hash inversion proof-of-work. This is depicted in Figure 7.7, arguably the most important figure in this book. First, each block includes a group of (valid) transactions, the hash of the previous block, and a nonce. The nonce in a block solves the partial hash inversion problem. That is, the nonce is a number such that the hash of the entire block (including the nonce) starts with a certain number of zero bits. In Figure 7.7 the hash of the block 271,076, mined on November 23, 2013, is 0x000000000000006e1163..., starting with 61 zero bits (or 15 zero characters in hexadecimal representation).

It is easy to adjust the **block difficulty** by increasing the number of starting zero bits. The Bitcoin protocol adjusts this difficulty to target **10 minutes between blocks**. This difficulty adjustment is part of the rules of Bitcoin and it is coded into every Bitcoin client.

The block difficulty is adjusted every 2,016 blocks or roughly every 2 weeks[10]. The adjustment takes into account the change in the computational power of the whole network since the last adjustment, by comparing the time-stamps of two blocks 2,016 positions apart. When mining power is being added to the network, blocks will be mined quicker than the target 10 minute period. The difficulty will be adjusted higher but trailing the growing network power.

If the correction to the difficulty is greater than 4 or lower than 1/4, it is clamped to those values. This helps protect against large swings in the difficulty if the power of the network changes too quickly.

Finding a partial hash collision is done through brute forcing: trying random (or consecutive) nonces until one of them creates the partial hash collision. Thus the time it takes to solve a block is random.

The persons or institutions contributing their computational power to solve the partial hash inversion power are called **miners**. The assistance of the miners is critical to secure the blockchain, and the protocol compensates them with newly minted bitcoins. A **block reward** is the compensation that is paid every time a miner solves the hash inversion

[10] It would be exactly two weeks if the difficulty during the 2 weeks was perfectly adjusted to the power of the network, and blocks would get mined exactly every 10 minutes.

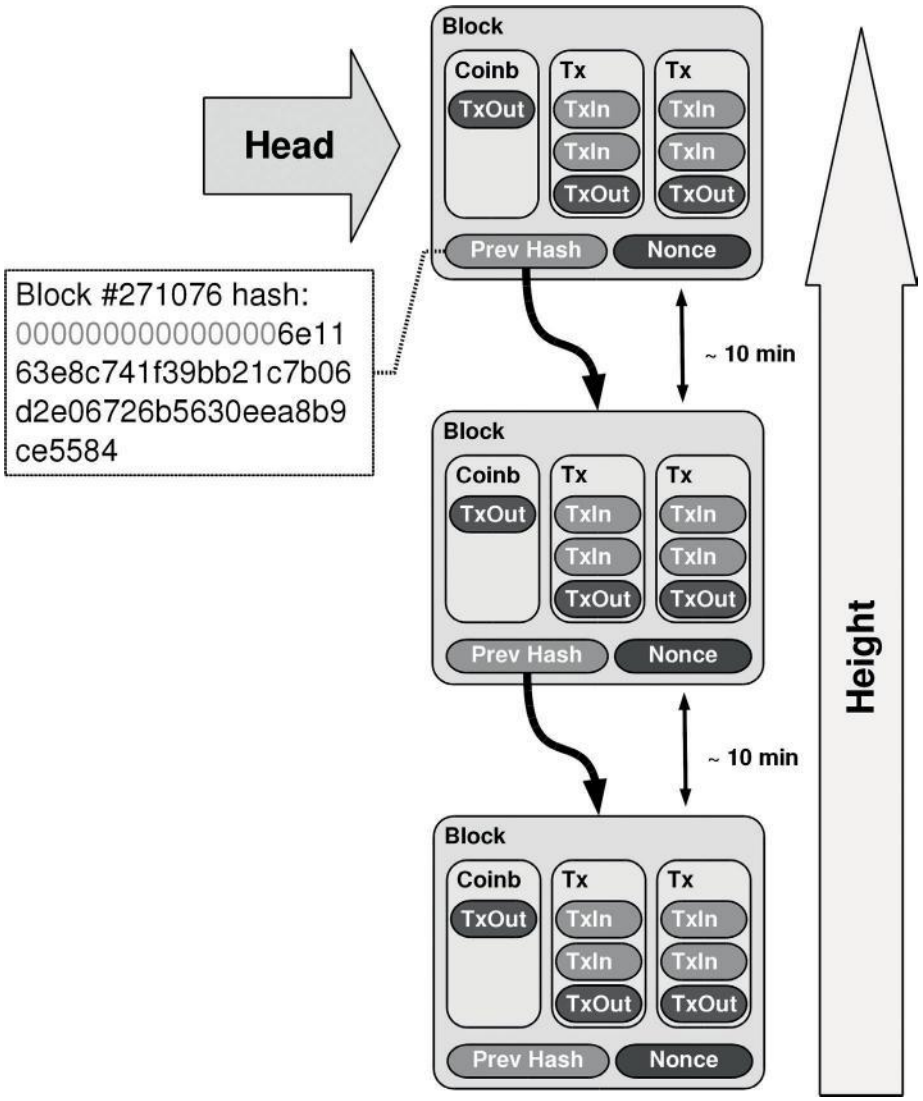


FIGURE 7.7 The blockchain

problem. Thus *new bitcoins issued are allocated to the miners that contribute their computational power to secure the blockchain.*

Every block includes a special transaction called the **coinbase**[11]. The coinbase is the first transaction in a block—see Figure 7.7. It has only one transaction input (TxIn), which is not linked to any previous transaction output (TxOut) and does not serve any purpose[12]. On the other hand, the coinbase can have many outputs. The sum of values

[11] Technically the coinbase is the only input to the first transaction in a block. Abusing the notation, throughout this book the first transaction in a block will be referred to as the coinbase transaction.
[12] This input can present any value. Some miners use it to publish a message, or to secure data in the blockchain (section 12.6). The coinbase of the very first block by Satoshi has the message: "The

in these outputs equals the block reward, plus the sum of all the fees granted by the transactions collected into the block.

BLOCK DIFFICULTY

The partial hash inversion problem can be understood as finding a nonce such that the resulting hash is lower than a **target hash**. The target hash is a 256-bit number that starts with several zero bits. Internally the Bitcoin client compresses this 256-bit number into a 32-bit unsigned integer, named **bits**. As each block saves a copy of the *bits* integer, this compression saves considerable space in the blockchain.[a]

In the example of Figure 7.7, *bits* has a value of 419892219, or **0x19070bfb** hexadecimal. This gives a target hash of **0x070bfb** $2^{8\cdot(\mathbf{0x19}-3)}$ = 0x0000000000000070b-fb000… where all the last zero nibbles have been omitted. The mined block had a hash of 0000000000000006e1163e8... thus meeting the target.

Sometimes the **difficulty** of mining a block is represented as a floating point number. This floating point number is the quotient of the initial target hash 0xffff $\cdot\ 2^{208}$ over the current hash target. The difficulty started at 1 at the beginning of the blockchain (January 3, 2009) and it cannot be lower than 1.

[a] The *bits* 32-bit value is stored in a variable named `nBits`, member of the class CBlock-Header. The definition of this class can be found in `core.h`. The conversion between the full 256-bit hash and the compressed 32-bit is performed in CBigNum::SetCompact and CBigNum::GetCompact, both in `bignum.h`.

Blocks usually include many transactions aside from the coinbase. But a valid block can be created without including any transaction except for the coinbase. Indeed, this type of block was very common at the beginning of the Bitcoin network, when very few transactions were broadcast. These “empty” blocks help secure the blockchain and award miners the block reward for doing so. Miners can choose which transactions to include in the blocks they mine, and they usually decide to include them based on the fee they pay (section 9.3).

The miner who solves the hashing problem uses the coinbase to pay the block reward to herself. This is why solving the partial hash inversion problem is called **mining**. All miners in the network compete to mine blocks. The **hash rate** of a miner is her computational power, measured in hashes/second. The **network hash rate** is the total hash rate of all the miners in the network, and it stands at around 30,000 TH/s[13] at the time of writing. Mining a block can be compared to a lottery, where the chances of a single miner are proportional to her share of the network hash rate.

The block reward was initially set to 50 bitcoins per block. Every 210,000 blocks, or roughly every 4 years, the block reward is halved. At the time of writing the block reward stands at 25 bitcoins. Figure 2.7 from Chapter 2 shows the total number of bitcoins in circulation assuming each block is mined exactly every 10 minutes. As expected, the

Times 03/Jan/2009 Chancellor on brink of second bailout for banks.”

[13] That is 30,000,000,000,000,000 hashes/second.

number of bitcoins awarded (the slope in the figure) decreases exponentially, making the total amount of bitcoins in circulation settle to a value of approximately 21 million. This issuance schedule is programmed in the logic of Bitcoin Core[14]. This **currency generation algorithm** is considered immutable by the community. At the time of writing, the total amount of bitcoins in circulation stands at around 12.5 million.

Historically roughly 99% of the compensation of miners has come from the block reward, and only 1% of their retribution from transaction fees. It is expected that over time a larger percentage of the retribution will be due to transaction fees.

The process of solving blocks is called mining in analogy to mining precious metals. Miners are rewarded with new currency. This analogy, although useful, can be carried too far. The block reward is fixed by the protocol and is not affected by the number of miners or the power they contribute. Contrary to precious metals mining, an increase in the mining investment does not increase the number of bitcoins in circulation. New investment in mining increases the total hash rate, thus reducing the share of original miners, keeping the total reward for the network constant[15].

The block preceding a given block is called its **parent block**. Blocks reference their parent block in the blockchain by including its hash in their data structure, as in Figure 7.7. Thus the blockchain keeps blocks in chronological order. The first block in the blockchain is called the **genesis block** and it was created by Satoshi in January 3, 2009. The order of a block in the blockchain, starting from the genesis block is called the **block height**. The last block added to the blockchain is called the **blockchain head**. New blocks are added on top of the blockchain head.

A **fork** occurs when two miners arrive at a new block at roughly the same time. Both blocks solve the partial hash inversion problem, but only one of them can be part of the long-term blockchain. The discarded block is called an **orphan block**. The decision of which branch of the blockchain is the valid one is not taken by any party[16]. Rather the dispute resolves itself organically.

A fork can persist for several blocks, as shown in Figure 7.8. This happens when there is a split in the network, and some miners believe one branch of the fork is the legitimate blockchain, while the others follow the other branch. The protocol determines that *the correct blockchain is the longest one*. So miners have an incentive to stop working in a branch as soon as it is clear that it will be orphaned, because work on an orphaned branch is wasted. Therefore forks resolve themselves quickly, usually in just 1 block. The average number of forks has been around 2%, i.e. on average every 50 blocks there is a fork in the blockchain. Forks of more than one block are very rare.

Transactions included in the blocks of a fork are not lost. When a fork is resolved and a branch of the blockchain is discarded, the transactions in that branch are introduced again into the unconfirmed transactions' memory pool, ready to be included in the next block mined. Some of these transactions might already appear in a block of the

[14] The path followed by the money supply is controlled by the variable CMainParams::nSubsidyHalvingInterval in `chainparams.cpp`. This variable determines the numbers of blocks after which the block reward will be halved. It is later used in the function GetBlockValue in `main.cpp` to assign the reward to a block.

[15] There is a temporary increase in the speed at which new blocks are mined until the feedback mechanism catches up and settles the expected time between blocks back at 10 minutes. Thus new investment temporarily accelerates the release of new bitcoins.

[16] An exception was the fork that occurred in March 2013 following the 0.8 release of Bitcoin Core, see box "Forks Due to Changes in the Protocol."

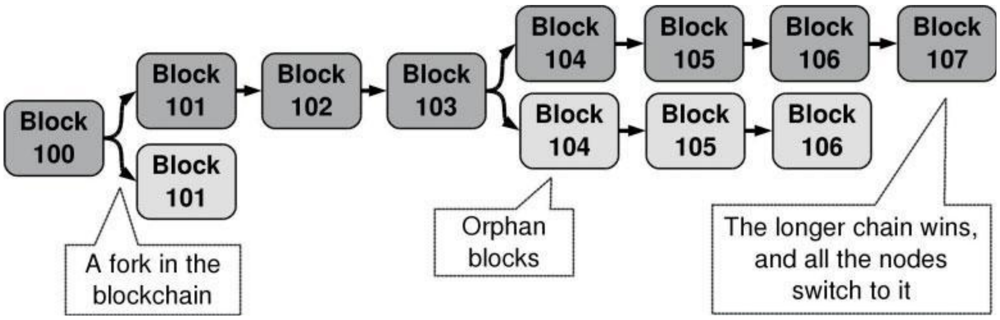


FIGURE 7.8 Dynamics of the blockchain

legitimate branch of the fork. In this case, these transactions are discarded and excluded from the unconfirmed transactions' memory pool.

Every fork resolution produces winners (the miners that solved blocks in the accepted branch) and losers (miners whose solved orphaned blocks). The protocol avoids having a central party or group deciding about the correct branch, in line with the decentralization philosophy of Bitcoin.

The Bitcoin protocol solves a fork in favor of the longest blockchain. **Blockchain length** is measured by the combined difficulty of all the blocks in the chain. If blockchain difficulty were measured by the number of blocks instead, an attacker could generate many "valid" blocks with a lower difficulty than the legitimate blockchain, thus winning the blockchain race by cheating. This is not possible, and an attacker must resort to having a significant percentage of the network hash rate at her disposal to pull off such an attack (section 7.5).

FORKS DUE TO CHANGES IN THE PROTOCOL

Forks can also occur due to differences in the code run by various nodes. One famous example occurred in March 2013 after the release of version 0.8 of Bitcoin Core, which changed the database used for the UTXO from BerkeleyDB to LevelDB. The new version introduced a subtle change in the rules of the Bitcoin protocol (Buterin, 2013a; Andresen, 2013b). This change resulted in a block being recognized as valid by the miners running version 0.8, but invalid for miners running version 0.7 of the software, creating a fork in the blockchain. As there was more computational power running version 0.8, this branch of the blockchain pulled ahead. However, this was leaving miners running the older version behind.

Developers and mining pool operators discussed the course of action, and decided to revert to the branch followed by miners with the older version. Miners and users running version 0.8 downgraded to version 0.7 and reset their blockchains to that branch. Some services had to shut down temporarily while performing the downgrade to protect from possible double-spending attacks.

Forks due to changes in the protocol are an exception to the rule that forks should resolve themselves organically.

The Bitcoin network is composed of **nodes**. Nodes are computers connected to the internet, running the Bitcoin software. The Bitcoin network is a peer-to-peer network: all nodes are homogeneous. Nodes receive transactions and blocks from other nodes, and relay these transactions and blocks to other nodes. Each node keeps a full copy of the blockchain.

A newly created transaction that has not been included into any block is called an **unconfirmed transaction**. Once a transaction is included in a block it is said to be **confirmed**. Whether a transaction is confirmed is a matter of degree: the more blocks that have been added to the top of the blockchain, the harder it is to perform a double-spending attack against a transaction. Section 7.5 further explores the topic of transaction confirmation.

Nodes store the unconfirmed transactions they receive in an database called the **unconfirmed transactions' memory pool**, often referred to simply as **mempool**[17]. Not all received transactions are added to the mempool. If a transaction double-spends an input of another transaction already in the mempool, it is dropped. A transaction is also dropped if it is not a standard transaction (section 6.7). Once a node receives a new block, or mines a block itself, the unconfirmed transactions' memory pool is updated, removing all transactions included in the block.

Transactions are not deemed confirmed until they are included in a block. This usually takes several minutes, but it could take longer. If the network is under heavy usage and there are many unconfirmed transactions, it can take several blocks for a particular transaction to be included into a block. The question if a transaction can be considered valid before it is included in a block is explored in section 7.5.

When a transaction is created, it is relayed to the Bitcoin network through a small amount of nodes. The nodes that receive the new transaction check that it is valid (Chapter 6) and is not a double-spend of a transaction already in the mempool. If the transaction passes the check, it is relayed to other nodes in the network, and is otherwise dropped. This is the way unconfirmed transactions propagate through the network.

In summary, aside from a full copy of the blockchain, a node also keeps additional data structures, such as the unspent transaction outputs cache or the unconfirmed transactions' memory pool, so that it can quickly validate new received transactions and mined blocks. If the received transaction or block is valid, the node updates its data structures and relays it to the connected nodes. It is important to note that a node does not need to *trust* other nodes, because it validates independently all the information it receives from them.

When a miner finds a new block, it broadcasts it to the network. All receiving nodes first check the validity of the block, i.e. that it solves the partial hash inversion problem with the required difficulty. They then update their internal data structures to reflect the new information contained in the block:

- Update the unspent transaction outputs cache (UTXO)[18].

[17] The mempool is implemented in the class **CTxMemPool** found in `txmempool.h`.
[18] The code performing this update can be found in the function **ConnectTip** in `main.cpp`.

- Update the unconfirmed transactions' memory pool. This involves going through the list of transactions, and dropping those that are in conflict with (spend the same outputs) as a transaction in the new mined block[19].

Nodes maintain a number of connections to other nodes in the network. Some nodes will like to keep as many connections open to other nodes as the available resources (CPU, network bandwidth) allow, usually in the upper hundredths. For instance, a wallet node might want to keep connections to many other nodes, as geographically distributed as possible, to quickly detect and act upon double-spending attempts. Similarly, a mining node might want to have as many connections open as possible, so that it receives prompt notice of mined blocks. A faster reception of new mined blocks minimizes the time wasted trying to mine a block that will become orphan. For other nodes, having up-to-the second information is not so important, and so they usually connect to only a handful of nodes.

The propagation of a block in the network is delayed by the usual network delays, but also because each node checks the full validity of a block before relaying it. Researchers have found in Decker and Wattenhofer (2013) that new mined blocks take on average around 10 seconds to propagate through the network. Propagation delay is proportional to the size of the block, so large blocks can take considerable time to propagate. They also found that most of the forks in the blockchain could be attributed to the delay in block propagation. Finally, they propose several measures and changes in the protocol to reduce propagation delay.

There are two types of nodes, **mining nodes** and **passive nodes**. Mining nodes are nodes actively trying to solve blocks, in order to reap the rewards. They include transactions from their unconfirmed transactions' memory pool into the block they are mining[20]. Passive nodes are usually employed by services such as wallets, payment processors, market data providers and so on. All nodes keep an up-to-date snapshot of the state of the network, including a full copy of the blockchain, the UTXO cache and the unconfirmed transactions mempool.

As an example, nodes providing services to wallet clients perform these tasks on behalf of their clients:

- Relay to the network the transactions created by the clients.
- Keep track of the confirmation status of a transaction. That is, notify the client when a transaction has been included in a block, and when new blocks have been added on top of the block where the transaction was included.
- Send the client the unspent transaction outputs belonging to certain addresses. This allows the client to keep track of the funds available.

Mining nodes cannot change transactions. They can only decide whether to include them in the block they are currently mining or not, and whether to relay them to the rest of the network or not. The contents of the transactions are secured by public key cryptography

[19] The code performing this update can be found in the function **ConnectBlock**, which is called by **ConnectTip**. **ConnectTip** is in turn called by **ActivateBestChain**. All these functions can be found in `main.cpp`.

[20] Section 9.3 explores how transactions are selected into a block.

and cannot be changed by the node processing the transaction[21]. At the time of writing there are roughly 10,000 nodes connected to the network—that is, there are 10,000 copies of the Bitcoin transaction database online.

BOOTSTRAPPING THE BLOCKCHAIN

When a node connects to the network for the first time, it requests a copy of the whole blockchain since the beginning (January 3, 2009). As a node downloads the blocks, it checks their validity. First it checks that all transactions included in the block are valid:

- Their inputs must spend previously unspent outputs.
- The sum of the funds in their outputs must be equal to or lower than the sum of the funds in the inputs.
- The <*scriptPubKey*> must be a valid one, all the inputs must be correctly signed.

Next, the node checks that the hash of the block is well constructed (section 7.6) and that it satisfies the required proof-of-work. By making all these checks, a new node cannot be fooled by a malicious node sending it an invalid copy of the blockchain. The only way a malicious node can feed an incorrect copy of the blockchain would be to perform a 51% attack (see next section).

At the beginning of the bootstrapping procedure, the node creates the databases for the unspent transaction outputs cache (UTXO) and the unconfirmed transactions' memory pool. These are then updated with the information in each block as the block is processed.

Bootstrapping a node uses a lot of bandwidth and computational power. At the time of writing the size of the downloaded blockchain exceeds 17GB. On a modest laptop computer with 2 cores on a residential ADSL connection, downloading and verifying the blockchain can take several days.

It may seem that nodes have an incentive not to relay a transaction, especially if the transaction awards a considerable fee to the miner. There are, however, various factors against not relaying transactions:

- Clients usually connect to various nodes when relaying a new transaction, because it is in their interest that the transaction is relayed, so that it is included in the next block mined by the network. A client can monitor whether the nodes it is using are relaying its transactions, and punish the nodes not relaying them. This is an undesired outcome for a node, which will prefer to see transactions as soon as they are created.
- Nodes connect to each other, and can police whether a particular node is refraining from relaying transactions. If a "cheater" is caught, other nodes in the network can punish it, say in a tit-for-tat strategy that stops relaying transactions to that node.

[21] Transaction malleability is an exception to this rule (7.6.1).

- A node that does not relay transactions to other nodes can be enlisted by a malicious node to help perform a double-spending attack, by feeding it a transaction that is double-spent somewhere else in the network. This can be very detrimental for the node, because it could mean losing the block reward when the malicious node achieves its goal. Given that transaction fees represent a small percentage of the total compensation of a block, holding transactions is usually not worth this risk for a node.

On the other hand, mining nodes have an incentive to broadcast mined blocks as fast as possible in order to reduce the probability that another node might mine a different block.

7.5 DOUBLE-SPEND AND OTHER ATTACKS

A **double-spend attempt** occurs when two different transactions try to spend the same funds. The Bitcoin protocol defends against this attack by deciding that the valid transaction is the one that finds its way into the blockchain first. Thus Bitcoin solves the double-spending problem in a decentralized way, without the need for a central authority to decide which transaction is valid.

A transaction is further secured as more blocks pile on top of the block where it was originally included. This is illustrated in Figure 7.9. An attacker wishing to change the blockchain at a certain block would have to mine again all blocks from that block up to the blockchain head. Moreover, as the Bitcoin network keeps on adding blocks to the blockchain, the attacker would not only have to redo all the past work, but would have to keep up with the pace of new legitimate block creation and surpass it. The only way an attacker could perform such a feat is to command a hash rate as large as the hash rate of the rest of the network. Thus this attack is called a **51% attack**[22]. A "51%" attack is still possible for an attacker that controls less than half the network hash rate. But in this case the probability of success depends on what percentage of the network hash rate the attacker controls and the number of blocks she has to outpace. The probability of success decreases exponentially on both. Only when the attacker controls >50% of the network hash rate is the probability of success 100%.

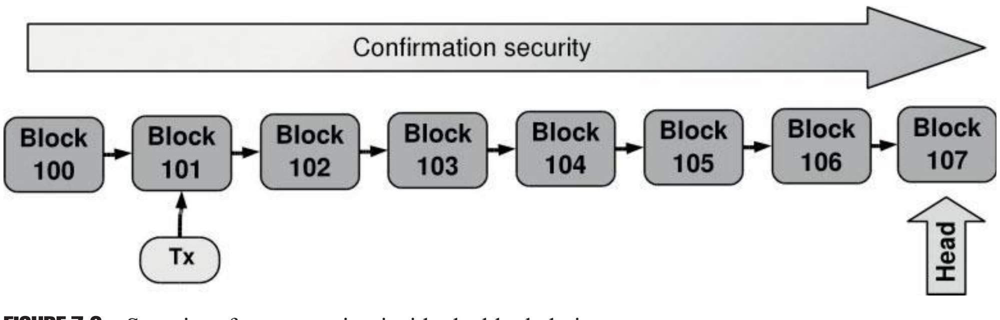


FIGURE 7.9 Security of a transaction inside the blockchain

[22] To be precise, it should be called a 50%+ attack.

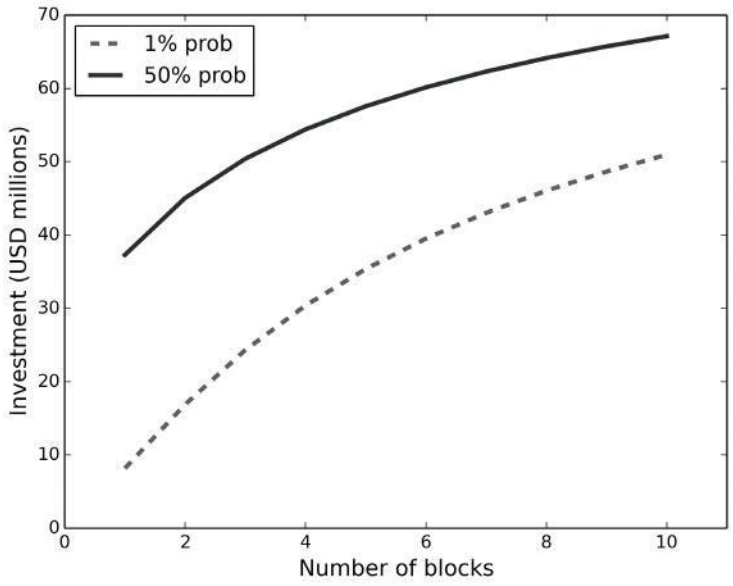


FIGURE 7.10 Investment required to pull off a 51% attack with 1% and 50% probability of success

Figure 7.10 shows an analysis of the investment an attacker should devote to be able to perform a double-spending attack. The analysis assumes that the network hash rate is 30,000 TH/s and that the cost of mining equipment is 3 USD per GH/s. An attacker needs to invest 90 USD million[23] to have as much hash rate as the rest of the network and to be able to perform a 51% attack. Figure 7.10 shows the amount an attacker should invest in new mining equipment to succeed with a probability of 1% and 50%. Note that with only 1 block confirming a transaction, an attacker should invest more than 8 USD million to have a 1% of success, or invest more than 37 USD million to succeed with a 50% probability. This analysis can be used to assess the risk a user is willing to accept when choosing the number of confirmations. For most purposes a few confirmations should be sufficient.

A 51% attacker can change transactions that she has control over, performing a double-spend on those funds. Although a 51% attacker could remove transactions from other users from the blockchain, she cannot alter the transactions, because these are protected by the ECC signature. That is, a 51% attacker cannot modify the amount or the recipient of arbitrary transactions in the blockchain. However, a 51% attacker can effectively prevent other miners from mining new blocks, by refusing to mine on top of the blocks produced by these other miners. Thus, a 51% attacker can successfully perform an attack *against other miners*.

Researchers have shown in Sompolinsky and Zohar (2013) that it is possible to pull off a successful 51% attack controlling significantly less than 50% of the network hash rate if there are delays in the propagation of mined blocks. They propose a modification

[23] The cumulative investment in mining equipment has been greater than this figure, because much of it was spent on less efficient technology (Chapter 9).

to the Bitcoin protocol, called GHOST, to make a 51% attack harder in the presence of block propagation delays. This proposal allows high block creation rates and large block sizes without increasing the risk of an attack.

The **GHOST (Greedy Heaviest-Observed Sub-Tree)** policy changes how the main chain is selected in the Bitcoin protocol. The GHOST policy takes into account the proof-of-work performed in orphaned blocks when selecting the longest chain. Starting from the genesis block, the main chain is selected by computing the total proof-of-work of every sub-tree in a fork (a sub-tree includes orphaned blocks). The sub-tree with the highest total proof-of-work is selected.

The 51% attack is the most cited attack on the blockchain. However, there are other types of attack, which are explored in the rest of this section.

7.5.1 Race Attack

When a node receives a new unconfirmed transaction that conflicts with a transaction in its unconfirmed transactions cache, i.e. it spends the same outputs, the new transaction is ignored. Thus nodes only keep in their cache a copy of the first transaction received[24]. A **race attack** occurs when a vendor accepts payment on an unconfirmed transaction by checking only a few nodes. The attacker could send a transaction to the nodes close to the vendor and a different transaction to many other nodes in the network. Thus only the nodes closer to the vendor show the transaction sending the funds to the vendor, while the rest of the network include in their mempool the double-spend. To defend against this attack a vendor should wait for the transaction to be included in at least one block.

A double-spend of a transaction is very unlikely if the majority of the mining nodes in the network already have a valid transaction in their unconfirmed transactions' memory pool. A payment processor or wallet service can take advantage of this fact by keeping connections open to a large portion of the mining nodes and checking that the correct transaction is in their unconfirmed transactions' memory pool. This allows the service operator to offer very fast transaction confirmation with limited risk.

In an ideal situation, all pending transactions in the network would be included in the next block mined. This assumes that all transactions propagate through the network with no delays, reach all miners and that miners include all pending transactions in their next block. In practice some transactions can suffer delays before being included in a block, and can even be dropped by the network and never make it to the blockchain[25]. It can even be possible for a transaction that was forgotten by the network to reappear after a node starts to broadcast it again. In cases where a transaction is not confirmed, a **double-spend** of that transaction to a different address should be forced. This has the advantage of preventing or at least making apparent problems with transaction malleability (7.6.1).

[24] Saving all unconfirmed transactions—even conflicting ones—carries several problems. First, a mining node would have to decide which of the conflicting transaction it should include in the block it is mining. Second, this could be easily abused in a DoS attack, where the attacker floods the network with double-spends of the same transaction.

[25] Nodes could drop transactions from the unconfirmed transactions' memory pool after a certain period, or after a restart of the node. However, most wallet implementations keep on resending a transaction periodically, i.e. after 1 hour, if it is not confirmed.

7.5.2 Finney Attack

This attack was discovered by Hal Finney (Bitcoin wiki, 2014h). The attacker (who is also a miner) secretly mines a block, including in it a transaction from one of her addresses to another of her addresses. This transaction has not been broadcast to the network, and is only included in the block that the attacker is secretly mining. Right before releasing the block, the attacker sends a double-spend of the TxOut included in her secret transaction. This attack would succeed in bypassing the protection of a user who just monitors that the transaction has propagated through the network. Once the victim has accepted the payment, the attacker would release the secretly mined block, thus performing a double-spend.

This attack has an associated opportunity cost because there is a delay from the time the block is mined to the time the block is released to the network. This delay is consumed in closing the deal with the victim and waiting for the double-spend transaction to propagate through the network. During this delay, there is the possibility that some other miner finds a block, and thus the double-spend attack fails and the attacker loses the block reward. This opportunity cost makes the attack impractical for low value transactions. For high value transactions, a defense against this attack is to require at least two confirmations.

In contrast to the 51% attack, an attacker does not have to control most of the hash rate in the network. A miner with a low hash rate could perform this attack, just waiting for the time when she mines the next block.

7.5.3 Transaction Spamming

An attacker could try to perform a denial of service on the Bitcoin network by creating many transactions where she sends the funds to herself. As space in the blocks is limited, this attack could prevent legitimate transactions to confirm. However, this attack is unlikely to work in practice because of the following factors:

- The number of “free” transactions an attacker can send to the network is limited. Blocks mined have a maximum block size of 50kB reserved for priority (free) transactions[26]. Transactions above this limit have to pay fees. So, at the end, an attacker would have to pay fees to perform this attack.
- The fees an attacker would have to pay would have to compete with the fees of legitimate transactions. As time passes the fees that legitimate transactions are willing to pay to be included in the blockchain will presumably rise, making this attack costlier to sustain over time.
- The transactions an attacker has to send have to be above a certain threshold, currently set at 0.0000054 bitcoins. Below this threshold a transaction is considered **dust** and dropped by the network. Therefore an attacker would have to possess some funds in Bitcoin to be able to perform the attack.

[26] More precisely, the limit for priority transactions in a block is defined in the static variable **DEFAULT_BLOCK_PRIORITY_SIZE**, currently set to 50,000 bytes. This definition can be found in `main.h`. Furthermore, priority transactions are themselves ordered based on the age of their outputs and their size, see function CCoinsViewCache::GetPriority in `coins.cpp`.

A variant of this attack would be to send invalid transactions to some nodes. An invalid transaction is checked by the node that receives it and dropped. Checking the signature is a CPU-intensive operation (section 7.7) so this attack could succeed in slowing the nodes that are flooded with transactions. However, as the flooded nodes do not relay the invalid transaction to the network, this attack does not have an effect in the rest of the network. Nodes in the bitcoin network have to protect against this DoS attack in the same way regular internet servers have to protect against DoS attacks.

7.6 MERKLE TREES

Up to this point, block hashes have been assumed to be the hash of the whole block, which includes the hash of the previous block in the blockchain, and all the transactions in the block. But nothing has been said about how this hash is constructed. The naive solution, to concatenate the byte representation of the block header and all the transactions and then hash the whole byte string, has several disadvantages:

- If one transaction is changed, the byte string has to be updated, and the hash computed again. This forces nodes to keep the whole block byte string in memory. If a transaction in the middle of the byte string is substituted for a bigger transaction, memory has to be allocated in the middle of the byte string, a costly operation.
- To verify that a transaction belongs to a block, the whole block must be available. Only then can the hash be computed and verified.

A **Merkle tree**, or **hash tree**, is a data structure that addresses these issues. It was proposed by Ralph Merkle in 1982. According to Nakamoto (2008a), Satoshi took the idea of using Merkle trees to time-stamp a block of transactions from Massias et al. (1999). Merkle trees are also widely used in file sharing applications, to keep track of the blocks of a file that have been correctly downloaded.

Figure 7.11 shows how a Merkle tree is used to compute the hash of a block[27]. First, a **binary tree** is formed with the hashes of the individual transactions as their leaves. These are denoted by Hash0, … Hash3 in the figure. A binary tree is a directed graph, where each parent has two children. The hash of the parent node is the hash of its two children, themselves hashes[28]. This is shown in Figure 7.11, where Hash01 = Hash(Hash0 || Hash1) and Hash is the SHA256 hash function. Eventually the hash of the root node, the **root hash** or **Merkle root**, is computed. With this Merkle root, the block header can now be assembled.

The **block header** contains the hash of the previous block in the blockchain, the root of the Merkle tree of the transactions and the nonce included by the miner. The hash of

[27] The implementation of the Merkle tree in the Bitcoin source code can be found in function CBlock::BuildMerkleTree in `core.cpp`.

[28] If a node in the tree does not have a matching node, it is hashed with itself to arrive at the hash of its parent.

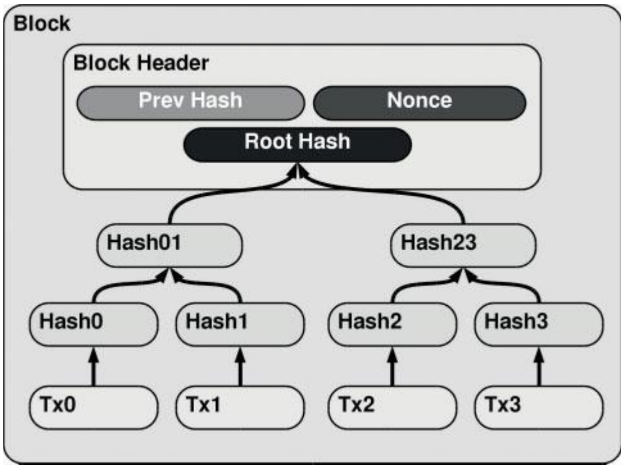


FIGURE 7.11 Merkle tree of the transactions in a block

the block is then the hash of the block header only: transactions are represented in this hash through the root of the Merkle tree[29].

One of the big advantages of using Merkle trees is transaction verification. Suppose a node wants to verify that a transaction, say Tx3, belongs to a block as in Figure 7.12. A node can perform this operation within a time that is related logarithmically to the number of nodes in the tree. Following Figure 7.12, the node only has to compute Hash3, Hash23 and the root hash, and check the result against the root hash stored in the block.

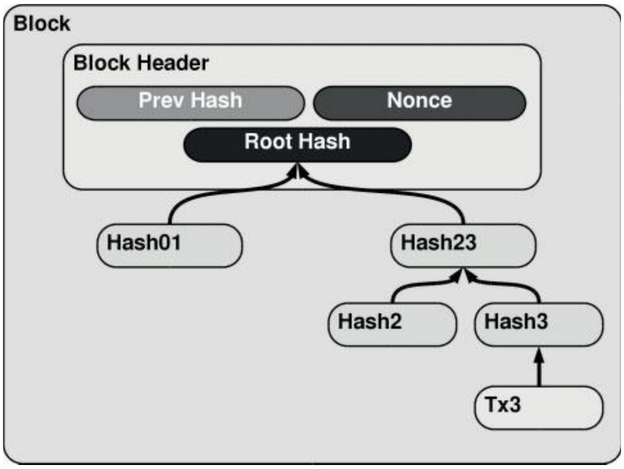


FIGURE 7.12 Pruning of transactions in a block

[29] The hash of a block is computed in the function **CBlockHeader::GetHash** in `core.cpp`. To be precise it includes: nVersion (version of the block; current version is 2), hashPrevBlock (hash of previous block), hashMerkleRoot (root hash of the Merkle root), nTime (time when the block is mined), nBits (block difficulty in compressed form; see box in section 7.4) and nNonce (nonce).

That is, the node only has to verify the **Merkle branch**, the part of the tree that allows to cryptographically prove that a transaction is included in the tree. The Simplified Payment Verification (SPV) protocol (section 8.8) makes extensive use of the fact that transactions are represented in Merkle trees.

7.6.1 Transaction Malleability

Transaction malleability is a "quirk" of the Bitcoin protocol, known since at least 2011 (Bitcoin wiki, 2014aa). Transactions in the blockchain are referred by their hash, whose value is included in the Merkle tree for the block. On the other hand, transactions are signed with the private key associated with the transaction inputs.

Transaction malleability refers to the fact that the subsets of transaction information that are used for the hash and for the digital signature are not equal. In other words, a transaction could be changed in such a way that its hash changes, but the digital signature is still valid.

The transaction is hashed before signing, as presented in section 6.5. Thus the transaction is hashed twice: once to sign it and another time to include it in the Merkle tree. The hash before signing does not include the <*scriptSig*> but in its place the <*scriptPubKey*> is copied. Thus a change in the <*scriptSig*> would not invalidate the digital signature, but would change the hash of the transaction that appears in the Merkle tree.

For a list of sources of malleability as well as proposed solutions, see Wuille (2014). The following is a non-exhaustive list of sources of malleability:

- Pushing extra data to the stack at the beginning of <*scriptSig*> that does not affect the evaluation of the script.
- Adding a sequence of non-push operations to <*scriptSig*> that do not yield any result, like appending an operation that pushes data to the stack followed by OP_DROP.
- Prepending one or more zeros to the ECC signature does not invalidate the signature[30].
- If s is the ECC signature to a message, then $-s$ is also a valid signature (section 5.4). Thus changing the sign of the ECC signature in <*scriptSig*> does not invalidate the signature but changes the transaction hash.

The upshot of transaction malleability is that if a user is tracking transactions based exclusively on their hash, the transaction could be changed so that the hash changed, but the transaction was still valid and thus could be included in a block. The user would not notice that the transaction had been confirmed, and would not adequately update her records.

Transaction malleability climbed to fame in February 2014 when a large exchange discovered that this attack had been used against it[31]. Attacker(s) allegedly filled a Bitcoin

[30] Bitcoin uses OpenSSL for ECC signature verification. OpenSSL uses DER-encoded data, but it does not enforce it. So the ECC signature can be slightly changed, such as prepending zeros, and OpenSSL would still accept it. This was fixed in version 0.8 of Bitcoin Core, by rejecting transactions with invalid DER-encoded signatures.

[31] *The Economist* ran a piece in its print edition (*The Economist*, 2014a).

refund order with the exchange. As confirmation for the order, the exchange sent them the hash of the refund transaction after sending it to the network. The attacker(s) waited for this transaction to appear in the network, quickly changed it and flooded the network with the modified version. Once the modified transaction had been included in a block, the attacker(s) allegedly complained to the exchange that the order with the associated hash had not been confirmed, and the exchange would issue a new refund order. The second refund order would have to spend different funds than the first one, otherwise it would be an invalid transaction. This sequence was allegedly repeated many times. Basic accounting checks would have quickly spotted the leak of funds, thus raising the question of whether there were not such checks in place or whether the drain of funds was due to other factors (Decker and Wattenhofer, 2014).

Exchange software should force a double-spend on transactions sent to the network that have not been confirmed. This would have helped detect the double-spend problem after getting feedback that the forced double-spend transactions were not being confirmed.

7.7 SCALABILITY

With increased usage the size of the blockchain has been growing steadily. Figure 7.13 shows the size of the blockchain on a logarithmic scale. As of the time of writing the size stands at roughly 17 GB.

One of the criticisms often raised against Bitcoin is that it will not scale to handle a rate of transactions comparable to that of more mature payment processing networks. As of the time of writing, blocks include an average of 300 transactions. Thus the network currently handles on average 1 transaction every 2 seconds or 0.5 transactions/second (tps). This is two to three orders of magnitude lower than the average transaction rates of established online payment systems, which range from 50 tps to 2,000 tps (Bitcoin wiki,

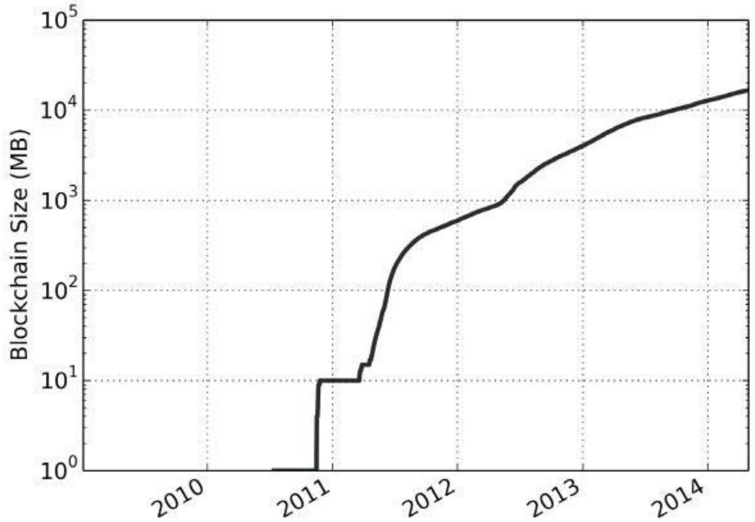


FIGURE 7.13 Blockchain size (MB) on a logarithmic scale. Data from blockchain.info

2014u). This section explores the bottlenecks that could prevent Bitcoin scaling to match the rates of established payment processing networks. This section is based on Bitcoin wiki (2014u).

There is a limit to the maximum block size hard-coded into Bitcoin Core. This limit is currently set at 1 MB[32]. This limit could be increased in future versions of Bitcoin Core if and when the tps of the Bitcoin network increases.

This section will present the limits imposed by current technology. The bottlenecks can be divided in three groups:

- **Computational power**. Nodes perform two main computing tasks: checking block hashes and checking transaction signatures[33]. Of these two tasks, signature checking consumes considerably more CPU. Transactions currently include an average of two inputs. This average is set to increase as multisignature and P2SH transactions increase in importance (Chapter 6). A current single core is able to process 8,000 ECDSA signatures/second[34]. Therefore a single core machine could process 4,000 tps. As signature checking is a highly parallelizable task, adding more cores can further increase the tps a machine could handle. Therefore CPU load does not seem to present a bottleneck on the increased utilization of Bitcoin.
- **Network**. Transactions currently average 0.5 kB in size. Again, this average will probably increase in the future with the increased importance of multisignature transactions. Residential ADSL connections have a bandwidth of 10 Mbits/s on the low end, and this bandwidth could accommodate a flow of 2,500 tps[35]. Higher bandwidths are readily available for network servers. Therefore network bandwidth does not seem to present a bottleneck on increase utilization either.
- **Storage**. As transactions take on average 0.5 kB of storage, storage utilization could quickly escalate for high transaction rates. For instance, a rate of 2,000 tps generates around 1 MB of data every second, or 30 TB of data every year. Furthermore, with the increased utilization comes an increase in the number of unspent transaction outputs, thus increasing the RAM memory requirements for the UTXO, probably making impossible for the UTXO to fit in RAM.

In summary, storage requirements are the biggest bottleneck for scalability. The problem of blockchain size has been amply debated in the community, and many ideas have been proposed, such as:

- **Ultimate blockchain compression**. For a going concern, the only information useful in the blockchain are the unspent transaction outputs. The rest of the data in the blockchain is useful for verification purposes, i.e. when a node bootstraps a new copy of the blockchain. It was proposed in Reiner (2012) to create an alternative

[32] More precisely 1,000,000 bytes, see static variable **MAX_BLOCK_SIZE** in `main.h`.

[33] Note that block mining difficulty is independent of the number of transactions: the difficulty of the partial hash inversion problem only depends on the total network hash rate.

[34] Optimizations have been proposed that increase the rate of ECDSA signature verification tenfold (Bitcoin wiki, 2014u).

[35] Denial of Service attacks have to be accounted for, which requires higher bandwidth requirements. In this respect, a Bitcoin node is not much different from a regular internet server.

blockchain to store a copy of the unspent transaction outputs cache (UTXO). The UTXO would be stored in a tree or similar data structure[36], sometimes called an **unused output tree** (UOT). This new blockchain would then be merge-mined with the regular Bitcoin blockchain. For details about how merged mining works see section 14.3, but the basic idea of merged mining is that the hash at the root of the UTXO tree is included in each regular block, thus securing the new blockchain using Bitcoin's computational power. Most of the nodes—even mining, wallet, or payment processing nodes—would only need to keep a copy of the UTXO blockchain. A full copy of the blockchain would then only need to be kept in **archival nodes**.

- **Finite blockchain**. A divide and conquer strategy was proposed in Bruce (2013). The blockchain would be divided into three pieces: an account tree, a mini-blockchain, and a proof chain. Although this proposal may be too radical to implement in Bitcoin in the short run, it could be picked up by other alternative cryptocurrencies.

For the time being, Bitcoin is not breaking at the seams. With the current value of the maximum block size set at roughly 1MB there is ample room to collect all the new transactions in the network: this block size could accommodate 3 to 4 tps. As the number of transactions increases, this value could be increased accordingly[37].

If space in the blockchain starts to become scarce transaction fees will increase. This would, on the one hand, increase investment in mining and, on the other hand, push the settlement of small transactions off the blockchain, thus decreasing the load of the system. Market forces, human ingenuity, and advances in technology will probably solve the scalability problems faced by Bitcoin.

[36] Different data structures were proposed: from Merkle trees, to binary-search trees, to a tree-like structure (Reiner, 2012).

[37] The maximum block size is represented by the **MAX_BLOCK_SIZE** variable. It was increased once from 250,000 bytes to the current 1,000,000 bytes. This required careful coordination of the miners to avoid a hard fork of the blockchain.

CHAPTER 8

Wallets

As explained in Chapters 6 and 7, bitcoins do not reside in a user's computers, rather bitcoins are an entry into a distributed ledger, the blockchain. This ledger holds the amount of funds available for each address. The private key associated with an address must be used to sign a transaction spending the funds from that address. A Bitcoin wallet is simply a collection of private keys. Although a Bitcoin wallet derives its name from an analogy with physical wallets, it is important to note the differences between the two:

- A physical wallet holds the physical money. Therefore it cannot be copied. In contrast, **a Bitcoin wallet can be copied**. Whoever controls a copy of the Bitcoin wallet can spend the funds. An attacker can "steal" a Bitcoin wallet by making a copy of it. In Bitcoin, ownership of funds is determined by spendability. Anyone in possession of the private key can spend the funds.
- **A Bitcoin wallet can be distributed across several devices**, in such a way that accessing the funds might require cooperation between the devices. This can be achieved with multisignature transaction outputs where the private keys of all the required signatures are distributed across several devices. The flexibility provided by the Bitcoin protocol allows for more types of wallets than physical wallets.
- **Receive-only Bitcoin wallets** are possible. These wallets only hold a copy of the public keys or the Bitcoin addresses. Receive-only wallets can receive funds but cannot spend funds from their addresses. Thus a compromised device that holds this wallet does not lead to a loss of the funds. Receive-only wallets (sometimes called watch-only wallets) are useful for situations where the user of the wallet is not trusted, or when she is at risk of theft. They can also be useful to monitor the funds and transactions in a wallet.

A **wallet software** is a piece of software that allows spending funds from the addresses in a wallet and to manage the wallet. It is very important for a Bitcoin user to have some knowledge of how a wallet software works, and so this is arguably the most important practical chapter in the book. The tasks performed by the wallet software usually include:

- Query the blockchain and present to the user a total of the funds available.
- Generate new addresses to receive new funds, or to receive the change coming from transactions performed by the wallet software.
- Interface with the user, for instance generating or reading QR codes representing addresses, transactions, etc.

- Allow the user to send funds to addresses of her choosing. To accomplish this, the wallet must gather addresses with enough funds under its control, assemble a transaction spending the funds from these addresses, create a new address or select an existing one to send back the change, sign the transaction with all the private keys, and publish the transaction to the blockchain.
- Track transactions' confirmation status.
- Make a backup of the wallet.
- Restore a backup of the wallet.

A Bitcoin address is like a bank account. The Bitcoin equivalent of opening an account is downloading wallet software (or signing up for a web wallet) and letting the software generate a private-public key pair. This key pair must be difficult to guess. For instance, if the wallet software were to choose $d = 1$ for the private key, then the public key would be just $B = d \cdot A = 1 \cdot A = A$ (section 5.4). This public key $B = A$ would then be converted to an address and used in the blockchain. Reversing this address to retrieve the private key is easy for an attacker. An attacker could brute-force "easy" private keys, such as 1, 2, 3, …, checking at every iteration whether the generated address matches an address in the unspent transaction outputs cache (UTXO cache). If a match is found, the attacker could use the private key to steal the funds from that address. There is evidence that there are attackers actively performing this attack on weak private keys (section 8.4).

To secure against this attack, the wallet software must generate private keys with a high degree of entropy. One way to achieve this is to use a good (pseudo)random number generator. This is the approach followed by many wallet software implementations, including the Bitcoin Core Wallet. The wallet software keeps a copy of the private keys, but only the addresses are usually shown to the user. Wallets generally allow the user to obtain a copy of the private keys, such as when the user requests a backup.

As new private keys are generated by the wallet software, the only place where this critical piece of information is saved is in the storage of the device running the wallet software. It is thus very important to **back up a wallet regularly and keep several copies of the backup in different safe locations**.

In an **address tampering** attack, a user is misled into sending funds to an incorrect address. An address tampering attack takes places when a user is about to send funds to an address and an attacker intercepts the legitimate address and substitutes it by an address under her control. An attacker that achieves man-in-the-middle status (section 5.1) could easily change the receiving address in a payment. Original wallets did not protect against the address tampering attack. This changed with the introduction of the "Payment Protocol" (BIP 70), that uses the established Public Key Infrastructure (PKI) to authenticate payment addresses (section 8.9). Aside from the man-in-the-middle technique, there are other ways to perform an address tampering attack, for instance superimposing a sticker with the QR code for the attacker's address in the menu of a restaurant that accepts Bitcoin payments. Bitcoin users should be aware of these types of attack and take the necessary steps to protect themselves.

This chapter will explain the technology behind wallet software, the different types of wallets and their security/usability trade-offs.

8.1 SYMMETRIC-KEY CRYPTOGRAPHY

To prevent unauthorized users from accessing the funds in a wallet, the private keys stored on the device are usually encrypted. When the user needs to access them, such as when signing a transaction, she provides the device with the password. The device temporarily decrypts the private keys, uses them to sign the transaction and wipes clean the memory where the unencrypted keys were stored. This process is shown in Figure 8.1.

Wallet encryption also works as a first line of defense in case the device is compromised and an attacker seizes a copy of the wallet. The strength of this protection is only as good as the chosen password. Given the weakness of most human-generated passwords and the recent advances in password cracking technology, a leaked password-protected wallet is at risk of being fleeced[1]. Thus it is advisable to keep a wallet only with limited funds online while storing most of the funds in offline wallets. Several technologies are available to keep funds offline (section 8.2).

To encrypt a wallet's private keys, symmetric-key cryptography is used. Symmetric-key cryptography was briefly introduced at the beginning of Chapter 5. The goal of a symmetric-key cipher is to "entangle" the original clear-text with the provided key in such a way that it is impossible to disentangle it without this key. Symmetric-key cryptography uses the same key for encryption and decryption. There are two types of symmetric ciphers:

- **Stream ciphers** which encrypt a stream of plain-text, one bit at a time.
- **Block ciphers** which encrypt blocks of plain-text, usually of 128 bits length.

Bitcoin Core Wallet, the reference implementation of the Bitcoin wallet, uses the AES-256 block cipher[2]. **AES** stands for Advanced Encryption Standard and was established

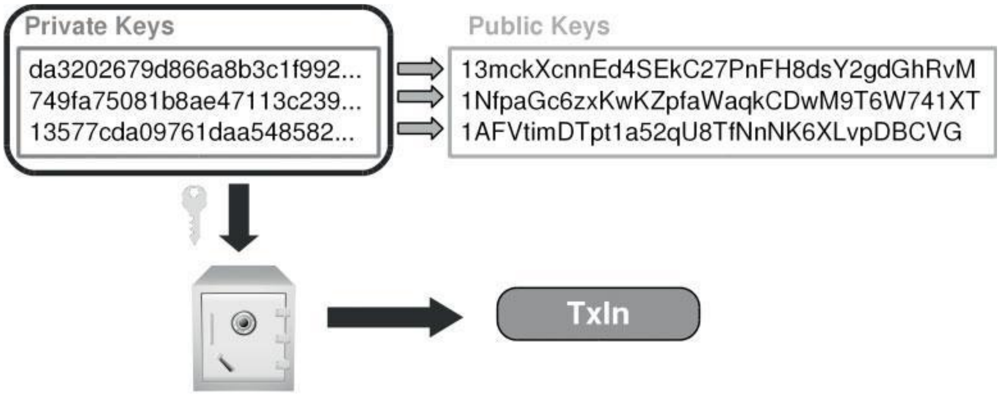

Private Keys
da3202679d866a8b3c1f992...
749fa75081b8ae47113c239...
13577cda09761daa548582...
Public Keys
13mckXcnnEd4SEkC27PnFH8dsY2gdGhRvM
1NfpaGc6zxKwKZpfaWaqkCDwM9T6W741XT
1AFVtimDTpt1a52qU8TfNnNK6XLvpDBCVG
TxIn

FIGURE 8.1 Encryption of wallet private keys

[1] To slow brute-forcing attempts, Bitcoin Core Wallet derives the symmetric-cipher key from the user password using a salt and 25,000 rounds of SHA512—see variable **nDeriveIterations** from class **CMasterKey** in `crypter.h` and function **CCrypter::SetKeyFromPassphrase** in `crypter.cpp`.

[2] Wallet encryption is implemented in the class **CCryptoKeyStore** in `crypter.h`.

by NIST in 2001, substituting DES (Wikipedia, 2014a). On a very high level, AES-256 performs 14 rounds of the following tasks:

- Key addition. This task mixes the symmetric key (actually a subkey derived from the symmetric key) with the message being encrypted.
- Byte substitution. Performs a non-linear transformation of the message being encrypted, which introduces *confusion* into the ciphertext.
- *Diffusion*. Shifts the bits of the ciphertext around, in order to spread the entropy added by the confusion task.

More details on AES can be found in any standard cryptography textbook, such as Paar and Pelzl (2010) and Wikipedia (2014a).

Bitcoin Core Wallet generates a new random change address for every transaction. This behavior is recommended to increase user privacy (Chapter 13). To guard against a loss of keys, a user should back up the wallet after every transaction. To make the backup process easier, Bitcoin Core Wallet generates a **key pool** of 100 new unused addresses in advance[3]. When the wallet software needs a new address, it taps the key pool following a first-in-first-out (FIFO) approach. The key pool is refilled every time a new key is used[4], so that the key pool always has 100 new addresses. Every time a backup is made, it includes not only the addresses holding funds, but also the 100 addresses from this keypool. Thus a backup is current for a while, until 100 new addresses are used. This does not remove the necessity to make frequent backups, but it alleviates it somewhat.

A Bitcoin Core wallet is not intended to be shared and used in several computers simultaneously. The fact that the Bitcoin Core wallet generates random addresses can lead to erratic behavior if the wallet is cloned and installed in another computer, because each computer will generate its own new addresses after a period of use.

Section 8.5 will introduce a technology that allows the generation of as many addresses as desired, only requires a single backup and can be shared among many devices.

8.2 OFFLINE WALLETS

Usually the device that holds a wallet is connected to the internet, in order to communicate with the Bitcoin network (receiving account status, sending transactions, observing confirmations, and so on). This is called an **online wallet** or a **hot wallet**. As every device connected to the internet is at risk of being compromised, it is good practice to only hold in online wallets the funds necessary for day-to-day operations. The rest of a user's funds should be kept in **offline wallets**, whose private keys are not accessible from the internet. Note that an offline wallet can sign transactions offline.

Cold storage refers to a place where the private keys are kept, that is not accessible from the internet. Private keys kept in cold storage have to be imported to a wallet (either an online or offline wallet) before spending the funds.

[3] This key pool default size of 100, can be changed with the command line argument -keypool=<n>.
[4] Every time a key from the key pool is used in **CWallet::ReserveKeyFromKeyPool**, it calls **CWallet::TopUpKeyPool** to fill up the key pool back to its original size (100 by default). Both functions can be found in `wallet.cpp`.

This section explores several options available to create an offline wallet and to handle cold storage.

8.2.1 External Storage Media

One way to create cold storage for the private keys is to save them in external storage media, such as USB flash drives or optical disks. Only when a private key it needed, such as when signing a transaction, is the private key retrieved from the external storage.

Note that storage media can degrade over time and the information contained is sometimes corrupted. Thus it is sensible to make several copies of the private keys.

If the private keys are retrieved by an online wallet, the device with the online wallet could be compromised and the file with the private keys stolen by an attacker. Therefore it is advisable to encrypt the file holding the private keys when saved in external media. Encrypting the files with the private keys would also protect the owner when the physical storage media is stolen.

Note that even if the file with the private keys is encrypted, an attacker that obtains a copy could attempt a brute-force attack. Moreover, if an attacker reaches root privileges on the device she could even get a copy of the encryption key or the clear-text of the private keys.

In summary, when using external storage, the weak link is the period where the private keys are imported to the online client to sign transactions. It is more secure to import the private keys from cold storage to an offline wallet.

8.2.2 Paper Wallets

Another way to create cold storage for the private keys is to print them on a piece of paper and to protect that piece of paper from physical theft. These are called **paper wallets**, although they are not technically a wallet.

In a paper wallet, public keys or Bitcoin addresses are usually printed alongside the private keys, so that the paper wallet can be easily identified without having to import the private key into a client. If the private keys are generated randomly, such as in the Bitcoin Core Wallet, a copy of each private key should be printed. Private keys are 256 bit (32 byte) integers, and can be represented in several ways:

- In **hexadecimal format**, for example "2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824". This format does not have any error correction and is thus susceptible to transcription errors. Variants of this format include decimal representation, Base64 encoding, etc.
- Base58 **Wallet Import Format (WIF)**. This format uses the Base58 encoding of Bitcoin addresses with error correction. WIF keys start with a "5" and have 51 characters, for example "5Kb8kLf9zgWQnogidDA76MzPL6TsZZY36hWXMssSzNydYXYB9KF". This is the most widely used format to export private keys. For details on the encoding and decoding of WIF keys, see Bitcoin wiki (2014ab).
- **Mini private key format**, encodes a private key in 30 characters, including some address verification to guard against typos. Mini private keys start with the letter "S", such as in "S6c56bnXQiBjk9mqSYE7ykVQ7NzrRy". This format is used for

applications where space is critical, such as inside physical Bitcoin coins or in QR codes where low dot density is needed (Bitcoin wiki, 2014n).

- **Passphrase-protected private keys** were introduced in BIP38 (Caldwell, 2013). BIP38 keys are protected by a password, encrypted with AES-256, and the key derivation is performed using the scrypt hash function (Percival, 2012). They start with "6P", such as in "6PRVWUbkzzsbcVac2qwfssoUJAN1Xhrg6bNk8J7Nzm2ZoGg". As of the time of writing, this BIP is under discussion.

All these formats can be represented in QR codes, making the process of importing easier and less prone to typing errors. Figure 8.2 shows a paper wallet generated by bitaddress.org. The left of the image shows the address together with its QR code representation. The right side shows the private key (WIF encoded) and its associated QR code. Note the higher information density of the QR code of the private key—37 bytes—compared to that of the address—25 bytes—both including checksums.

Paper wallets can be a very secure way of storing Bitcoins if good security practices are followed when generating them. It is recommended that paper wallets are generated using offline devices, i.e. a computer connected to a printer, with both of them offline. The devices should be rebooted before and after generating the paper wallet to avoid leaving sensitive information in the devices' memory. It is advisable that the device generating the paper wallet is booted from a live CD (such as the Live CD provided by some Linux distributions) to avoid spyware stealing the private keys[5]. For a full list of these security practices, see Bitcoin wiki (2014o).

Once generated, the owner should avoid exposing the private keys, be it the WIF string or the QR code, to anyone[6].

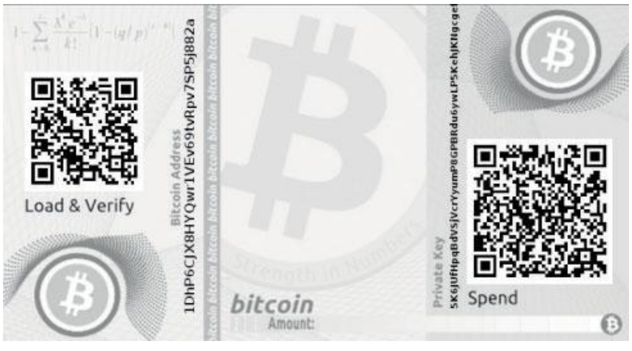

Load & Verify
Bitcoin Address
1DhP6CJX8HYQwr1VEv69tvRpv7SP5J882a
bitcoin
Amount:
Private Key
5K6JUfHpqBdVSJVcrYyumP8GPBRdu6ywLP5KehJKNgcge
Spend

FIGURE 8.2 Paper wallet generated using bitaddress.org

[5] This does not provide definitive security. An attacker could compromise the ISO of the Live CD on the distribution website, so that when the device is booted using the compromised Live CD, it could mount the bootable drive of the device and compromise it. Then it could capture the private keys that are being generated and save them in the device's main boot drive. Then when the user boots the device from the regular boot drive, the malware would be able to "phone home" with the private keys. Thus it is not totally impossible for malware to steal the private keys when booting from a Live CD, but it considerably raises the difficulty for the attacker.

[6] On December 2013 during a TV show, a host exposed the QR code of the private key of a paper wallet to the cameras. The funds were immediately taken and later returned to his owner.

As was the case for external storage, the private keys from the paper wallet must be imported into the wallet software to be able to sign transactions. If the paper wallet is imported to an online wallet, the keys are at risk of being seized by any malware that the device could be running. Thus paper wallets should only be imported moments before the funds will be used[7].

A **physical bitcoin** is a coin or piece of paper that hides a private key behind a tamper-proof mechanism. The address associated with the private keys holds certain amount of bitcoins. To spend the funds from the physical bitcoin, the private key must be recovered and imported to a wallet, much in the same way that a private key is imported from a paper wallet. The manufacturer of the physical bitcoins must be a trusted party, as the funds in a physical bitcoin are provided by the manufacturer.

8.2.3 Offline Devices

An offline wallet can be created using two devices, one connected to the internet and another offline. The connected device keeps a copy of the public keys or addresses only. When a user wants to create a transaction, the connected device assembles a transaction gathering the funds from the addresses in the wallet it sees fit, just like a regular wallet. However, this transaction cannot be signed yet, because the connected device does not have a copy of the private keys. The transaction must be sent to the offline device for signing, and the signed transaction transferred back to the connected device. Thus there must be communication between the two devices. This communication can take place, through external media, such as a USB flash drive. However, using storage media creates an indirect communication link between the two computers that can be exploited by an attacker[8].

A different approach would be to communicate the unsigned/signed transaction through out-of-band means, such as through scanned QR codes. In this configuration, the unsigned transaction could be encoded in a QR code. This QR code could be scanned by the offline device. Then, after signing, the signed transaction could be encoded in a new QR code and scanned by the online device.

In summary, the online device keeps a balance of the funds and initiates the transactions. Because the offline device is not connected to the internet it cannot know what funds are available in the addresses belonging to the wallet. The offline device is thus just a signing authority. If the online device is compromised, it constitutes a breach of the privacy of the wallet, but not of its security.

[7] There is the possibility that an attacker could have previously compromised the computer and tried to spend the funds in the wallet right before the legitimate owner. Both transactions would be competing to enter the blockchain first and spend the funds. An even safer practice would be to import the paper wallet to an offline device that would communicate with an online device via an out-of-band channel. This last procedure would probably leave the more paranoid security-minded users happier.

[8] This attack is not only theoretical: in 2010 a computer virus named **stuxnet** was discovered, that was allegedly used against Iranian nuclear facilities (Zetter, 2011). The malware spread through, among other means, USB flash drives, eventually reaching its target: an industrial control system computer not connected to the internet.

8.2.4 Hardware Wallets

Hardware wallets[9] are devices that both store private keys and sign transactions with these private keys. The private keys never leave the device, so they cannot be seized by malware installed on the computer of the user.

The hardware wallet communicates with a client wallet software in a computer. This client can be either a wallet program or a web wallet running inside a web browser. In any case, the client only acts as an intermediary between the hardware wallet and the blockchain, merely transmitting the transactions signed inside the hardware wallet. Some wallet software projects include support for hardware wallets.

Transactions are sent from the client wallet on the computer to the hardware wallet through some connection, usually USB. Signed transactions are returned from the hardware wallet through the same connection. Hardware wallets usually have a small screen to show the user information about the transaction, and some buttons to allow the user to decide whether to sign the transaction or decline it. If malware is installed in the user's computer, it could change the details of the transaction being sent over to the hardware wallet. The screen in the hardware wallet showing details of the transaction about to be signed is meant as a protection against this kind of attack. It is also common for the hardware wallet to require a PIN or a password to accept transactions coming through the connection.

Figure 8.3 shows a Trezor™ hardware wallet in operation, with an address and a bitcoin amount, prompting the user to either confirm or deny the transaction. The hardware wallet is connected to a computer through a USB port, and prompts the user for a PIN via the computer's wallet software.

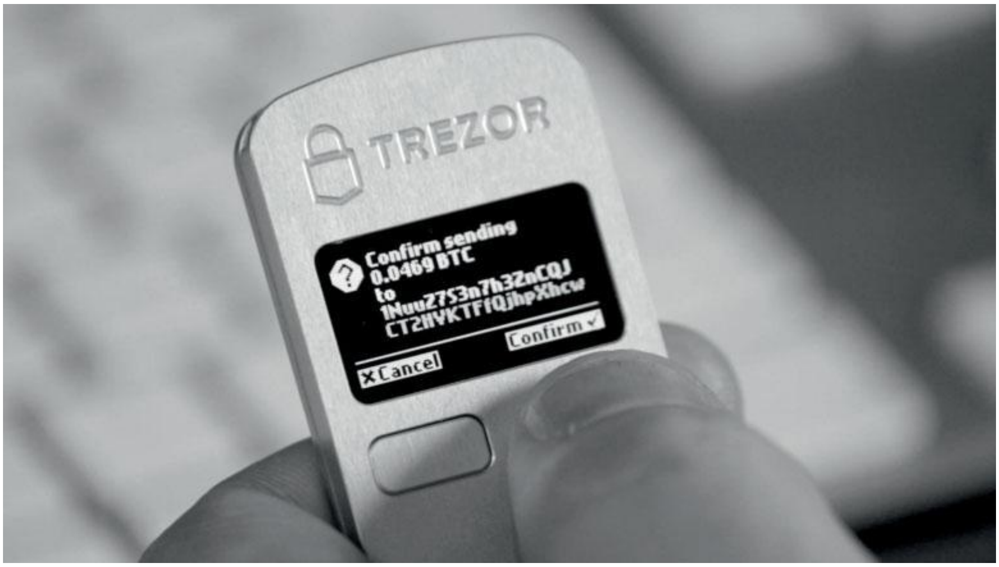


FIGURE 8.3 Trezor™ hardware wallet. Picture by SatoshiLabs (www.bitcointrezor.com)

[9] This discussion will be based on the Trezor™ hardware wallet, see www.bitcointrezor.com. For a list of other hardware wallets, see Bitcoin wiki (2014j).

To protect against loss, theft, or malfunction of the hardware wallet, the user must keep a copy of the private key on some other medium, such as paper or external storage. Hardware wallets usually allow the user to make a copy of the private key during initialization of the device. In particular, Trezor™ uses deterministic wallet technology (section 8.5) to generate many addresses from a single master private key, and therefore only a backup of this master key is necessary.

8.3 WEB WALLETS

Web wallets (also called **hosted wallets** or **cloud wallets**) are online accounts with an external provider, where a user can deposit her funds. The funds are controlled by the web wallet provider. By authenticating with the web wallet provider, the user can later access those funds, i.e. make transactions. The main advantage of web wallets are that they are easy to set up (just signing up to a web service) and that the management of the private keys is done by the web wallet provider, thus lowering the entry barrier for new users. There are additional advantages, such as low commissions[10] for transactions, or the possibility to settle transactions between users of the same service instantaneously and at zero fees. Many of the services offered by web wallet providers are also offered by online exchanges and so-called Bitcoin banks, so the considerations of this section also apply to these businesses.

Web wallets are similar to online banking, in the sense that funds are kept by the web wallet provider. However, in contrast to regular banks, where deposits are covered by deposit insurance, the user has no recourse against the web wallet provider absconding with the funds. Furthermore, web wallet providers are not as thoroughly regulated as banks are, which can increase doubts about their solvency (section 8.3).

Security practices for web wallet providers or exchanges are similar to the practices for individual users: users should store in online wallets only the funds necessary for day-to-day operations, and keep the rest in offline wallets/cold storage. Note that, in addition to the risk of the web wallet provider stealing their users' funds, there is the risk that the provider is hacked and loses the funds, thus making the service insolvent.

The use of web wallets has privacy implications too. On the one hand they can increase the anonymity of the user, because the addresses used in transactions are those of the web wallet with no direct relation to the user. On the other hand, anonymity is decreased because the web wallet provider usually keeps a record of the transactions, and also retains personal information about their users.

Hybrid web wallets are web wallets where the private keys are stored in the user's computer, but managed by software (usually written in Javascript) from the wallet service provider. Transactions initiated by the users are communicated first to the web wallet provider, who later publishes them on the blockchain. The advantage is a reduced exposure to the service provider, but at the cost of increasing the burden to the user of keeping a secure system.

[10] A web wallet service provider can pool many transactions before pushing them onto the blockchain, reducing transaction costs.

8.4 BRAIN WALLETS

Brain wallets generate a private key by hashing a long password or passphrase. Bitcoin private keys are 256 bits long, so a hashing function that yields a 256 bit hash, such as SHA256, can be used. A brain wallet's password does not need to be stored in a device; rather it should be stored in the brain of the user—hence the name. This has the advantage that no backups are needed, provided the user is able to remember the password. As was the case with paper wallets, they are called brain wallets although they do not constitute actual wallets: the password still needs to be imported to an actual wallet to access the funds.

Brain wallets have a big disadvantage that should dissuade anyone from using them: they are subject to brute-force attacks that could steal all the funds in the wallet if successful. A brute-force attack tries many passwords and checks if the address generated from the password exists and has funds. The chances of success of these attacks are very high. First, users are very bad at choosing passwords with a high degree of entropy. Second, password cracking technology has got much better in the last years, thanks to technologies such as dictionary attacks, rainbow tables, GPUs and the fact that several big password databases have been leaked[11]. An account of the recent advances in password cracking can be found in Goodin (2012).

Key stretching is a technique used to slow down password cracking. Key stretching uses a slow hash function, such as SHA256 applied 1,000 times in a row to derive the private keys from the password. There are also deliberate slow hash functions such as bcrypt or scrypt[12], that were specifically developed to curb password crackers—see "Password Hashing" in section 7.1.

Researchers tested sending small amounts to addresses generated using weak passwords: the funds have been quickly stolen. There are attackers actively searching the blockchain for addresses generated using weak passwords to steal the funds. As Bitcoin developers warn (Andresen, 2012): "Do not use a brainwallet! You are likely to lose your coins." In summary, **do not use brain wallets**.

8.5 DETERMINISTIC WALLETS

A **deterministic wallet** is a wallet that can generate many addresses (and their associated private keys) from a common private key. Deterministic wallets have several advantages over regular wallets:

- **Easier backups**. Regular wallets must be backed up every time a new (random) address is created. Some wallets generate addresses in advance, e.g. Bitcoin Core Wallet generates 100 addresses in advance by default. In contrast, a deterministic wallet requires only one backup when the wallet is set up.

[11] As a user posted in Reddit: "Just lost 4 BTC out of a hacked brain wallet. The pass phrase was a line from an obscure poem in Afrikaans. Somebody out there has a really comprehensive dictionary attack program running." (www.reddit.com/r/Bitcoin/comments/1ptuf3/).

[12] For instance some wallets use scrypt or 100,000 rounds of SHA256 for key stretching.

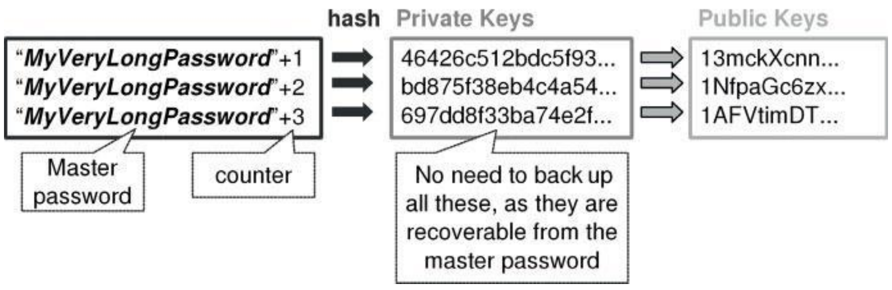


FIGURE 8.4 Type-1 deterministic wallet

- **Smaller backups**. The backup size of regular wallets grows over time, as more and more addresses are added to the wallet. This can be cumbersome when backing up a wallet on paper. The backup of a deterministic wallet has a small constant size, as only the master password needs to be backed up (see below).
- **Generation of new addresses without knowledge of the private key**. This is a feature of type-2 deterministic wallets, see below.

A deterministic wallet can be built from a brain wallet. Figure 8.4 shows how a sequence of addresses can be generated using a **master password** "MyVeryLongPassword" by appending a counter to the password before hashing it to yield the private keys:

$$priv = H(pw|n)$$
$$B = priv \cdot A \bmod p$$

Where pw is the master password, n is the counter, H is a secure hash function, $priv$ is the private key, A is the generator of the elliptic curve, B is the public key from which the address can be generated and p is the order of the prime field under which the elliptic curve operations are performed (section 5.4). This type of wallet is called **type-1 deterministic wallet** (Maxwell, 2011). The addresses generated by a type-1 deterministic wallet are impossible to relate to one another by anyone who does not know the master password.

As with brain wallets, type-1 deterministic wallets are susceptible to brute-force attacks and should not be used unless the master password has been generated with enough entropy.

Type-2 deterministic wallets (Maxwell, 2011) allow the separation of the roles of generating a private key and generating an address. These wallets use the properties of elliptic curves to calculate new public keys without revealing private keys (section 5.4). The main idea is to split the ECC private key into two parts:

$$\begin{aligned} priv &= mpk + H(pw|n) \\ B &= priv \cdot A \bmod p \\ &= mpk \cdot A + H(pw|n) \cdot A \bmod p \\ &= B_{mpk} + H(pw|n) \cdot A \bmod p \end{aligned}$$

Where *mpk* is a **master *private* key** and B_{mpk} is the **master *public* key**. The private key *priv* is composed of two elements: *mpk*, which is kept secret, and $H(pw|n)$, which can be generated from a master password *pw*. Receive-only wallets are given only *pw* and B_{mpk}, so that they can compute addresses, such as *B*. However, only the owner of the master private key, *mpk*, can compute the final private key *priv* for the generated address.

Type-2 deterministic wallets increase security because of the separation of the roles of the master password *pw* and the master private key *mpk*. First, the *mpk* is a regular ECC private key which can be randomly generated and backed up following standard procedures. Second, although the master password *pw* is shared among many users, its compromise does not lead to a compromise of the funds (as with type-1 deterministic wallets), only to a loss of privacy.

An application of type-2 deterministic wallets is an online business with customer facing servers. The business administrator can give the servers a copy of the master public key B_{mpk} and the master password *pw*. Servers can then use them to generate new addresses used to receive payments from customers. As the servers do not have a copy of the private keys for those addresses, the funds cannot be stolen in case of a compromise. All an attacker can do is observe the funds stored in the addresses derived from the master public key.

Wallets that depend upon offline devices to sign transactions usually use the key-splitting feature of elliptic curves discussed. Only the offline device holds the master private key *mpk*, whereas the online wallet client is given only the master public key B_{mpk}. Thus the online client can generate new addresses and assemble transactions, but only the offline device can correctly sign them.

An undesirable property of type-2 deterministic wallets is that receive-only wallets can generate all addresses in the wallet, and can therefore observe all the funds available and all the transactions taking place. A useful property for a wallet would be that a receive-only wallet could only generate a subset of the addresses in the wallet. Hierarchical deterministic wallets, introduced later, have this property.

8.5.1 Message Authentication Code (MAC)

This section introduces Message Authentication Codes (MACs), a cryptographic primitive that will be used in Hierarchical Deterministic Wallets.

A **Message Authentication Code (MAC)** is a piece of data of fixed length that is appended to a message between two parties sharing a *symmetric key* known only to them. A **MAC function** takes two inputs, a message of arbitrary length and the symmetric key, and outputs a MAC. The sender generates a MAC using the symmetric key and appends the MAC to the message. The receiver can then verify that the message originated from the sender (*authenticity*) and that the message has not been tampered with (*integrity*). A MAC function can also be viewed as a hash function with the symmetric key as an extra input.

A MAC serves a similar purpose to a digital signature, but with an important difference: the key used to generate it is a symmetric key. Therefore, a MAC does not provide *non-repudiation*: as the receiver keeps a copy of the symmetric key, she can produce the MAC of any message of her choosing. Therefore, a MAC does not prove that the message was generated by the sender; it could have been generated by the receiver. The advantage of MACs over digital signatures is that they are much faster to compute.

MACs can be constructed from hash functions or from block ciphers. A **Hash-based Message Authentication Code (HMAC)** is a way to construct a MAC using a hash function as a building block. HMACs are provably secure under the assumption that the hash function is collision-resistant (Paar and Pelzl, 2010). Hierarchical deterministic wallets use an HMAC to derive the private keys, using the HMAC-SHA512 algorithm. For more details on HMACs, see Paar and Pelzl (2010), Ferguson et al. (2010), or Katz and Lindell (2007).

8.5.2 Hierarchical Deterministic Wallets

Hierarchical Deterministic wallets (HD wallets) are deterministic wallets whose derived addresses form a hierarchy. As with deterministic wallets, all their addresses can be derived from a starting secret. As was the case with type-2 deterministic wallets, public keys and private keys can be derived separately. The novelty introduced by HD wallets is that the addresses it generates are sorted in a tree structure such that a node has visibility of its descendants but not of its ascendants.

There are two types of nodes in the tree: **private nodes** that hold the private keys to the sub-tree originating from them, and **public nodes** that hold only the public keys to their sub-tree. Aside from the private and public keys, each node has an additional 32-byte field called the **chain code**. The goal of the chain code is to add additional entropy to each node. Thus revealing an address does not automatically reveal the tree derived from that node.

A node is represented by the key (either private or public) and the chain code. Figure 8.5 shows how nodes use an HMAC function to derive branches. There are two types of derivation: **private child derivation** that can be performed only by private nodes and **public child derivation** that can be performed by both private and public nodes. The figure shows how a private node (left) derives a private and a public node, and how a public node (right) derives a public node. The **child number** is a 4-byte integer that determines the branch that will be derived. Values over 0x80000000 denote private child derivation, and values below that number denote public derivation. Thus every private node in the tree can have 2^{31} private descendants and 2^{31} public descendants. Every public node can only

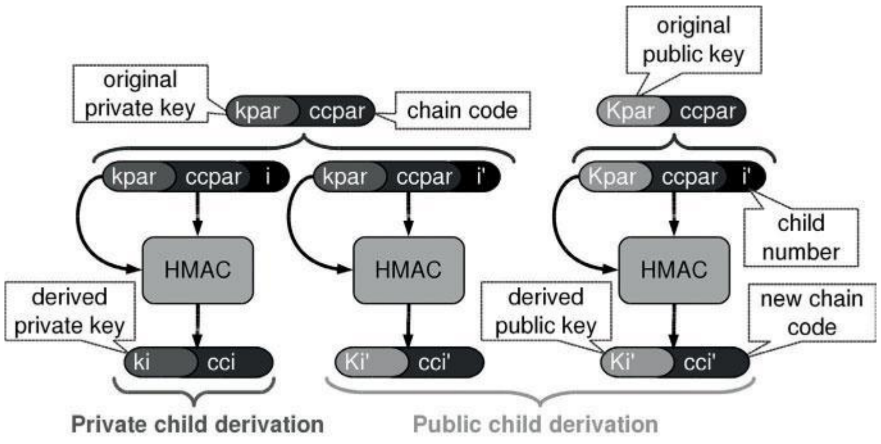


FIGURE 8.5 Hierarchical Deterministic Wallet (BIP 32)

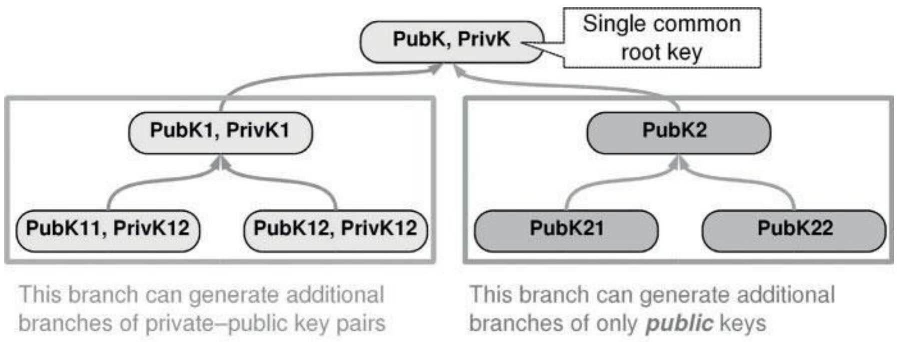

PubK, PrivK
Single common root key
PubK1, PrivK1
PubK11, PrivK12
PubK12, PrivK12
PubK2
PubK21
PubK22
This branch can generate additional branches of private–public key pairs
This branch can generate additional branches of only ***public*** keys

FIGURE 8.6 Tree created by an HD wallet

have 2^{31} public descendants. Their descendants in turn can have their own descendants and so on. Nodes in the tree are represented by their path from the root (Wuille, 2012).

Hierarchical Deterministic wallets allow generating private and public tree branches from a single root key. Figure 8.6 represents two trees, one private and one public derived from a single common master private key. The owner of this master private key controls the funds in the whole tree. Additionally the holders of private nodes have control of the funds in those branches of the tree. Holders of public nodes can only generate public branches, but do not have control over the funds in those branches.

The often-cited application of HD wallets is a company where each business unit controls a private node, while the accountants and auditors receive the root public node. However, as explained in Buterin (2013i), this setup can be attacked if the holder of a branch private key and the holder of the root public key decide to cooperate, for instance if the department head and the auditor decide to run off with the money.

Historically, different wallet implementations have been incompatible: a wallet file generated with one wallet implementation is hard—if not impossible—to import to another wallet implementation. To help create compatibility between different wallets, hierarchical deterministic wallets have been standardized in BIP 32 (Wuille, 2012). This standardization will hopefully increase portability of wallet files. BIP 32 describes how nodes in an HD wallet can be represented by a 111 character Base58 encoded string that starts with “xprv” for private nodes and “xpub” for public nodes. This representation includes information about both the ECC keys and the chain code.

8.6 MULTISIGNATURE WALLETS

One way to increase the security of a wallet is to store the funds in multisignature outputs: outputs that require several signatures to unlock the funds. Multisignature outputs can be achieved either with multisignature transactions (section 6.3) or with P2SH transactions (section 6.6).

There is a push in the Bitcoin community towards wallets that handle multisignature transactions to avoid the single point of failure that the private keys in a wallet represent. A multisignature wallet presents the user the funds unlockable by multiple signatures and how many of the required private keys are part of the wallet. As of the time of writing, multisignature support in most wallets is a work in progress. **Second-generation web wallets** use multisignature transactions (section 8.3).

Multisignature wallets can be combined with hierarchical deterministic wallets in what are called **Hierarchical Deterministic Multisignature wallets (HDM wallets)** (Buterin, 2014d; CryptoCorp, 2014). HDM wallets use several HD wallets, each with its own independent seed, and combine the addresses generated by these independent HD wallets to form multisignature transactions. The advantage of HDM wallets is that many multisignature transactions can be generated from the private master keys, increasing convenience.

8.7 VANITY ADDRESSES

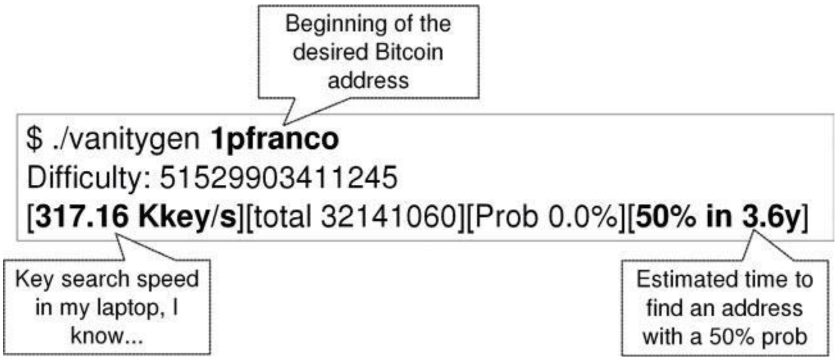
Most Bitcoin addresses are random in nature. But some users, like an online retailer, might wish to have a "personalized" address that starts with a message. For instance the author of this book might wish to have an address that starts with 1pfranco. This type of address is called a vanity address. Vanity addresses can be searched for by trial and error, using any Bitcoin wallet software. The user can ask the wallet to generate an address, inspect the address, i.e. to see if it starts with 1pfranco, and, if it doesn't, request a new address from the Bitcoin wallet software. Of course, this process can be automated. Originally users of Bitcoin who wanted to create a vanity address applied a patch to Bitcoin Core, called the vanity patch, that turned Bitcoin Core into a search engine for vanity addresses. Nowadays most users use a tool called vanitygen,[13] which, aside from being faster, offers several other advantages over Bitcoin Core with the vanity patch applied.

Vanitygen employs a brute-force algorithm. It works by generating a random 256-bit private key, computing the corresponding public key, i.e. the EC point, and then following the procedure of section 5.6 to arrive at the Bitcoin address. If there is a match with the desired prefix specified by the user, vanitygen presents the user with the private key and the address and exits. If there is no match, vanitygen generates a new random private key and follows the procedure again. The private key resulting from running vanitygen can then be imported to a wallet, and used regularly as with any other Bitcoin address.

Figure 8.7 shows vanitygen running in the author's laptop, trying to arrive at a Bitcoin address that starts with 1pfranco. The program shows the difficulty of the problem, i.e. the number of addresses that should be tested on average until the desired vanity address is found. It also shows the rate at which addresses are tested (317.16 Kkeys/s), the progress thus far (32141060 keys tested) and the estimated time to arrive at the desired vanity address with a 50% probability (3.6 years). Finding a vanity address is a computationally intensive problem. Furthermore, it increases exponentially with the number of characters to match. Figure 8.8 shows the output of vanitygen trying to find an address that starts with 1pfran. The estimated time with a 50% probability is 9.4 hours in this case. Being a brute-force algorithm, vanitygen can be used to find a match to a complete Bitcoin address. However, this operation would take an astronomical amount of time, as shown in Figure 8.9.

Vanitygen can run on a CPU or on a GPU. It is advisable to use it on a GPU to greatly speed up the task of finding a vanity address. Vanitygen also allows using regular

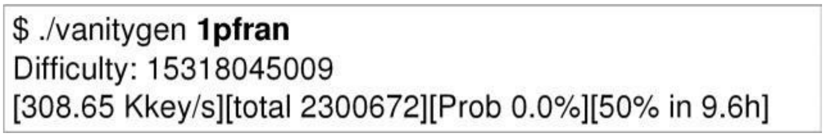
[13] Vanitygen can be found at github.com/samr7/vanitygen.



```
$ ./vanitygen 1pfranco
Difficulty: 51529903411245
[317.16 Kkey/s][total 32141060][Prob 0.0%][50% in 3.6y]
```

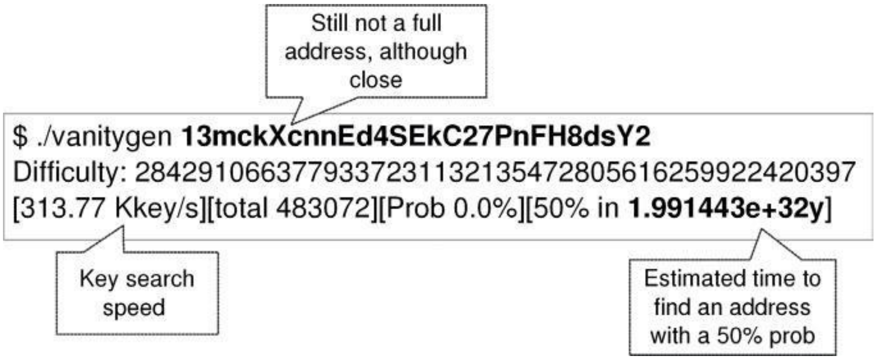
Beginning of the desired Bitcoin address
Key search speed in my laptop, I know...
Estimated time to find an address with a 50% prob

FIGURE 8.7 Vanitygen address generator



```
$ ./vanitygen 1pfran
Difficulty: 15318045009
[308.65 Kkey/s][total 2300672][Prob 0.0%][50% in 9.6h]
```

FIGURE 8.8 Vanitygen address generator with a shorter prefix



```
$ ./vanitygen 13mckXcnnEd4SEkC27PnFH8dsY2
Difficulty: 2842910663779337231132135472805616259922420397
[313.77 Kkey/s][total 483072][Prob 0.0%][50% in 1.991443e+32y]
```

Still not a full address, although close
Key search speed
Estimated time to find an address with a 50% prob

FIGURE 8.9 Vanitygen address generator with a string close to the size of an address

expressions, not just prefixes. It also allows the input of a file with the desired prefixes or regular expressions to search, which permits searching for several unrelated vanity addresses concurrently, thus achieving economies of scale. Someone with vast amounts of computing power could offer a service to search for vanity addresses, thus reaping the benefits of these economies of scale. The problem is, however, that the client of this service cannot be sure that the vanity generator she has hired would not keep a copy of the private key.

There is a workaround to this problem by using split-key elliptic curve math (section 8.5). The user generates a EC pair. She keeps the private key to herself and sends the public key to the vanity generator. The vanity generator *starts the search from the public key provided, instead of from the generator of the elliptic curve*. That is, the vanity generator searches for a partial private key. When found, the vanity generator sends this partial private key to the client. The client adds up the two partial parts of the private key to arrive at the final private key. The vanity generator never gets to know the full private key, and thus is not able to steal the funds in the address. There are services, called vanity pools, that allow users to post a partial public key and a desired vanity address. Then anybody with computational power can try to solve for those vanity addresses, getting a fee when a vanity address is found. The operator of the pool gets a cut of the fee.

Vanity addresses have their advantages, especially in terms of branding. However, using vanity addresses gives away with the (pseudo)anonymity of Bitcoin addresses, and makes it easier to data mine the transaction graph, facilitating the process of matching Bitcoin addresses with real world identities (Chapter 13). Also note that vanity addresses are usually reused, i.e. multiple transaction outputs are signed out of them. This could be a security problem if used with a wallet software that uses a weak random number generator (5.4.2).

Finally, a word of caution with vanity addresses: vanity addresses are subject to the address tampering attack introduced at the beginning of this chapter. A vanity address with a certain message in it does not necessarily belong to the company or institution on the address because anyone can generate it. For instance, a Bitcoin address that starts with 1pfranco does not necessarily belong to this author. Furthermore, there can be many addresses that start with 1pfranco, generated and controlled by different people. All that can be said about a vanity address is that someone has spent computational power to arrive at such an address.

8.8 SIMPLIFIED PAYMENT VERIFICATION (SPV)

Full Bitcoin nodes verify new blocks that are added to the blockchain, checking that all transactions included are properly signed and that the hash of the blocks satisfy the proof-of-work difficulty. During this process, full nodes maintain a cache of unspent transaction outputs, UTXO (Chapter 6). The UTXO can be queried by a wallet to determine the amount of funds available in the addresses controlled by that wallet. A wallet implementation following this approach has two disadvantages that make it unpractical for lightweight clients:

- It requires downloading and storing the full blockchain. The size of the full blockchain is already beyond the reach of smartphones both in terms of storage and network bandwidth.
- It requires a full node verifying all blocks and all transactions contained in those blocks. At the current level of network utilization of 0.5 tps (transactions per second) (section 7.7) validating all transactions in the block is within the reach of the average smartphone. The problem is, however, that this transaction verification would quickly drain the battery of the device.

One possible solution would be to run a wallet client that would connect to a trusted server. The wallet would use the trusted server as a proxy to connect to the Bitcoin network. It could query the server for new transactions to certain addresses periodically, and it could use the server to relay signed transactions to the network.

Another possible solution is to use a **Simplified Payment Verification (SPV)** wallet. SPV was introduced by Satoshi Nakamoto in the original Bitcoin paper (Nakamoto, 2008a), where Nakamoto proposed that an SPV client only keep a copy of the block headers. When an SPV client needs to verify a transaction it downloads the Merkle branch that binds the transaction to the block header (section 7.6). Note that SPV clients do not need to rely on trusted nodes. The SPV approach is followed by several open source Bitcoin wallets.

When there are conflicting branches, i.e. a fork in the blockchain, a regular node will assume the longest blockchain (in terms of difficulty) is the legitimate one. Thus a regular node determines the validity of transactions by their inclusion in a block that belongs to the longest chain. This is called the **block height** validity check. In contrast, an SPV client determines the validity of a transaction by how many blocks have been mined on top of the block where the transaction is included. This is called the **block depth** validity check.

The original Bitcoin paper did not address how an SPV client would receive information about the relevant transactions, i.e. transactions involving the addresses controlled by the wallet. A query mechanism between the SPV client and the node is needed. This need is filled by connection Bloom filtering, specified in BIP 37 (Hearn, 2012). **Connection Bloom filtering** allows nodes in the network to set up filters in their connection with other nodes.

A **Bloom filter** is a probabilistic data structure that represents a set. Testing if an element belongs to the set is an efficient operation. The idea behind bloom filters is to use an array of bits that are filled with the results of applying several hash functions to the elements[14]. That is, the hash functions perform a mapping from elements to the array of bits. To test if an element belongs to the filter, the result of its hashing is compared against the bits set in the filter. If all bits that are 1 in the hash are set in the filter the element belongs to the filter. Conversely if any bit that is 1 in the hash is not set in the filter the element does not belong to the filter (Bloom, 1970).

Bloom filters are probabilistic data structures because their results are not exact. False positives, where an element which does not belong to the set is flagged as part of the set, are possible. However, false negatives, where an element which does belong to the set is not flagged as part of the set, are not possible. This is not a limiting factor for SPV clients: false positives are just transactions that the client is not interested in and can be discarded.

A client can set up a filter with a node. The node will then serve filtered blocks, i.e. blocks that only include transactions (and their branch of the Merkle tree) captured by the filter[15]. To create a filtered block, the node parses all incoming transaction testing the hash of each transaction and every data element in the inputs and outputs of a transaction against the Bloom filter. If there is a match against the filter, the transaction is included in the filtered block. Because all data elements of a transaction are tested, addresses, public keys and even the hashes of P2SH transactions can all be added to a filter[16].

[14] Bloom filters proposed in BIP 37 use the Murmur algorithm to generate the hash functions used in the filter.

[15] Bloom filters are implemented in Bitcoin Core in the class **CBloomFilter** in `bloom.(h|cpp)`. The **CBloomFilter** class is then used by CMerkleBlock in `main.(h|cpp)`.

[16] Some meta-coins (section 12.7) use Bitcoin's blockchain to store their meta-coin transactions. Bloom filters can be set up to filter transactions belonging to these meta-coins.

The precision of a bloom filter is defined as the fraction of false positives. This precision depends on the percentage of set bits in the filter. A filter that includes many elements would have a large percentage of its bits set and would lead to many false positives. This behavior can be used by clients to increase the privacy of their queries to a node. Increasing the number of bits set in the filter would include more transactions than the desired ones, making it harder for the node to single out the addresses controlled by the client. However, this comes at the cost of an increased bandwidth. Thus Bloom filtering allows a trade-off between privacy and bandwidth.

Both lightweight clients that use trusted nodes and those that use SPV are called **thin clients**. A review of the security implications of thin clients can be found in Bitcoin wiki (2014y).

8.9 THE "PAYMENT PROTOCOL" (BIP 70)

The Payment Protocol is a communication protocol between a merchant and a client wishing to pay with bitcoins, specified in BIP 70 (Andresen, 2013c). Before the Payment Protocol, when a client paid a merchant with bitcoins, she had to copy and paste the merchant's Bitcoin address in her wallet client, select the amount and make the wallet send the payment[17]. This procedure is vulnerable to man-in-the-middle attacks (section 5.1): an attacker that controls the connection to the network can replace the Bitcoin address of the merchant with an address of her own.

The Payment Protocol extends Bitcoin addresses with a message that also contains the name of the merchant, the amount to pay, and an additional message from the merchant. The whole message is signed by the merchant with the private key of his X.509 certificate. **X.509** is one of the most widely used standards for public key infrastructure (PKI, see section 5.1) in the internet.

A transaction following the Payment Protocol takes these steps:

- The client decides to make a payment, i.e. clicks the "pay button."
- The merchant creates a payment request and signs it with its X.509 certificate. This payment request includes the address where the merchant wishes to receive the payment.
- The merchant sends the signed payment request along with the certificate to the client wallet.
- The wallet checks that the message is correctly signed with the X.509 certificate included in the payment request and that the X.509 certificate is signed by a Certificate Authority whose certificate is found in the device[18]. It then shows the user the name of the merchant and the solicited amount.

[17] This process had been somewhat streamlined with the introduction of the URI scheme in BIPs 20 and 21 (Dashjr, 2011; Schneider, 2012). Using the URI scheme, clicking on a payment link would open the wallet software with the address of the merchant and the correct amount already filled in. The URI scheme has been extended to support the Payment Protocol in BIP 72 (Andresen, 2013d).

[18] This procedure is standard and is followed by every browser when connecting to a website through https.

- If the user accepts the payment, the wallet creates and signs a transaction sending the funds to the requested address(es), and includes it in a message for the merchant. This message can contain additional information, such as a return Bitcoin address or a message to the merchant.
- The merchant receives the payment message, extracts the signed transaction and publishes it to the network[19]. It then returns a signed payment receipt to the client, who thus receives immediate confirmation of the payment.

Note how using X.509 certificates provides protection against man-in-the-middle attacks, as the merchant signs the payment address using her X.509 certificate's private key. Some additional features of the Payment Protocol include:

- The merchant and the client can use the Payment Protocol to send messages between them. In particular, messages sent by the merchant would be presented to the user by the wallet client.
- The merchant can split the funds to be received between many addresses. This could be used to implement merge avoidance (13.4.4).
- The client can include refund addresses to be used by the merchant in case the order cannot be fulfilled[20]. Return addresses can be generated automatically by the wallet software, for instance using BIP 32 (8.5.2), to automatically avoid address reuse.
- The payment receipt can include a proof of payment signed by the merchant that the client can later use in case of a dispute.

For a full specification of the protocol, consult BIP 70 in Andresen (2013c).

[19] The client could also publish the signed transaction to the network herself.

[20] Before this feature was introduced, the merchant would have to get in contact with the client—say by email—and request a refund address. Note that the merchant should not assume that the address it received funds from is a valid return address: the private keys for this address could have been deleted by the wallet after depleting the funds, the user could have been using a web wallet where the address belongs to the web wallet provider, or the user could have stopped using that wallet.

CHAPTER 9

Mining

Bitcoin mining was introduced in section 7.4, in the context of the blockchain. **Mining** is the process of adding blocks to the blockchain. Miners contribute their computational power to solve the blocks that are added to the blockchain, and the network remunerates them with the block reward and the fees collected from all the transactions included in the block.

Miners solve the partial hash inversion problem. To find a solution, mining software usually increments the block nonce and runs the proof-of-work algorithm to check if the chosen nonce generates a correct block hash (i.e. a block hash that meets the difficulty requirements)[1].

A typical optimization used by miners is to pre-compute the hash of the initial part of the block header that contains the previous block hash and the root of the Merkle transaction tree (section 7.6). This part of the block header is constant during the mining process and therefore can be stored in a buffer.

One of the advantages of the mining mechanism is that it rewards early adopters for supporting the network. This was very important in the beginning, when Bitcoin bootstrapped itself into relevance. Bitcoin does not have a corporation backing it, so marketing had to be done virally. This would have been impossible without the help of early adopters. Rewarding miners is a way to enlist them to create word of mouth.

Mining is similar to a market with perfect competition: as long as there is profit to be made, new entrants will enter the market until the profit opportunity is depleted. As described in 7.4, the mining difficulty increases as more miners enter the network, but the total block reward stays the same. At the creation of Bitcoin, the block reward was 50 bitcoins. This block reward is halved every 210,000 blocks[2], or roughly every 4 years, to comply with the pace of money creation set in the protocol. Figure 9.1 shows the amount of bitcoins issued. Note that the issuance of new bitcoins is not a smooth line, as the introduction of new mining capacity temporarily increases the rate of new block creation until the feedback mechanism catches on. Thus, under an increasing network hash rate, the issuance of new bitcoins accelerates somehow. On November 28, 2012—"halving day"—more than a month ahead of schedule, the block reward was halved to 25 bitcoins. As of the time of writing, the protocol awards a fixed reward of roughly $24 \cdot 6 \cdot 25 = 3{,}600$ bitcoins every day.

[1] Mining is implemented in Bitcoin Core in the function **BitcoinMiner**. The actual mining loop can be found in the function **ScanHash_CryptoPP**. Both functions are located in `miner.cpp`. The probabilities of finding a partial hash inversion are the same whether the block nonce is incremented or the nonce is generated randomly, so mining algorithms will implement whichever is faster. Mining was disabled by default in the Bitcoin client, as it became uneconomical to mine using CPUs.

[2] See variable **nSubsidyHalvingInterval** in `chainparams.cpp`.

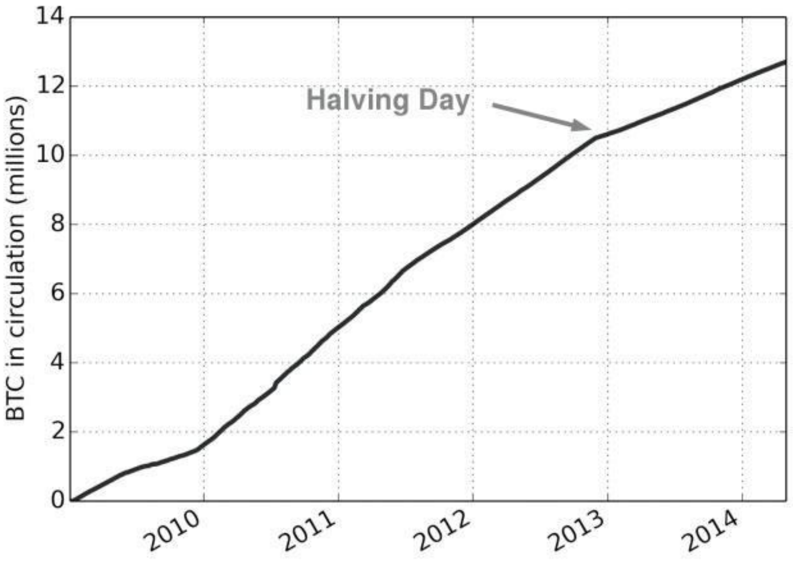


FIGURE 9.1 Bitcoins in circulation. Data from blockchain.info

Bitcoin is a peer-to-peer network; anyone can connect to it and start mining right away. New entrants do not have to ask for permission or adhere to a set of rules or regulations before they enter the mining market. Nor can incumbents collude to prevent new participants from entering. Thus new investment will enter the contest to capture the block reward, lowering the reward of all miners already in the network. Thus, in an scenario of increasing bitcoin price (or increasing technological advancement), miners have to keep increasing their hashing rate in order to obtain the same reward, in a process similar to the Red Queen Effect[3]. This process will continue until the marginal cost of the last miner to enter equals her expected reward. At this point the network has reached an equilibrium, which can only be perturbed by some external factor, such as a further increase in bitcoin prices.

There are, however, some factors that could confer a sustainable advantage to some participants, allowing them to enjoy higher profits:

- **Technological advantage**. This technological advantage could either stem from an innovation in the implementation of the proof-of-work algorithm (SHA256^2) in silicon hardware[4] or it could stem from a miner controlling a better chip manufacturing process, such as a big chip manufacturer entering the mining business.

[3] The Red Queen Effect refers to situations where competitors must constantly evolve, not to gain an advantage but merely to survive in a highly competitive environment. It gets its name from Lewis Carroll's Red Queen character when she explained to Alice that it took all the running she could do just to keep in the same place.

[4] At the time of writing, an SHA256 hash function takes approximately 20,000 gates to build. A technological breakthrough that reduces that number significantly could spark a new episode in the ASICs arms race.

- **Hedging Bitcoin volatility**. A miner could get an advantage if she were able to hedge the Bitcoin price volatility more effectively than her competitors. Any miner could in principle hedge the Bitcoin price volatility using Bitcoin futures[5] but, as of the time of writing, this market is almost non-existent. This advantage could be specially important during periods where the price of Bitcoin is depressed and competitors could be forced to shut down[6]. Furthermore, a miner who is able to hedge the volatility of her income would require a lower rate of return for her investment.
- **Lower electricity prices**. Miners who are able to secure low electricity prices have a cost advantage. Bitcoin mining would likely migrate to places with cheap and abundant electricity, such as Iceland. This might even decrease the environmental impact of Bitcoin mining, as places with cheap electricity are usually able to generate it from environmental-friendly sources, such as hydro-electrical plants.

In summary, barriers to entry to the mining business are generally low, as there is no way for the incumbents to collude and prevent new competition from entering the network. Therefore the network hash rate will probably stabilize at a rate where the mining reward just covers the marginal costs of running the mining equipment.

The marginal costs of running mining equipment include the cost of electricity, but also the renting costs of the datacenter, refrigeration costs, maintenance, and so on. Then there is the amortization cost of the equipment itself, or its opportunity cost. The only currently viable technology—ASIC—is highly optimized for Bitcoin mining, and does not have any other alternative use[7]. These factors, coupled with the lag in the production of mining equipment in response to Bitcoin's price increases, could create boom and bust cycles in the mining market.

It has been argued in Güring and Grigg (2011) that as botnet operators do not have to pay the operating costs of running the equipment (notably the electricity costs), mining botnets would displace legitimate mining, leading to the collapse of Bitcoin mining. However, with the current network hash rate (section 9.1), it is more economical for a bot-herder to use her botnet for other nefarious purposes (click fraud, email spam, or plain spyware) than to use it for Bitcoin mining.

[5] A short position in a bitcoin future would pay the difference between the price of bitcoin at the inception of the contract and the price of bitcoin at a predetermined date. For example, if the price of a bitcoin has dropped from 600 USD to 500 USD, the short future would pay 600 USD – 500 USD = 100 USD. A miner who holds a short position in this future contract would effectively lock a future bitcoin price of 600 USD: 500 USD coming from the market at the expiration of the contract and 100 USD coming from the payoff of the future contract.

[6] This might turn out not to be a significant advantage because, even though some miners would turn off the mining hardware during certain periods, the hardware is still there and could be turned on again if the price of bitcoin recovers.

[7] An alternative use to mining Bitcoin would be mining other cryptocurrency whose proof-of-work hash function is SHA256 or SHA256^2, such as Peercoin or Namecoin. As the prices of most cryptocurrencies are highly correlated, a drop in bitcoin prices would most likely lead to a switch-off of the uneconomical mining equipment.

9.1 MINING TECHNOLOGY

As of the time of writing, the network hash rate stands at around 30,000,000 GH/s (= 30,000 TH/s = 30 PH/s). Figure 9.2 presents the evolution of the network hash rate since Bitcoin's inception, on a logarithmic scale[8]. The figure shows the dates of introduction of new technologies and the corresponding "eras" in mining history. The exponential growth in the network hash rate has been due to two trends:

- Exponential growth in the price of Bitcoin itself, which has attracted a lot of mining investment.
- Advances in mining technology, as mining equipment manufacturers have caught up with state-of-the-art chip manufacturing.

Mining hardware has followed a trend toward more specialized hardware where a larger part of the circuitry of the chip is dedicated to the hashing function. There have been four phases in this transition:

- **CPUs**. CPU stands for Central Processing Unit: the main chip inside computers and other devices. It is general purpose hardware: its computational power can be applied to many tasks, including mining Bitcoin. The initial release of the Bitcoin Core implemented mining on the CPU. During the first phase of Bitcoin mining, running from 2009 to the summer of 2010, mining was performed only using CPUs. During this phase, the growth of the hash rate was due to new enthusiasts entering

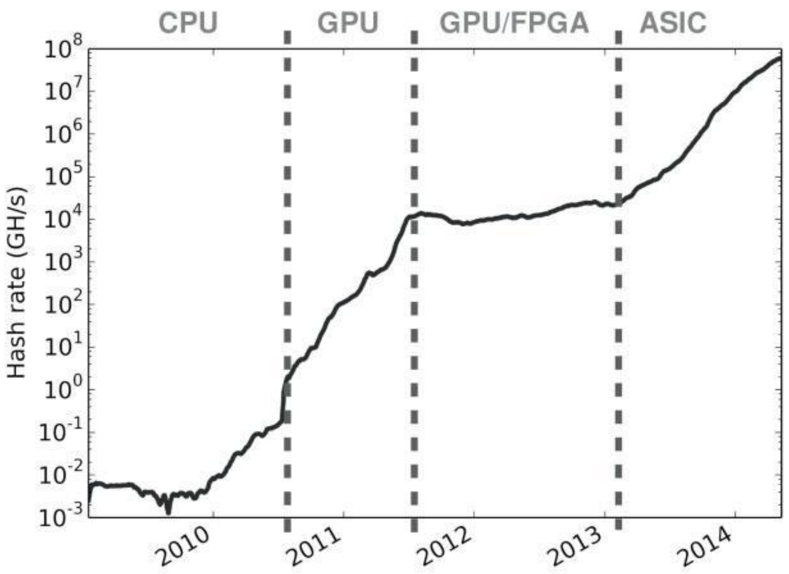


FIGURE 9.2 Hash rate of the Bitcoin network. Hash rate data from blockchain.info

[8] Linear growth on a logarithmic scale equals exponential growth on a linear scale. The growth of the network hash rate has been exponential so far.

the mining space. The latest retail processors offer a hash rate of approximately 20MH/s.

- **GPUs**. GPU stands for Graphics Processing Unit: the specialized computer chip originally used for graphic acceleration. There is a trend in computing of using the parallel power of GPUs to perform general computations, known as GPGPU or General-Purpose computing on GPU. Starting in mid-2010, GPUs were programmed to mine Bitcoins, quickly rendering CPU mining uneconomical[9]. GPUs offer an advantage over CPUs because they are composed of hundreds or even thousands of computational units, compared with the handful in a typical CPU. The computational units of a GPU are much more limited than those of a CPU, but enough to perform SHA256 hashes. For a more detailed explanation of why GPUs offer a greater hash rate than CPUs, see Bitcoin wiki (2014ac). The latest GPUs offer a hash rate ranging from 100MH/s to 500MH/s.
- **FPGAs**. FPGA stands for Field-Programmable Gate Array. FPGAs are chips built of logic blocks that can be programmed and interconnected to perform a particular task. As the name suggest, FPGAs are designed to be programmable "in the field," i.e. after shipping. FPGAs were introduced in Bitcoin mining in mid-2011 and for a time competed with GPUs. GPUs held the advantage on cost per GH/s and resale value, while FPGAs had an advantage in lower power consumption (Taylor, 2013). Typical FPGAs have a hash rate of approximately 1 GH/s.
- **ASICs**. ASIC stands for Application-Specific Integrated Circuit. ASICs are chips built for a specific application, in contrast to CPUs (or, to a lesser degree, GPUs) that accept software running many possible applications. ASIC parts have the logic of the SHA256 function copied as many times as the area of the chip allows, in order to run as many hash tries in parallel as possible. Early ASIC design reused the technology developed for FPGAs (Taylor, 2013). The hash power of an ASIC depends on its manufacturing process technology. At the time of writing, 28nm ASICs offer a hash rate of approximately 500 GH/s, with 20nm 3TH/s parts in sight.

Some of the periods of exponential increase (or even jumps) in the network hash rate have coincided with the introduction of new mining technologies (see Figure 9.2). The latest period—ASICs era—is still in progress at the time of writing. However, as the next ASIC iteration (20nm) catches up with the state-of-the-art in chip manufacturing process, the exponential trend in hash rate is set to level off. From that point on, economics suggest that increases in the network hash rate will follow advances in chip manufacturing process and bitcoin prices[10].

[9] Sathosi Nakamoto initially envisioned mining as computational democracy, saying "proof-of-work is essentially one-CPU-one-vote" in the original Bitcoin paper (Nakamoto, 2008a). He (she?) commented on the forums: "We should have a gentleman's agreement to postpone the GPU arms race as long as we can for the good of the network" (Marion, 2014). He felt the introduction of GPU mining would be detrimental to participation in Bitcoin mining, as GPU hardware is less widespread than CPUs.

[10] If, once the mining steady state is reached, the price of bitcoin falls, some mining equipment might be disconnected. This equipment will still be there, waiting for an increase in the price to make it viable. This would create a cap on the remaining miners' profit margin, as a subsequent increase in the bitcoin price would prompt the disconnected equipment to reconnect again.

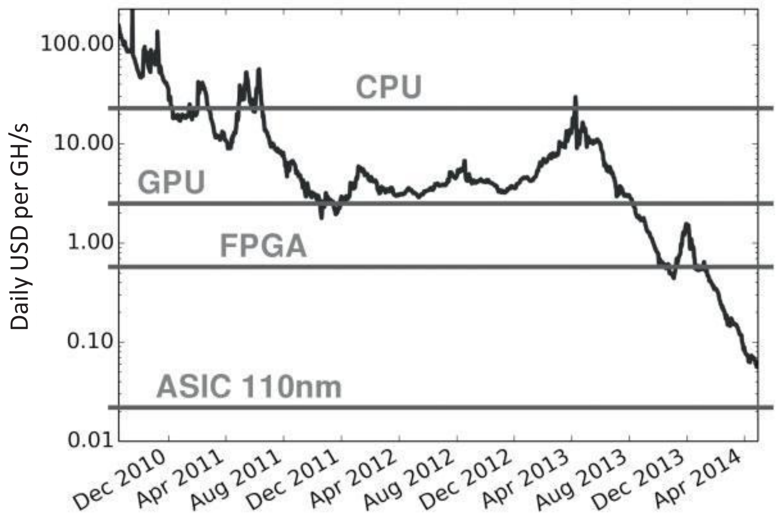


FIGURE 9.3 Mining revenue compared to electricity cost of different technologies. Price and hash rate data from blockchain.info

Figure 9.3 shows the daily revenue obtained from Bitcoin mining, in daily USD per GH/s. Superimposed in the figure are the levels showing the electricity cost of each technology that can be used for Bitcoin mining. The cost levels have been taken from Taylor (2013) with an estimated cost of electricity of 0.2 USD/kWh. As the expected revenue from mining has decreased and become lower than the electricity costs of running the hardware, mining technologies have become obsolete. As the figure shows, CPUs, GPUs and FPGAs are no longer profitable. The only viable technology to mine Bitcoin is ASICs. If the trend in Figure 9.3 is extrapolated, most ASICs based on old process technologies will progressively become obsolete, and the mining revenue will stabilize close to the electricity cost of the state-of-the-art ASIC technology[11]. As of the time of writing, the estimated electricity cost of 28nm ASIC technology stands at around 0.003 USD per GH/s.

With the increase in mining difficulty, miners holding hardware that is no longer competitive—like CPUs and GPUs—have migrated to mining other cryptocurrencies, notably Litecoin. Alternative cryptocurrencies will be covered in Chapter 11.

The current network hash rate is equivalent to approximately 1,250,000,000 latest-generation CPUs or 90,000,000 latest-generation GPUs, assuming that this hardware was put solely to the task of mining bitcoins. However, the comparison is not completely fair, because CPUs and GPUs are general-purpose hardware that can perform many more tasks than just mining Bitcoin. On the other hand, most Bitcoin mining is performed by ASICs that can only perform a very narrow computation (SHA256

[11] The mining revenue will likely be higher than just the electricity costs, to cover the rest of the costs associated with running a mining operation.

hashing)[12]. Following some of these measures, Bitcoin is claimed to be one of the biggest computational networks in the world.

The takeaway from these comparisons is that an attacker that wished to perform a 51% attack on Bitcoin would have to realize a large investment in mining equipment. Thus the network hash rate is an indication of the security underlying the blockchain. Increases in the hash rate raise the bar for an attacker wishing to perform a 51% attack. Conversely, a decrease in the hash rate would be detrimental to the security of the distributed database.

Some commentators have suggested a positive feedback loop between the price of bitcoin and the network hash rate. While it is true that mining investment follows the price of Bitcoin, the converse is not necessarily true. After a decrease in the price of Bitcoin, some mining power will be disconnected, and the security of the blockchain going forward will be lowered. But this mining dynamic should only affect the price indirectly, as the effect of a decrease in the network hash rate, if any, should be already incorporated in the new price where the market has found an equilibrium.

There has been some controversy regarding the environmental impact of Bitcoin mining. This controversy has been fuelled by incorrect estimates of the total **electricity consumption** of the network. Some journalists have quoted an estimate of the electricity consumption based on CPU technology. As mining technology has moved on, and the only current viable mining technology—ASIC—has a much lower energy consumption per GH/s, these figures overestimated the total energy consumption of the Bitcoin network.

As mining technology catches up with state-of-the-art process technology, mining cost will be driven primarily by electricity costs and mining power will then probably shift to locations where electricity is cheaper. The environmental impact of electricity in places with low electricity costs might be smaller, as these are usually places with large natural sources of energy. Besides, the energy consumed by Bitcoin mining is arguably not wasted: it is employed in securing the blockchain. A fairer comparison might be to the emissions produced by the current financial system in achieving a similar goal.

A compilation of the specifications of a long list of mining equipment spanning all four technologies can be found in Bitcoin wiki (2014m). A good account of the different steps in the evolution of the mining technology can be found in Taylor (2013).

9.2 POOLED MINING

Assuming the arrival of new blocks follows a Poisson process, the time between two arrivals (mining of a new block) should follow an exponential distribution. To test this hypothesis, a sample of times between blocks has been assembled. These times are computed as the difference between time-stamps of consecutive blocks in the blockchain,

[12] An even less meaningful comparison is sometimes made between mining FLOPs (floating point operations per second) and supercomputers' FLOPs. The problem with this comparison is that the SHA256 algorithm does not perform any floating point operation and mining ASICs do not have a floating point unit (FPU). Thus the conversion of the "computational power" of an ASIC to FLOPs is somewhat arbitrary.

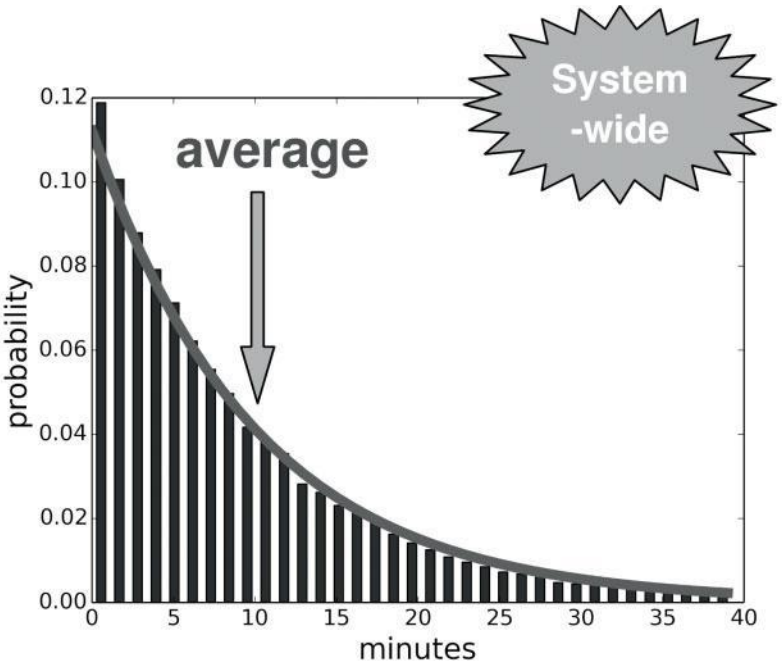


FIGURE 9.4 Probability of mining a block (network-wise)

using blocks between 200,000 and 250,000[13]. Figure 9.4 shows the empirical probability distribution of the sample, and Figure 9.5 shows an exponential distribution Q-Q plot of this sample. The fit is reasonably good, with a high $R^2 = 0.9993$, but there is some divergence from the exponential distribution in the right tail.

The sample average time between blocks is 553 seconds. The theoretical exponential probability distribution function, with a mean arrival time of 10 minutes (600 seconds), is superimposed in Figure 9.4.

Figure 9.4 shows the distribution of block arrivals for the network as a whole. An individual miner solving a block also follows a Poisson distribution, but with a larger time between arrivals. Individual miners are subject to a high degree of uncertainty as to when they will mine a new block. As an example, right before the introduction of ASIC mining when GPU mining was still profitable, the expected time between solved blocks for an individual miner with one GPU would have been in the order of 150 days.

To help miners manage this risk, mining pools started to appear at the end of 2010. A **mining pool** is an aggregation of miners, who contribute their hash power to the pool and share the mining rewards. Figure 9.6 shows an illustration of the expected time between mined blocks for several miners alone and for a pool that aggregates all their hash power. By forming a pool, miners can have a much more predictable income stream to

[13] Some differences between timestamps were negative, due to inaccuracies in the time settings of mining servers. These negative differences have been rounded to 0 in the analysis of figures 9.4 and 9.5. The sample average time during these periods was 544 seconds without adjusting for negative arrival times.

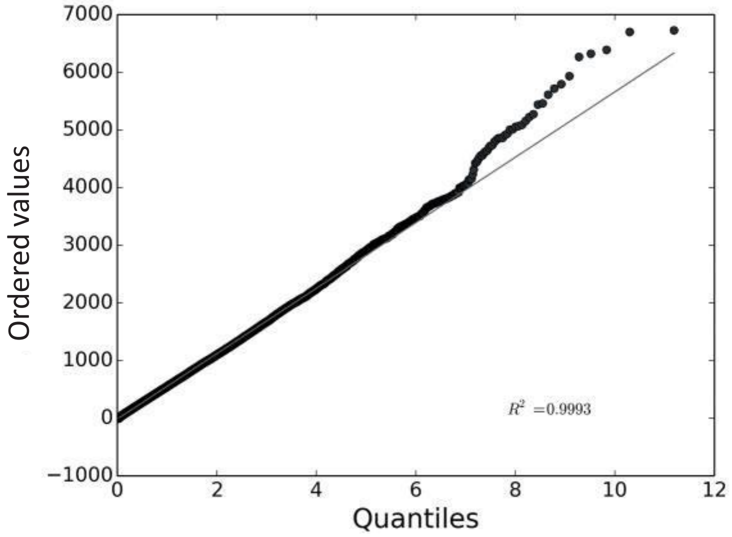


FIGURE 9.5 Q-Q plot (exponential distribution) of the empirical time between blocks

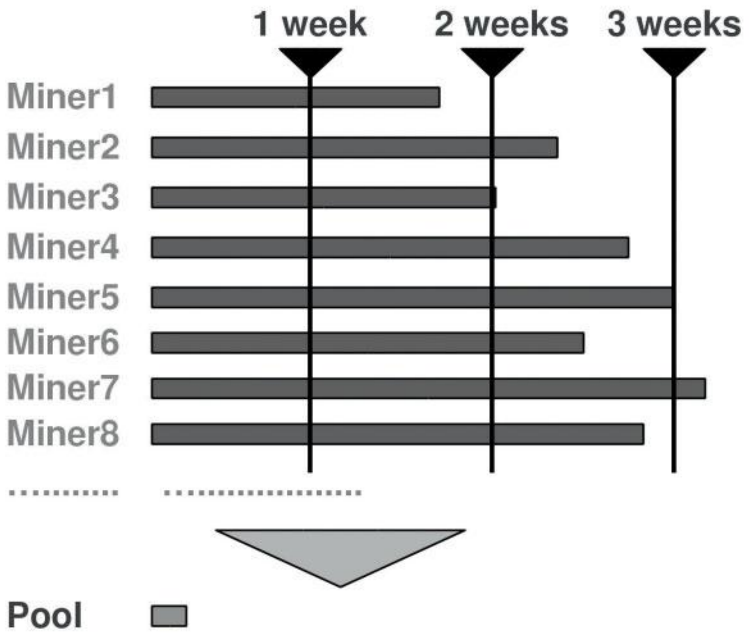


FIGURE 9.6 Expected time to mine a block

share among them. Revenue sharing in a mining pool is proportional to the hash rate contributed by each miner, minus a small fee charged by the pool operator, which is usually run for profit. An additional advantage is that miners participating in a pool do not have to keep a copy of the full blockchain or process all incoming transactions: it is enough for the pool operator to feed miners a copy of the block header[14].

After the introduction of ASICs, the need for mining pools may decrease, due to increased professionalization of the mining activity, such as the arrival of hosted mining services. Still, at the time of writing, most of the mining is done by a handful of pools, see Table 9.1.

A mining pool whose participants "promise" to share their computational work with the mining pool and whose operator "promises" to share the pool rewards honestly among its members, is fraught with conflicts of interest. Both miners and pool operators have an incentive to cheat:

- Miners have an incentive to overstate their hash rate, or contribute only a portion of their hash rate, while mining solo with the rest of the hash rate. A pool can control the work done by the miners by requiring them to present a valid proof-of-work of the block they are mining, but of a lower difficulty. These are called **shares**. The pool operator can measure shares received by its miners, and allocate the block reward proportionally to these number of shares. Another approach to control the work done by miners is for miners to submit **metahashes**. Metahashes are hashes of many hashes produced by the miner. The pool operator then checks the validity of the metahashes provided by its miners. Checking metahashes is computationally intensive: to check all the metahashes from all the miners, the pool operator would have to redo all the work of all miners, which defeats the purpose of a mining pool. Thus the pool operator checks the metahashes only periodically, usually in a round-robin fashion[15]. As the metahash approach is much more computationally intensive than the share approach, it is rarely used in practice.
- Miners have an incentive to publish a new block on their own when they find it. This can easily be avoided if miners are given the hash of the block header by the pool operator, which includes an address under the control of the pool operator in the coinbase transaction. Thus a miner cannot change the address that will be credited with the block reward.
- Block operators have an incentive to cheat miners. When presented with a new block, the pool operator has an incentive not to share the block reward, or share it only with the miner who presented the block, leaving the rest of the pool in the dark. This issue could be solved if miners were to request the whole block header from the pool operator, not just the hash of the beginning, so they would be able to monitor the blockchain themselves. Another approach to solve this problem is for the miners to receive a fixed payment for their work, irrespective of whether a block is mined

[14] There are several competing mining protocol standards. In some of them, pool operators only share the block header with the miners. But in others, such as the GetBlockTemplate (Bitcoin wiki, 2014i), operators share the whole block with miners. In these latter protocols, miners can choose which transactions to include or even include additional transactions of their own, that have not been broadcasted to the network.

[15] In a round-robin schedule, jobs are chosen from the available processes in a circular fashion. This scheduling assigns the same amount of work to each process. Applied in the context of metahash checking, it means that the pool operator checks the metahashes of each of its miners in circular order. Round-robin scheduling might be cheated if a miner is able to guess the frequency of metahash checking, thus some randomness is usually added to the process.

by the pool. This is the approach followed by pay-per-share (PPS) mining pools, see below.

Different pools follow different protocols to handle these conflicts of interest. Some examples of mining pools and the protocols they follow are:

- **Bitcoin Pooled Mining (BPM).** Miners submit shares, i.e. solved blocks with a lower difficulty. Older shares weigh less, to dissuade miners from switching pools in the middle of a round. A round finishes the moment a block is solved by the pool and the next round starts right away. Rewards are distributed to miners at the end of each round according to the number of shares submitted, but there is no mechanism to address the pool operator cheating.
- **Pay-per-share (PPS).** Miners submit shares and the pool operator pays an immediate flat fee for each share paid from the pool's existing balance, thus eliminating the risk that miners are cheated by the pool operator. The uncertainty of new block arrivals is transferred to the pool operator. The operator of a pay-per-share pool charges a small fee as compensation for this risk, making the payoff to miners lower than their expected payoff for solo mining.
- **Pay-per-last-N-shares (PPLNS).** This protocol is similar to pay-per-share, with the difference that the last N shares submitted by miners are taken into account to distribute the mining reward. The last-N-shares window can span several blocks: a share can be rewarded several times (if the pool gets lucky and it mines several blocks in quick succession) or not at all (if the pool gets unlucky and no block is mined during the window). In PPLNS, the uncertainty of mining a block is transferred from the pool operator to the participants in the pool, which usually translates into the pool operator collecting a smaller fee. Many of the largest mining pools at the time of writing use variants of this protocol.
- **P2Pool** is a peer-to-peer network of mining nodes that create a **share chain**. A share chain is a chain of shares, i.e. solved blocks with a difficulty lower than the network difficulty. Peers in the pool check that blocks in the share chain have a coinbase that distributes the reward proportionally to the *previous* shares in the share chain. When a block is solved, the miner who solves it publishes it in the blockchain, and the reward is automatically distributed among the peers through the coinbase. The chain of shares plays the same role as the pool operator in a pay-per-share protocol, but in a distributed way. As there is no pool operator, all mining rewards are distributed to the peers. The main disadvantage of P2Pool is that all miners have to operate a full node. This contrasts with the other protocols where miners are fed only the block header by the pool operator. Thus participating in P2Pool requires computing power and bandwidth to process the blockchain, increasing the cost to miners. Further details on P2Pool can be found in Bitcoin wiki (2014p).

Mining protocols based on miners creating shares of lower difficulty—such as PPS, PPLNS, or P2Pool—are subject to a block withholding attack. In a **block withholding attack** a malicious miner presents the pool operator with valid shares, but withholds solved blocks, depriving the pool from collecting the reward on these blocks. An attacker performing a block withholding attack suffers a small loss due to the pool's loss of revenue which will impact all users' payoff equally. The block withholding attack is akin