

Blockchain

Muchang Bahng

Fall 2024

Contents

1	Cryptography	3
1.1	Elliptic Curve Cryptography	5
2	Base58 Encoding and WIF Format	7
3	Bitcoin Keys and Addresses	8
4	Cryptocurrency Wallets	10
4.1	Mnemonic Code Words	10
4.2	Hierarchical Deterministic Wallets (BIP0032/BIP0044)	11
4.3	Extended Child Key Derivation - Non-Hardened Extended Private Key	13
4.4	Extended Child Key Derivation - Non-Hardened Extended Public Key	14
4.5	Extended Child Key Derivation - Hardened Extended Private Key	15
4.6	HD Wallet Key Identifier (Path)	17
5	Advanced Keys and Addresses	18
5.1	Encrypted Private Keys (BIP0038)	18
5.2	Vanity Addresses	18
5.3	Paper Wallets	19
6	Transaction Chains and UTXOs	19
6.1	Transaction Fees	20
6.2	Transaction Chaining and Orphan Transactions	21
7	Locking/Unlocking Scripts and Script Language	22
7.1	Script Language	23
8	Standard Transactions	23
8.1	Pay-to-Public-Key-Hash (P2PKH)	24
8.2	Pay-to-Public-Key (P2PK)	25
8.3	Multi-Signature	25
8.4	Data Output (OP_RETURN)	26
8.5	Pay-to-Script-Hash (P2SH)	27
9	The Bitcoin Network	28
9.1	Full Nodes vs Simplified Payment Verification (SPV) Nodes	29
9.2	Transaction Pools	31
9.3	Alert Messages	31
10	The Blockchain	31
10.1	Block Header	32

10.2 Linking Blocks in the Blockchain	33
10.3 Merkle Trees	33
11 Mining and Consensus	35
11.1 Mining Nodes & Aggregating Transactions into Blocks	36
11.2 Constructing the Block Header & Difficulty Representation	37
11.3 Mining the Block & Proof-of-Work Algorithm	37
11.4 Difficulty Target and Retargeting	38
11.5 Successfully Mining and Validating the Block	39
11.6 Blockchain Forks and Assembling Chains of Blocks	39
11.7 The Hashing Race and Mining Pools	42
11.8 P2Pools and Consensus Attacks	44

1 Cryptography

The most primitive types of encryption before the advent of computers consisted of Caesar Cipher, Vigenere cipher, Hill's Cipher, and Verman cipher (for brief descriptions, see here). All of these were easily decipherable by foreign parties using computers, so more sophisticated methods were developed, such as RSA, SHA, and AES, to list a few.

These advanced encryption methods can be categorized into 2 categories (plus an extra group for hash functions), followed by examples. These example algorithms differ from each other by how much data they can handle at once (when splitting strings into blocks of text and encrypting each of the blocks at a time) and what kind of key it needs for its decryption.

1. **Symmetric encryption** requires both parties (the sender and the receiver) to have access to the same **key**. So, the recipient needs to have the key before the message is decrypted. This works best for closed systems, which have less risk of a third-party intrusion. Symmetric encryption is faster than asymmetric encryption, but the limit is that both parties must securely store their keys before any message is sent.
 - **AES (Advanced Encryption System)** is one of the most secure and widely used encryption types, used by governments and businesses. It encrypts data in blocks, splitting up the plaintext into certain blocks, encrypting them separately. Additionally, each encryption method has a different number of rounds of encryption.
 - AES-128 encrypts blocks of a 128-bit size with 10 rounds
 - AES-192 encrypts blocks of a 192-bit size
 - AES-256 encrypts blocks of a 256-bit size with 14 rounds
 - **DES (Data Encryption Standard)** was the accepted standard of encryption in the 1970s but is considered unsafe on its own. It encrypts 56 bits of data in a block, i.e., a block/key length of 56 bits.
 - **3DES/TDEA (Triple Data Encryption Algorithm)** is an upgraded version of DES used today, which uses three separate 56-bit keys for triple protection. The shorter block lengths are encrypted three times, but newer forms may take over soon.
2. **Asymmetric encryption**, also called **public-private key encryption**, uses two keys for the encryption process, a public-private key pair (public, private). If user A, with pair (g, g') , wants to send user B, with pair (f, f') , a message M, A can take B's public key and encrypt the message to get the ciphertext $f(M)$. They can send this encrypted message over the internet to B, where B will use their own private key to decrypt the message to get $f'(f(M))$. Mathematically speaking, letting T be the set of all strings, the public key $f : T \rightarrow T$ is a mapping that encrypts a message into a jumble of characters, known to the world, while the private key $f' : T \rightarrow T$ is precisely the inverse of f . Usually, f' is almost impossibly hard to compute, which is what makes asymmetric encryption so secure. Even though public-key encryption is slower, it has the huge advantage of allowing *any* two parties to communicate to each other, without the need to exchange keys in advance.
 - **RSA (Ron Rivest, Adi Shamir, Leonard Adleman)** takes advantage of the mathematical fact that it is easy to multiply 2 large prime numbers together, but it is not easy to factor a large number into two prime numbers (by large we're talking on the order of hundreds of digits). The math behind RSA is here. It is extremely secure, with an estimated breaking time of thousands or millions of years before an RSA encryption is brute-force cracked. Another great advantage of RSA is its scalability, coming with various encryption key lengths such as 768-bit, 1024-bit, 2048-bit, 4096-bit, etc., but it is not practical for large or numerous files due to speed.
 - **ECC (Elliptic Curve Cryptography)** is based on the algebraic structure of elliptic curves over finite fields. It also works on the principle of irreversibility (easy to compute in one direction but painfully difficult to reverse it). More specifically, the security of ECC depends on the ability to compute a point multiplication and the inability to compute the multiplicand given the original

and product points. The size of the elliptic curve, measured by the total number of discrete integer pairs satisfying the curve equation, determines the difficulty of the problem.

3. Another type of encryption used is called the **cryptographic hash function**, which is an algorithm that takes an arbitrary amount of data input and produces a fixed-size output of enciphered text called a **hash value**, or **hash**. It is extremely hard to decode, so it is mainly used as a one-way encryption.
 - **MD5 (Message Digest 5)** creates 128-bit outputs but is an outdated method.
 - **SHA-1** is the second version of the Secure Hash Algorithm, after SHA-0. SHA-1 creates 160-bit outputs and replaced MD5.
 - **SHA-2** is a suite of hashing algorithms much more secure than SHA-1 algorithms, with each algorithm represented by the length of its output.
 - SHA-224 creates 224-bit outputs.
 - SHA-256 creates 256-bit outputs.
 - SHA-384 creates 384-bit outputs.
 - SHA-512 creates 512-bit outputs.
 - **LANMAN (LAN manager)** is the Microsoft LAN Manager hashing algorithm, used by legacy Windows systems, but due to weaknesses, it is not used today.
 - **NTLM (NT LAN Manager algorithm)** is used for password hashing during authentication and is the successor of the LANMAN algorithm.

Let's talk about public-key encryption for a second. User A has no problem sending a secure message to B, but there is a huge problem: B has no way of verifying that the message actually came from A! Some hostile third party C could have sent a misleading message N to B by encrypting N with B's public key to get $f(N)$. B has no way to know whether $f(N)$ or $f(M)$ is the correct one.

In order to verify the identity of the sender, **digital signatures** come into play. In order for A to prove the authenticity of the message M (or to be exact, the encrypted message $f(M)$), A must digitally sign the document/message. First, A uses a hash function H to generate the encrypted version of the message, getting $H(f(M))$ (this hash function is well known). Now, this is further encrypted using A's private key g' to produce $g'(H(f(M)))$. This is appended to the document and sent to B with A's public key as a triplet $(f(M), g'(H(f(M))), g)$. B, upon receiving this triplet, has everything they need to verify the sender's identity.

- B first takes the well-known hash function H and computes $H(f(M))$ himself.
- B takes the string $g'(H(f(M)))$ and decrypts it using the public key g to get $g(g'(H(f(M)))) = H(f(M))$
- If the two values match, then B can verify that the sender who has that public key g also has the correct private key g' , and is therefore user A.

Let's observe the futility of (third-party) C's efforts when trying to send a false message. Assume that C has key pairs (k, k') .

- C can send the key triple using their own public key as $(f(M), k'(H(f(M))), k)$. Certainly, the hash functions will match since $H(f(M)) = k(k'(H(f(M))))$, but B can clearly see that the public key received is k , and not g like it's supposed to be.
- C can try sending the public key as g (since A's public key g is open to everyone) to send the triple $(f(M), k'(H(f(M))), g)$. However, B would find out that $H(f(M)) \neq g(k'(H(f(M))))$, and so the hashes don't match.

Either way, C has no hope of impersonating A. Note that this is all dependent on sender A keeping their private key g' secure. If user C had access to g' , then this entire security system is compromised. In other words, two separate entities in possession of the same key-pair are indistinguishable within the network.

1.1 Elliptic Curve Cryptography

We first introduce the **Discrete Logarithm problem**. It is well known that for any given $a, b \in \mathbb{R}$, the logarithm $\log_b a$ is a number x such that $b^x = a$. Analogously, in any algebraic group G , powers b^k can be defined for all integers k , and the **discrete logarithm** $\log_b a$ is an integer k such that $b^k = a$. Discrete logarithms are quickly computable in a few special cases, but no efficient method is known for computing them in general.

For our purposes, we can define an elliptic curve as a plane curve over a finite field which consists of the points satisfying the equation

$$y^2 = x^3 + ax + b \quad (1)$$

along with a distinguished point at infinity, denoted ∞ . The coordinates here are chosen from a fixed finite field of characteristic not equal to 2 or 3. This set together with the group operation of elliptic curves (described here and more informally below) is an abelian group, with the point at infinity as an identity element.

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

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

or

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

where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, produces an elliptic curve that looks like the following. Note that it is a collection of points since the "curve" is embedded in a discrete field rather than a continuum. In actuality, the following visual is for when $p = 17$.

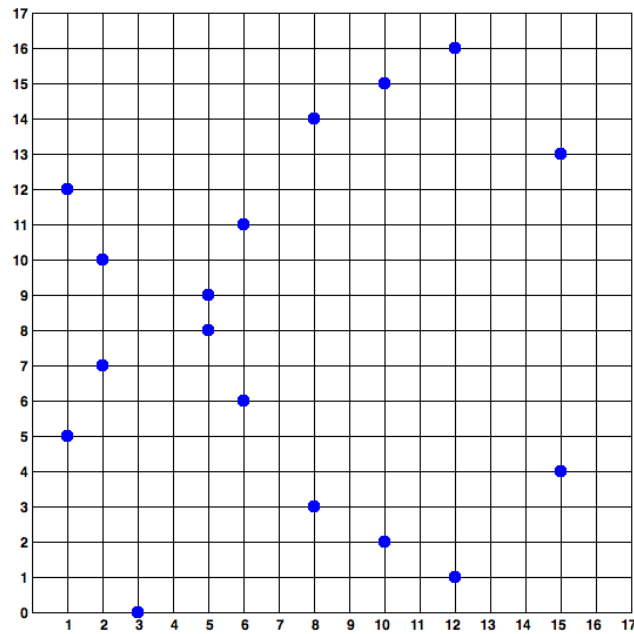


Figure 1: Discrete Elliptic Curve

One solution is the point

```

1 x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
2 y = 32670510020758816978083085130507043184471273380659243275938904335757337482424

```

On a larger scale, the curve has more of a sense of randomness to it as we zoom out or embed it into different 2-manifolds.

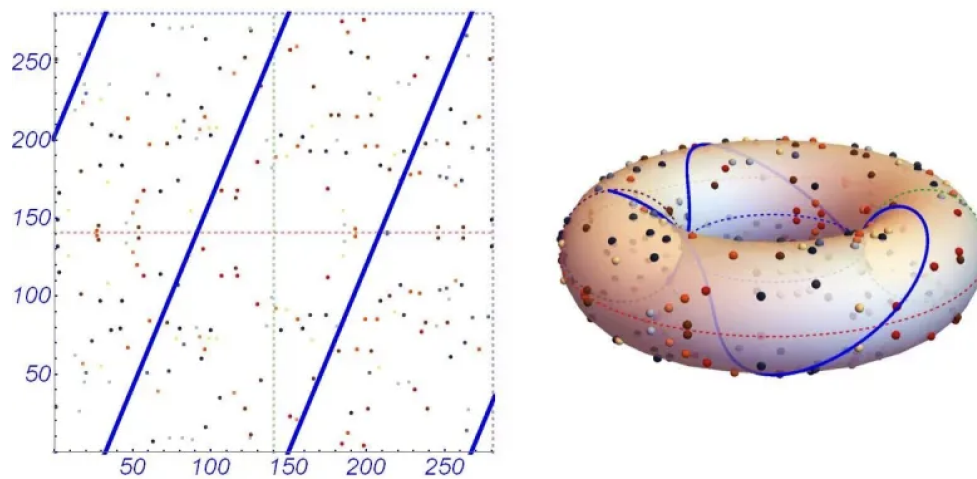


Figure 2: Elliptic Curve on a larger scale

On this set of points on the elliptic curve, we can define an additive (group) operator $+$ as such:

- Given any two different points P_1, P_2 on the elliptic curve, there is a third point $P_3 = P_1 + P_2$ on the elliptic curve. Geometrically, this third point P_3 is calculated by drawing a line between P_1 and P_2 ,

and this line will intersect the elliptic curve in exactly one additional place. Call this point $P'_3 = (x, y)$, and then reflect this in the x-axis to get $P_3 = (x, -y)$.

- If $P_1 = P_2$, then the line drawn is merely just the tangent line of the extension of the curve into the real plane at P_1 , and this line is guaranteed to intersect the curve in exactly one point.
- If P_1 or P_2 is the point at infinity, then it acts as the identity element. That is, $P + \infty = \infty + P = P$

It is also true that $+$ is associative in this group, so we can extend this to define multiplication as repeated addition.

$$kP = \underbrace{P + P + \cdots + P}_{k \text{ times}} \quad (4)$$

It turns out that multiplying (i.e., repeatedly adding) is easy to do, but to go the other way around is notoriously hard.

2 Base58 Encoding and WIF Format

Since this encoding occurs so often in the following sections, let us review it separately. Say that we have a string of hexadecimal characters, called the **payload**. Usually, payloads in hexadecimal are conventionally converted into a **Wallet Import Format (WIF)** using a method called **Base58Encode**.

Base58Check has the following features:

- An arbitrarily sized input.
- A set of 58 alphanumeric symbols consisting of easily distinguished uppercase and lowercase letters (00, II) are not used. The alphabet is:

```
1 123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

- Unlike a hash function, one can easily convert a number in any base from and to Base58Check.

The steps to encode it are shown:

- We determine a **version prefix** (in hex) for the payload that tells us what type of information it represents and attach the prefix onto the payload to get

```
1 Version Prefix + Payload
```

- We compute the **checksum** by taking the first four bytes (i.e., 8 hex digits) of the double-SHA of the Version Prefix + Payload and adding it to the end to get:

```
1 Version + Payload + Checksum = Version + Payload + SHA256(SHA256(Version Prefix + Payload))
```

- Then we convert the entire hex number into Base58, which has the alphabet ultimately producing

```
1 Base58(Version + A + Checksum)
```

Some common version prefixes and the prefixes they produce after Base58Check encoding:

Type	Version Prefix (hex)	Base58 Result Prefix
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K, or L
BIP38 Encrypted Private Key	0x0142	6P
BIP32 Extended Public Key	0x0488B21E	xpub

Table 1: Version Prefixes and Their Base58 Result Prefixes

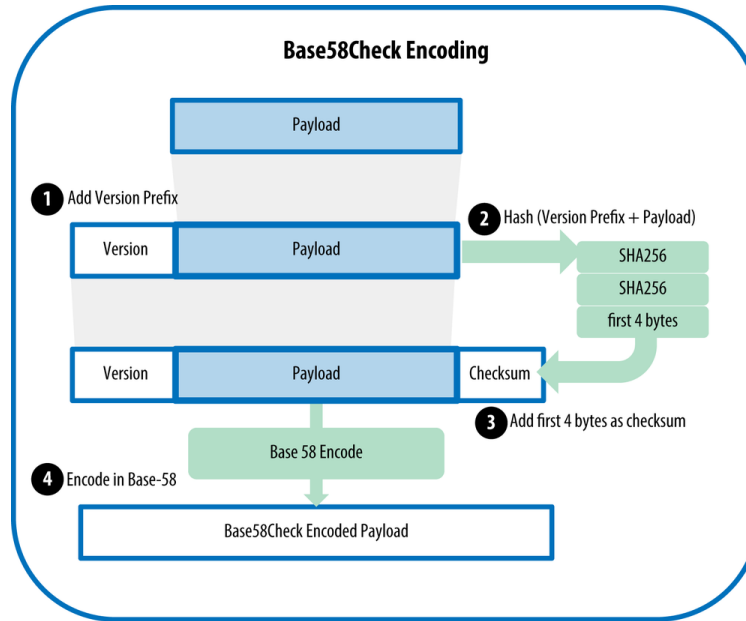


Figure 3: Base58Check Encoding Process

3 Bitcoin Keys and Addresses

In the early days of Bitcoin, it was possible to send payments to an IP-address like 104.25.248.32. This was planned to be a convenient method to use Bitcoins without dealing with public keys and addresses. However, after realizing that this process was vulnerable to man-in-the-middle attacks, the option was disabled and newer forms of addresses came out.

Fundamentally, a Bitcoin wallet is basically a file that contains a public-private key pair (K, k) following the Elliptic Curve Cryptography system. Our fundamental base will be in hexadecimal. Most generally, the steps are as such:

- The private key k is a randomly generated 64-digit hexadecimal number (256-bits). It is stored in one of two main forms, with WIF-compressed being the more recently used form:
 - **WIF-Uncompressed**: Base58Check with a version prefix of 0x80, resulting in Base58 Result prefix of 5.
 - **WIF-Compressed**: Base58Check with a version prefix of 0x10, resulting in Base58 Result prefix of K or L.
- The public key K can easily be computed by k using elliptic curve multiplication $K = k * G$ (which is irreversible), where k is the private key, G is a constant point (x_0, y_0) called the **generator point** (called this because it is the generating point of the entire group), and K is the resulting public key.

Now, depending on which WIF form that k is in, K is stored in one of two ways. Note that K is a pair of hexadecimal numbers.

- If k is WIF-Uncompressed, the pair of numbers $K = (K_1, K_2)$ are simply concatenated with the prefix 04

```
1 04 + K_1 + K_2
```

- If k is WIF-Compressed, only the first number K_1 is stored with the prefix being 02 if K_2 is odd or 03 if K_2 is even. This actually turns out to be a lossless form of compression, which is explained here.

```
1 K_2 even --> 02 + K_1
2 K_2 odd  --> 03 + K_1
```

- The **Bitcoin address** A is derived from the public key through the use of one-way cryptographic hashing with the functions **SHA256** and **RIPEMD160**. This composite function, called **SHA160**, produces a 160-bit/20-byte number, i.e. a 40-digit hexadecimal. Note that the compressed and uncompressed forms produce *different* bitcoin addresses!

```
1 A = RIPEMD160(SHA256(K)) = SHA160(K)
```

- This address A is almost always presented in WIF using Base58Check.

For bitcoin, the specific elliptic curve that is used is referred to as **secp256k1**, and the generator/base point is always the same for all bitcoin addresses. It has hexadecimal coordinates:

```
1 x = 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
2 y = 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8
```

which in compressed form is

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

and uncompressed form is

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

The diagram below shows a comprehensive outline of the key-producing processes, which has two paths (for when a WIF-uncompressed or a WIF-compressed private key is created).

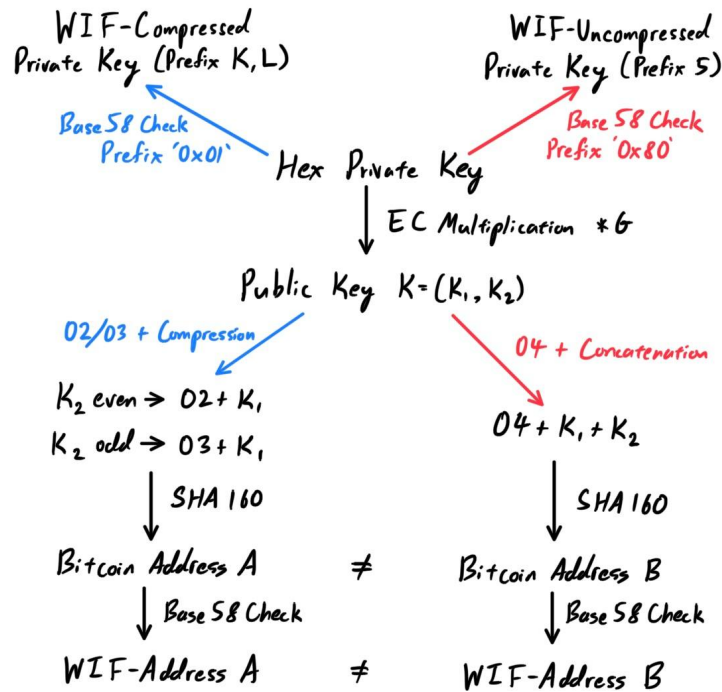


Figure 4: WIF Compressed and Uncompressed Comparison

4 Cryptocurrency Wallets

Wallets are containers for private keys, usually implemented as structured files or databases. One common misconception is that Bitcoin wallets contain coins, but that's not true at all. Wallets contain just the keys, not coins. Each user has a wallet containing keys. Wallets are really keychains containing pairs of private/public keys, and users sign transactions with the keys, thereby proving they own the transaction outputs (their coins). The coins are stored on the blockchain in the form of transaction-outputs (often noted as vout or txout). There are two types of wallets:

- **Nondeterministic (Random) wallets** are simply collections of randomly generated private keys, but they are cumbersome to manage, back up, and import, especially if you want to avoid address re-use because that means managing many keys, which creates the need for frequent backups.
- **Deterministic (Seeded) wallets** are wallets that contain private keys that are all derived from a common seed, through the use of a one-way hash function from a previous private key, linking them in sequence. As long as you can re-create that sequence, you only need the first key (known as the *seed* or *master key*) to generate them all. The seed is also sufficient for a wallet export or import, allowing for easy migration of all the user's keys between different wallet implementations.

4.1 Mnemonic Code Words

To make storing these words easier, we use **mnemonic code words** (a sequence of 12-24 English words) to encode the seed. The mnemonic code represents 128 to 256 bits, which are used to derive a longer (512-bit) seed through the use of the key-stretching function PBKDF2. The resulting seed is used to create a deterministic wallet and all of its derived keys.

	128-bit	256-bit
Entropy Input	0c1e24e5917779d29 7e14d45f14e1a1a	2041546864449caf f939d32d574753 fe684d3c947c33467 13dd8423e74abcf8c
Mnemonic	army van defense carry jealous true garbage claim echo media make crunch	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Seed	3338a6d2ee71c7f28eb5b88215963 4cd46a898463e9d2d0980f8e80dfb ba5b0fa0291e5fb888a599b44b931 87be6ee3ab5fd3ead7dd646341b2c db8d08d13bf7	3972e432e99040f75ebe13a660110 c3e29d131a2c808c7ee5f1631d0a9 77fcf473bee22fce540af281bf7cd eade0dd2c1c795bd02f1e4049e205 a0158906c343

Table 2: Examples of Mnemonic Code Words

4.2 Hierarchical Deterministic Wallets (BIP0032/BIP0044)

The most advanced form of deterministic wallets is the **hierarchical deterministic wallet**, or **HD wallet**, defined by the BIP0032 standard. Hierarchical deterministic wallets contain keys derived in a tree structure, such that a parent key can derive a sequence of children keys, each of which can derive a sequence of grandchildren keys, and so on, to an infinite depth. This type of wallet is more specifically called a **Type-2 HD wallet**.

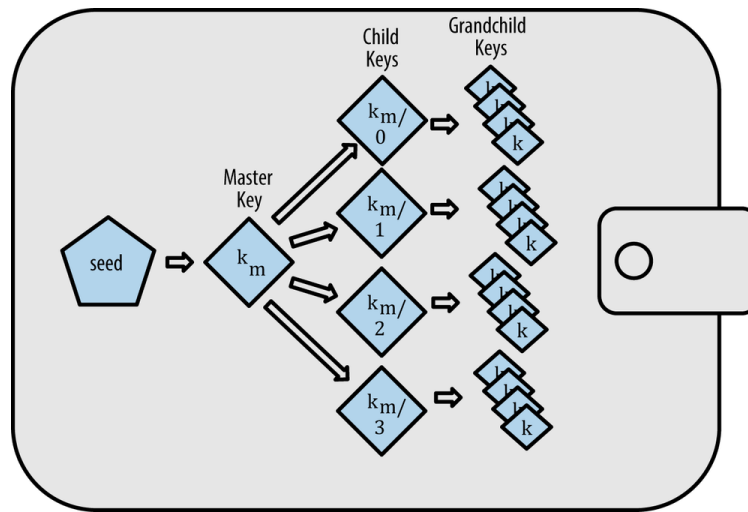


Figure 5: Hierarchical Deterministic Wallets

These wallets give both more of an organizational structure to the wallet. Additionally, users can create a sequence of public keys without having access to the corresponding private keys. This allows HD wallets to be used on an insecure server or in a receive-only capacity, issuing a different public key for each transaction.

HD wallets are created from a single *root seed*, which is a 128, 256, or 512-bit random number. The mnemonic code words are derived for the user to back up, and the root seed is inputted into the HMAC-SHA512 algorithm to output the **master private key m** (which itself produces the **master public key M** using normal elliptic curve multiplication $m * G$) and a **master chain code**, which is used to introduce entropy in the function that creates child keys from parent keys.

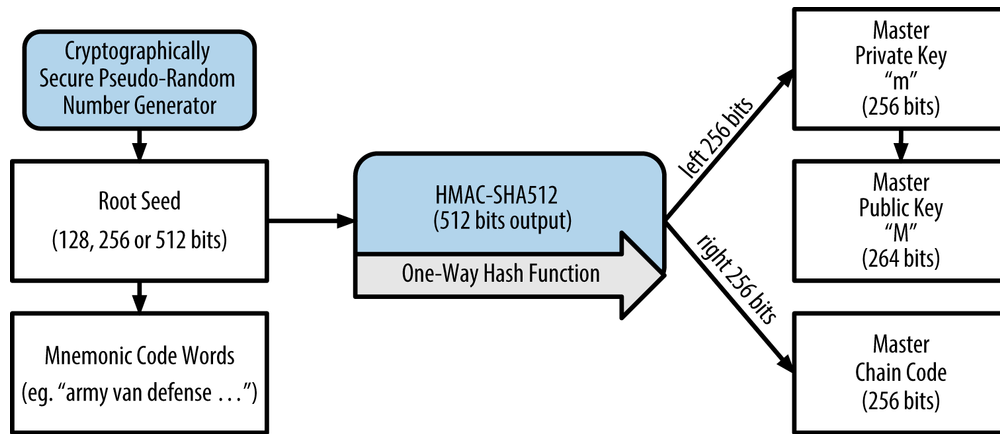


Figure 6: HD Wallet Creation Process

Hierarchical deterministic wallets use a **child key derivation (CKD)** function to derive children keys from parent keys. It takes in the inputs:

- A parent key, which could be private or public.
- A seed called a chain code (256 bits)
- An index number (32 bits) which represents the i th children of a parent (e.g. Child 0, Child 1, Child 2, etc.)

and outputs a 512-bit hash, which is really just a concatenation of the child 256-bit key and a 256-bit chain code, called an **extended key**. We can represent the hash mathematically as a function CKD where

$$1 \quad \text{CKD}(\text{parent key}, \text{parent chain code}, \text{index}) = (\text{child key}, \text{child chain code})$$

Let us concentrate on the parent key input for the CKD function. One of the most important things about CKD is that it suffices the commutative diagram below, where k represents a private key, K represents a public key, $*$ represents the normal elliptic multiplication that generates a public from a private key, and CKD is the child key derivation function.

$$\begin{array}{ccc} k_{\text{parent}} & \xrightarrow{CKD} & k_{\text{child}} \\ \downarrow * & & \downarrow * \\ K_{\text{parent}} & \xrightarrow{CKD} & K_{\text{child}} \end{array}$$

Figure 7: CKD Commutative Diagram

This means that inputting the private parent key will generate a private child key, and a public child key will generate a public child key. More specifically, there are two types of extended keys.

- An **extended public key** is the concatenation of a public key and a chain code, which is generated by inputting a parent public key, the chain code, and the index. Note that you can't generate any private keys with the extended public key; only public keys. This ability to derive public child keys from public parent keys *without* having the private keys is especially useful since it gives us two ways to derive a child public key:
 - from the child private key, or

- directly from the parent public key

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

One common application of this solution is to install an extended public key on a web server that serves an ecommerce application. The web server can use the public key derivation function to create a new bitcoin address for every transaction (e.g., for a customer shopping cart). The web server will not have any private keys that would be vulnerable to theft. Another common application of this solution is for cold-storage or hardware wallets. In that scenario, the extended private key can be stored on a paper wallet or hardware device (such as a Trezor hardware wallet), while the extended public key can be kept online.

- An **extended private key** is the concatenation of a private key and a chain code. This allows you to generate both the private and public keys of all future generations, giving you complete control of future wallets.

4.3 Extended Child Key Derivation - Non-Hardened Extended Private Key

The diagram below shows the generation of an extended child private key with an extended child private key. Let us walk through the steps:

- The parent private key and the parent chain code are the inputs.
- The parent public key is generated through elliptic curve addition from the parent private key.
- An index between 0 and $2^{31} - 1$ is chosen and concatenated to the public key. We now have the parent chain code and the

`1 parent public key + index`

- The parent chain code and the **parent public key+index** is put in through the **HMAC-SHA512** function to generate a 512-bit output.
 - The new chain code is the last 256 bits of the result from the **HMAC**.
 - The new private key is the first 256 bits of the result from the **HMAC** added to the original private key. This essentially just takes the original private key and increases it by a random 32-byte number (mod whatever the order of the field in which the elliptic curve exists).
- Notice how even though the initial inputs are the parent private key and the parent chain code, the child chain code is really dependent upon the parent public key. Furthermore, the security of the child private key comes from adding the randomly-produced 32-byte number (produced by the parent public key and chain code) to the unknown parent private key.

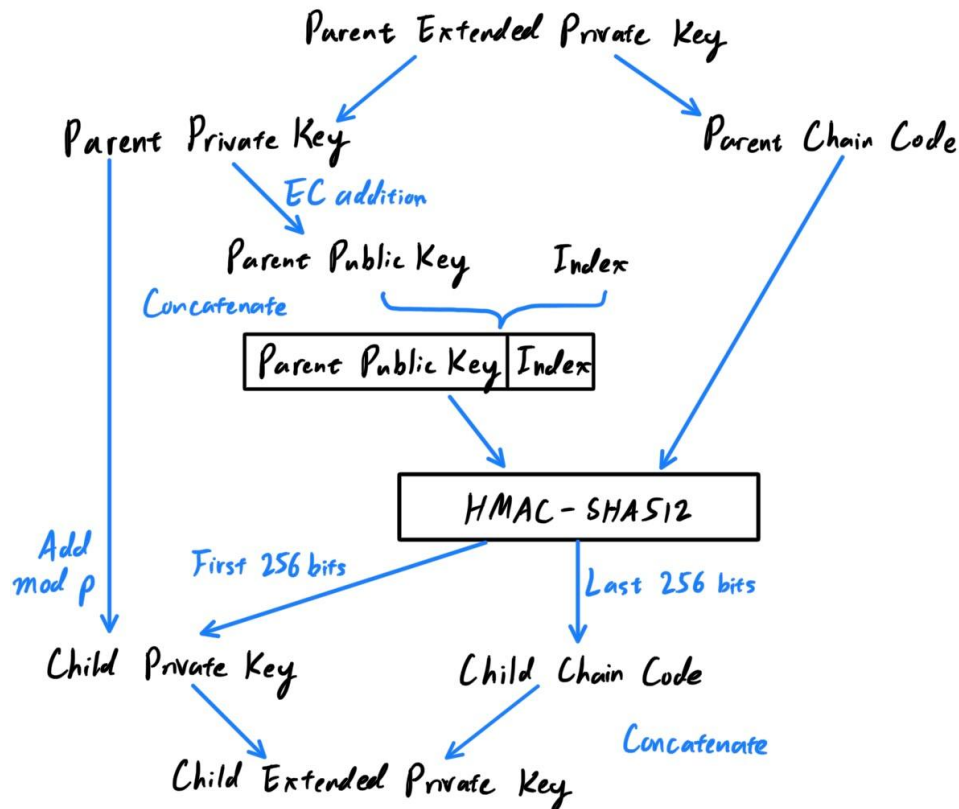


Figure 8: Non-Hardened Extended Private Key Derivation

4.4 Extended Child Key Derivation - Non-Hardened Extended Public Key

The diagram below shows the generation of an extended child public key with an extended child public key. This is really just a sub-diagram of the one above.

- The parent public key and the parent chain code are the inputs.
- An index between 0 and $2^{31} - 1$ is chosen and concatenated to the public key. We now have the parent chain code and the

```
1 parent public key + index
```

- The parent chain code and the parent public key+index is put in through the HMAC-SHA512 function to generate a 512-bit output.
 - The new chain code is the last 256 bits of the result from the HMAC.
 - Note that a public key in ECC is really a point (x, y) . We first take the first 256 bits from the HMAC, multiply it by the generator point G to get some point in \mathbb{R}^2 , and finally use elliptic curve addition to get the point

```
1 Parent Public Key + (First 256 bits of HMAC * G)
```

which is the new child public key.

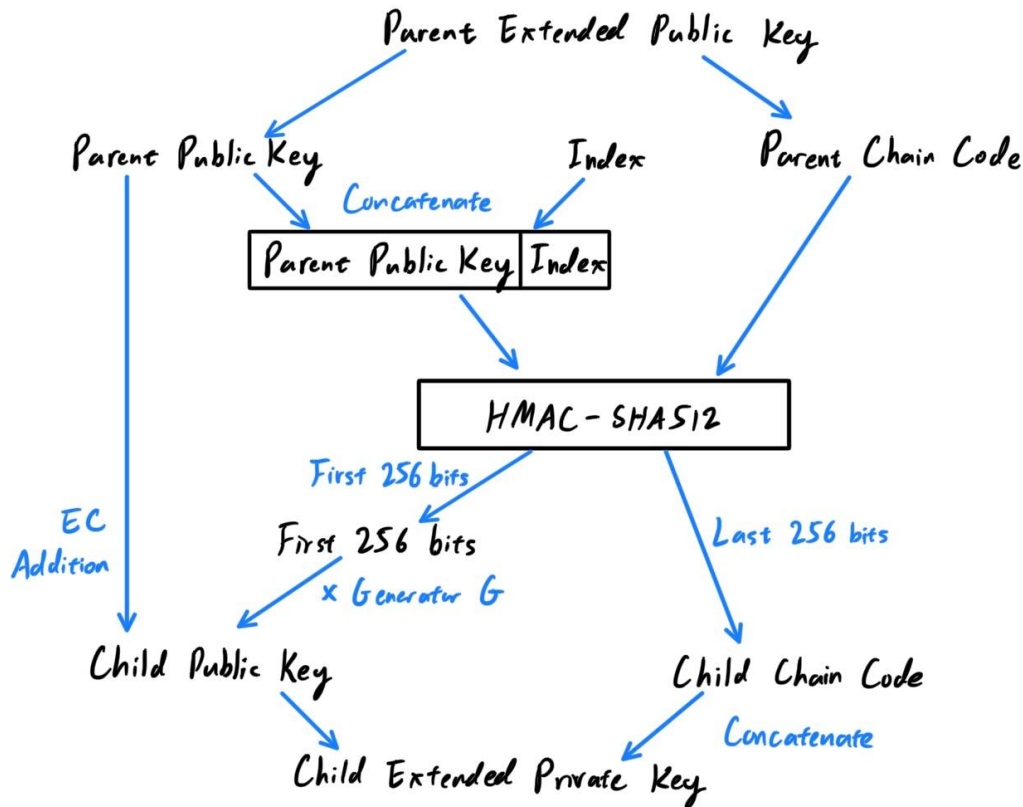


Figure 9: Non-Hardened Extended Public Key Derivation

A few more points to quickly hit:

- Child private keys are indistinguishable from nondeterministic random keys. Because the derivation function is a one-way function, the child key cannot be used to find a parent key nor any of its siblings since this requires you to backtrack across the hash function.
- The child private key alone can be used to make a public key and thus a bitcoin address. Then, it can be used to sign transactions to spend anything paid to that address, which is still a lot of power. The lack of the chain code just restricts you from making *new* keys.
- A child private key, the corresponding public key, and the bitcoin address are all indistinguishable from keys and addresses created randomly. The fact that they are part of a sequence is not visible, outside of the HD wallet function that created them. Once created, they operate exactly as "normal" keys.
- The initial chain code seed (at the root of the tree) is made from random data, while subsequent chain codes are derived from each parent chain code.

4.5 Extended Child Key Derivation - Hardened Extended Private Key

The ability to derive a branch of public keys from an extended public key is very useful, but it comes with a potential risk. Access to an extended public key does not give access to child private keys. However, because the extended public key contains the chain code, if a child private key is known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children. Worse, the child private key together with a parent chain code (which can be found in the extended parent public key) can be used to deduce the parent private key, quite easily, in fact.

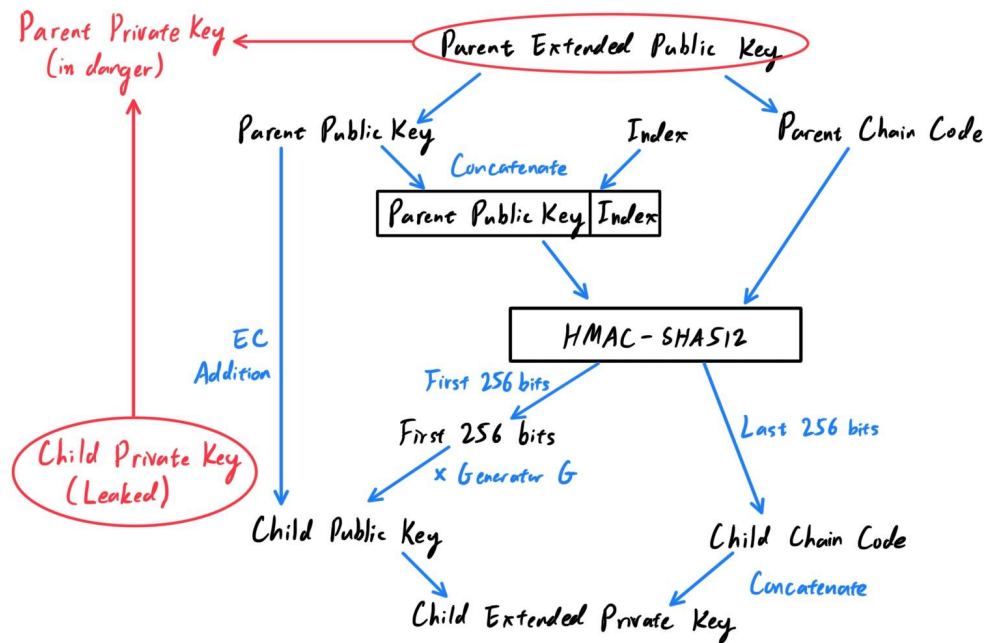


Figure 10: Parent Private Key Compromise Scenario

The risk is that with this information, a hostile party could have access one level up the wallet tree (but not beyond), allowing outsiders to spend the bitcoins at will. To prevent this, HD wallets use an alternative derivation function called **hardened derivation**, which breaks the relationship between the parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a "firewall" in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key.

The diagram below shows the generation of a hardened extended child private key with an extended child public key. This is really just a sub-diagram of the one above.

- The parent private key and the parent chain code are the inputs.
- An index between 2^{31} and $2^{32} - 1$ is chosen and concatenated to the private key (note that unlike the non-hardened version, we do not convert the private key into a public key first). We now have the parent chain code and the

1	parent private key + index
---	----------------------------

- The parent chain code and the parent public key+index is put in through the HMAC-SHA512 function to generate a 512-bit output.
 - The new chain code is the last 256 bits of the result from the HMAC.
 - The new private key is the first 256 bits of the result from the HMAC added to the original private key. This again just takes the original private key and increases it by a random 256-bit number.
- Note that this hardened child key was constructed by putting the private key into the HMAC function (which an extended public key does not have access to), which means that child extended private keys derived this way will have a public key that cannot be derived by a corresponding extended public key.

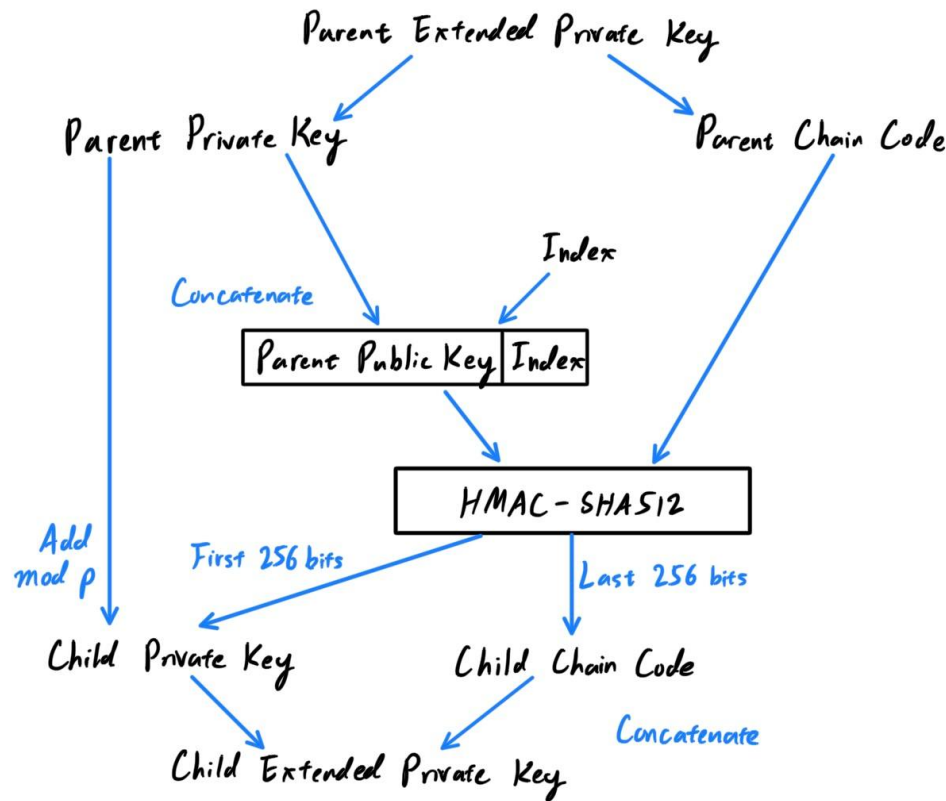


Figure 11: Hardened Extended Private Key Derivation

When the hardened private derivation function is used, the resulting child private key and chain code are completely different from what would result from the normal derivation function. The resulting "branch" of keys can be used to produce extended public keys that are not vulnerable, because the chain code they contain cannot be exploited to reveal any private keys. Hardened derivation is therefore used to create a "gap" in the tree above the level where extended public keys are used. In simple terms, if you want to use the convenience of an extended public key to derive branches of public keys, without exposing yourself to the risk of a leaked chain code, you should derive it from a hardened parent, rather than a normal parent. As a best practice, the level-1 children of the master keys are always derived through the hardened derivation, to prevent compromise of the master keys.

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

4.6 HD Wallet Key Identifier (Path)

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

The "ancestry" of a key is read from right to left, until you reach the master key from which it was derived.

For example, identifier $m/x/y/z$ describes the key that is the z th child of key $m/x/y$, which is the y th child of key m/x , which is the x th child of m . Some examples of paths are shown in the table.

HD Path	Key Described
$m/0$	The first (0) child private key from the master private key (m)
$m/0/0$	The first grandchild private key of the first child ($m/0$)
$m/0'/0$	The first normal grandchild of the first hardened child ($m/0'$)
$M/23/17/0/0$	The first great-great-grandchild public key of the first great-grandchild of the 18th grandchild of the 24th

Table 3: Examples of HD Wallet Key Paths

The HD wallet tree structure offers tremendous flexibility. Each parent extended key can have 4 billion children: 2 billion normal children and 2 billion hardened children. Each of those children can have another 4 billion children, and so on. The tree can be as deep as you want, with an infinite number of generations. With all that flexibility, however, it becomes quite difficult to navigate this infinite tree. It is especially difficult to transfer HD wallets between implementations, because the possibilities for internal organization into branches and subbranches are endless. Two Bitcoin Improvement Proposals (BIPs) offer a solution to this complexity by creating some proposed standards for the structure of HD wallet trees, but we will not get into them here.

5 Advanced Keys and Addresses

5.1 Encrypted Private Keys (BIP0038)

For more advanced security without expending too much accessibility, BIP0038 proposes a common standard for encrypting private keys (based off of AES encryption) that uses an extra password to further encrypt a WIF-formatted private key. For example, the WIF-Uncompressed private key with a prefix of 5 would be encrypted to a Base58Check-encoded encrypted private key with prefix 6P. Therefore, a key starting with 6P means that a password must be needed to decrypt it. The most common use case for BIP0038 encrypted keys is for paper wallets that can be used to back up private keys on a piece of paper. As long as the user selects a strong passphrase, a paper wallet with BIP0038 encrypted private keys is incredibly secure and a great way to create offline bitcoin storage (also known as "cold storage").

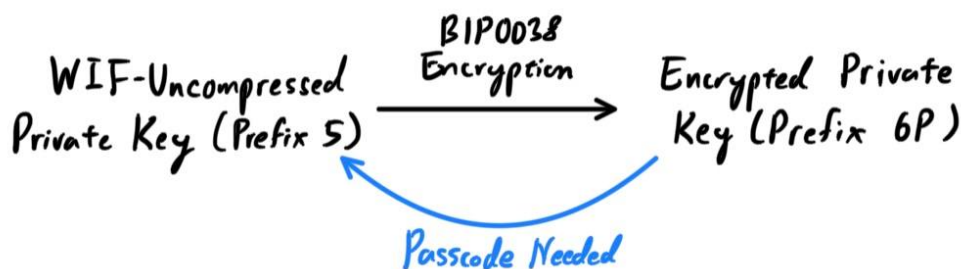


Figure 12: BIP38 Encryption Process

5.2 Vanity Addresses

Vanity addresses are valid bitcoin addresses that contain human-readable messages. For example,

```
1 LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33
```

contains the letters forming the word "Love" as the first four Base58 letters. Vanity addresses require generating and testing billions of random candidate private keys, until one derives a bitcoin address with the

desired pattern. Vanity addresses are no less or more secure than any other address. You can no more easily find the private key of an address starting with a vanity pattern than you can any other address. Looking at the pattern "KidsCharity", we can approximate how frequently we might find this pattern in a bitcoin address in the figure below. An average desktop PC without specialized hardware can search approximately 100,000 keys per second.

Length	Pattern	Frequency	Average Search Time
1	1K	1 in 58 keys	1 millisecond
2	1Ki	1 in 3,364	50 milliseconds
3	1Kid	1 in 195,000	2 seconds
4	1Kids	1 in 11 million	1 minute
5	1KidsC	1 in 656 million	1 hour
6	1KidsCh	1 in 38 billion	2 days
7	1KidsCha	1 in 2.2 trillion	3-4 months
8	1KidsChar	1 in 128 trillion	13-18 years
9	1KidsChari	1 in 7 quadrillion	800 years
10	1KidsCharit	1 in 400 quadrillion	46,000 years
11	1KidsCharity	1 in 23 quintillion	2.5 million years

Table 4: Frequency and Search Time for Vanity Addresses

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

5.3 Paper Wallets

Paper wallets are basically bitcoin private keys printed on paper, effective for cold storage due to its immunity from hacking. Often the paper wallet also includes the corresponding bitcoin address for convenience, but this is not necessary because it can be derived from the private key. They can be generated in this website. The simplest form of a paper wallet is the table below.

Public Address	1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x
Private Key (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

Table 5: Simple Paper Wallet

6 Transaction Chains and UTXOs

Fundamentally, a **transaction** is a data structure that encodes a transfer of value from a source of funds, called an **input**, to a destination, called an **output**. A transaction contains the following:

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

Table 6: Transaction Structure

Unspent transaction outputs, or **UTXOs**, are indivisible chunks of bitcoin currency locked to a specific owner, recorded on the blockchain, and recognized as currency units by the entire network. The bitcoin network tracks all available (unspent) UTXO currently numbering in the millions, with each user's bitcoin scattered as UTXO amongst hundreds of transactions and hundreds of blocks. In effect, there is no such thing as a stored balance of a bitcoin address or account; there are only scattered UTXO, locked to specific owners. Note that even though UTXOs can be any arbitrary value, once created it is indivisible just like a coin that cannot be cut in half. If a UTXO is larger than the desired value of a transaction, it must still be consumed in its entirety and change must be generated in the transaction. For example, if you have a 20 BTC UTXO and want to pay 1 bitcoin, your transaction must consume the entire 20 bitcoin UTXO and produce two outputs: one paying 1 bitcoin to your desired recipient and another paying 19 bitcoin in change back to your wallet. All this is automatically done by your bitcoin wallet. The diagram shows a sequence of (three) transactions.

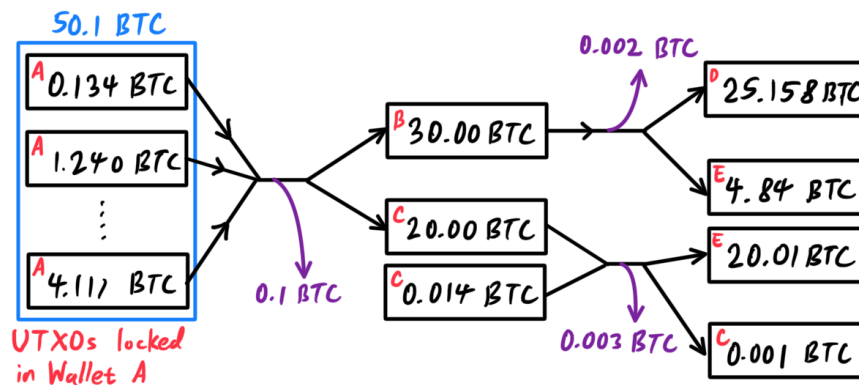


Figure 13: Sequence of Bitcoin Transactions

The misleading concept of a user's *bitcoin balance* is derived from the use of the bitcoin *wallet*, but it is really the case that the wallet calculates the user's balance by scanning the blockchain and aggregating all UTXOs belonging to that user. This system allows chunks of bitcoin to move from owner to owner in a chain of transactions consuming and creating UTXO. Transactions consume UTXO by unlocking it with a signature of the current owner and create UTXO by locking it to the bitcoin address of the new owner. The exception to the output and input chain is a special type of transaction called the **coinbase** transaction, which is the first transaction in each block. This transaction is placed there by the "winning" miner and creates brand-new bitcoin payable to that miner as a reward for mining. This is how bitcoin's money supply is created during the mining process.

6.1 Transaction Fees

Furthermore, **transaction fees** exist, which serve as an incentive to include (mine) a transaction into the next block and also as a disincentive as "spam" transactions of any kind of abuse of the system, by imposing

a small cost on every transaction. Transaction fees are calculated based on the *size of the transaction in kilobytes* (by multiplying the size of the transaction in kb by the per kilobyte fee), not the value of the transaction in bitcoin and are collected by the miner who mines the block that records the transaction on the blockchain. Miners prioritize transactions based on many different criteria, including fees, and might even process transactions for free under certain circumstances. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next-most-mined block, whereas a transaction with insufficient or no fees might be delayed, processed on a best-effort basis after a few blocks, or not processed at all. Transaction fees are not mandatory, and transactions without fees might be processed eventually; however, including transaction fees encourages priority processing.

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

$$1 \quad \text{Fees} = \text{Sum(Inputs)} - \text{Sum(Outputs)}$$

This is a somewhat confusing element of transactions and an important point to understand, because if you are constructing your own transactions you must ensure you do not inadvertently include a very large fee by underspending the inputs. That means that you must account for all inputs, if necessary by creating change, or you will end up giving the miners a very big tip! For example, if you consume a 20-bitcoin UTXO to make a 1-bitcoin payment, you must include a 19-bitcoin change output back to your wallet. Otherwise, the 19-bitcoin "leftover" will be counted as a transaction fee and will be collected by the miner who mines your transaction in a block.

6.2 Transaction Chaining and Orphan Transactions

As we have seen, transactions form a chain, whereby one transaction spends the outputs of the previous transaction (known as the parent) and creates outputs for a subsequent transaction (known as the child). Sometimes an entire chain of transactions depending on each other—say a parent, child, and grandchild transaction—are created at the same time, to fulfill a complex transactional workflow.

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

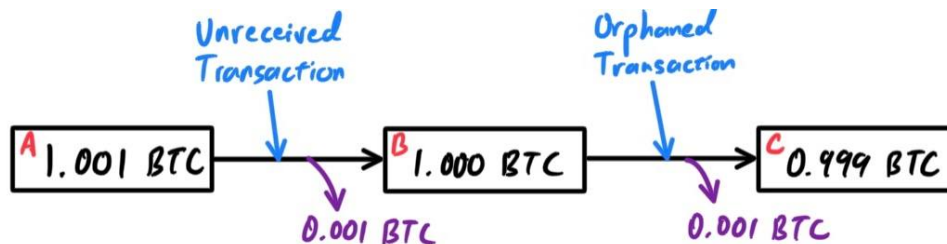


Figure 14: Orphaned Transaction

There is a limit to the number of orphan transactions stored in memory, to prevent a denial-of-service attack

against bitcoin nodes. The limit is defined as `MAX_ORPHAN_TRANSACTIONS` in the source code of the bitcoin reference client. If the number of orphan transactions in the pool exceeds `MAX_ORPHAN_TRANSACTIONS`, one or more randomly selected orphan transactions are evicted from the pool, until the pool size is back within limits.

7 Locking/Unlocking Scripts and Script Language

Bitcoin clients validate transactions by executing the following two types of scripts, which are written in a Forth-like stack-based scripting language called **Script**. Note that the bitcoin transaction script language Script contains many operators, but is deliberately limited in that there are no loops or complex flow control capabilities, making it Turing incomplete.

- A **locking script**, also called a `scriptPubKey` is an encumbrance placed on an output, and it specifies the conditions that must be met to spend the output in the future.
- An **unlocking script**, also called `scriptSig`, is a script that solves/satisfies the conditions placed on an output by a locking script and allows the output to be spent. Unlocking scripts are part of every transaction input, and most of the time they contain a digital signature produced by the user's wallet from his or her private key.

For each input in the transaction, the validation software will first retrieve the UTXO referenced by the input. That UTXO contains a locking script defining the conditions required to spend it. The validation software will then take the unlocking script contained in the input that is attempting to spend this UTXO and execute the two scripts. Let's take an example of user C in the diagram above. Say that C wants to send 20.01BTC to user E. It first creates a transaction of 20.01 BTC from C → E (but does not validate it yet). The software must first confirm that C has access to enough BTC to pay E this much BTC (if there are insufficient funds, the transaction is rejected by the blockchain), so C's wallet software searches through the blockchain and finds a 20BTC UTXO (created from the conglomeration of outputs from users A₁₂₂ to A₂₄₁) and a 0.014BTC UTXO (created from the output from user F). Both the 20BTC and 0.014BTC UTXO contain a locking script, but C's wallet contains the credentials, i.e. the unlocking script, that takes the two locked UTXOs and unlocks it for use. C pays E in a 20.01BTC locked UTXO (unlockable only by E's wallet signature) and sends the "change" back to C's own wallet in the form of a 0.001BTC locked UTXO (unlockable only by C's wallet signature).

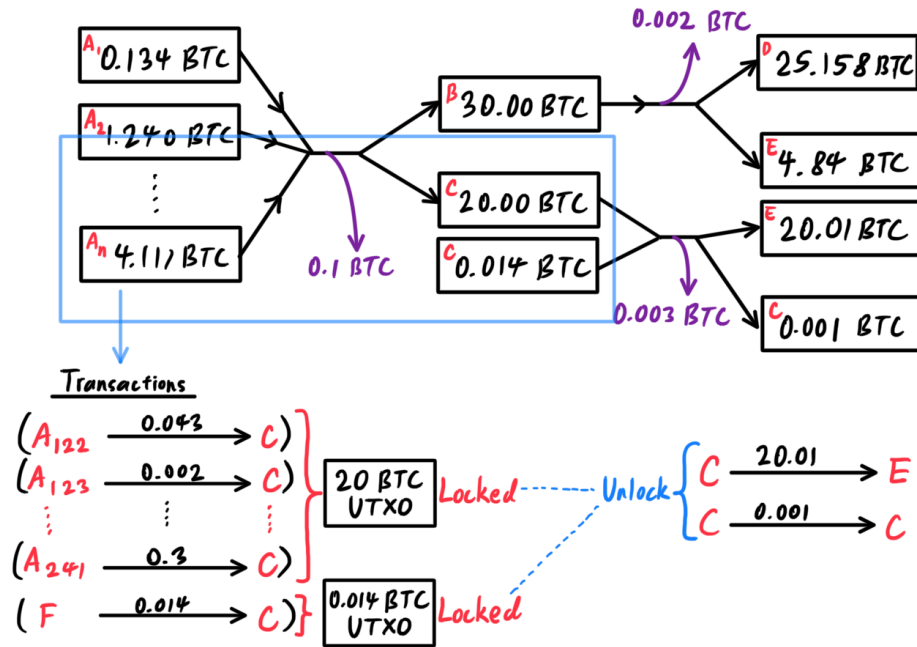


Figure 15: UTXO Transaction Example

7.1 Script Language

Bitcoin's scripting language is called a stack-based language because it uses a very simple data structure called a **stack** (which can be visualized as a stack of cards) with two operations:

- *push* adds an item on top of the stack
- *pop* removes the top item from the stack

The scripting language executes the script by processing each item from left to right. **Numbers** (data constants) are pushed onto the stack, and **operators** push/pop one or more parameters from the stack, act on them, and might push a result onto the stack. Three basic operators are:

- **OP_ADD** pops two items from the stack, adds them, and push the resulting sum onto the stack.
- **OP_SUB** pops two items from the stack, subtracts the second from the first number, and pushes the resulting difference onto the stack.
- **OP_EQUAL** is a conditional operator that pops two items from the stack and pushes **TRUE** (represented by 1) if they are equal or **FALSE** (represented by 0) if they are not equal.

For example, the following script is just a simple way of determining whether $2 + 7 - 3 + 1 = 7$. It should output **TRUE**.

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

8 Standard Transactions

Developers of bitcoin introduced some limitations in the types of scripts that could be processed by the reference client, encoded in a function called `isStandard()` defining 5 types of "standard" transactions:

- Pay to Public Key Hash (P2PKH)

- Public Key
- Multi-Signature (limited to 15 keys)
- Pay to Script Hash (P2SH)
- Data Output (OP_RETURN)

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

The vast majority of transactions processed on the bitcoin network are P2PKH transactions. In P2PKH transactions, the locking scripts that restrict these UTXOs are hashes of public keys of the recipient, i.e. the recipient address ($A = \text{SHA160}(K)$). The locking script is of the form:

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

These "locks" can be unlocked by the unlocking script, which is the pair consisting of:

- the recipient signature (which can only be produced by the recipient private key), and
- the recipient public key

The unlocking script is of the form:

```
1 <Recipient Signature> <Recipient Public Key>
```

The two scripts together would form the combined validation script:

```
1 <Recipient Signature> <Recipient Public Key> OP_DUP OP_HASH160 <Recipient Public Key Hash>
  OP_EQUAL OP_CHECKSIG
```

When executed, this combined script will evaluate to **TRUE** if and only if the unlocking script matches the conditions set by the locking script, i.e. if the unlocking script has a valid signature from the cafe's private key that corresponds to the public key hash set as an encumbrance. The step-by-step execution of the combined script

```
1 <sig> <PubK> DUP HASH160 <PubKHash> EQUALVERIFY CHECKSIG
```

is described below:

- The value **<sig>** is pushed to the top of the stack.
- The execution continues, and the value **<PubK>** is pushed to the top of the stack, on top of **<sig>**
- The **DUP** operator duplicates the top item in the stack, and the resulting value is pushed to the top of the stack.
- The **HASH160** operator hashes the top item in the stack with **HASH160(PubK)**. The resulting value **<PubKHash>** is pushed to the top of the stack.
- The value **PubKHash** from the script is pushed on top of the value **PubKHash** calculated previously from the **HASH160** of the **PubK**.
- The **EQUALVERIFY** operator compares the **PubKHash** encumbering the transaction with the **PubKHash** calculated from the user's **PubK**. If they match, both are removed and execution continues.
- The **CHECKSIG** operator checks that the signature **<sig>** matches the public key **<PubK>** and pushes **TRUE** to the top of the stack if true.

8.2 Pay-to-Public-Key (P2PK)

Pay-to-public key is a simpler form of bitcoin payment than P2PKH. With this script form, the public key itself is stored in the locking scripts rather than the public-key hash as with P2PKH. P2PK was invented by Satoshi to make bitcoin addresses shorter for ease of use, but is mainly outdated.

The locking script is of form:

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

and the corresponding unlock script is of form:

```
1  <Signature from Private Key>
```

The two scripts together would form the combined validation script:

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

The step-by-step execution of the combined script

```
1  <sig> <PubK> CHECKSIG
```

is described below:

- The value **<sig>** is pushed to the top of the stack.
- The execution continues, and the value **<PubK>** is pushed to the top of the stack, on top of **<sig>**
- The **CHECKSIG** operator checks that the signature **<sig>** matches the public key **<PubK>** and pushes **TRUE** to the top of the stack if true.

8.3 Multi-Signature

Multi-signature scripts set a condition where N public keys are recorded in the script and at least M of those scripts must provide signatures to release the encumbrance, known as an **M-of-N scheme**. Note that there is an inherent limit on how many public keys N there can be in a multisig script (with clearly $M < N$). The general form of a M -of- N multisig locking script is:

```
1  M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_CHECKMULTISIG
```

With an unlocking script of form:

```
1  OP_0 <Signature 1> ... <Signature M>
```

For example, the locking and unlocking scripts together of a 2-to-3 multisig scheme is of form

```
1  OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3
   OP_CHECKMULTISIG
```

When executed, this combined script will evaluate to **TRUE** if and only if the unlocking script matches the conditions set by the locking script.

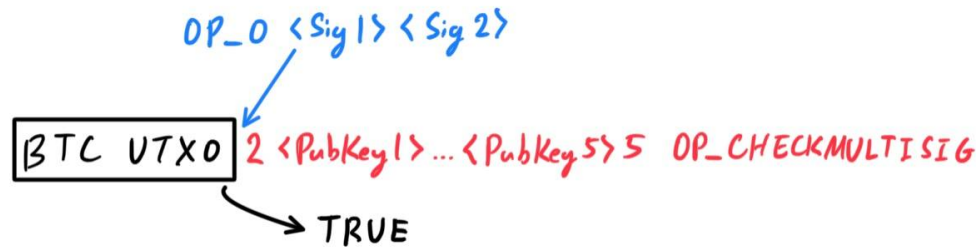


Figure 16: Multi-Signature Scheme

8.4 Data Output (OP_RETURN)

Bitcoin's distributed and timestamp ledger, the blockchain, has potential uses far beyond payments, such as digital notary services, stock certificates, and smart contracts. Early attempts to use bitcoin's script language for these purposes involved creating transaction outputs that recorded data on the blockchain; for example, to record a digital fingerprint of a file in such a way that anyone could establish proof-of-existence of that file on a specific date by reference to that transaction.

The use of bitcoin's blockchain to store data unrelated to bitcoin payments is a controversial subject, since the inclusion of non-payment data in bitcoin's blockchain causes a "blockchain bloat", burdening those running full bitcoin nodes with carrying the cost of disk storage for data that the blockchain was not intended to carry. Moreover, such transactions create UTXO that cannot be spent, using the destination bitcoin address as a free-form 20-byte field. Because the address is used for data, it doesn't correspond to a private key and the resulting UTXO can never be spent; it's a fake payment. This practice causes the size of the in-memory UTXO set to increase and these transactions that can never be spent are therefore never removed, forcing bitcoin nodes to carry these forever in RAM, which is far more expensive.

In version 0.9 of the Bitcoin Core client, a compromise was reached with the introduction of the `OP_RETURN` operator. `OP_RETURN` allows developers to add 40 bytes of nonpayment data to a transaction output. However, unlike the use of "fake" UTXO, the `OP_RETURN` operator creates an explicitly provably unspendable output, which does not need to be stored in the UTXO set. `OP_RETURN` outputs are recorded on the blockchain, so they consume disk space and contribute to the increase in the blockchain's size, but they are not stored in the UTXO set and therefore do not bloat the UTXO memory pool and burden full nodes with the cost of more expensive RAM. `OP_RETURN` scripts look like this:

```
1 OP_RETURN <data>
```

The data portion is limited to 40 bytes and most often represents a hash, such as the output from the `SHA256` algorithm (32 bytes). Many applications put a prefix in front of the data to help identify the application. For example, the Proof of Existence digital notarization service uses the 8-byte prefix `DOCPROOF`, which is ASCII encoded as `44f4350524f4f46` in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to `OP_RETURN` that could possibly be used to "spend" an `OP_RETURN` output. The whole point of `OP_RETURN` is that you can't spend the money locked in that output, and therefore it does not need to be held in the UTXO set as potentially spendable—`OP_RETURN` is provably unspendable. `OP_RETURN` is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If an `OP_RETURN` is encountered by the script validation software, it results immediately in halting the execution of the validation script and marking the transaction as invalid. Thus, if you accidentally reference an `OP_RETURN` output as an input in a transaction, that transaction is invalid.

8.5 Pay-to-Script-Hash (P2SH)

To motivate this example, let us first describe some limitations of the P2PK multi-sig. Say that my company uses a multi-sig script for all customer payments, meaning that any payments made by customers are locked in such a way that they require at least 2 signatures to release, from me and one of my partners/attorney. This kind of scheme offers corporate governance controls and protects against theft, embezzlement, or loss. To give an example, say that there is a transaction from customer A \rightarrow Company that produces a UTXO of 1BTC. This UTXO would of course be locked by a multi-sig script of form

```
1 2 <my PubK> <Partner1 PubK> <Partner2 PubK> <Partner3 PubK> <Attorney PubK> 5 OP_CHECKMULTISIG
```

Even though multi-sig scripts are a powerful feature, they are cumbersome to use, since I would have to communicate this script to every customer prior to payment, each customer would have to create a special bitcoin wallet software, and the transaction sizes would be much larger (with more public keys added to the locking script). The burden of that extra-large transaction would be borne by the customer in the form of fees. Finally, a large transaction script like this would be carried in the UTXO set in RAM in every full node until it was spent. To resolve these issues, pay-to-script-hash (P2SH) was developed.

The key characteristic of P2SH payments is that the complex locking script (in this context referred as the **redeem script**) is replaced with its digital fingerprint, a cryptographic hash. When a transaction attempting to spend the UTXO is presented later, it must contain the script that matches the hash, in addition to the unlocking script. The following table shows a complex script without P2SH

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 7: Complex Script without P2SH

while the next table shows a complex script as P2SH:

Redeem Script	2 <PubKey1> <PubKey2> <PubKey3> <PubKey4> <PubKey5> 5 OP_CHECKMULTISIG
Locking Script	OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
Unlocking Script	<Sig1> <Sig2> <Unlocking Redeem Script>

Table 8: Complex Script with P2SH

As you can see from the tables, with P2SH the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeem script itself is presented later, as part of the unlocking script when the output is spent. This shifts the burden in fees and complexity from the sender to the recipient (spender) of the transaction. Let us walk through this step by step:

- Given a transaction from a customer to the company, the transaction UTXO, which would originally be locked with the P2PSH script, referred to this as the **original redeem script** (which would be a very long script when converted to hex),

```
1 2 <PubKey1> <PubKey2> <PubKey3> <PubKey4> <PubKey5> 5 OP_CHECKMULTISIG
```

is in fact locked with the shortened form (called the **hash of the redeem script**)

```
1 OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
```

- When I (the CEO of the company) want to spend this UTXO, I must present an **unlocking redeem script** (which should match the original redeem script) and the signatures necessary to unlock it, like

this:

```
1 <Sig1> <Sig2> <Unlock Redeem Script>
```

- The two scripts are combined in two stages. First, the redeem script is checked against the locking script to make sure the hash matches:

```
1 <original redeem script> OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
```

- If the redeem script hash matches, the unlocking script is executed on its own, to unlock the redeem script:

```
1 <Sig1> <Sig2> <Original Redeem Script>
```

which is really

```
1 <Sig1> <Sig2> 2 <PubKey1> <PubKey2> <PubKey3> <PubKey4> <PubKey5> 5 OP_CHECKMULTISIG
```

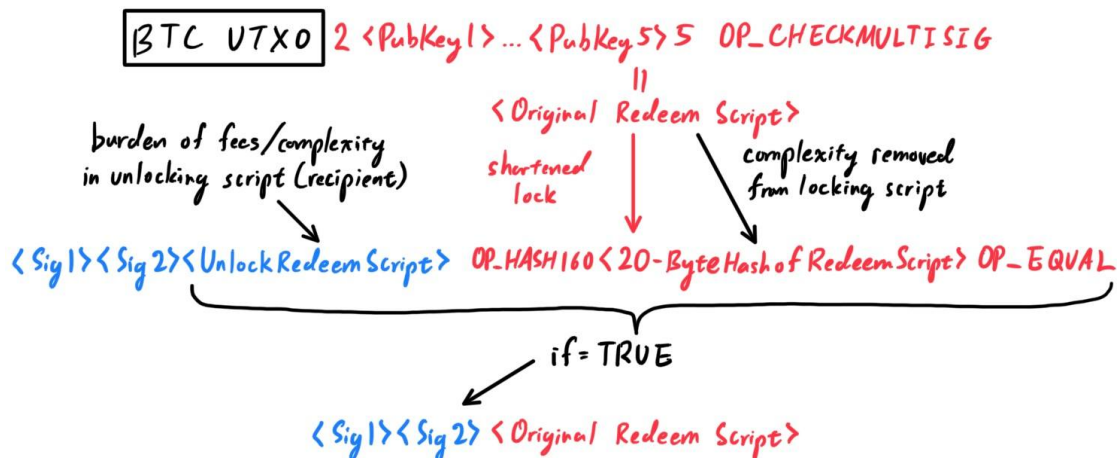


Figure 17: Pay-to-Script-Hash (P2SH) Process

9 The Bitcoin Network

Bitcoin is structured as a **peer-to-peer (P2P)** network architecture on top of the Internet, with a flat network topology and decentralized, non-hierarchical structure between nodes. The **bitcoin network** refers to the collection of nodes running the bitcoin P2P protocol, but there exists other protocols such as Stratum (used for mining and lightweight/mobile wallets) and pool-mining ones. Therefore, the term **extended bitcoin network** will refer to the overall network that includes the Bitcoin P2P protocol, pool-mining protocols, the Stratum protocol, and other related ones.

The network consists of about 12,000 nodes with different functionalities/characteristics:

- **Full Blockchain nodes**, also called **full nodes**, refer to nodes that maintain a complete and up-to-date copy of the bitcoin blockchain with all the transactions, which they independently build and verify, starting with the very first block and building up to the latest known block in the network.
 - **SPV nodes**, or **lightweight nodes**, contain a subset of the blockchain and can verify transactions using a method called **simplified payment verification (SPV)**.

- **Routing functions** are contained in every node in order to participate in the network.
- **Mining nodes** compete to create new blocks by running specialized hardware to solve the proof-of-work algorithm. Some mining nodes are also full nodes while others are lightweight nodes participating in pool mining and depending on a pool server to maintain a full node.
- **User wallets** might be part of a full node, as is usually the case with desktop bitcoin clients. Increasingly, many user wallets, especially those running on resource-constrained devices such as smartphones, are SPV nodes.

When a new node boots up, it must discover other bitcoin nodes on the network in order to participate. Through the TCP Internet protocol, the new node can first connect with any bitcoin node at random with the familiar "three-way handshake," but it by default connects with some long-running stable nodes listed on the client as **seed nodes** (which can be used to quickly discover other nodes in the network). Once this connection is established, the node will send an **addr** (address) message containing its own IP address to its neighbors, who will then forward it to its neighborhoods to ensure that the new node is better connected. The handshakes allow each node to exchange their version, which contains the following information:

```
1  "addr" : "85.213.199.39:8333",
2  "services" : "00000001",
3  "lastsend" : 1405634126,
4  "lastrecv" : 1405634127,
5  "bytessent" : 23487651,
6  "bytesrecv" : 138679099,
7  "conntime" : 1405021768,
8  "pingtime" : 0.00000000,
9  "version" : 70002,
10 "subver" : "/Satoshi:0.9.2.1/",
11 "inbound" : false,
12 "startingheight" : 310131,
13 "banscore" : 0,
14 "syncnode" : true
```

Additionally, the newly connected node can send **getaddr** to the neighbors, asking them to return a list of IP addresses of other peers. This way, a node can advertise its existence on the network for other nodes to find it. This entire process is called **bootstrapping**.

9.1 Full Nodes vs Simplified Payment Verification (SPV) Nodes

In the early days of bitcoin, every node was a full node. Running a full blockchain node lets you independently verify all transactions without the need to rely on any other systems. As of October 2021, the entire blockchain was about 360GB. The first thing a full node will do once it connects to peers is try to construct a complete blockchain. If it is a brand-new node and has no blockchain at all, it only knows one block, the **genesis block**, which is statically embedded in the client software. We describe the steps of the blockchain catching up:

1. A node joins the bitcoin network. This could be a brand-new node, or a node that had been offline for a while and needs to catch up. It bootstraps to the network and sends/receives the version message between its peers. This version message contains the "BestHeight" variable, which tells a node's current blockchain height (number of blocks). Peered nodes will see each other's version and BestHeight variables and compare how many blocks they each have. These peered nodes will exchange a "getblocks" message that compares the top block on their local blockchain.

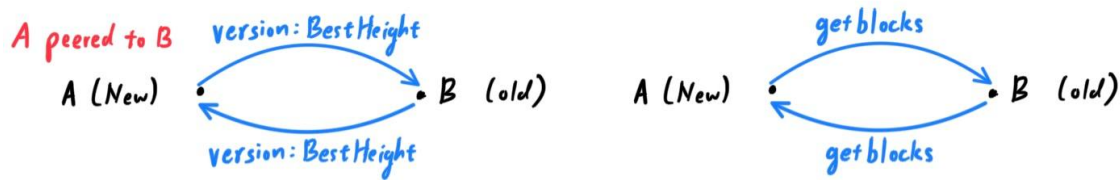


Figure 18: Step 1: Node Joining the Network

2. The old node will send the new node an "inv" message, which tells the new node which blocks must be exchanged for it to catch up. Upon receiving this inv message, the newer node issues a series of "getdata" messages requesting the full block data and identifying the requested blocks using the hashes from the inv message.

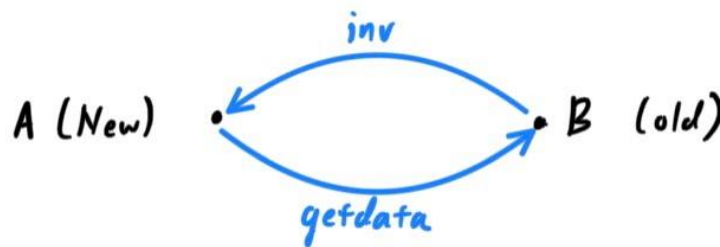


Figure 19: Step 2: Exchange of Block Information

3. This is all done in a controlled manner by sending 500 blocks at a time all spread out among peers to avoid overwhelming the network with requests. This process continues until the node catches up with the rest of the network.

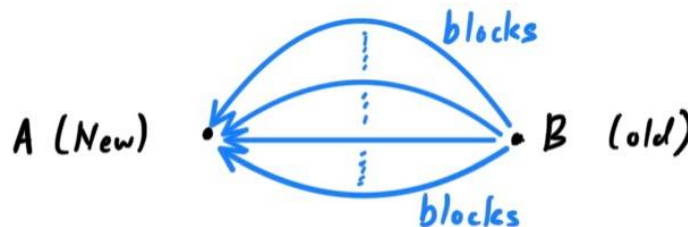


Figure 20: Step 3: Controlled Block Transfer

Not all nodes have the ability to store the full blockchain, so the SPV method, which is much more common than full nodes, is used to allow them to operate without storing everything. SPV nodes download only the block headers without the included transactions, making the resulting chain 1000 times smaller than the full blockchain. Since SPV nodes do not know about all the transactions on the network, they verify transactions using a different methodology that relies on peers to provide partial views of relevant parts of the blockchain on demand. Let us compare them with an example:

- When examining a new transaction in block 300,000, a full node links all 300,000 blocks down to the genesis block and builds a full database of UTXO, establishing the validity of the transaction by confirming that the UTXO remains unspent.
- An SPV node cannot validate whether the UTXO is unspent and will instead establish a link between the transaction and the block that contains it using **merkle paths** (which is an efficient verification

algorithm of the contents of large data structures). Each block in the bitcoin blockchain contains a summary of all the transactions in the block using a merkle tree. Then, the SPV node waits until it sees six additional blocks 300,001~300,006 (these 6 blocks could be existing ones or ones that are now being created) piled on top of the block (in order to make sure that a false blockchain is not being created) and verifies the new transaction in block 300,000 by establishing its depth under blocks 300,006 to 300,001. The fact that other nodes on the network accepted block 300,000 and then did the necessary work to produce six more blocks on top of it is proof, by proxy, that the transaction was not a double-spend. An SPV node cannot be persuaded that a transaction exists in a block when the transaction does not in fact exist. The SPV node establishes the existence of a transaction in a block by requesting a merkle path proof and by validating the proof of work in the chain of blocks.

In summary, a full blockchain node verifies a transaction by checking the entire chain of thousands of blocks below it in order to guarantee that the UTXO is not spent, whereas an SPV node checks how deep the block is buried by a handful of blocks above it.

9.2 Transaction Pools

Almost every node on the bitcoin network maintains two temporary separate lists stored in local memory and not saved on persistent storage, along with maybe one more:

- The **orphan pool** contains transactions without a parent.
- The **transaction pool** contains unconfirmed transactions. For example, a node that holds a user's wallet will use the transaction pool to track incoming payments to the user's wallet that have been received on the network but are not yet confirmed.
- The **UTXO pool** (maintained by some implementations of the bitcoin client) contains millions of entries of unspent transaction outputs, including some dating back to 2009. It may be housed in local memory or as an indexed database table on persistent storage.

When a node starts, both pools are empty but are gradually populated with new transactions received on the network. When a transaction is added to the transaction pool, the orphan pool is checked for any orphans that reference this transaction's outputs and any matching orphans are then validated (which can trigger a cascade reconstruction of an entire chain of interdependent transactions for chains missing the first parent UTXO). To compare and contrast:

- The transaction and orphan pools represents a single node's local perspective and might vary significantly, while the UTXO pool represents the emergent consensus of the network and will therefore vary little between nodes.
- The transaction and orphan pools only contain unconfirmed transactions, while the UTXO pool only contains confirmed outputs.

9.3 Alert Messages

Alert messages are a seldom used function, but are nevertheless implemented in most nodes as bitcoin's "emergency broadcast system." It is a means by which the core bitcoin developers can send an emergency text message to all bitcoin nodes of perhaps some serious problem in the bitcoin network, most notably in early 2013 when a critical database bug caused a multiblock fork to occur in the bitcoin blockchain. They are also cryptographically signed by a public key, with the corresponding private key held by few select members of the core development team. Each node receiving this alert message will verify it, check for expiration, and propagate it to all its peers, thus ensuring rapid propagation across the entire network.

10 The Blockchain

A **block** is a container data structure that aggregates transactions for inclusion in the public ledger, the blockchain. It can be stored as a flat file or in a simple database. The Bitcoin Core client stores the

blockchain metadata using Google's LevelDB database. For quick reference, the block header is 80 bytes, the average transaction is at least 250 bytes, and the average block contains more than 500 transactions. A complete block with all transactions is therefore 1000 times larger than the block header. The structure of the block can be seen below.

Block Size (4B)	Size of the block, in bytes, following this field
Block Header (80B)	Version (4B): A version number to track software/protocol upgrades
	Previous Block Hash (32B): A reference to the hash of the previous (parent) block in the chain
	Merkle Root (32B): A hash of the root of the merkle tree of this block's transactions
	Timestamp (4B): The approximate creation time of this block (seconds from Unix Epoch)
	Difficulty Target (4B): The proof-of-work algorithm difficulty target for this block
	Nonce (4B): A counter used for the proof-of-work algorithm
Transaction Counter (1~9B)	How many transactions follow
Transactions (Var)	The transactions recorded in this block

Table 9: Structure of a Bitcoin Block

10.1 Block Header

A **block header** consists of three sets of block metadata.

- The reference to the previous block hash, which connects this block to the previous block in the blockchain.
- The set (difficulty, timestamp, nonce) relate to the mining competition, which will be elaborated later.
- The third set is the merkle tree root, a data structure used to efficiently summarize all the transactions in the block.

The primary identifier of a block is its cryptographic hash, made by hashing the 80-byte block header twice through the SHA256 algorithm.

```
1 32-byte Block Hash = SHA256(SHA256(80-byte block header))
```

The resulting 32-byte hash is called the **block hash**, but more accurately called the **block header hash**, because only the block header is used to compute it. The block hash identifies a block uniquely and unambiguously and can be independently derived by any node by simply hashing the block header. For example, the following hash is the block hash of the genesis block, along with other variables:

```
1  "hash" : "00000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
2  "confirmations" : 308321,
3  "size" : 285,
4  "height" : 0,
5  "version" : 1,
6  "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
7  "tx" : [
8    "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
9  ],
10 "time" : 1231006505,
11 "nonce" : 2083236893,
12 "bits" : "1d00ffff",
13 "difficulty" : 1.00000000,
```



```
14  "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
```

Note that the block hash is not actually included inside the block's data structure, neither when the block is transmitted on the network, nor when it is stored on a node's persistence storage as part of the blockchain. Instead, the block's hash is computed by each node as the block is received from the network. The block hash might be stored in a separate database table as part of the block's metadata, to facilitate indexing and faster retrieval of blocks from disk.

A second way to identify a block is by its position in the blockchain, called the **block height**. Therefore, a block can thus be identified in two ways: by referencing the block hash or by referencing the block height. The block hash is a unique identifier, but the block height may not always be unique due to blockchain forks, when two or more blocks having the same block height compete for the same position in the blockchain. The block height is also not a part of the block's data structure; it is not stored within the block. Each node dynamically identifies a block's position (height) in the blockchain when it is received from the bitcoin network. The block height might also be stored as metadata in an indexed database table for faster retrieval.

10.2 Linking Blocks in the Blockchain

Given a full node that contains the current blockchain, the node will know the block header, and therefore the block hash, of the top block. As the node receives an incoming block from the network, it will validate the block by looking at the incoming block header's previous block hash. If the previous block hash matches the current top block hash already in the local blockchain, then the node adds this new block to the end of the chain, making the blockchain longer.

For example, assume that a full node has 277,314 blocks in the local copy of the blockchain, with the top block header hash of

```
1  0000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249
```

The bitcoin node then receives a new block from the network, which is parsed as follows:

```
1  "size" : 43560,
2  "version" : 2,
3  "previousblockhash" :
4    "0000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
5  "merkleroot" :
6    "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
7  "time" : 1388185038,
8  "difficulty" : 1180923195.25802612,
9  "nonce" : 4215469401,
10 "tx" : [
11   "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",
12
13   #[... many more transactions omitted ...]
14
15   "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
16 ]
```

The new block's "previousblockhash" field contains the hash of its parent block, which is already known to the node, that of the last block on the chain at height 277,314. Therefore, this new block is a child of the last block on the chain and extends the existing blockchain.

10.3 Merkle Trees

Each block in the bitcoin blockchain contains a summary of all the transactions in the block, using a **merkle tree**, or a **binary hash tree**. It is a data structure where each node branches into two (hence the name

binary) and containing cryptographic hashes (hence the name hash). Merkle trees provide a very efficient process to verify whether a transaction is included in a block. A Merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the root, or merkle root. The cryptographic hash algorithm used in bitcoin's merkle trees is **SHA256** applied twice, also known as double-SHA256. The time to parse a Merkle tree with N elements is $O(\log_2 N)$, which is extremely good.

To show how this merkle tree structure of transactions is formed, let us start with a block that must encode in it 4 transactions A, B, C, D.

- The transactions themselves are not stored in the merkle tree; rather, their data is double-hashed and the resulting hash is stored in each leaf node as H_A , H_B , H_C , H_D :

```
1 H_A = SHA256(SHA256(Transaction A))
```

- Consecutive pairs of leaf nodes are then summarized in a parent node, by concatenating the two hashes and hashing them together. For example, the parent node H_{AB} is created as:

```
1 H_AB = SHA256(SHA256(H_A + H_B))
```

- This process continues until there is only one node at the top, the node known as the **Merkle root**. This 32-byte hash is stored in the block header and summarizes all the data in all four transactions.

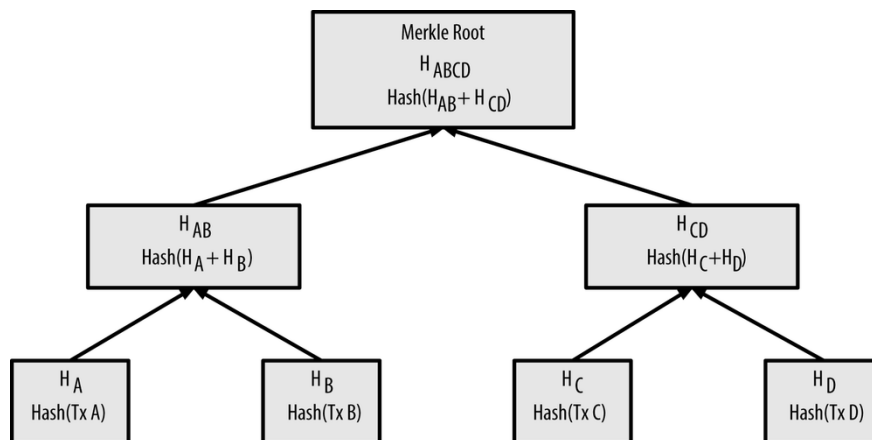


Figure 21: Merkle Tree with Four Transactions

- This can clearly be used to construct trees of any size containing any number of transactions. If there are odd numbers of transactions to summarize at any level, then the last transaction hash will be duplicated to create an even number of leaf nodes, known as a **balanced tree**.

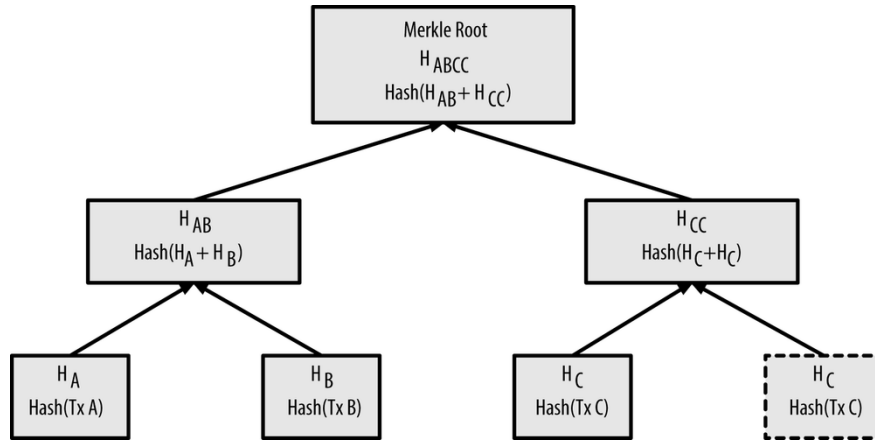


Figure 22: Merkle Tree with Odd Number of Transactions

To prove that a specific transaction is included in a block, all we need to do is create an **authentication path**, or a **merkle path**, connecting the specific transaction to the root of the tree. To give an example, let us have a merkle tree of 16 transactions, and we must prove that a given transaction K is included in the block. The path consists of the four hashes

$$H_L \rightarrow H_{IJ} \rightarrow H_{MNOP} \rightarrow H_{ABCDEFGH}$$

With these four hashes provided as a path, any node can prove that H_K (in green in the diagram) is included in the merkle root by computing four additional pair-wise hashes H_{KL} , H_{IJKL} , $H_{IJKLMNOP}$, and the merkle tree root (outlined in a dotted line in the diagram).

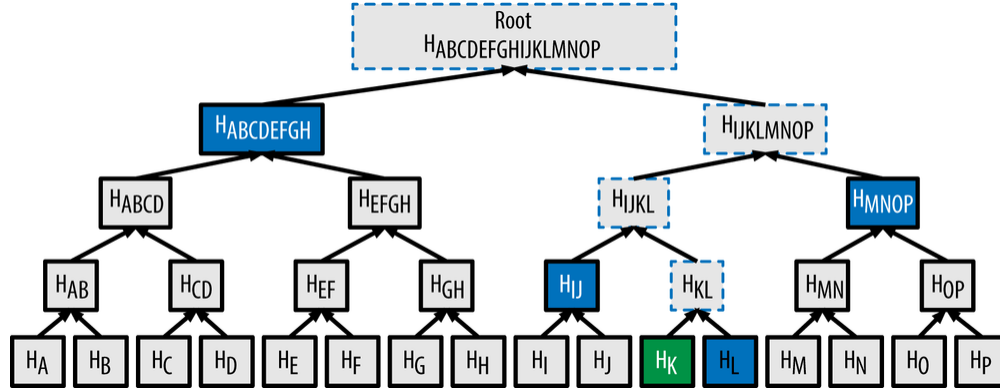


Figure 23: Merkle Path for Transaction K

With merkle trees, a node can download just the block headers (80 bytes per block) and still be able to identify a transaction's inclusion in a block by retrieving a small merkle path from a full node, without storing or transmitting the vast majority of the blockchain, which might be several gigabytes in size. Nodes that do not maintain a full blockchain, called simplified payment verification (SPV nodes), use merkle paths to verify transactions without downloading full blocks.

11 Mining and Consensus

Mining is the process by which new bitcoin is added to the money supply. There are multiple purposes and properties of mining:

- Mining serves to secure the bitcoin system against fraudulent transactions or transactions spending the same amount of bitcoin more than once, known as a double-spend.
- Miners provide processing power to the bitcoin network in exchange for the opportunity to be rewarded bitcoin.
- Mining is the invention that makes bitcoin special, a decentralized security mechanism that is the basis for peer-to-peer digital cash. The reward of newly minted coins and transaction fees is an incentive scheme that aligns the actions of miners with the security of the network, while simultaneously implementing the monetary supply.
- Each block, generated on average every 10 minutes, contains entirely new bitcoins, created from nothing.
- Every 210,000 blocks (approx. 4 years), the currency issuance rate is decreased by 50%. The rate of new coins decreases like this exponentially over 64 "halvings" until block 13,230,000 (mined approx. in 2137), when it reaches the minimum currency unit of 1 satoshi. At slightly less than 21 million bitcoins, no more bitcoins will be issued.
- The finite and diminishing issuance creates a fixed monetary supply that resists inflation. Unlike a fiat currency, which can be printed in infinite numbers by a central bank, bitcoin can never be inflated by printing.

Bitcoin has no central authority, yet somehow every full node has a complete copy of a public ledger that it can trust as the authoritative record. The blockchain is not created by a central authority, but is assembled independently by every node in the network. Somehow, every node in the network, acting on information transmitted across insecure network connections, can arrive at the same conclusion and assemble a copy of the same public ledger as everyone else. This chapter examines the process by which the bitcoin network achieves **global consensus** without central authority.

11.1 Mining Nodes & Aggregating Transactions into Blocks

A **miner**, or a **mining node**, is a specialized computer-hardware system connected to a server running a full bitcoin node designed to mine bitcoins.

- Like every other full node, it receives, validates, and relays new, unconfirmed transactions on the bitcoin network. These transactions are added to the node's transaction pool, where they await until they can be mined into a block. It is always listening for new blocks, too.
- The miner immediately constructs a new empty block (say, the Nth block in the chain), called a **candidate block**. It takes the unconfirmed transactions that are in the pool, puts them in the candidate block, and tries to find a solution to the proof-of-work algorithm in order to make the block valid.
 - If a new block is received before the miner can successfully solve the proof-of-work algorithm, this signifies the end of the competition for block N and the beginning of the competition to create block N+1. Upon receiving this block N and validating it, the miner will also check all the transactions in the transaction pool and remove any that were included in block N+1. Whatever transactions remain in the memory pool are unconfirmed and are waiting to be recorded in a new block.
 - If the miner successfully solves the algorithm first, then it takes the unconfirmed transactions in the pool, puts it into the now valid block, and adds it to the blockchain. This new updated blockchain is propagated throughout the network, telling other nodes that this block is solved. With this, the miner node *constructs* the **generation transaction**, or **coinbase transaction**, which represents the mining reward. It consists of the newly minted bitcoins (6.25BTC in 2021) along with all the transaction fees (difference of all the outputs from all the inputs). Unlike other transactions, the generation transaction has a "coinbase" input rather than one specifying a previous UTXO to spend.

11.2 Constructing the Block Header & Difficulty Representation

To construct the block header, the mining node needs to fill in the following fields.

Version (4B)	A version number to track software/protocol upgrades
Previous Block Hash (32B)	A reference to the hash of the previous (parent) block in the chain
Merkle Root (32B)	A hash of the root of the merkle tree of this block's transactions
Timestamp (4B)	The approximate creation time of this block (seconds from Unix Epoch)
Difficulty Target (4B)	The proof-of-work algorithm difficulty target for this block
Nonce (4B)	A counter used for the proof-of-work algorithm

Table 10: Block Header Fields

The version is self-explanatory and needs no further explanation. The mining node needs to add the "Previous Block Hash," which is the hash of the block header of the top block in its local blockchain. This can also be done since the node merely needs to double-SHA256 the header. To find the merkle root, the node constructs a merkle tree to summarize all the transactions and finds the root hash. Then, the mining node will add a 4-byte timestamp encoded as the Unix Epoch timestamp. The node then fills in the difficulty target, which defines the required proof-of-work difficulty to make this a valid block. The 4-byte difficulty is encoded as an 8-digit hexadecimal, for example 0x1903a90c, which is a mantissa-exponent encoding of the target where the first part 0x19 is a hexadecimal exponent and the next part 0x03a90c is the coefficient. The formula for the difficulty target of this number is:

```
1 target = coefficient * 2^(8 * (exponent - 3))
```

which gets, using 0x1903a30c,

[illegible]

Which, in decimal, is

$$1 \quad 238,348 * 2^{176} = 22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328$$

Obviously, this is a really huge number, and we will find out what this means. For now, it is important to just know that this is a predetermined number representing the "difficulty" of completing the algorithm. Finally, the nonce is a variable that is initialized to 0.

11.3 Mining the Block & Proof-of-Work Algorithm

The miner constructs a candidate block filled with transactions and then constructs the header. Now, at this point, the nonce is initialized to 0, and the miner calculates the hash of this block's header to get

```
1 hash0 = SHA256(Block Header with Nonce=0)
```

and sees if it is smaller than the current *target* (target difficulty). If it is not, then the miner changes the Nonce to a different value out of the 2^{32} (about 4 billion) ones to get a different hash. Usually, we would increment it by 1 to get the next hash:

```
1 hash1 = SHA256(Block Header with Nonce=1)
```

Once the miner goes through billions and billions of hashes and finds a hash that is less than the target difficulty, the node has successfully solved the proof-of-work algorithm. Note that it is often the case that the miner can go through all 2^{32} nodes and still not find a hash value that is less than the target. In this case, the mining node can take additional measures to refresh the nonce:

- The timestamp can be changed up to two hours in the future, which would change the header and therefore give a fresh new 4 billion new nonces to go through.
- The coinbase transaction, which can store 2~100 bytes of data, can be used as a source of extra nonce values. The coinbase transaction is included in the merkle tree, which means that any change in the coinbase script causes the merkle root to change.

The 8 bytes of extra nonce, plus the 4 bytes of standard nonce and the 7200 seconds (within the next 2 hours) of timestamp, allows miners to explore a total of almost 2^{109} possibilities per second, an extremely large number. For example, let us denote the block N header with nonce value M as **BlockNHeaderM**. Then, the mining node will compute the following:

```

1  BlockNHeader0 => a80a81401765c8eddee25df36728d732...
2  BlockNHeader1 => f7bc9a6304a4647bb41241a677b5345f...
3  BlockNHeader2 => ea758a8134b115298a1583ffb80ae629...
4  BlockNHeader3 => bfa9779618ff072c903d773de30c99bd...
5  BlockNHeader4 => bce8564de9a83c18c31944a66bde992f...
6  BlockNHeader5 => eb362c3cf3479be0a97a20163589038e...
7  BlockNHeader6 => 4a2fd48e3be420d0d28e202360cfbaba...
8  BlockNHeader7 => 790b5a1349a5f2b909bf74d0d166b17a...
9  BlockNHeader8 => 702c45e5b15aa54b625d68dd947f1597...
10 BlockNHeader9 => 7007cf7dd40f5e933cd89fff5b791ff0...
11 BlockNHeader10 => c2f38c81992f4614206a21537bd634a...
12 BlockNHeader11 => 7045da6ed8a914690f087690e1e8d66...
13 BlockNHeader12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
14 BlockNHeader13 => 0ebc56d59a34f5082aaef3d66b37a66...
15 BlockNHeader14 => 27ead1ca85da66981fd9da01a8c6816...
16 BlockNHeader15 => 394809fb809c5f83ce97ab554a2812c...
17 BlockNHeader16 => 8fa4992219df33f50834465d3047429...
18 BlockNHeader17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
19 BlockNHeader18 => 9989a401b2a3a318b01e9ca9a22b0f3...
20 BlockNHeader19 => cda56022ecb5b67b2bc93a2d764e75f...

```

For example, if the target was

```
1 0x10000000000000000000000000000000000000000000000000000000
```

then the winning nonce is 13, since it is the first nonce in which the BlockNHashM value was less than the target. Clearly, given the target, we just need a hash value that starts with a 0, which has a 1/16 chance of happening randomly. Furthermore, increasing the difficulty by 1 bit (or by 1 hex digit) causes an exponential increase in the time it takes to find a solution, more specifically by a factor of 2 (or by 16). This is why all bitcoin block hashes have a lot of zeroes in the beginning, meaning that the acceptable range of hashes is much smaller, hence it is more difficult to find a valid hash.

11.4 Difficulty Target and Retargeting

Note that the easiness, and thus the speed in which the proof-of-work algorithm is solved, is dependent on how small the target difficulty is. If we decrease the target, then the task of finding a hash that is less than the target becomes more difficult, and more importantly, this difficulty target is *adjustable*. Bitcoin's blocks are generated every 10 minutes, which underpins the frequency of currency issuance and the speed of transaction settlement. It must remain at a constant 10 minutes over many decades, adjusting to increased computing power and additional mining nodes/competition.

This difficulty adjustment algorithm is quite simple: It readjusts the difficulty every 2016 blocks, which should have an expected computing time of 20,160 minutes (2 weeks). The ratio between the actual timespan and desired timespan is calculated and a corresponding adjustment (up or down) is made to the difficulty.

$$1 \quad \text{New Difficulty} = \text{Old Difficulty} * (\text{Actual Time of Last 2016 Blocks} / 20160 \text{ minutes})$$

- If the actual time in the past 2016 blocks took longer than 2 weeks (meaning that the rate at which blocks are added to the blockchain is slower), then the new difficulty is increased from the old one to make the process easier.
- If the actual time of the past 2016 blocks took less than 2 weeks, then this means that blocks are added too fast and therefore the target must be decreased (making it more difficult).

To avoid extreme volatility in the difficulty, the retargeting adjustment must be less than a factor of 4 per cycle. If the required difficulty adjustment is greater than a factor of four, it will be adjusted by the maximum and any further adjustment will be accomplished in the next retargeting periods.

Note that the target difficulty is independent of the number of transactions or the value of transactions, but it is dependent on market forces as new miners enter the market to compete for the reward. This means that the hashing power and electricity expended is not at all dependent on the number/value of transactions within a block, but rather how many miners there are in the network (which determines how hard the difficulty is).

11.5 Successfully Mining and Validating the Block

Once a mining node has found a nonce that produces a block hash that is less than the target, the node transmits the block (with the header containing the nonce) to all its peers. They receive, validate, and then propagate the new block, and as it ripples out across the network, each node adds it to its own copy of the blockchain. The mining nodes that receives and validated the block abandon their efforts to find a block at the same height and immediately start computing the next block in the chain.

The node is validated through a long list of criteria that must all be met, some of which are:

- The block data structure is syntactically valid.
- The block header hash is less than the target difficulty (enforces the proof of work)
- The block timestamp is less than two hours in the future (allowing for time errors)
- The block size is within acceptable limits
- The first transaction (and only the first) is a coinbase generation transaction
- All transactions within the block are valid using the transaction checklist.

Independent validation is what prevents cheaters from constructing fake blocks that will profit themselves. A cheater who sends a block with a fake transaction will find their block rejected by the network, and the cheater would expend a lot of electricity in mining without any rewards.

11.6 Blockchain Forks and Assembling Chains of Blocks

Nodes maintain three sets of blocks, shown below. Invalid blocks are rejected as soon as any one of the validation criteria fails.

- The **main blockchain** is at any time whichever chain of blocks has the most cumulative difficulty associated with it. Under most circumstances this is also the chain with the most blocks in it, unless there are two equal-length chains and one has more proof of work.
- One or multiple **secondary chains** that form branches off the main block chain, due to forks. The main chain can have branches with blocks that are "siblings" to the blocks on the main chain. These

blocks are valid but not part of the main chain. They are kept for future reference, in case one of those chains is extended to exceed the main chain in difficulty.

- **Orphan blocks**, which are blocks that do not have a known parent in the known (both main and secondary) chains. Orphan blocks are saved in the orphan block pool where they will stay until their parent is received. Once the parent is received and linked into the existing chains, the orphan can be pulled out of the orphan pool and linked to the parent, making it part of a chain. Orphan blocks usually occur when two blocks that were mined within a short time of each other are received in reverse order (child before parent).

Under normal circumstances, once a valid block is found, it is sent, validated, and propagated through the network, and new nodes will stack the new block on top of its local blockchain. As said before, this is done by looking at the block's "previous block hash" field, which is the reference to the new block's parent. If the parent is in the chain, it can be successfully linked.

Blocks might arrive at different nodes at different times, causing the nodes to have different perspectives of the blockchain. We look at the simplified diagram of bitcoin as a global network. A **fork** in the blockchain occurs whenever there are two candidate blocks competing to form the longest blockchain. This occurs under normal conditions whenever two miners solve the proof-of-work algorithm within a short period of time from each other. As both miners discover a solution for their respective candidate blocks, they immediately broadcast their own "winning" block to their immediate neighbors who begin propagating the block across the network. Each node that receives a valid block will incorporate it into its blockchain, extending the blockchain by one block. If that node later sees another candidate block extending the same parent, it connects the second candidate on a secondary chain. As a result, some nodes will "see" one candidate block first, while other nodes will see the other candidate block and two competing versions of the blockchain will emerge. This leads to two completely valid "versions" of the blockchain that is kept in each node.

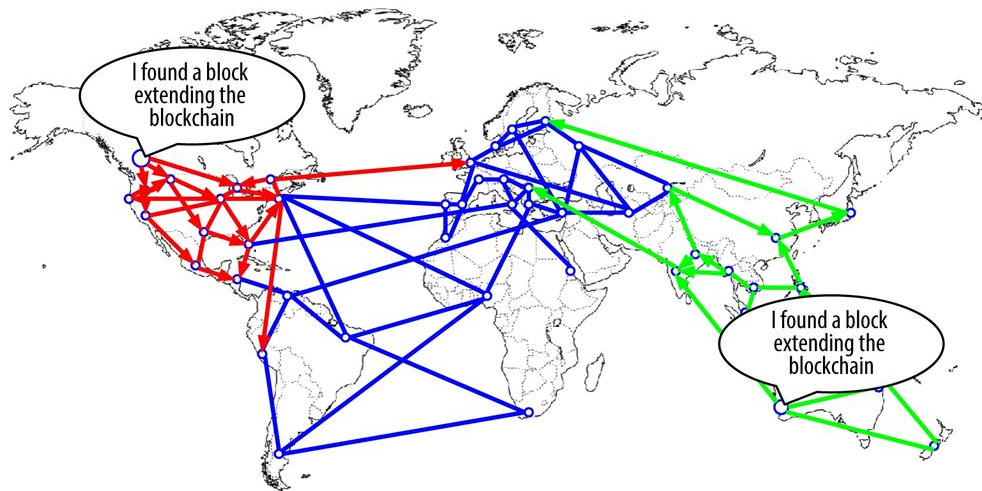


Figure 24: Fork in the Blockchain

As the two blocks propagate, some nodes receive block "red" first and some receive block "green" first. Therefore, the network splits into two different perspectives of the blockchain, one side topped with a red block, the other with a green block.

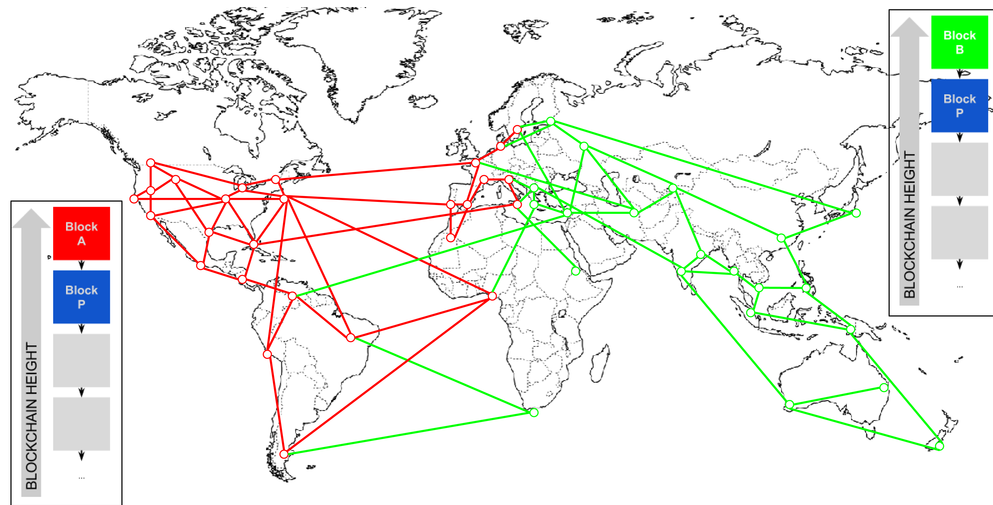


Figure 25: Network Split Due to Fork

The bitcoin network nodes closest to the Canadian node will hear about block "red" first and create a main blockchain with "red" as the last block in the chain. Meanwhile, nodes closer to the Australian node will take that block as the winner and extend the blockchain with "green" as the last block (e.g., blue-green), ignoring "red" when it arrives a few seconds later. Any miners that saw "red" first will immediately build candidate blocks that reference "red" as the parent and start trying to solve the proof of work for these candidate blocks. The miners that accepted "green" instead will start building on top of "green" and extending that chain.

Forks are almost always resolved within one block. As part of the network's hashing power is dedicated to building on top of "red" as the parent, another part of the hashing power is focused on building on top of "green." Even if the hashing power is almost evenly split, it is likely that one set of miners will find a solution and propagate it before the other set of miners have found any solutions. Let's say, for example, that the miners building on top of "green" find a new block "pink" that extends the chain (e.g., blue-green-pink). They immediately propagate this new block and the entire network sees it as a valid solution.

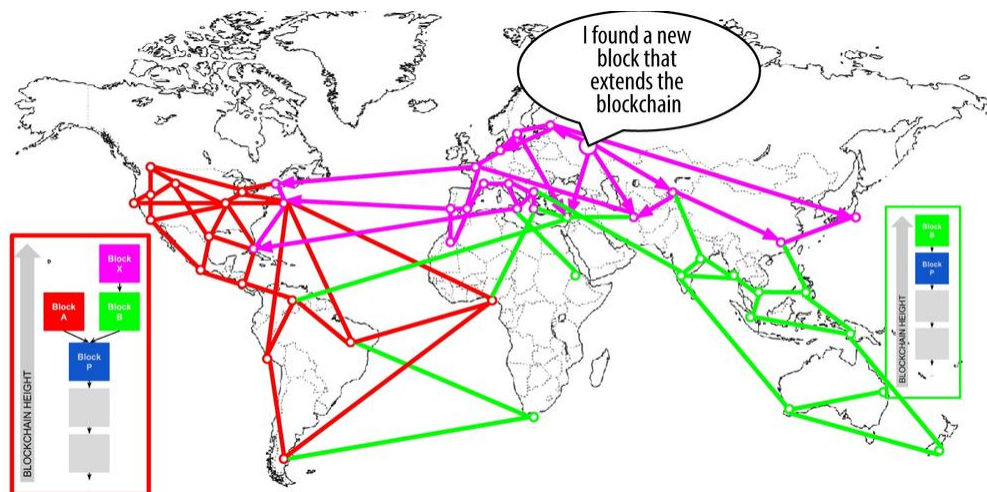


Figure 26: Fork Resolution

All nodes that had chosen "green" as the winner in the previous round will simply extend the chain one

more block. The nodes that chose "red" as the winner, however, will now see two chains: blue-green-pink and blue-red. The chain blue-green-pink is now longer (more cumulative difficulty) than the chain blue-red. As a result, those nodes will set the chain blue-green-pink as main chain and change the blue-red chain to being a secondary chain, as shown in Figure 8-6. This is a chain reconvergence, because those nodes are forced to revise their view of the blockchain to incorporate the new evidence of a longer chain. Any miners working on extending the chain blue-red will now stop that work because their candidate block is an "orphan," as its parent "red" is no longer on the longest chain. The transactions within "red" are queued up again for processing in the next block, because that block is no longer in the main chain. The entire network re-converges on a single blockchain blue-green-pink, with "pink" as the last block in the chain. All miners immediately start working on candidate blocks that reference "pink" as their parent to extend the blue-green-pink chain.

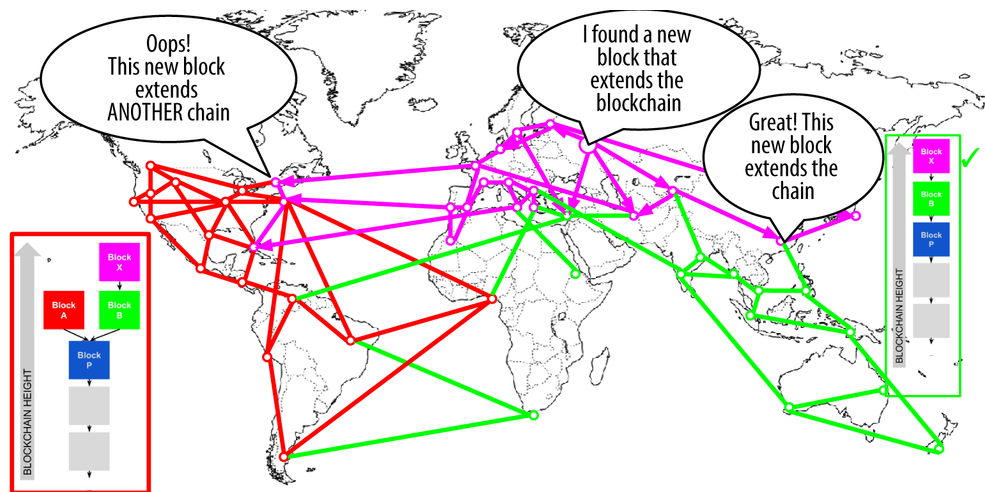


Figure 27: Network Reconvergence

A one-block fork may be quite common, occurring once a week, but a fork that extends to two blocks (i.e. if two blocks are found almost simultaneously by miners on opposite sides of a previous fork) is extremely rare. Bitcoin's block interval of 10 minutes is a design compromise between fast confirmation times (settlement of transactions) and the probability of a fork. A faster block time would make transactions clear faster but lead to more frequent blockchain forks, whereas a slower block time would decrease the number of forks but make settlement slower.

11.7 The Hashing Race and Mining Pools

The **hash rate** is a key measuring unit (and security metric) of the processing power of the Bitcoin network, generally represented in hashes per second. The more hashing power in the network, the greater its security and overall resistance to attack. Although bitcoin's exact hashing power is unknown, it is possible to estimate it from the number of blocks being mined and the current block difficulty. Daily numbers may periodically rise or drop as a result of the randomness of block discovery. A terahash, or Th, is 1 trillion hashes, and as of November 2021, the total hashrate of the entire bitcoin network is approximately 160m TH/s. The historical data (actually, the 30-day moving average) of the hash rate in Th/s is shown below.

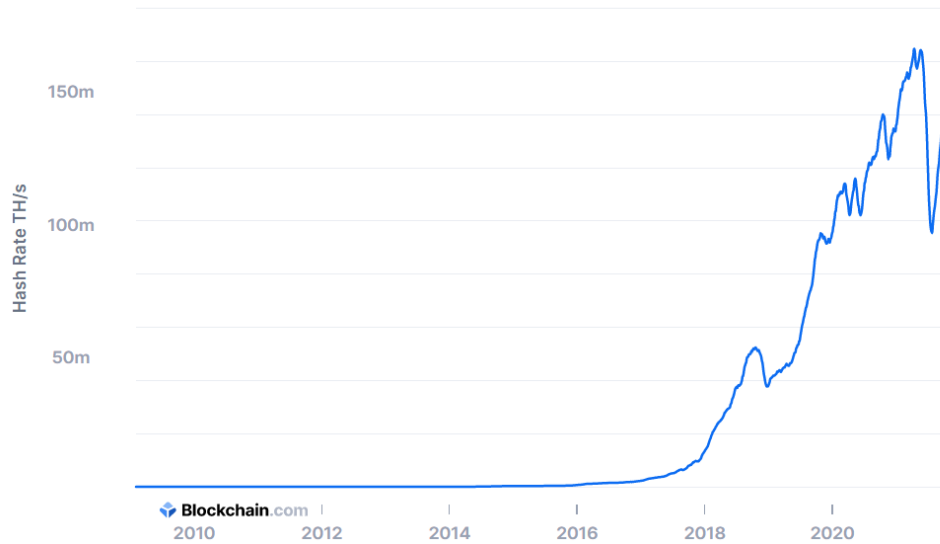


Figure 28: Historical Hash Rate of Bitcoin Network

As the amount of hashing power applied to mining bitcoin has exploded, the difficulty has risen to match it, as shown below in the graph of the 30-day moving average.

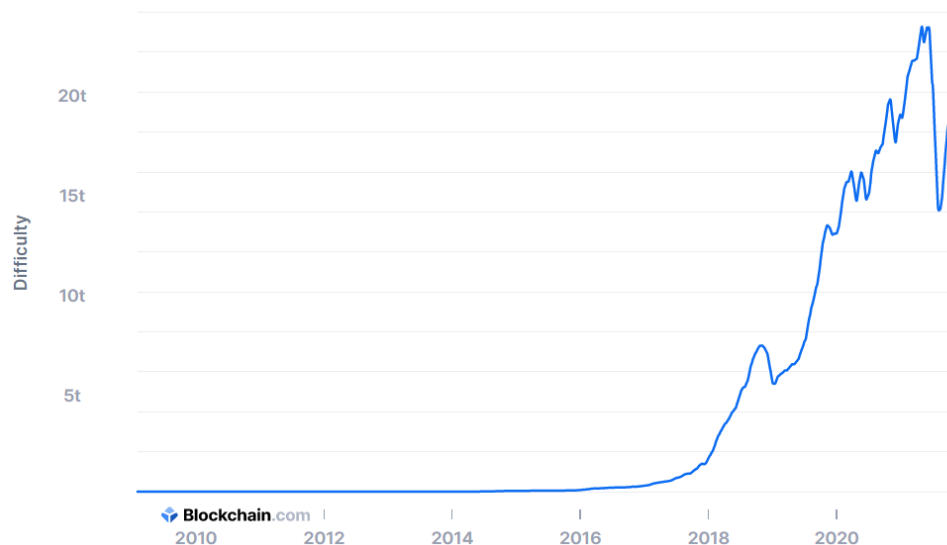


Figure 29: Bitcoin Mining Difficulty Over Time

These bitcoins are mined using ASIC mining chips, which have increased in density over the years (i.e. how many chips can be squeezed into a building, while still dissipating the heat and providing adequate power).

Because individual miners don't stand a chance, miners now collaborate to form **mining pools**, pooling their hashing power and sharing the reward among thousands of participants. They get a smaller share of the overall reward, but typically get rewarded every day, reducing uncertainty. Additionally, the ability to mine without running a full node is another big benefit of joining a pool. Mining pools coordinate many hundreds or thousands of miners, over specialized pool-mining protocols. The individual miners configure their mining equipment to connect to a pool server, after creating an account with the pool. Their mining hardware

remains connected to the pool server while mining, synchronizing their efforts with the other miners. Thus, the pool miners share the effort to mine a block and then share in the rewards.

Mining pools coordinate hundreds/thousands of miners over specialized pool-mining protocols. The individual miners connect to the pool server while mining, synchronizing their efforts with the other miners. Thus, the pool miners share the effort to mine a block and then share in the rewards. Successful blocks pay the reward to a pool bitcoin address, rather than individual miners, and periodical payments are made to the miners' bitcoin addresses. Since most mining pools are managed by an individual or company, the **pool operator** may charge a small management fee for the pool-services.

Miners participating in a pool split the work of searching for a solution to a candidate block, earning "shares" for their mining contribution. The mining pool sets a lower difficulty target for earning a share, typically more than 1,000 times easier than the bitcoin network's difficulty. As miners succeed in the easier difficulty target, they get more shares, and when someone in the pool successfully mines a block, the reward is earned by the pool and then shared with all miners in proportion to the number of shares they had. Pool miners connect to the pool server using a mining protocol such as Stratum (STM) or GetBlockTemplate (GBT). Both the STM and GBT protocols create block templates that contain a template of a candidate block header. The pool server constructs a candidate block by aggregating transactions, adding a coinbase transaction (with extra nonce space), calculating the merkle root, and linking to the previous block hash. The header of the candidate block is then sent to each of the pool miners as a template. Each pool miner then mines using the block template, at a lower difficulty than the bitcoin network difficulty, and sends any successful results back to the pool server to earn shares.

11.8 P2Pools and Consensus Attacks

Managed pools can be risky due to their single point of failure and the possibility for the pool manager to cheat (by directing the pool effort to double-spend transactions or invalidate blocks), which is why **P2Pool**, a new peer-to-peer mining pool without a central operator, was proposed and implemented in 2011.

P2Pool works by decentralizing the functions of the pool server, implementing a parallel blockchain-like system called a **share chain**. A share chain is a blockchain running at a lower difficulty than the bitcoin blockchain. The share chain allows pool miners to collaborate in a decentralized pool, by mining shares on the share chain at a rate of one share block every 30 seconds. Each of the blocks on the share chain records a proportionate share reward for the pool miners who contribute work, carrying the shares forward from the previous share block. When one of the share blocks also achieves the difficulty target of the bitcoin network, it is propagated and included on the bitcoin blockchain, rewarding all the pool miners who contributed to all the shares that preceded the winning share block. Essentially, instead of a pool server keeping track of pool miner shares and rewards, they are kept track by a decentralized blockchain called the share chain (basically a mini-blockchain tracking shares within a pool server).

Participation in P2Pool has increased due to concerns of a **51% attack**, a type of a **consensus attack**. A scenario of a 51% attack would play out as such: Assume that a group of miners, controlling a majority (51%) of the total network's hashing power, collude to attack bitcoin. With the ability to mine the majority of the blocks, the attacking miners can cause deliberate "forks" in the blockchain and double-spend transactions or execute denial-of-service attacks against specific transactions or addresses.

- A **fork/double-spend attack** is one where the attacker causes previously confirmed blocks to be invalidated by forking below them and re-converging on an alternate chain. With sufficient power, an attacker can invalidate six or more blocks in a row, causing transactions that were considered immutable (six confirmations) to be invalidated. Note that a double-spend can only be done on the attacker's own transactions, for which the attacker can produce a valid signature. Double-spending one's own transactions is profitable if by invalidating a transaction the attacker can get a nonreversible exchange payment or product without paying for it.

For example, assume that entity A buys something from entity B for 10BTC. Once the transaction is confirmed in a block in the form "A pays B 10BTC," B hands over the product to A. Once receiving the product, A works with an accomplice, C, who operates a large mining pool, and C launches a

51% attack as soon as A's transaction is included in a block. C directs the mining pool to re-mine the same block height as the block containing A's transaction, replacing the "A → B" with "A → A", a transaction that double-spends the same input as A's payment. The double-spend transaction consumes the same UTXO and pays it back to A's wallet, instead of paying it to B, essentially allowing A to keep the bitcoin. C then directs the mining pool to mine an additional block, so as to make the chain containing the double-spend transaction longer than the original chain (causing a fork below the block containing A's transaction). When the blockchain fork resolves in favor of the new (longer) chain, the double-spent transaction replaces the original payment to B. Ultimately, B lost both the product and the BTC.

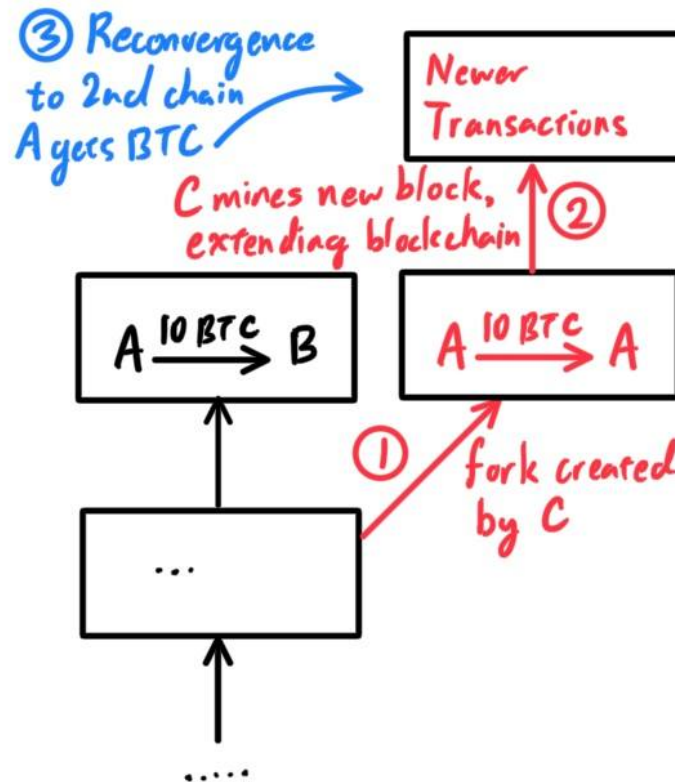


Figure 30: Double Spend Attack

To protect against this kind of attack, a merchant selling large-value items must wait at least 6 confirmations before giving the product to the buyer. The more confirmations elapse, the harder it becomes to invalidate a transaction with a 51% attack. For high-value items, payment by bitcoin will still be convenient and efficient even if the buyer has to wait 24 hours for delivery, which would ensure 144 confirmations.

- A **denial-of-service attack** is the other scenario for a consensus attack, which denies service to specific bitcoin participants (their addresses). An attacker with a majority of the mining power can simply ignore specific transactions. If they are included in a block mined by another miner, the attacker can deliberately fork and re-mine that block, again excluding the specific transactions. This type of attack can result in a sustained denial of service against a specific address or set of addresses for as long as the attacker controls the majority of the mining power.

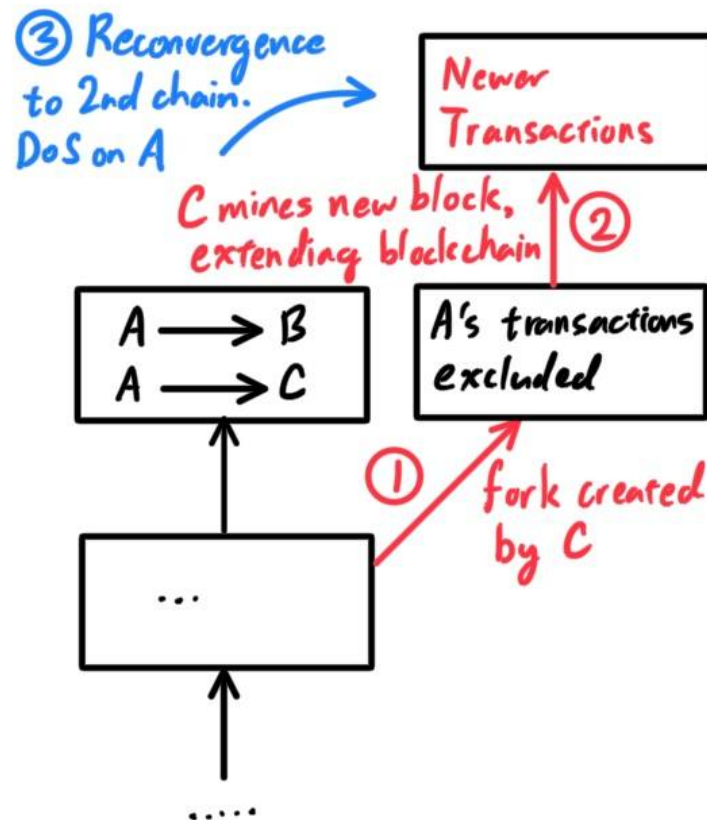


Figure 31: Denial-of-Service Attack

Despite its name, the 51% attack scenario doesn't actually require 51% of the hashing power, but can be attempted with as little as 30%. The 51% threshold is simply the level at which such an attack is almost guaranteed to succeed. A consensus attack is essentially a tug-of-war for the next block and the "stronger" group is more likely to win. With less hashing power, the probability of success is reduced, because other miners control the generation of some blocks with their "honest" mining power. One way to look at it is that the more hashing power an attacker has, the longer the fork he can deliberately create, the more blocks in the recent past he can invalidate, or the more blocks in the future he can control. Even though there is no possible way for solo miners to control even 1% of the total mining power, it may be possible to do so with mining pools.