Blockchain Course

Bitcoin

fausto.spoto@univr.it

https://github.com/spoto/blockchain-course

The internet of money

What do we expect from money

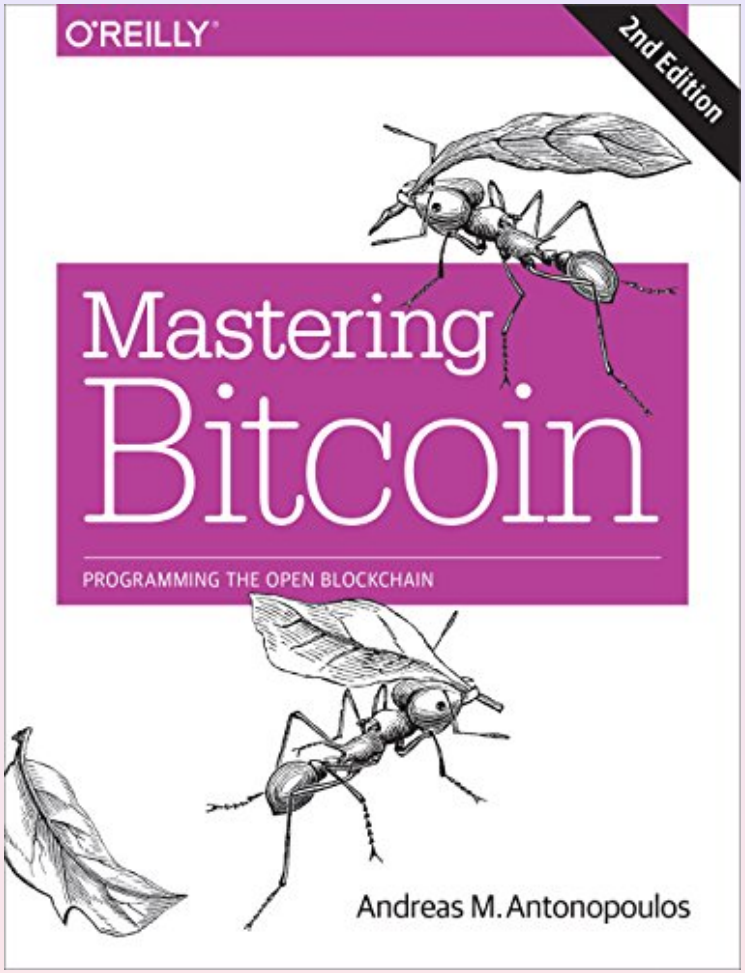
- money should be protected from counterfeiting (*legality*)
- money should not be spent twice (*uniqueness*)
- no one can claim that my money belongs to him (*ownership*)

Electronic money exists since decades (credit cards, online transactions)

Bitcoin provides a fully decentralized electronic cash system, for the first time (a single State cannot shut down the bitcoin network)

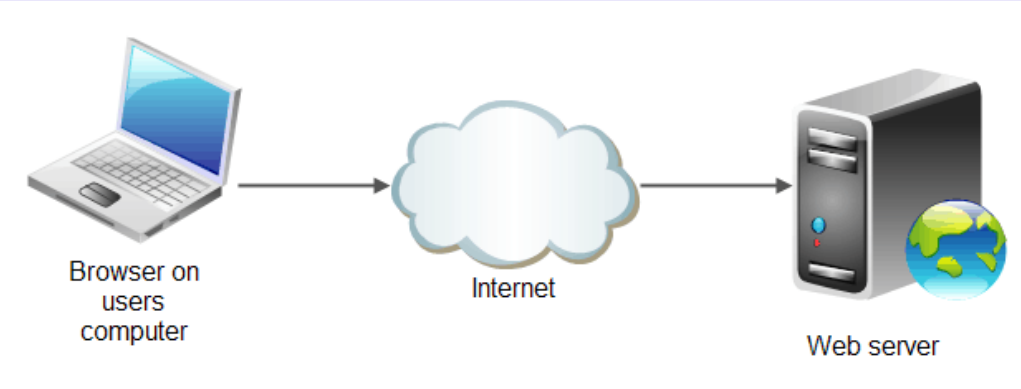
"Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto, 2008

Reference used in this course



https://github.com/bitcoinbook/bitcoinbook

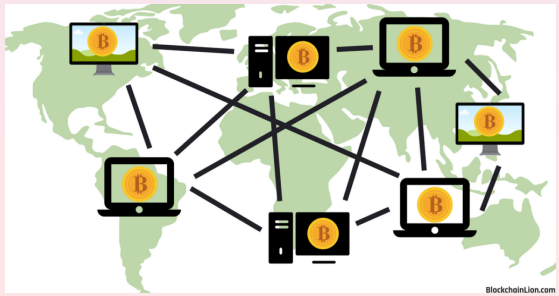
Bitcoin as a web service



The server keeps a map (ledger) $user\_id \Rightarrow balance$ and accepts transactions to transfer balances

Users interact through a browser (wallet) to ask to transfer balances

The server is actually a worldwide peer-to-peer (p2p) network of computers

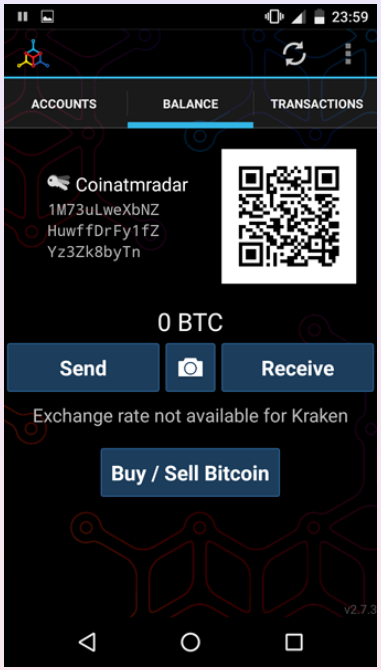


Wallets

- desktop wallets
- mobile wallets
- web wallets
- hardware wallets
- paper wallets

Mobile wallets

At the first start-up, a bitcoin address is created for you, then transactions from/to that address are tracked:



The address can be seen as our IBAN. Note that its creation is a local operation that does not do anything on the network and is consequently completely anonymous

Address creation

When Alice's wallet starts for the first time:

1. it generates a finite sequence of bits through a secure random generator (a secret private key)
2. it computes the bitcoin address as an abstraction of the private key (hashing)
3. it shows the bitcoin address as an alphanumeric string and as a picture (QR code)
4. the address is not sensitive information: Alice can publish it in her web page
5. the private key is sensitive information: Alice keeps it secret
   - a hardware wallet stores it in its internal memory
   - a desktop wallet stores it in Alice's computer's file system (!)
   - a mobile wallet stores it in Alice's phone (!!!)
   - a web wallet stores it at a third-party service (!!!!!!!)

Random generators

Cryptographically-secure random generators

Private keys should be generated by using a cryptographically-secure random generator, or otherwise keys might not really be randomly spread and might be more easily guessed

In Java

`java.security.SecureRandom`, not `java.util.Random`

The generation of private keys does not require any centralized service and occurs completely outside the blockchain, offline. The probability of computing an already used key is negligeable

Alice charges her wallet

- she asks a friend to sell for you bitcoins at your address
- meets a bitcoin seller in person
- earns bitcoin by working
- uses a bitcoin ATM
- uses a bitcoin currency exchange company

What is the price?

It is not set by the computer network! It's a social agreement, the average of the last sell operations. You can look online for it

The charge transaction

1. Joe (the seller) specifies in his wallet Alice's bitcoin address as destination (or scans Alice's QR code with his mobile)
2. Joe signs a transaction (a sequence of bits), with his private key, stating: *"I acknowledge to send X bitcoins from my address to Alice's destination address"*
3. Joe's wallet broadcasts the signed transactions to one (or more) servers of the bitcoin network
4. the network spreads the information and eventually the transaction is cleared (in 10 minutes or more)
5. Alice's wallet polls the bitcoin network for a transaction having Alice's address as destination and updates the balance on the screen accordingly (confirmation)

The spend transaction

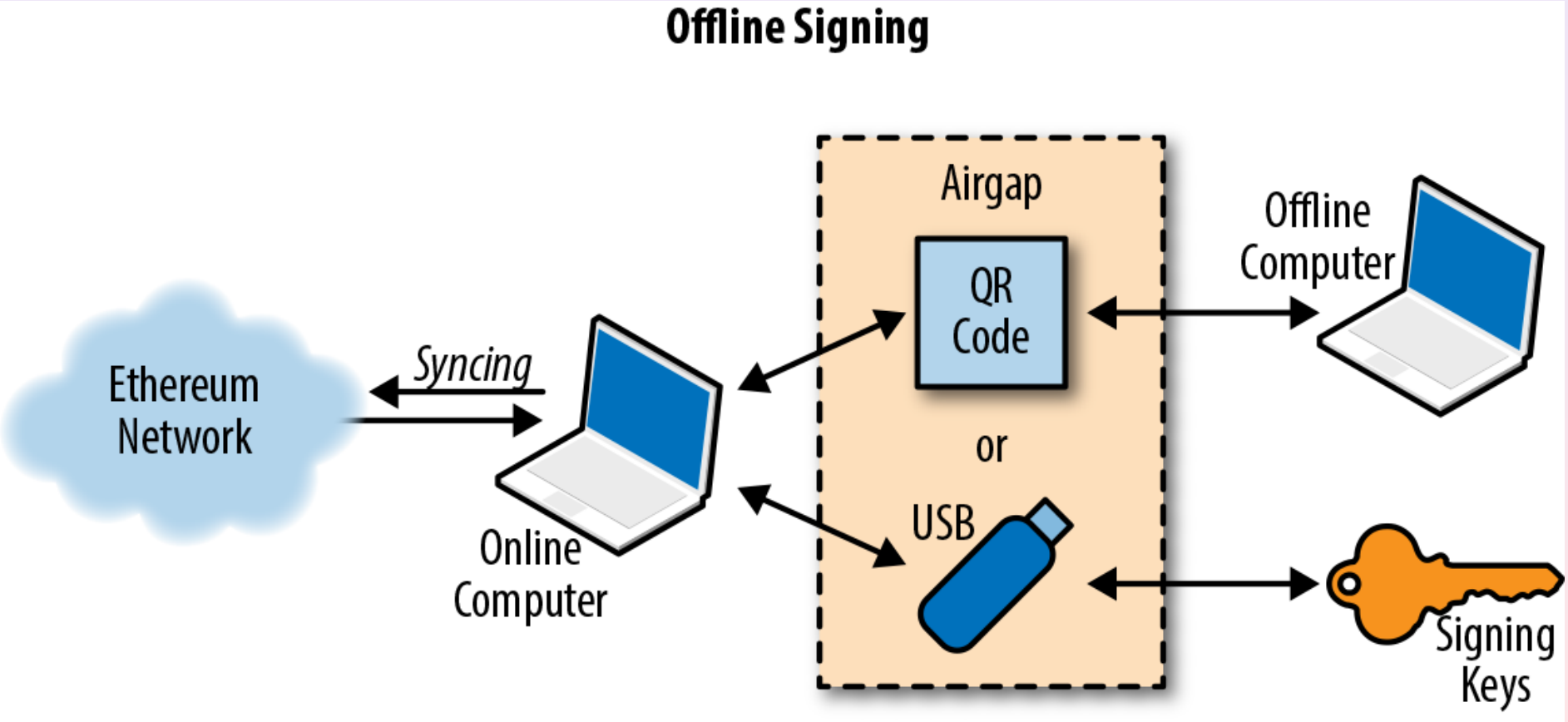
Alice's wallet is charged now and she wants to buy a coffee at Bob's coffee shop:

1. Alice's wallet signs, with her private key, a transaction (a sequence of bits) stating: *"I acknowledge to send Y bitcoins from Alice's address to Bob's destination address"* (some metadata can be added)
2. the transaction is broadcast to one (or more) nodes of the bitcoin network and eventually cleared
3. Alice's wallet polls the bitcoin network for a transaction having her address as source and updates the balance on the screen accordingly (confirmation)

See it online: `https://explorer.btc.com/btc/transaction/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2`

Offline signing

For extreme security, transactions might be signed offline, on an *airgapped* computer, so that private keys are not in memory of any connected computer



The transaction

Transaction Hash

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

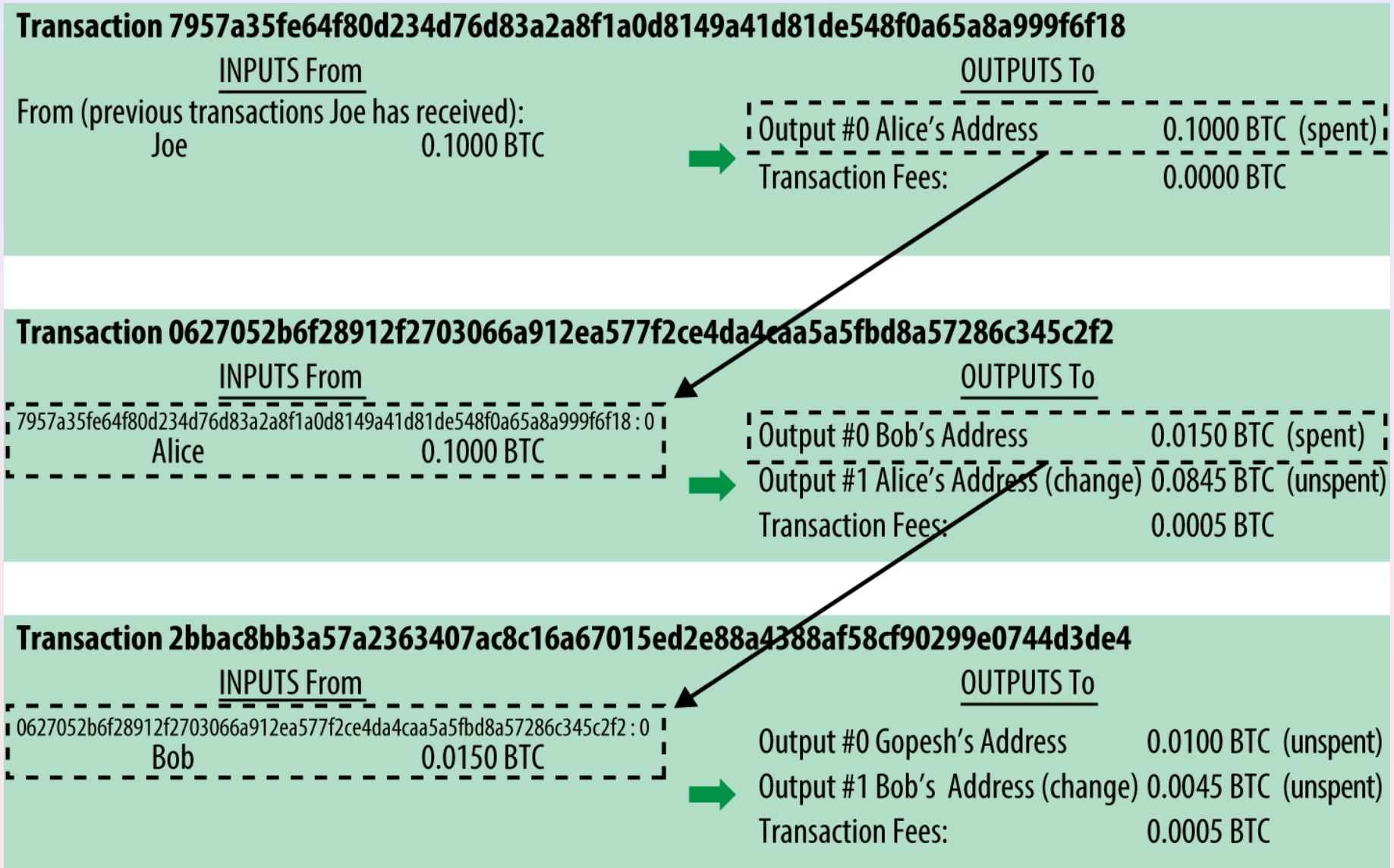
Summary

Height	277,316	Status	Confirmed
Confirmations	392,841	Input	0.10000000 BTC
Timestamp	2013-12-28 00:11:54	Output	0.09950000 BTC
Size (rawtx)	258 Bytes	Sigops	8
Virtual Size	258 Bytes	Fees	0.00050000 BTC
Weight	1,032	Fees Rate(BTC/kVB)	0.00193798 BTC

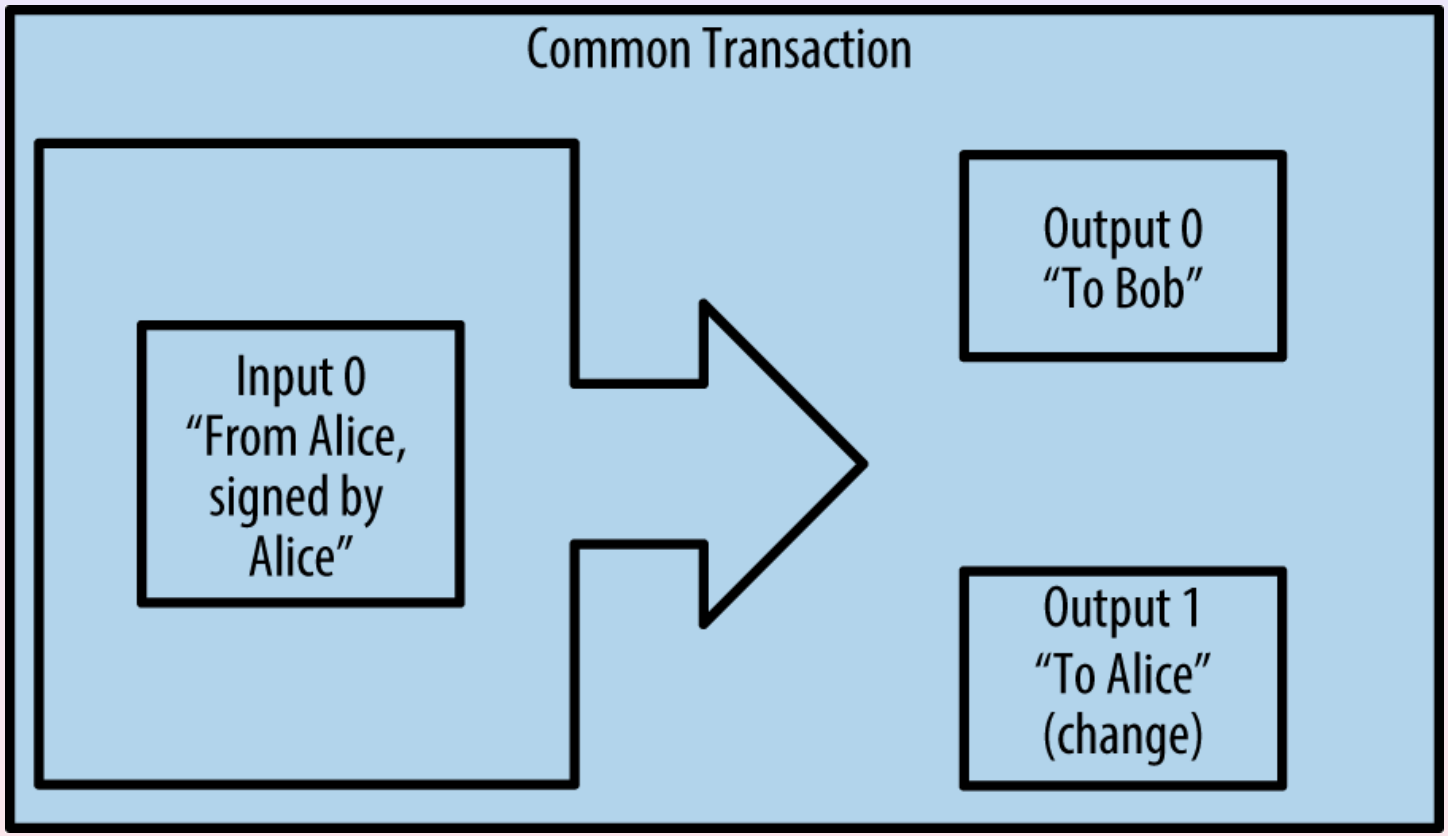
Transaction

Input (1)	0.10000000 BTC	Output (2)	0.09950000 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK	0.10000000	1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA	0.01500000
		1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK	0.08450000

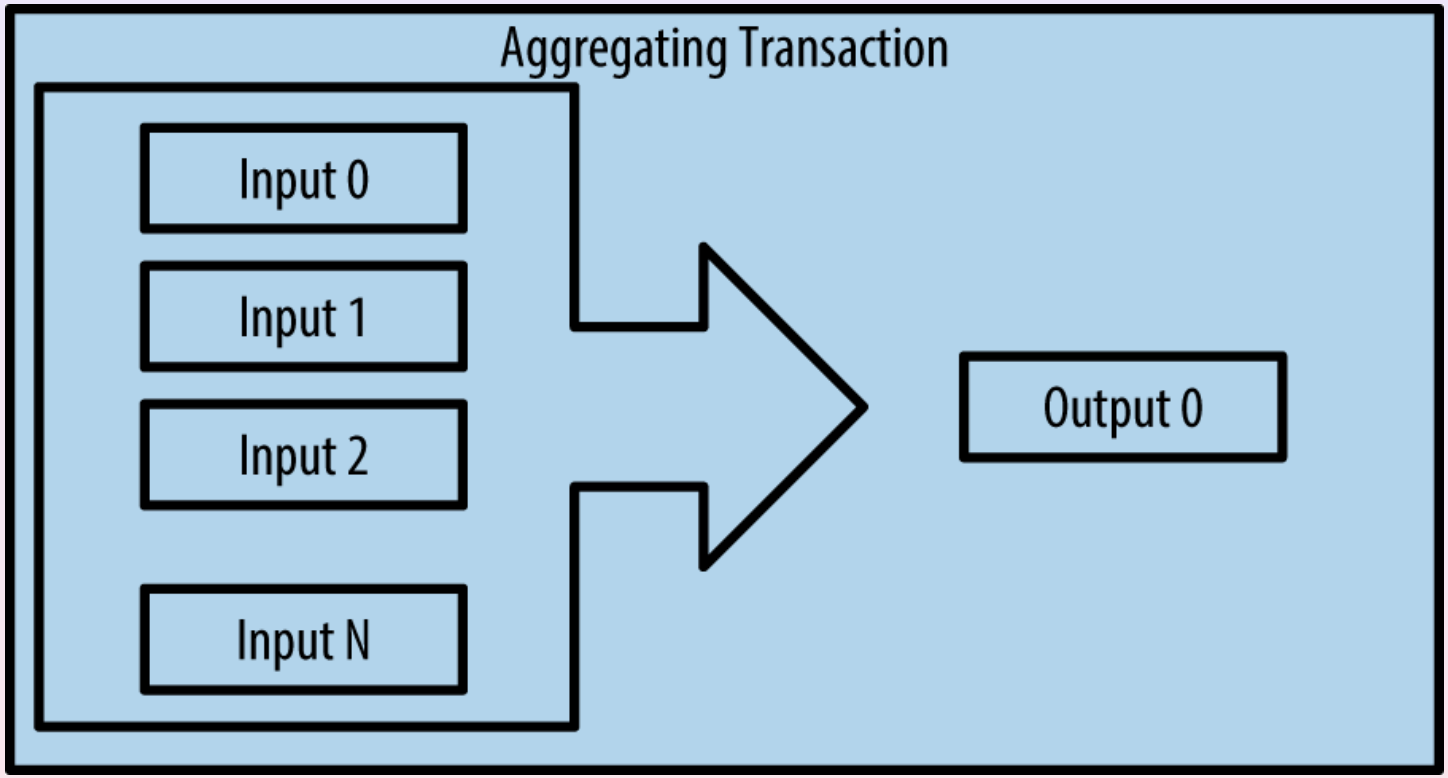
Transactions form a chain, outputs can be change



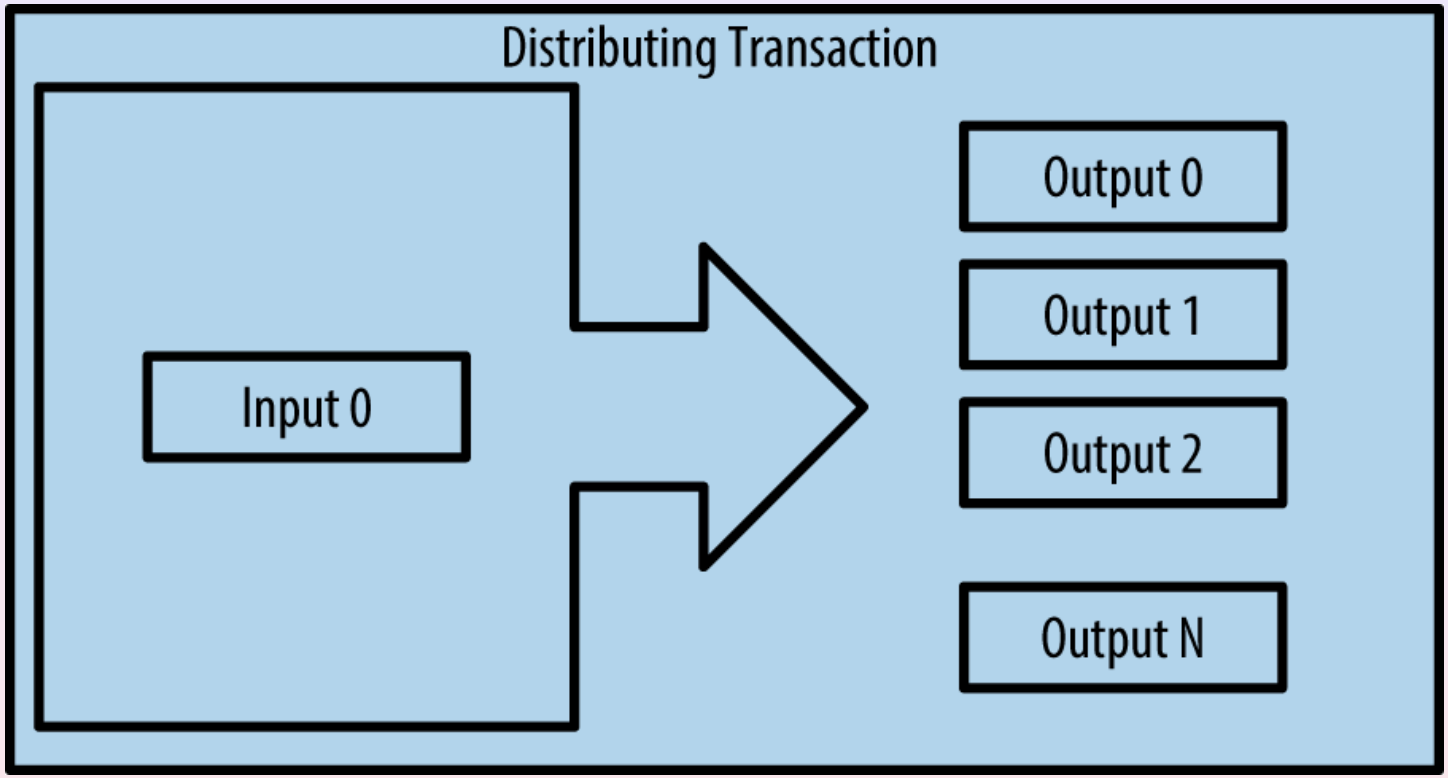
Typical transaction: pay somebody and gets the change



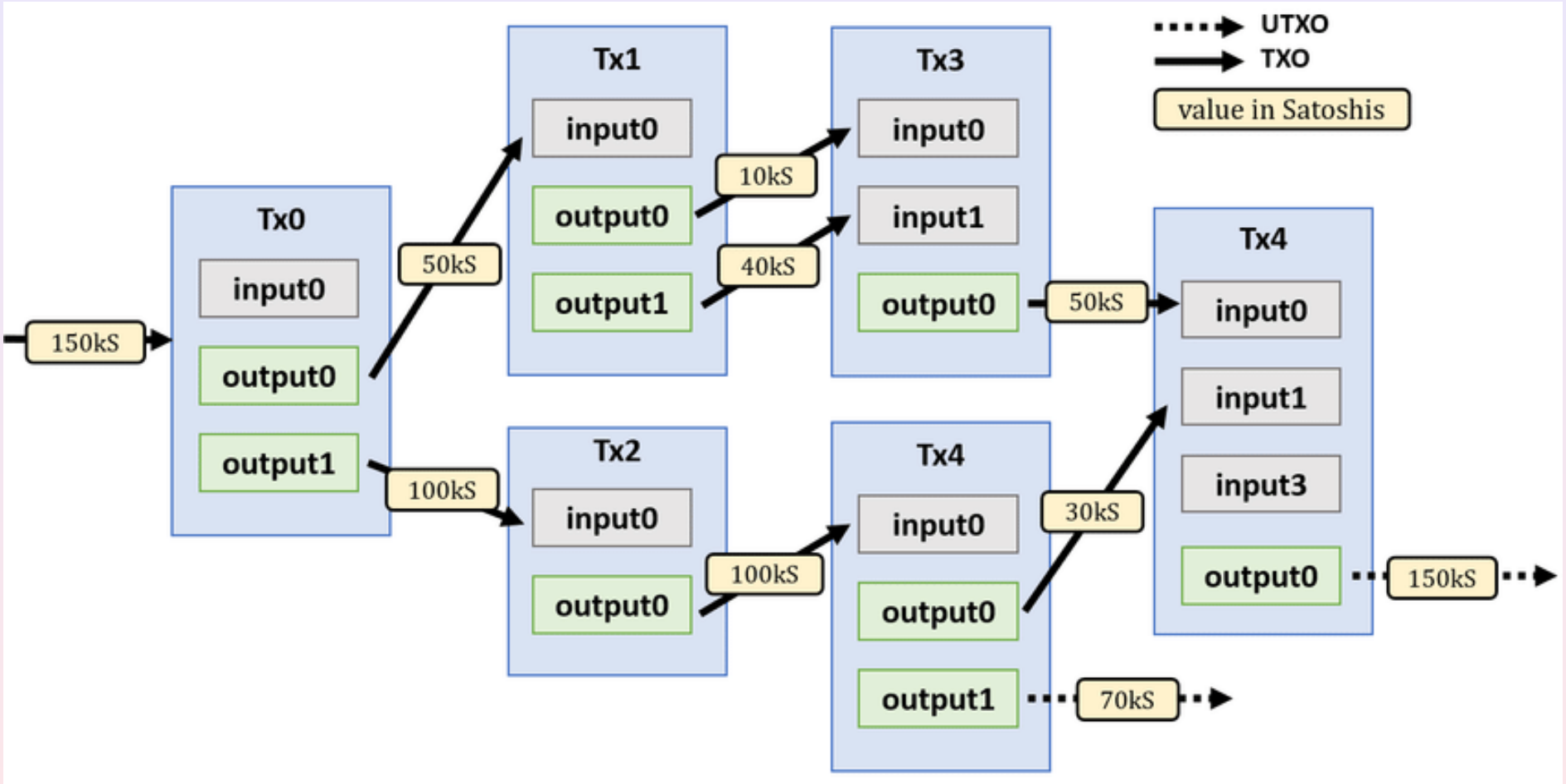
Typical transaction: aggregate small notes into a larger one



Typical transaction: distribution



A DAG of transactions



How Alice's wallet prepares a transaction

1. Alice's wallet keeps a list of all known unspent outputs for the address of Alice
   - if it does not know it, it can query the bitcoin network through an API
2. the wallet selects a subset *inputs* of the unspent outputs, enough to cover the *amount* of the transaction
   - any strategy can be applied here
3. the wallet specifies an output for the destination address of the transaction and the $amount \geq 0$ sent to that output
4. the wallet specifies a second output, for Alice's address itself, and the $change \geq 0$ sent back to Alice
5. the difference

$$fee = \sum inputs - amount - change \geq 0$$

is the network's reward (and protection) for processing the transaction

How Alice sends the transaction

1. Alice's wallet sends the bytes of the transaction to a node of the bitcoin p2p network
   - or it can store it on removable memory support and another device will send it
2. the transaction gets forwarded among all peers (flooding)
3. the wallet of the destination will very soon see a transaction for its address and can assume that it will eventually be processed (unconfirmed transaction)
4. eventually, around 10 minutes later, the transaction will be processed by the network and the wallet of the destination will notice that (confirmed transaction)
5. after some time, around one hour, the transaction can be considered as definitively processed (finalized transaction)

Merchants can wait for 3, 4 or 5 before handling over the good, dependening on the relevance of the transaction

Miners and Rewards

Miners are (some) nodes of the bitcoin network. They receive, forward and aggregate transactions into collectors, called blocks

When a node creates a new block, it has the right to tag the block with a bitcoin address μ, called the miner's address:

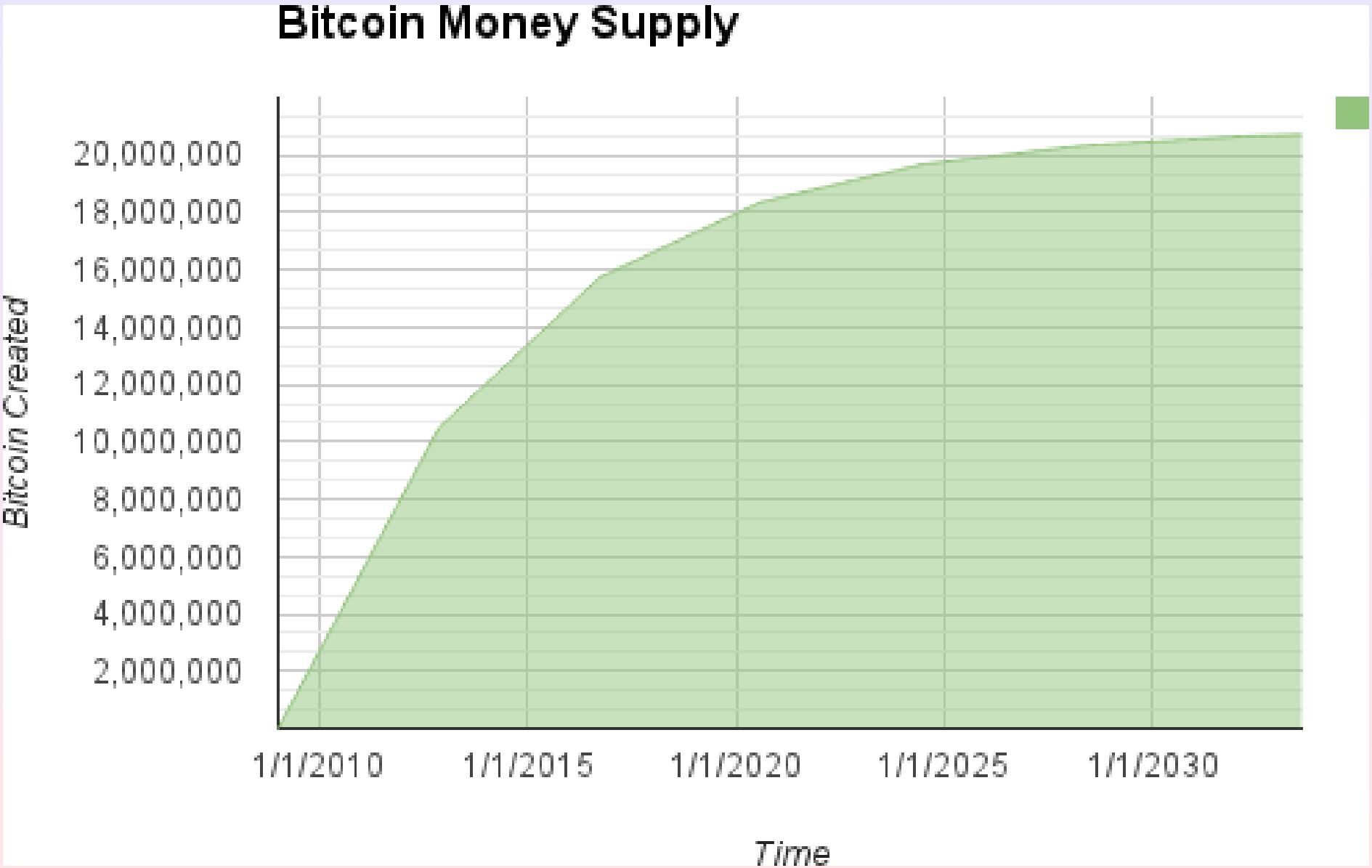
- the fees $\phi_1 \cdots \phi_n$ of the n transactions in the block go to μ
- some amount of money ι is created out of thin air and goes to μ

Typically, μ belongs to the person/organization who owns the machine that runs the node

ι is the inflation: it is computed through a fixed algorithm that makes it decrease with the time and will eventually reach 0, the day when 21,000,000 total bitcoins will be mined

$\Rightarrow$ bitcoin is deflationary

Bitcoin supply over the years



How miners work

1. Each miner listens the p2p network for new transactions and stores them in a temporary area called mempool
2. when enough new transactions are available in the mempool, it selects some of them
   - typically, it selects those with the largest fees, but any other choice is fine
3. it builds a new block (mining):
   - it adds the selected transactions
   - it adds a special coinbase transaction with no inputs, whose only output is μ and whose amount is $\iota + \sum_{i=1}^{n} \phi_i$
   - it tags the block with a reference to the previous block
   - it tags the block with its own miner's address μ
   - it tags the block with a nonce computed by solving an expensive puzzle
   - if no other peer has been faster, it forwards the new block to all its peers

Fees

`https://bitcoinfees.earn.com/`

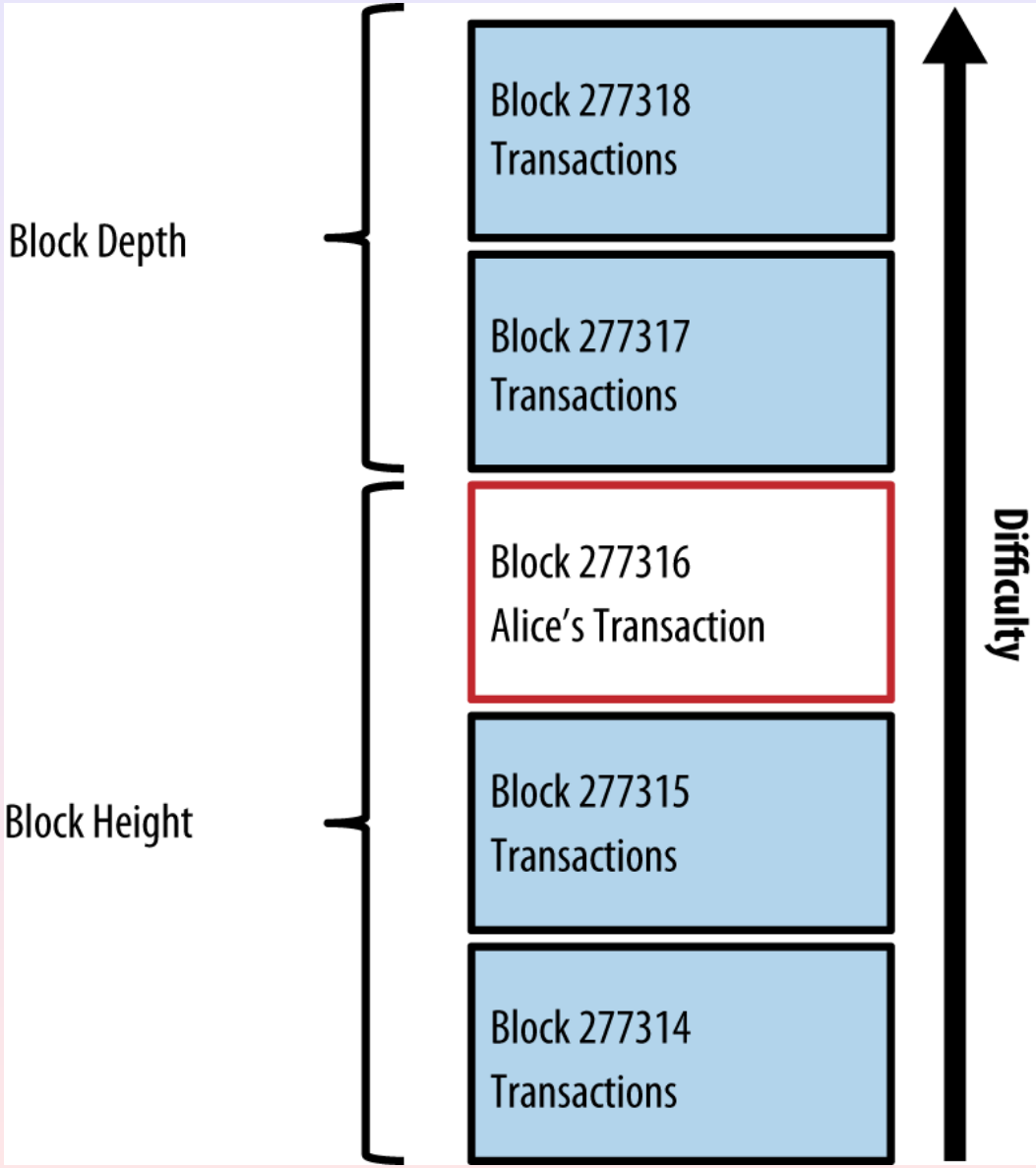
For example

- if I want a transaction processed in one hour at most, I must offer 0.0000009 BTC per byte
- a simple transaction is in average 250 bytes long
- a BTC is 42780 euros worth (as of today)

$\Rightarrow$ I must offer $0.0000009 * 250 * 42780 = 9.6255$ euros as fee

The size, not the value of the transaction is relevant!

Block's height, depth and confirmations



Cryptography

Bitcoin uses cryptography for many goals

Hashing

- as synthetic representation of very long information (compaction)
- as machine-independent pointers to data structures (reference)
- to solve a mathematical puzzle (mining)

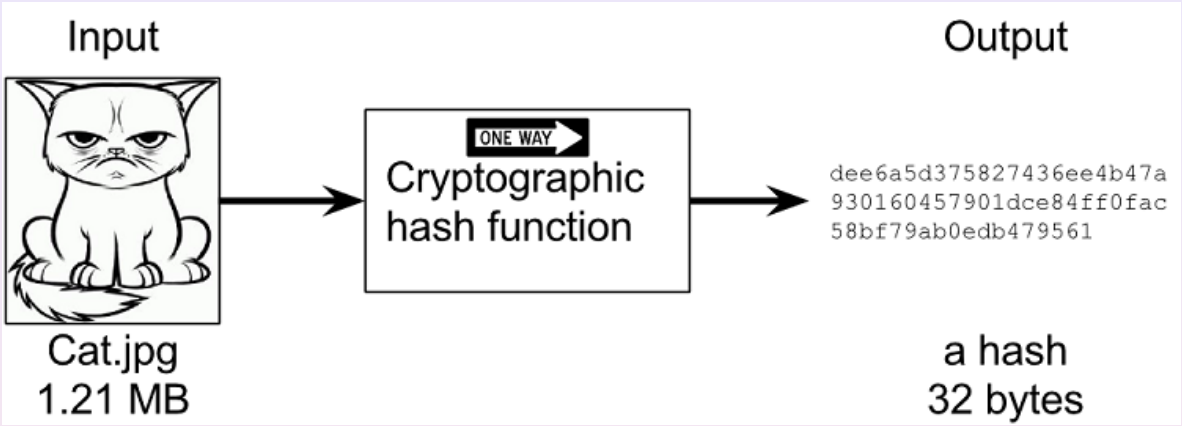
Signature

- to prove the identity of the sender of transactions (witness)

Bitcoin doesn't use cryptography to hide data (everybody sees everything)

Cryptographic hash functions

A hash function maps data of arbitrary size to data of fixed size



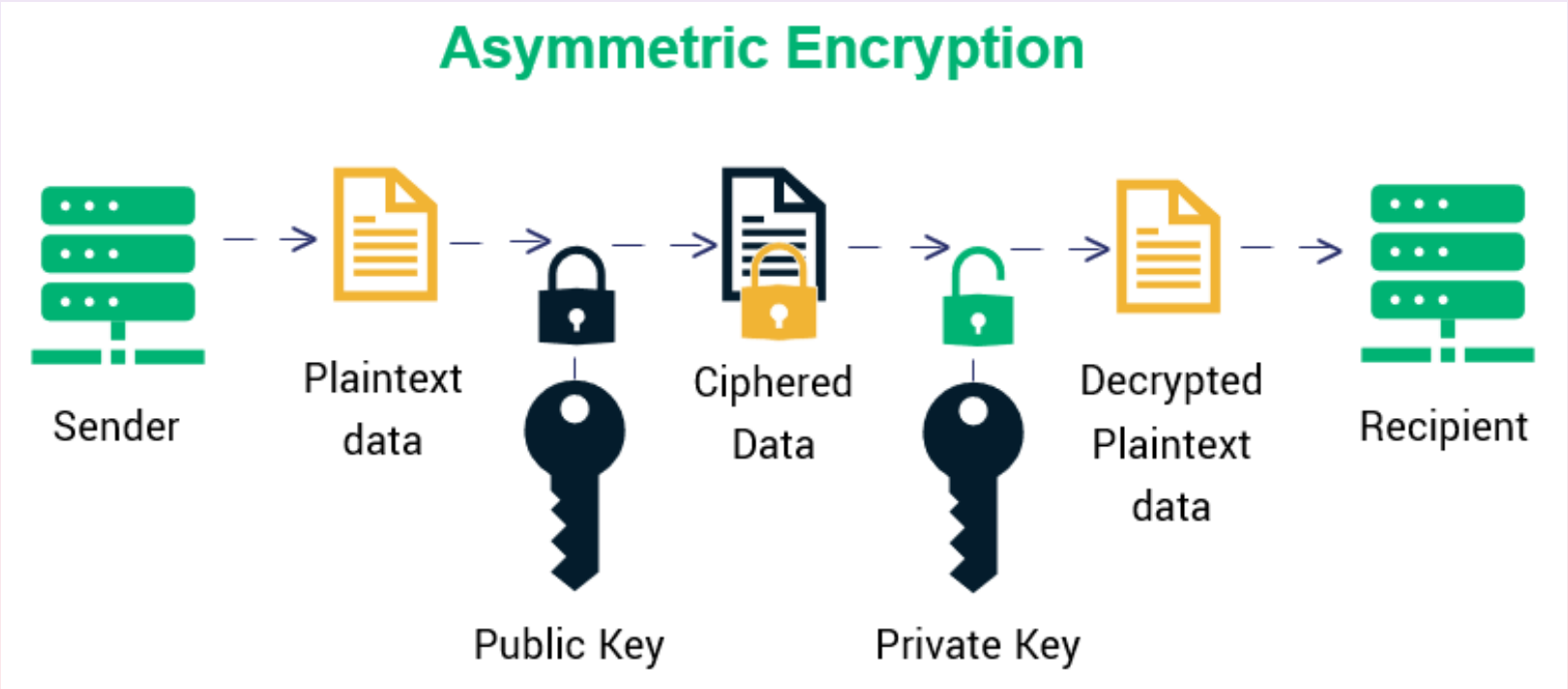
Cryptographic hash function

1. deterministic
2. verifiable (in linear time)
3. non-correlated (small change of input $\Rightarrow$ extensive change of hash)
4. irreversible (*one-way*)
5. collision-protected: difficult to compute two inputs with same hash

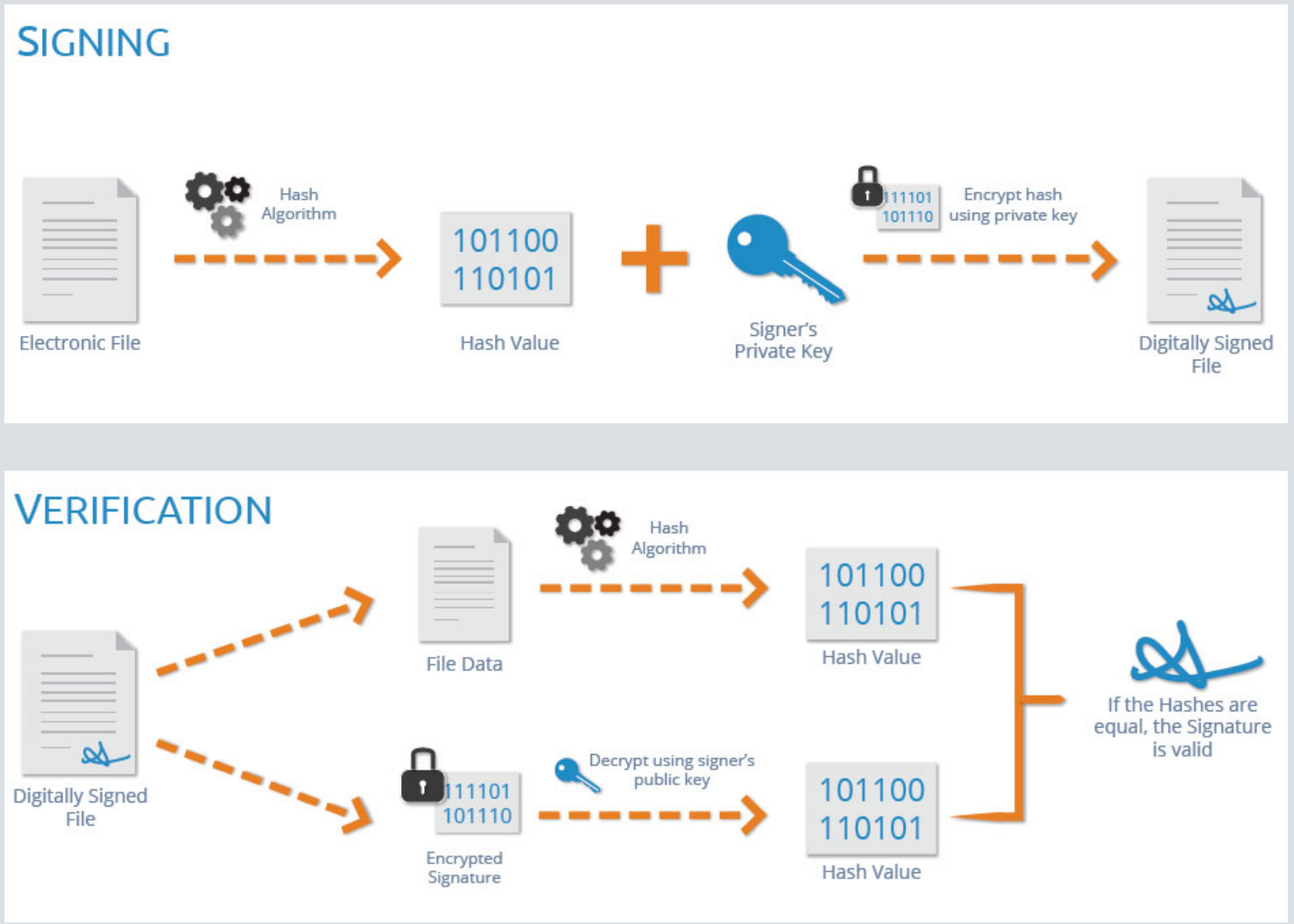
Asymmetric encryption

$$\text{Secure random generator} \Rightarrow \text{private key} \xrightarrow{\text{ECM}} \text{public key}$$

Ellyptic Curve Multiplication (ECM) is an abstraction or hashing algorithm

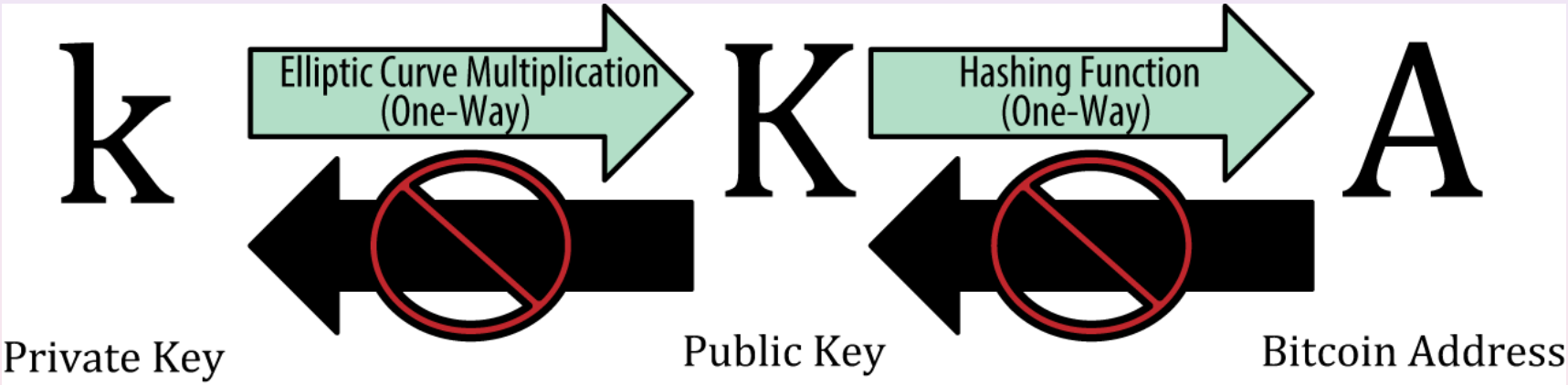


Digital signature



Bitcoin addresses

- private key: 256 bits (32 bytes)
- public key: 256 bits
- bitcoin address: 160 bits (20 bytes *ie.*, 40 hex digits)



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

Base58 representation of addresses

40 hex digits is too long!

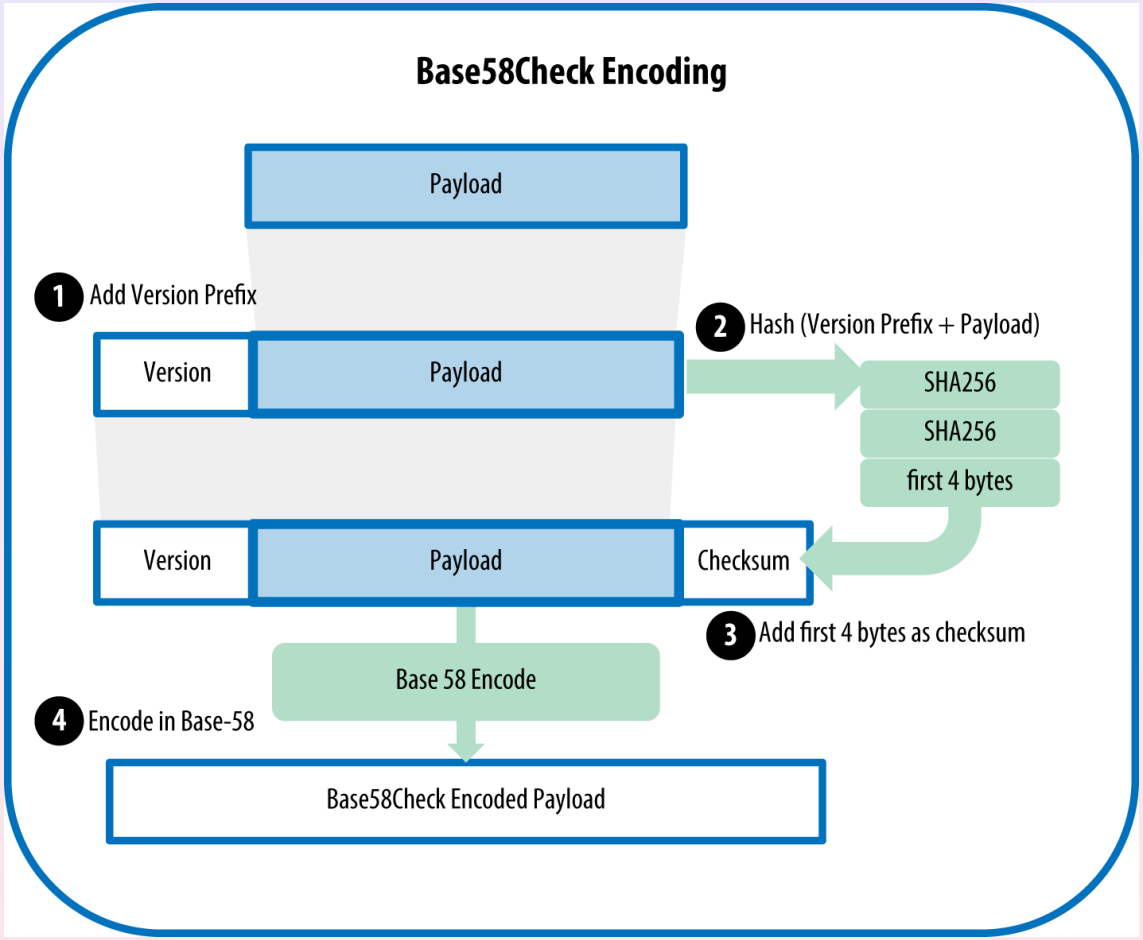
Base58

A representation of natural numbers in base 58, using the following symbols for the 58 digits:
`123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz`

28 Base58 digits are enough to cover 40 hex digits

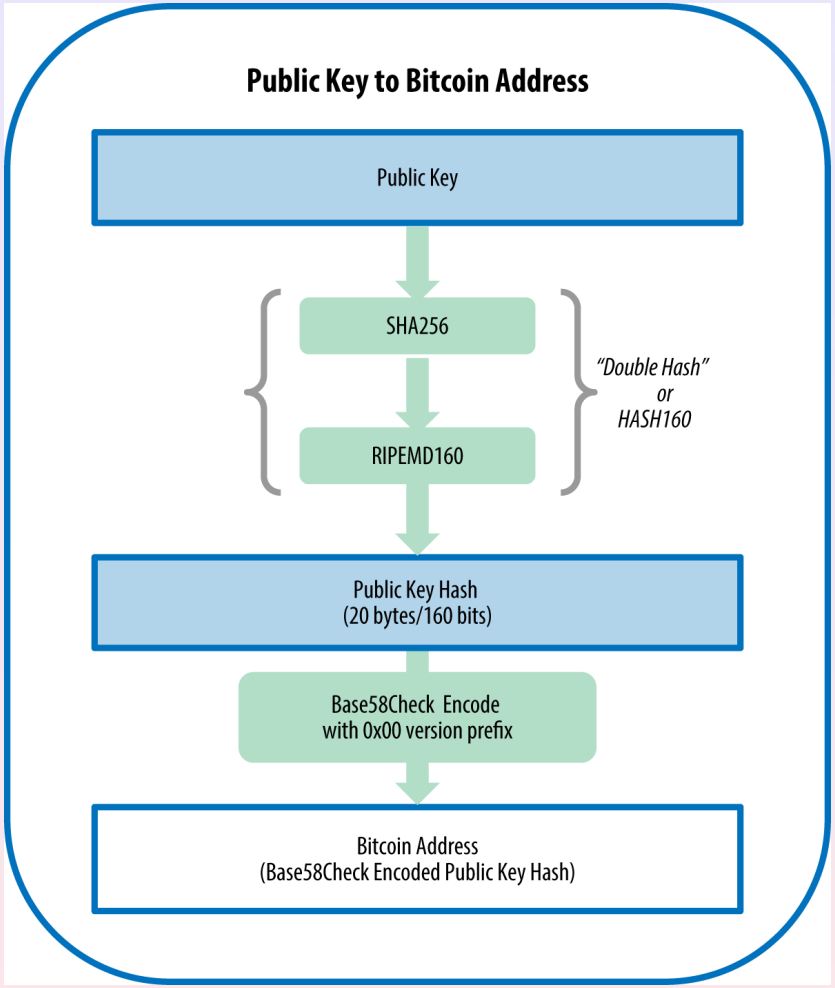
Base58Check adds a checksum

Base58 addresses are too easy to spell wrong!



34 Base58Check digits are enough for a bitcoin address

Put it all together



Public key: `0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa`

Address: `1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK`

Libraries

There are many libraries for computing private and public keys

- OpenSSL (`https://www.openssl.org`)
- libsecp256k1 (`https://github.com/bitcoin-core/secp256k1`)
- bitcoinj (`https://bitcoinj.org`)
- Bouncy Castle (`https://www.bouncycastle.org`)

Vanity addresses

Alice wants a bitcoin address whose first letters are "Love"

`1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33`

- vanity addresses can only be found by brute-force computation
- realistic up to 7 characters
- vanity addresses are typically recycled for many transactions, since they are hard to find

  $\Rightarrow$ reduced security

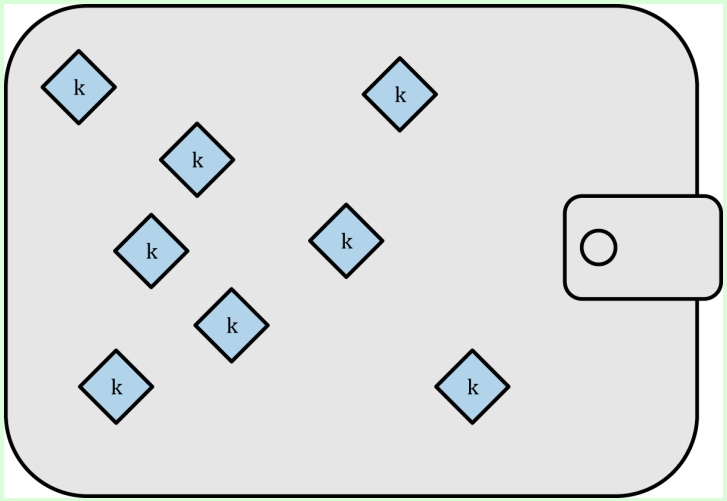
`https://vanitypool.appspot.com/`

Wallets

A wallet is a software application that keeps track of keys. It creates and broadcasts transactions signed with those keys

A wallet is the file where key pairs are stored

The simplest: a nondeterministic wallet (a bunch of keys, Type-0)

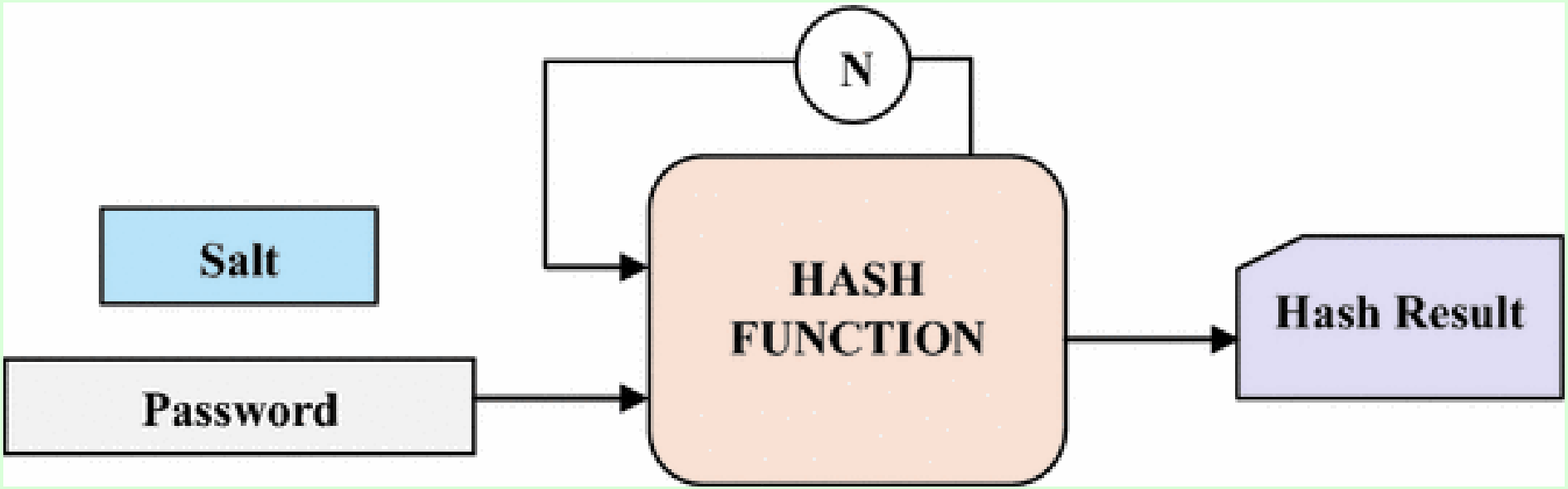


Hard to backup!

Nondeterministic wallets

Password stretching

To make dictionary attacks harder, nondeterministic wallets repeatedly apply a hash function to each password, using th hash result as the actual encryption password

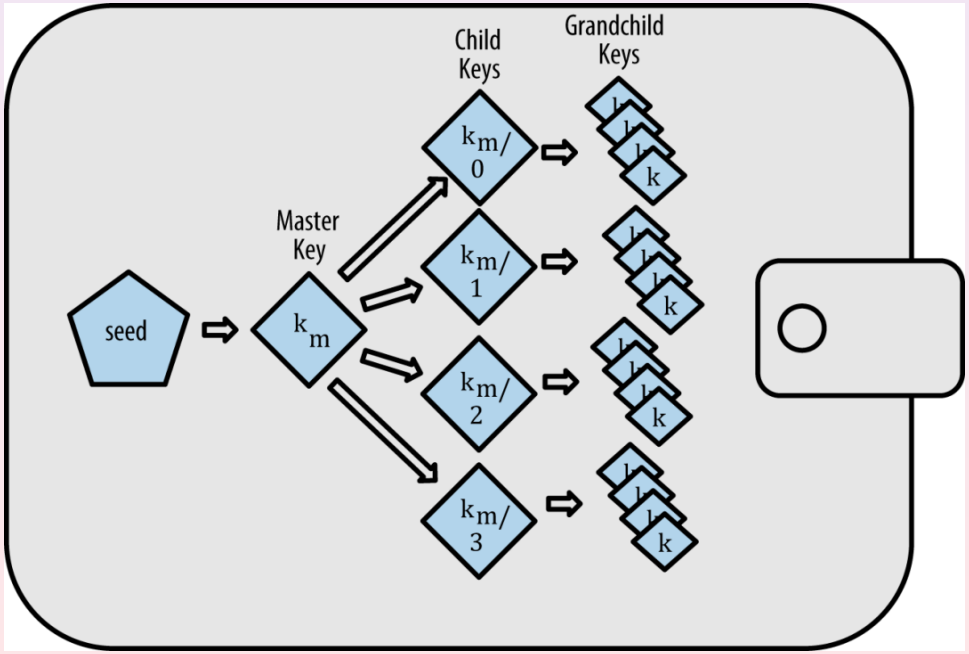


Deterministic wallets (BIP-32, seeded, Type-1)

Streams of keys

They store just a seed, from which streams of keys are derived:

- only the seed must be stored, backed up and kept private
- different streams can be allocated for different goals or to distinct enterprise departments



Encoding the seed as mnemonic code words (BIP-39)

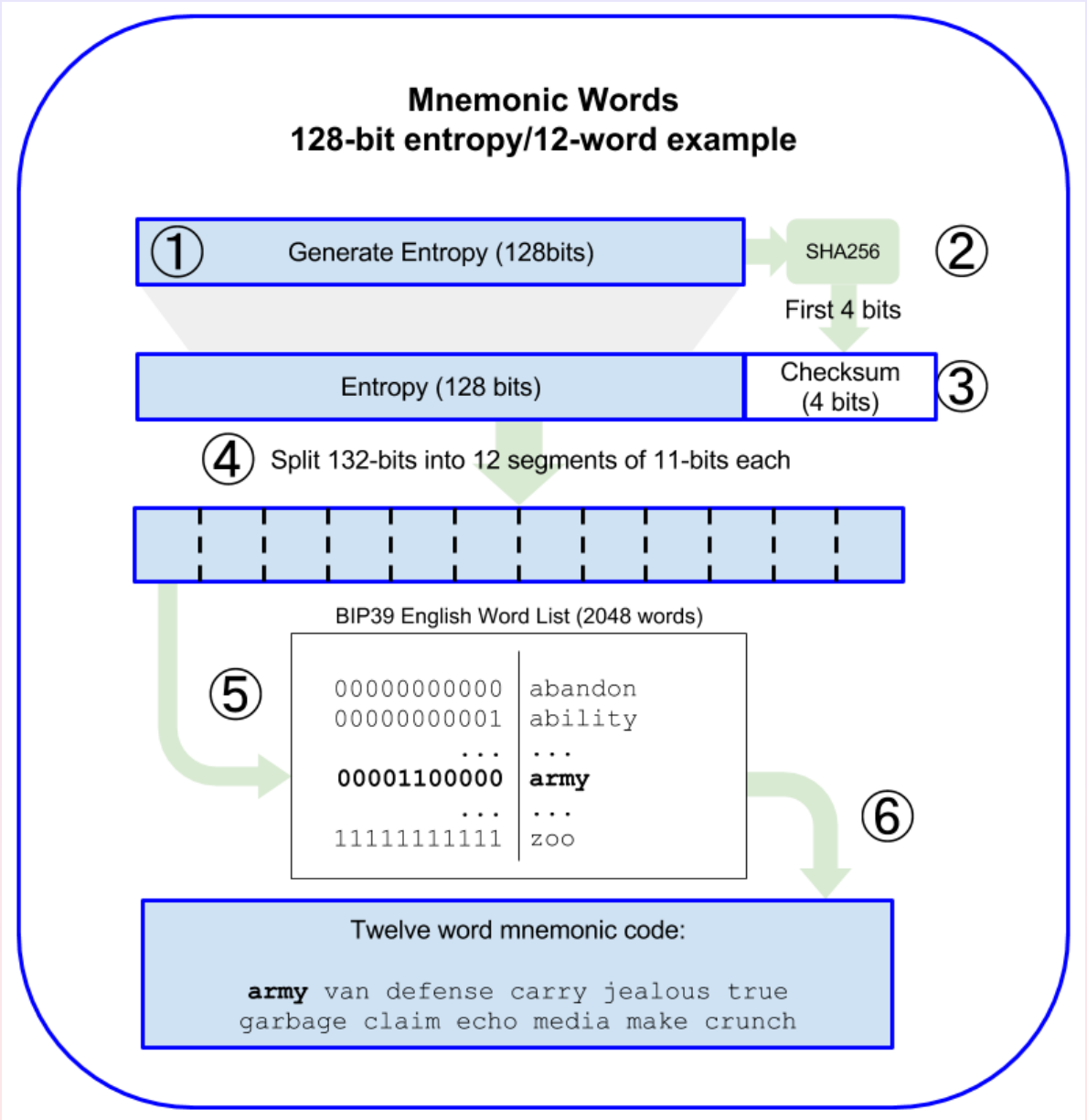
What is simpler and less error-prone to remember and backup?

`0C1E24E5917779D297E14D45F14E1A1A`

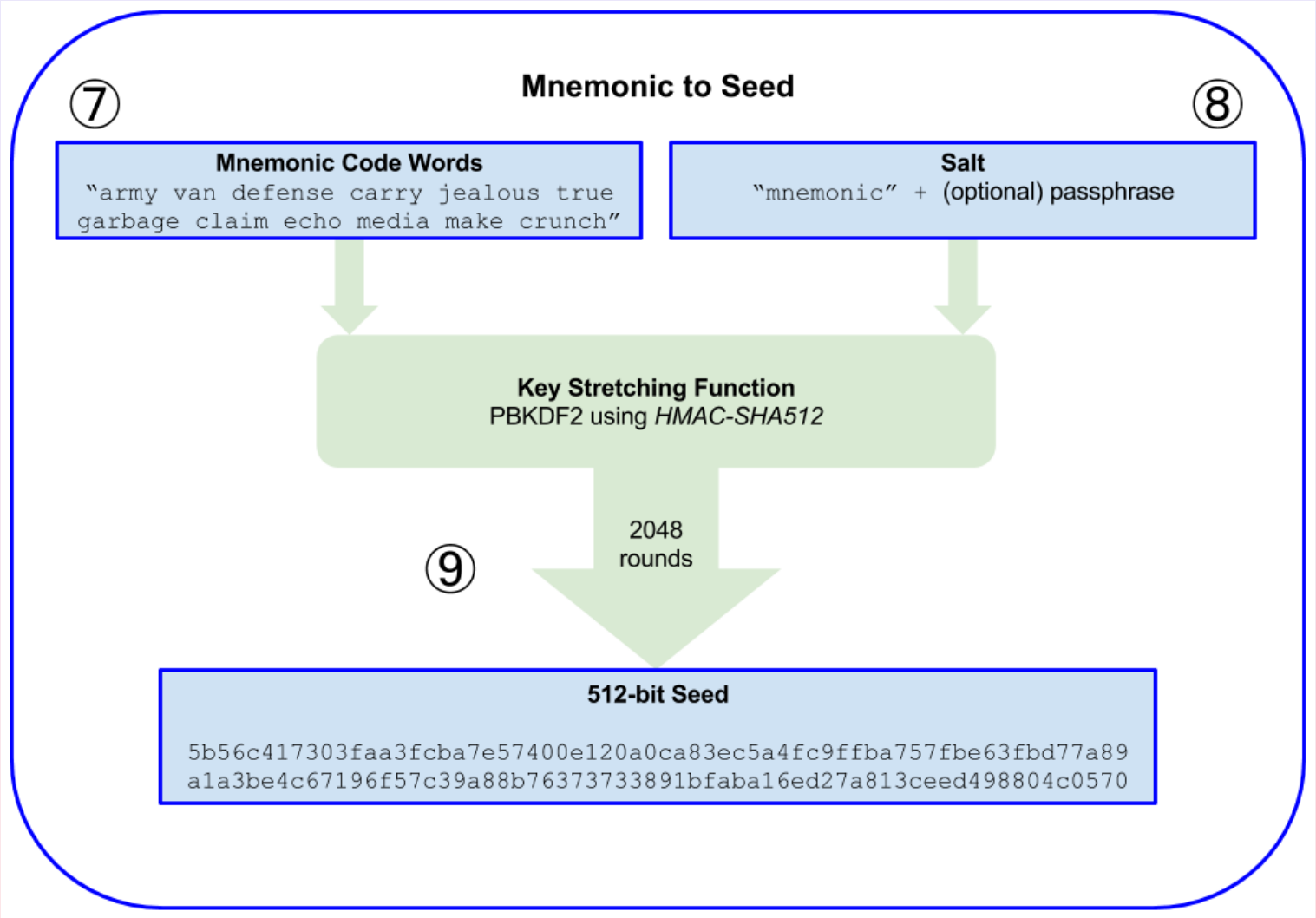
or rather

`army van defense carry jealous true`
`garbage claim echo media make crunch`

Generation of the mnemonic words



Generation of the seed from the mnemonic words



Practical considerations

- keep the mnemonic on paper, never in digital form
- memorize the passphrase
- let somebody else memorize the passphrase (at least, before you die)
- hint towards a wrong passphrase, leading to a *duress wallet*

There are no *wrong* passphrases

Every passphrase leads to some wallet, which unless previously used will be empty

Parent-to-child key derivation (BIP-32)

- there is an algorithm to derive private child keys from private parent keys
- there is an algorithm to derive public child keys from public parent keys
  - ideal for storing only public keys in a cloud server
- the derivation can be *hardened*, so that the leak of a single private key does not allow one to reconstruct the other private keys

HD path	Key described
`m/0`	The first (`0`) child private key of the master private key (`m`)
`m/0/0`	The first child private key of the first child (`m/0`)
`m/0'/0`	The first normal child of the first *hardened* child (`m/0'`)
`m/1/0`	The first child private key of the second child (`m/1`)
`M/23/17/0/0`	The first child public key of the first child of the 18th child of the 24th child

Giving structure to the key hierarchy (BIP-44)

`[m|M]/purpose'/coin_type'/account'/change/address_index`

The value of `purpose` is fixed to `44`. For Bitcoin, `coin_type` is fixed to `0` (while it is `60` for Ethereum, with `change` fixed to `0`):

`[m|M]/44'/0'/account'/[0|1]/address_index`

HD path	Key described
`M/44'/0'/0'/0/2`	The 3rd receiving public key for the primary bitcoin account
`M/44'/0'/3'/1/14`	The 15th change-address public key for the 4th bitcoin account
`M/44'/60'/0'/0/2`	The third public key for the primary Ethereum account
`m/44'/2'/0'/0/1`	The second private key for the primary Litecoin account

The transaction, again

Transaction Hash

0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

Summary

Height	277,316	Status	Confirmed
Confirmations	392,841	Input	0.10000000 BTC
Timestamp	2013-12-28 00:11:54	Output	0.09950000 BTC
Size (rawtx)	258 Bytes	Sigops	8
Virtual Size	258 Bytes	Fees	0.00050000 BTC
Weight	1,032	Fees Rate(BTC/kVB)	0.00193798 BTC

Transaction

Input (1)	0.10000000 BTC	Output (2)	0.09950000 BTC
1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK	0.10000000	1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA	0.01500000
		1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK	0.08450000

The real transaction

no coins, no senders, no recipients no balances, no accounts, no addresses


```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The real transaction

two new UTXOs (unspent transaction outputs)

```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG",
    }
  ]
}
```

The real transaction

reference to an old UTXO (soon to be TXO)


```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The real transaction

the amount of the first new UTXO (in satoshis)

```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The real transaction

the unlocking or witness script of the first new UTXO (crypto-puzzle)


```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The real transaction

the hash of the transaction whose voutth UTXO is being spent

```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The real transaction

the unlocking script (usually digital signature + public key)


```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG",
    }
  ]
}
```

The real transaction

scripts are written in the Script programming language


```
{
  "vins": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "unlock": "3045022100884d142d86652a3f47... 0484ecc0d46f..."
    }
  ],
  "vouts": [
    {
      "value": 0.01500000,
      "lock": "DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
               EQUALVERIFY CHECKSIG"
    },
    {
      "value": 0.08450000,
      "lock": "DUP HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8
               EQUALVERIFY CHECKSIG"
    }
  ]
}
```

The Script programming language

Reverse-polish stack-based stateless language

- ✓ sequence
- ✓ conditional
- ✗ repetition

$\Rightarrow$ Turing incomplete

Why Turing incomplete?

1. predictable execution time
2. guaranteed termination

denial-of-service attacks are impossible at language level

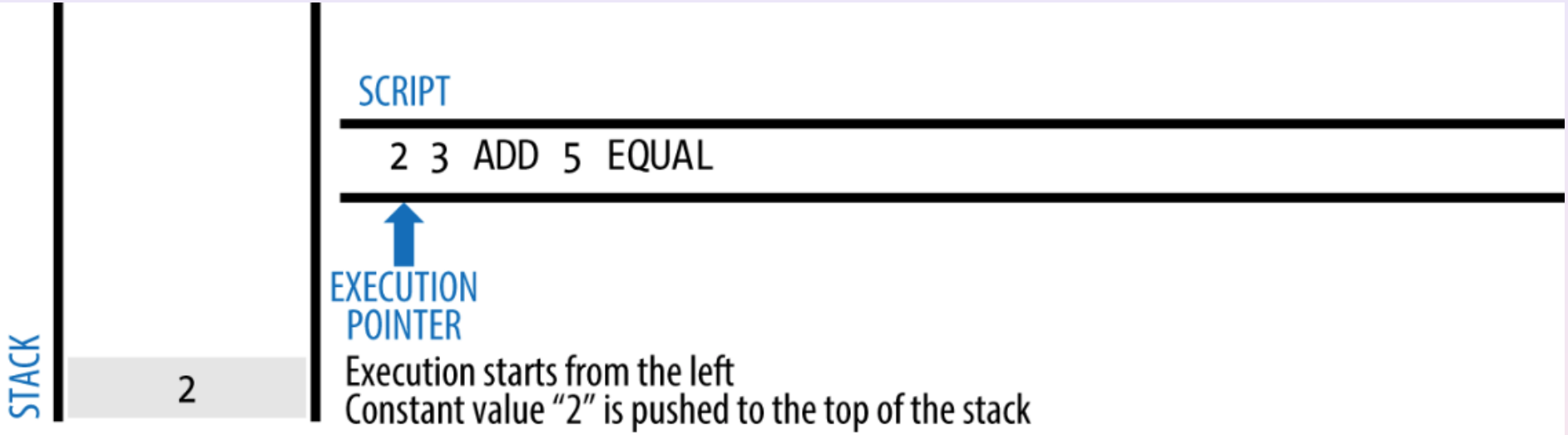
Script validity

A program in the Script language is valid if its execution does not stop with failure and terminates with a stack whose topmost element is `TRUE`

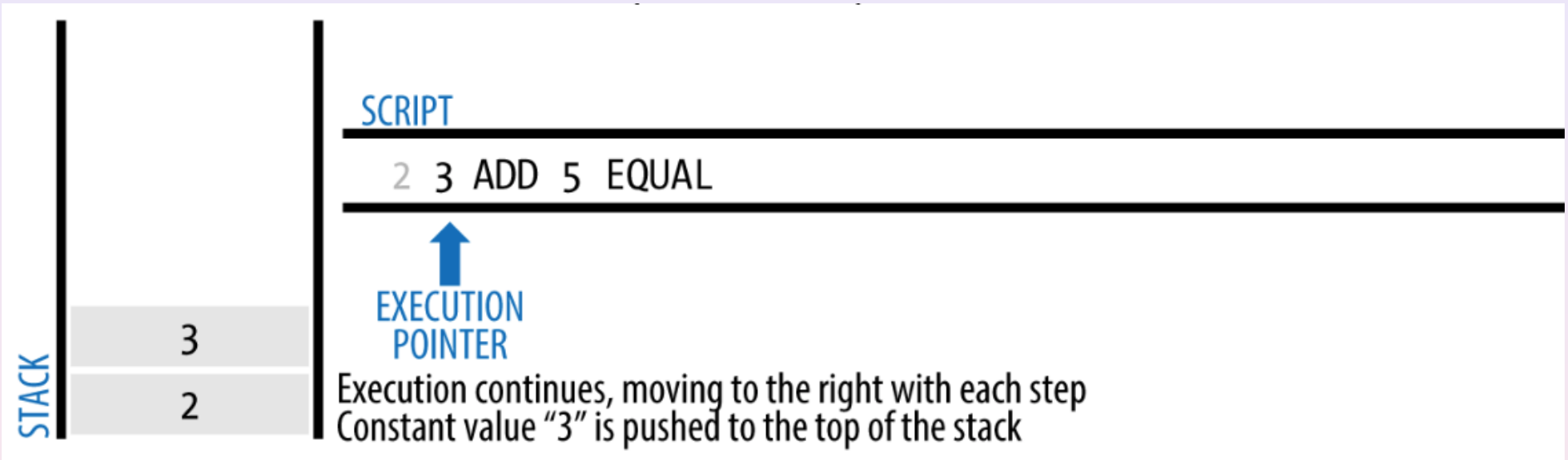
Execution proceeds left-to-right

Let us execute `2 3 ADD 5 EQUAL` to see if it's valid

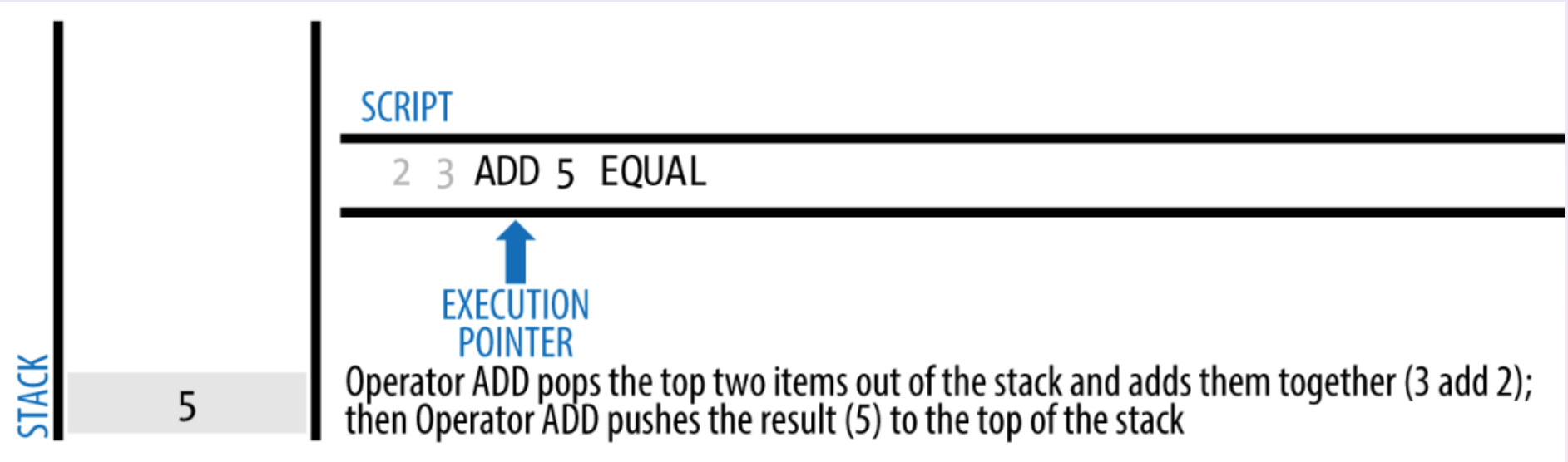
2 3 ADD 5 EQUAL



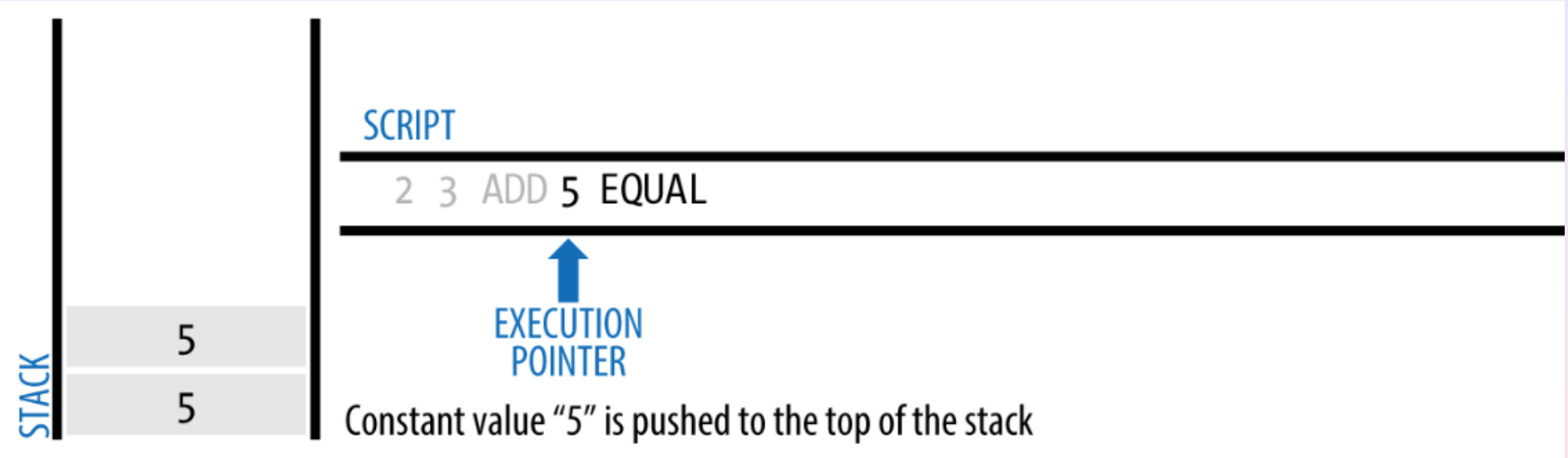
2 3 ADD 5 EQUAL



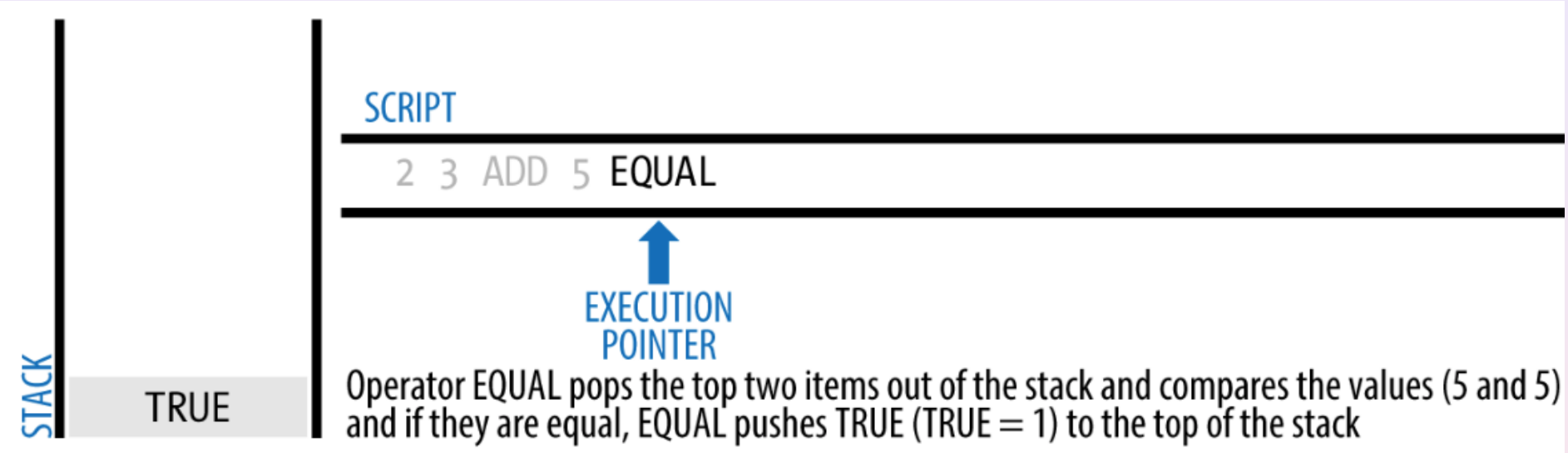
2 3 ADD 5 EQUAL



2 3 ADD 5 EQUAL



2 3 ADD 5 EQUAL



The program is valid!

Other examples of (in-)valid scripts

These are all valid

- `TRUE`
- `FALSE TRUE`
- `2 7 ADD 3 SUB 1 ADD 7 EQUAL`
- `2 7 EQUAL IF FALSE ELSE TRUE ENDIF`

These are all invalid

- `FALSE`
- `2 7 EQUAL`
- `2 7 EQUAL IF TRUE ELSE FALSE ENDIF`
- `2 7 EQUAL TRUE ENDIF`

The validation algorithm for bitcoin transactions

```
previous_tx = { // this transaction has hash H
  "vins": .....
  "vouts": [ { "value": .....,     "lock": "....." }, ..... ]
}

tx = {
  "vins": [ { "txid": H,     "vout": .....,     "unlock": "....." }, ..... ],
  "vouts": .....
}
```

```
boolean is_valid(Transaction tx) {
  for each (txid, vout, unlock) in tx.vins
    previous_tx = get_transaction(txid)
    lock = previous_tx.vouts[vout].lock
    if (unlock lock is invalid)
      return false

  return true
}
```

The typical P2PKH script (*pay to publickey hash*)

"I want to send some value to address"

```
previous_tx = { // this transaction has hash H
  "vins": .....
  "vouts": [{ "value": ..., "lock": DUP HASH160 <address> EQUALVERIFY CHECKSIG },
             .....]
}
```

"I'm address, here is my signature, use that value"

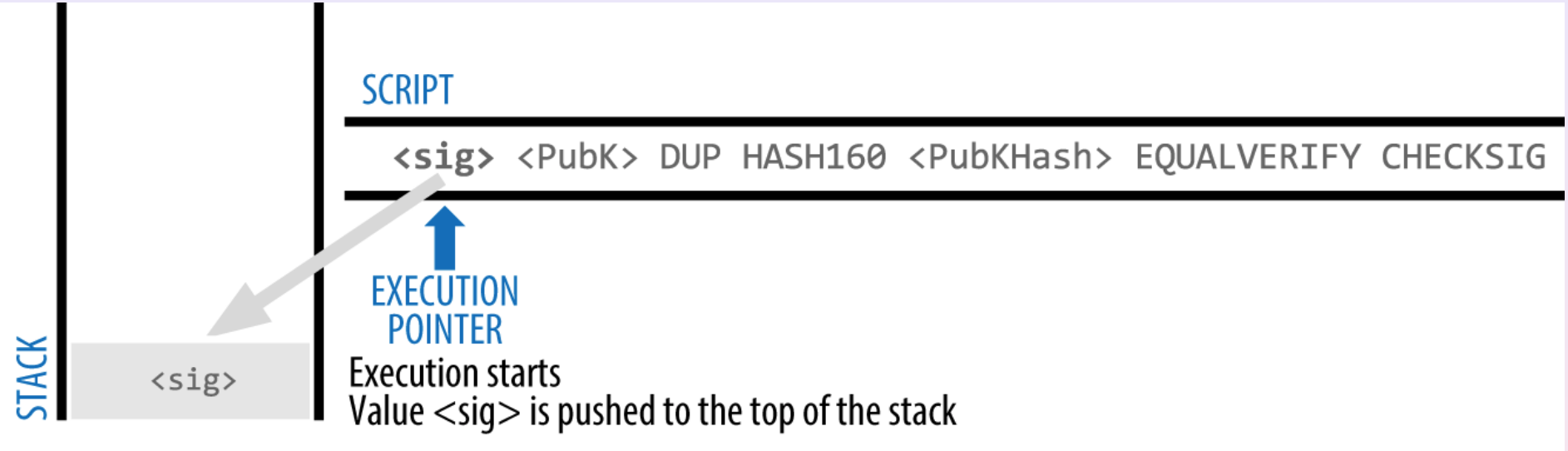
```
tx = {
  "vins": [ { "txid": H,   "vout": .....,   "unlock": <sig> <PubK>}, ..... ],
  "vouts": .....
}
```

unlock lock

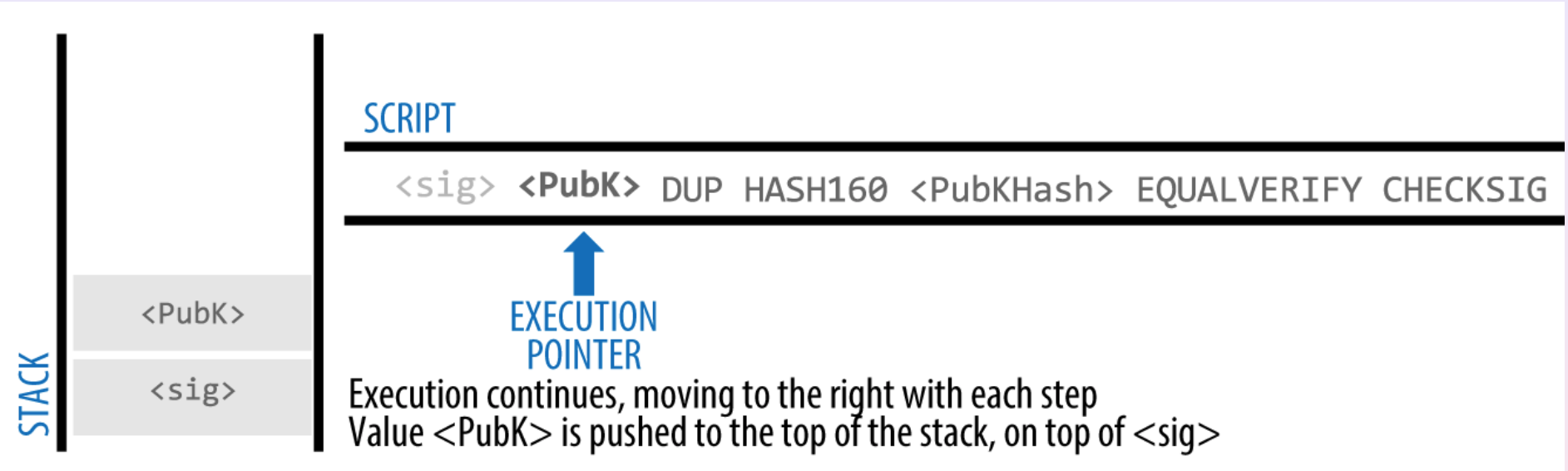
```
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG
```

The bitcoin address is often referred to as PublicKHash

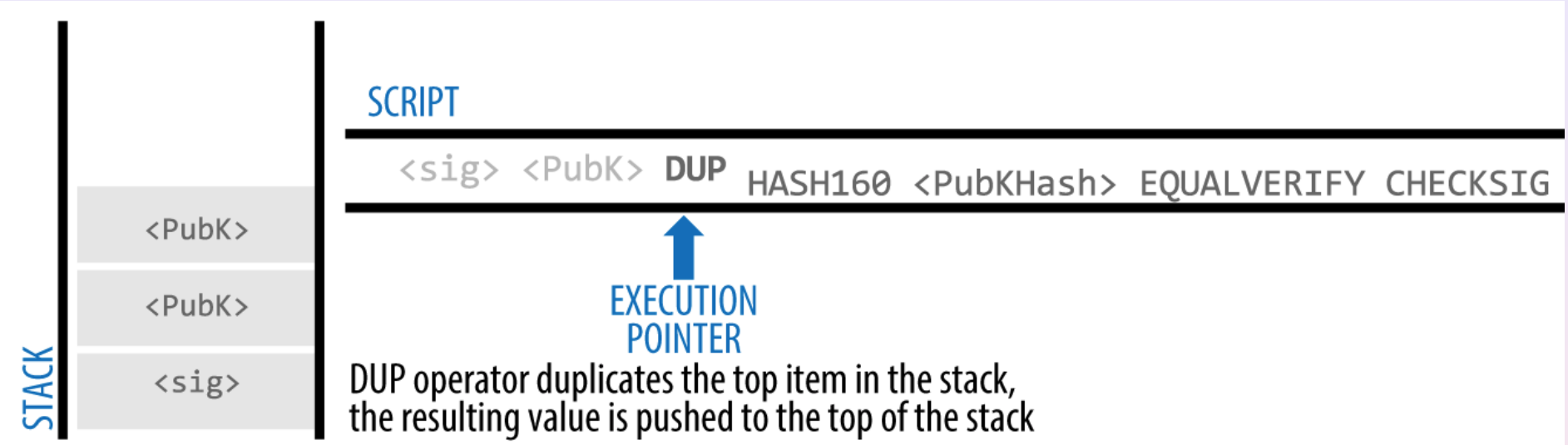
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



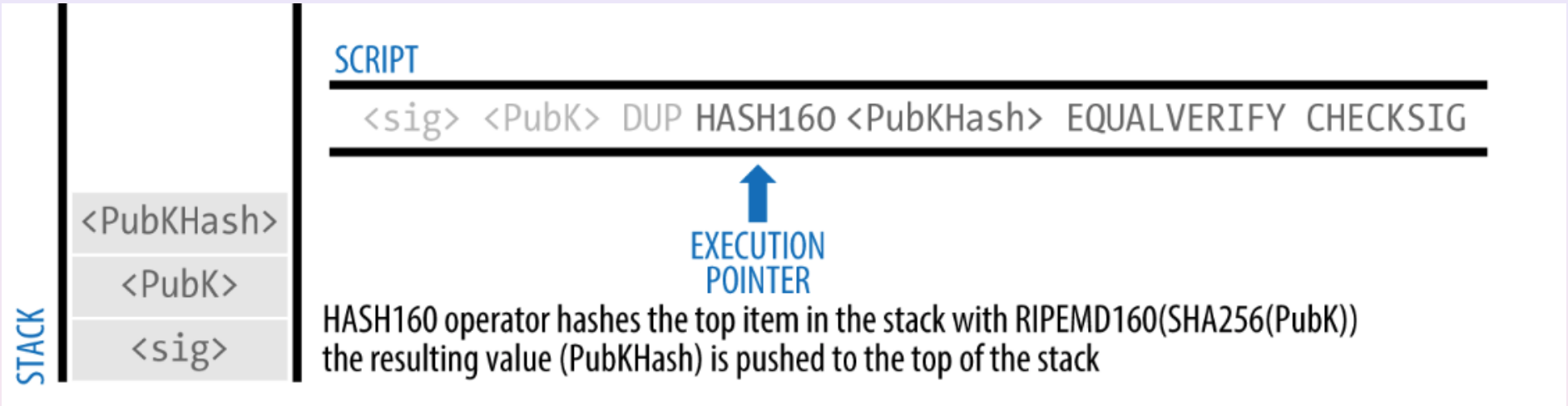
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



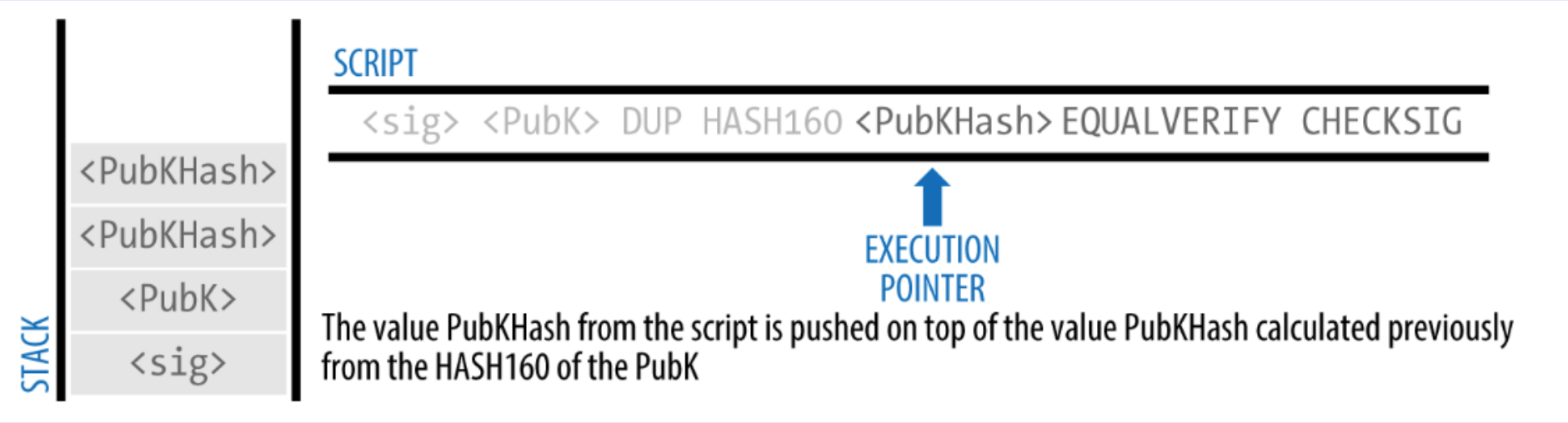
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



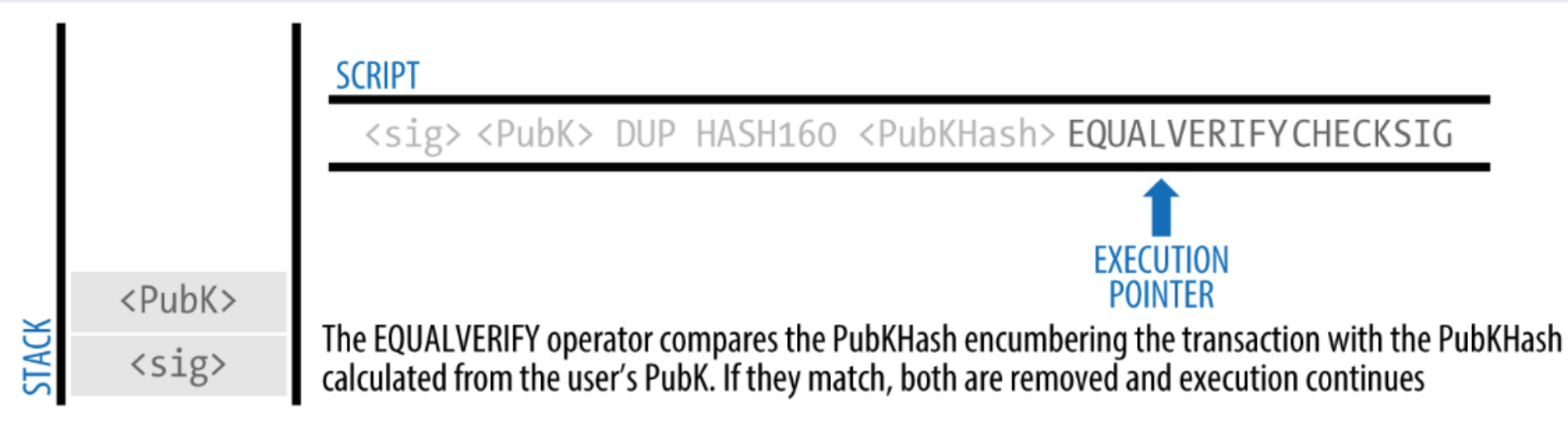
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



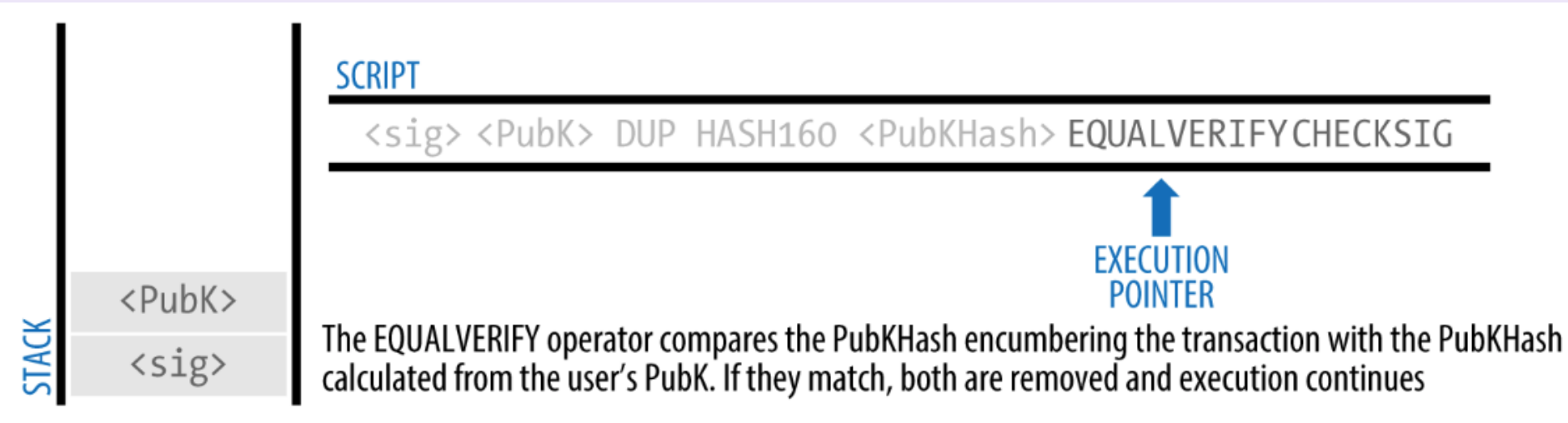
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG

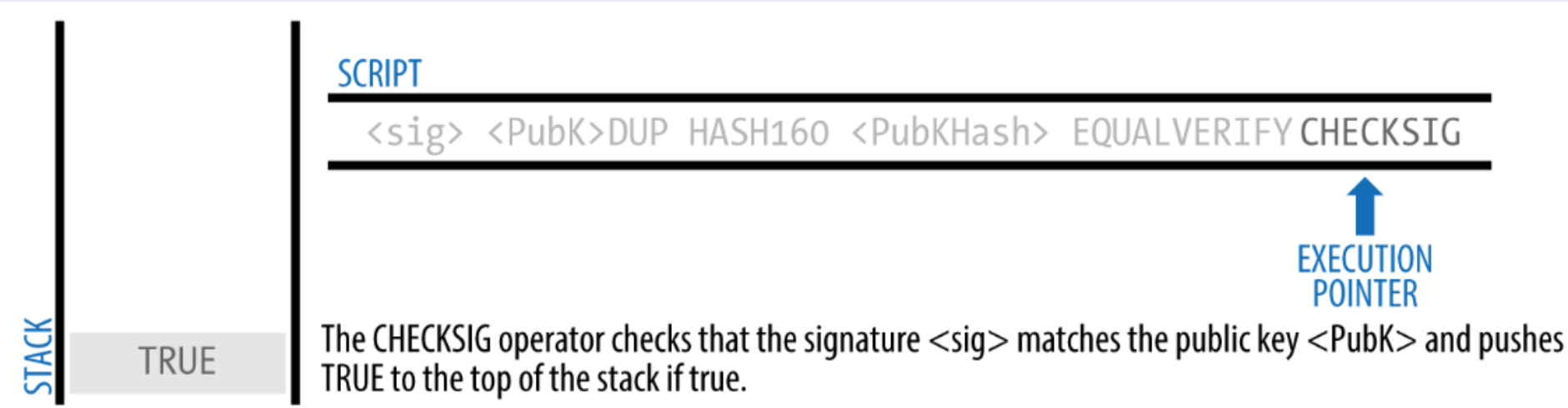


<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



`CHECKSIG` verifies that `sig` is a signature of the transaction by using the private key corresponsing to `pubK`

<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



This script gives proof of ownership!

How block explorers and wallets infer accounts

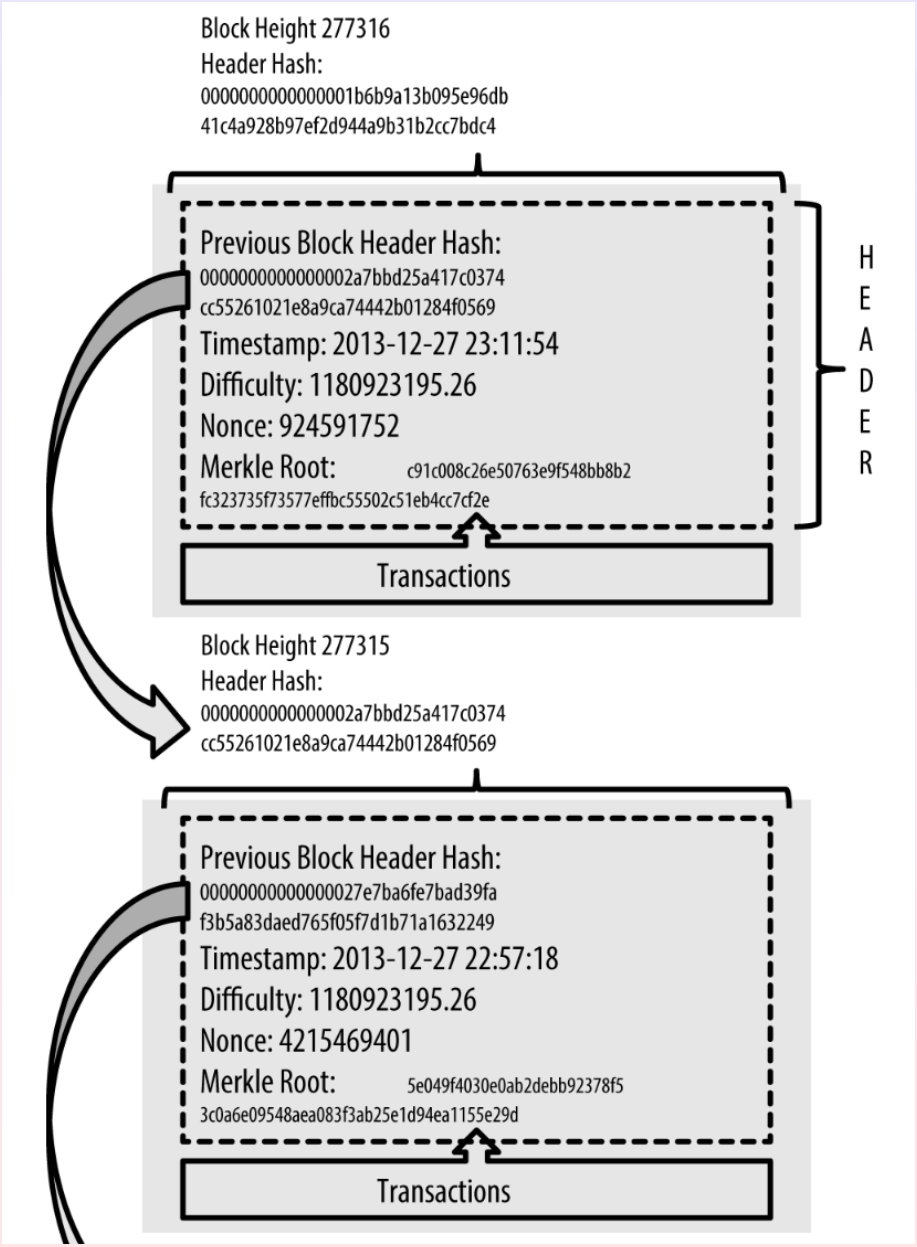
The typical transaction

```
tx = {
  "vins": [ { "txid": ..., "vout": ..., "unlock": <sig> <PubK>}, ..... ],
  "vouts": [{ "value": v, "lock": DUP HASH160 <address> EQUALVERIFY CHECKSIG },
            .....]
}
```

$$\text{RIPEMD160}(\text{SHA256}(\texttt{PubK})) \stackrel{v}{\Longrightarrow} \texttt{address}$$

Fragile! It works for the standard script only

Headers, blocks and blockchain, genesis block

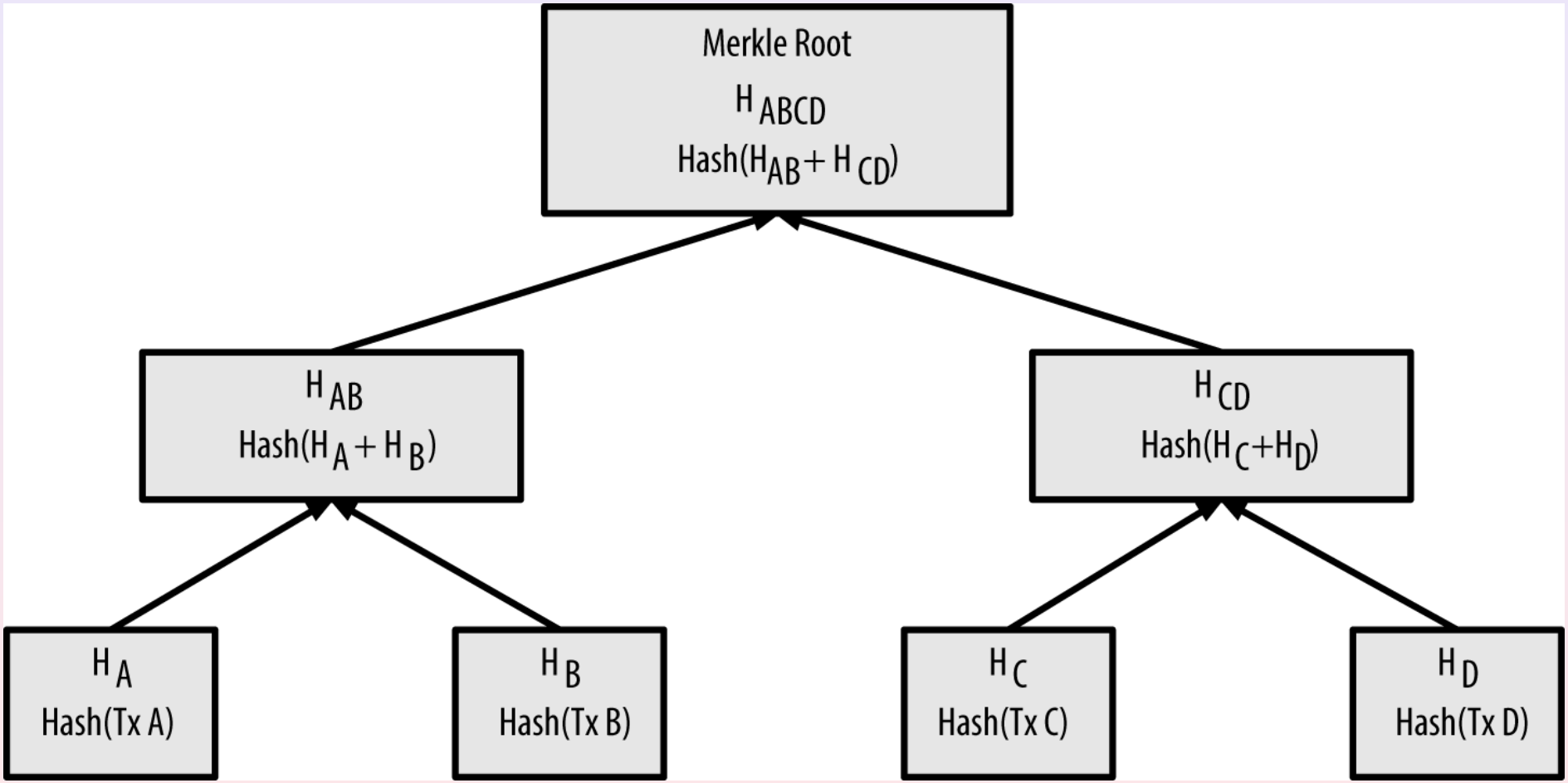


The blockchain grows

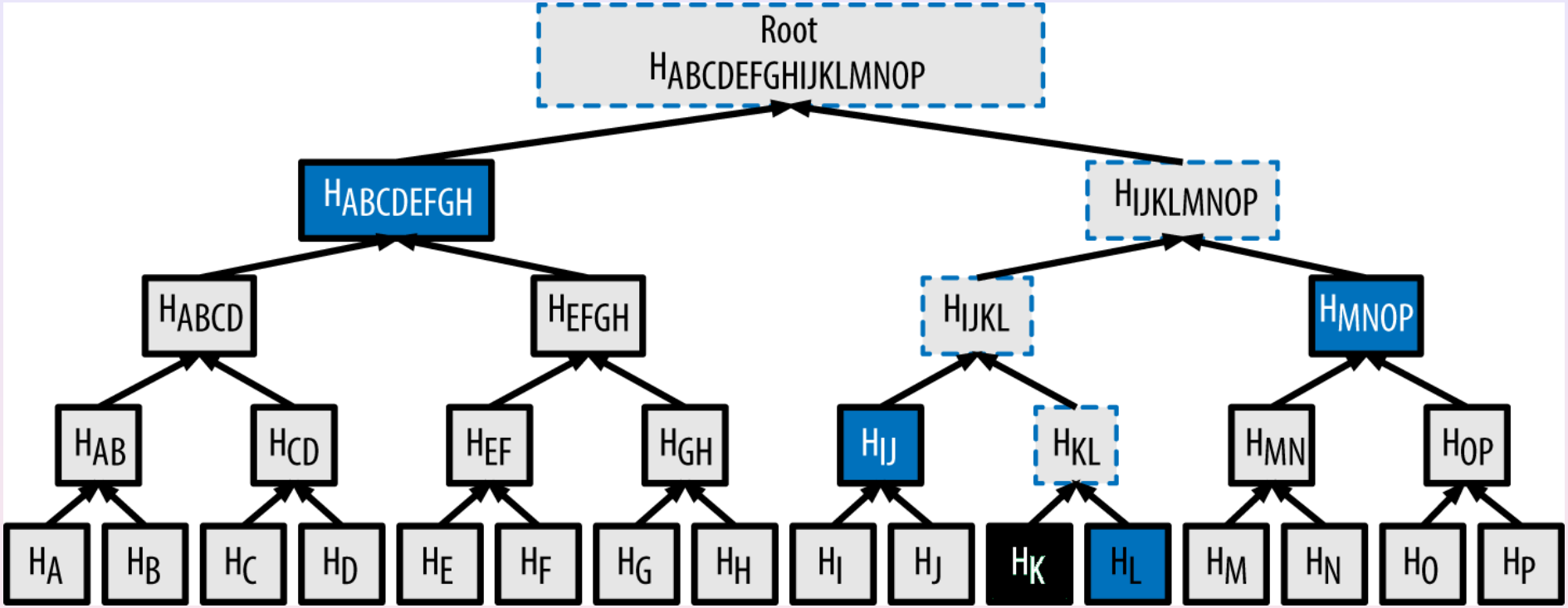
Full nodes are peers that keep the full blockchain in their database

- they receive transactions from other peers or the outside world
  - they keep them in their mempool
  - they whisper them to their peers
- they produce blocks by packing mempool transactions
  - they forward them to their peers
- they receive blocks from peers
  - they use the previous block hash of the received blocks to attach them at the right place in the blockchain
    $\Rightarrow$ the bitcoin blockchain is a tree, not a list

The header contains the hash of the transactions (Merkle root)



Merkle trees provide an efficient inclusion test



I know the root hash and want to know if the black H_K is included

The four blue hashes can be given to me as that proof of inclusion (*authentication path*)

The Merkle tree helps external tools

A wallet wants to know when its transaction t has been processed

- the wallet downloads the header of each new block from a full node; the header is much smaller than the block; it contains the Merkle tree's root
- the wallet asks the full node for a proof of inclusion of t in the block
- the full node answers with a proof consisting of only $32 \times \log_2(\mathit{transactions\_in\_block})$ bytes
- the wallet checks the proof by computing $\log_2(\mathit{transactions\_in\_block})$ hashes and comparing the last hash computed to the Merkle root in the header

The Merkle tree is not in blockchain!

- it is not in the blocks
- it is not in the header of the blocks
  - $\Rightarrow$ only the hash of the Merkle tree's root is in the header
- it is in the volatile memory of full nodes
  - $\Rightarrow$ to answer inclusion tests with a short, efficient proof

Mining

The vision of the miner

The goal of mining is to mint new coins and earn money

The vision of Nakamoto

The goal of mining is to secure the bitcoin network

Miners are expected to create new *valid* blocks

New valid block = it respects the consensus rules

- the structure of data in the header and transactions must be correct
- transactions have at least one input (but for coinbase transactions)
- transactions have at least one output
- transactions do not create money (but for coinbase transactions)
- coinbase transactions have a correct reward
- transactions are all valid (their unlocking scripts match the corresponsing locking scripts)
- transaction inputs refer to unspent UTXO only (no double-spending inside the same history)
- ...

Construction of a new valid block at height H

- addition of a selection of some valid transactions from the mempool
- addition of a coinbase transaction
- computation of their Merkle tree root
- creation of the block's header:
  - hash of block $H-1$
  - timestamp
  - Merkle tree root
  - nonce (for now, an arbitrary number)

If a miner receives from its peers a block b at height H while it's building that same block

It verifies the validity of b. If b is valid, it stops working at the block at height H, adds b to the blockchain and immediately starts working at block at height $H+1$ on top of b. In this sense, the miner *votes* for b

Whom should I trust?

The construction of a new block is too easy!

Just a small fraction of second

$\Rightarrow$ each miner can build as many blocks as it wants

$\Rightarrow$ each miner can decide its own history

Whom should I trust?

The construction of a new block is too easy!

Just a small fraction of second

$\Rightarrow$ each miner can build as many blocks as it wants

$\Rightarrow$ each miner can decide its own history

The real genius of Nakamoto:

make it randomly hard!

Proof-of-work (PoW)

Add the following consensus rule

The hash of valid blocks is smaller than a given constant *difficulty*

Each miner does work

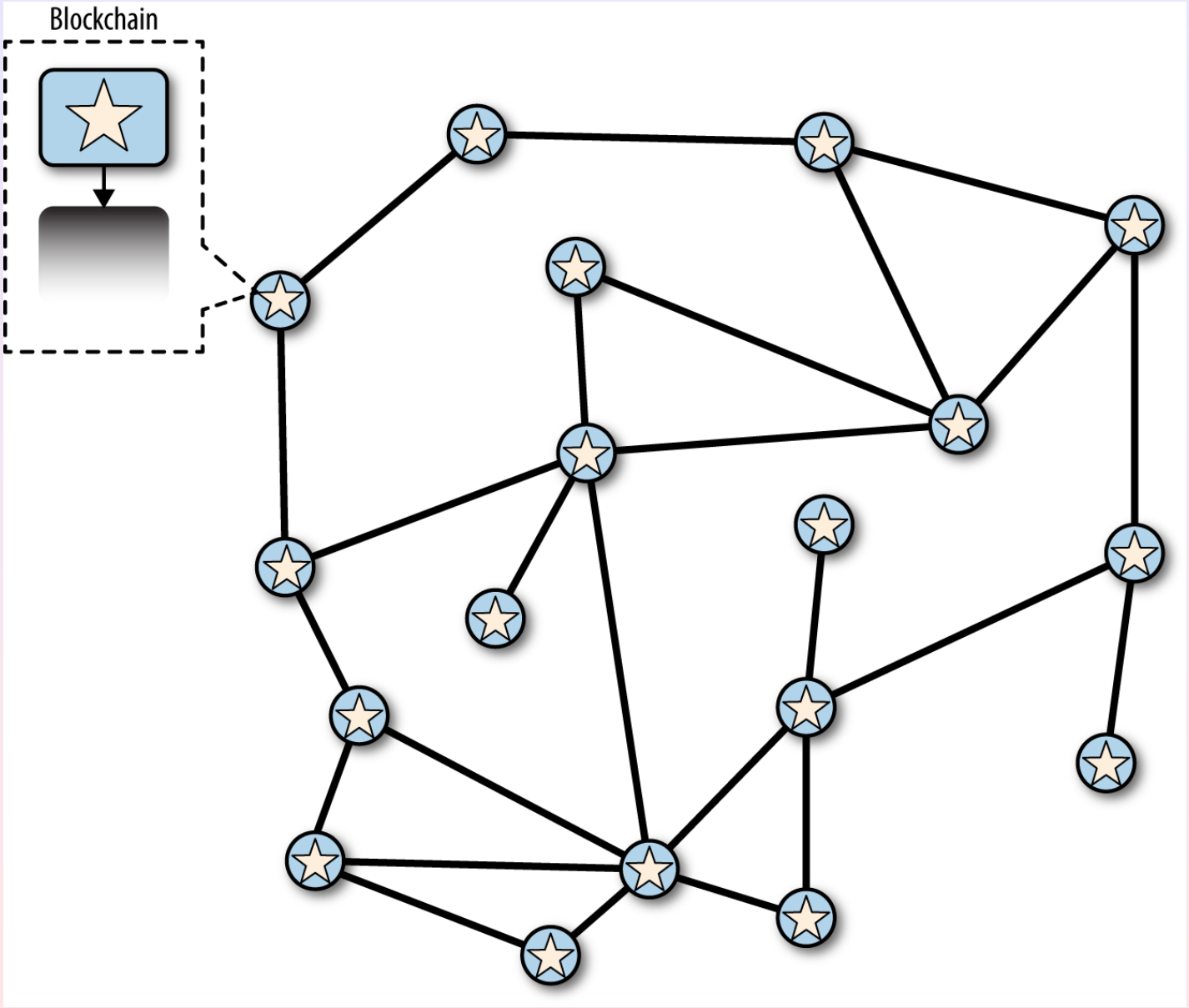
1. build a new block
2. set the nonce field of its header to a random value
3. compute the hash h of the header
4. if $h < difficulty$ stop
5. go back to step 2

- the header of the resulting block is the PoW
- the time to solve this puzzle is inversely proportional to *difficulty*
- the algorithm can be easily run in parallel

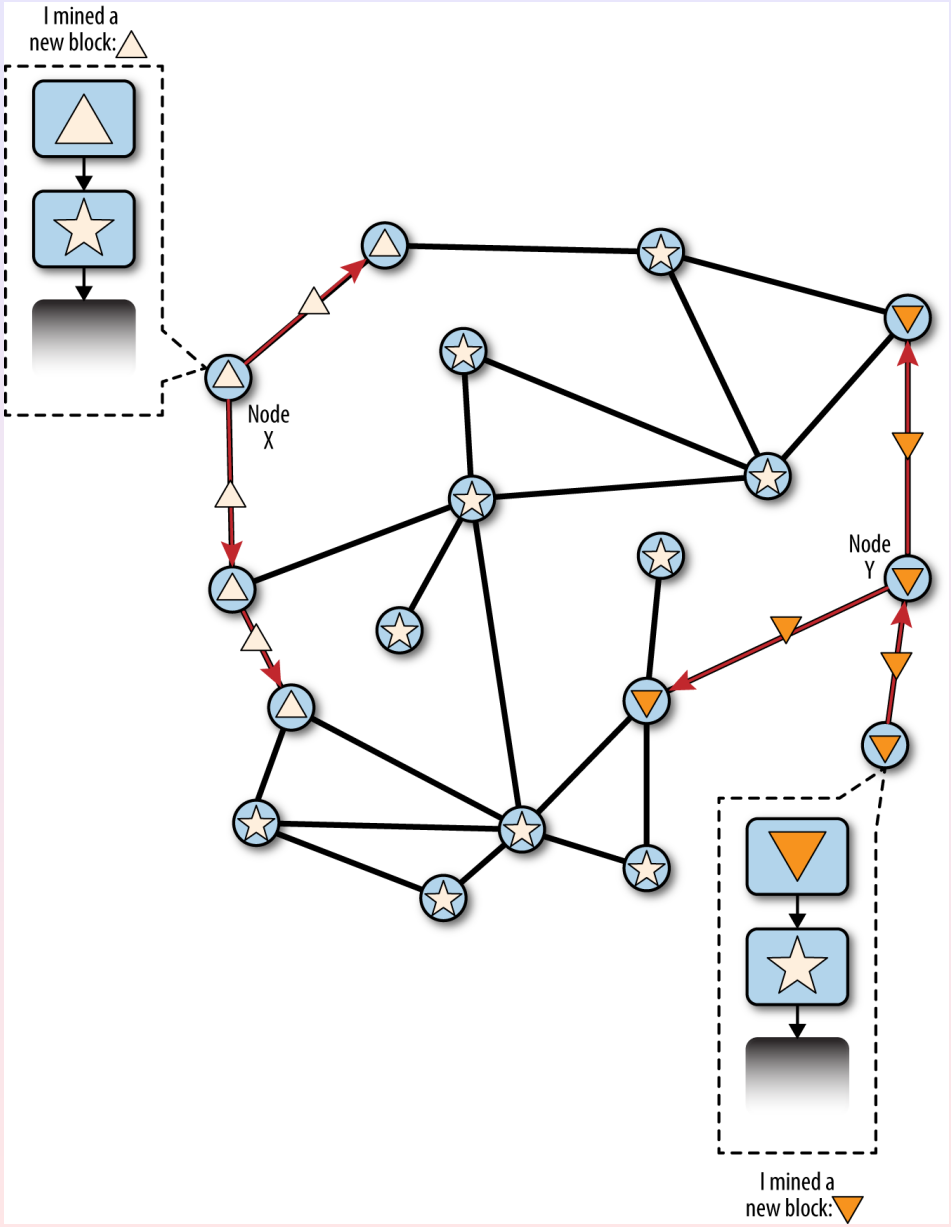
The blockchain grows

- the miner creates a new block on top of the main history
- the miner receives a valid new block from a peer
  - whose parent is the top of the main history, or
  - whose parent is another block of the blockchain (fork), or
  - whose parent is unknown to the miner (orphan block)
- in case of fork, the main history is the longest one, but the miner keeps all histories, in case they might become the new main history in the future

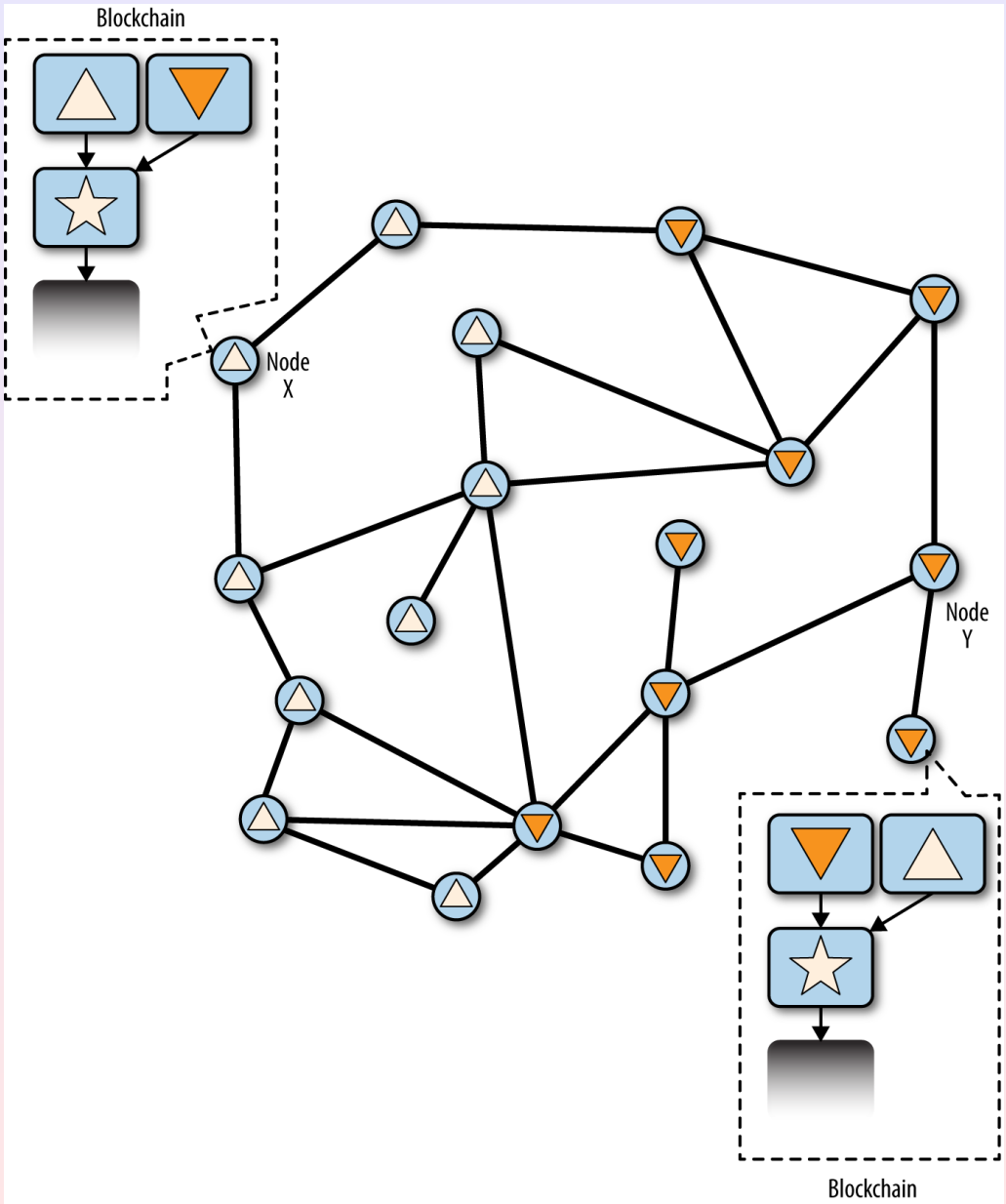
Fork: all nodes start with the same vision



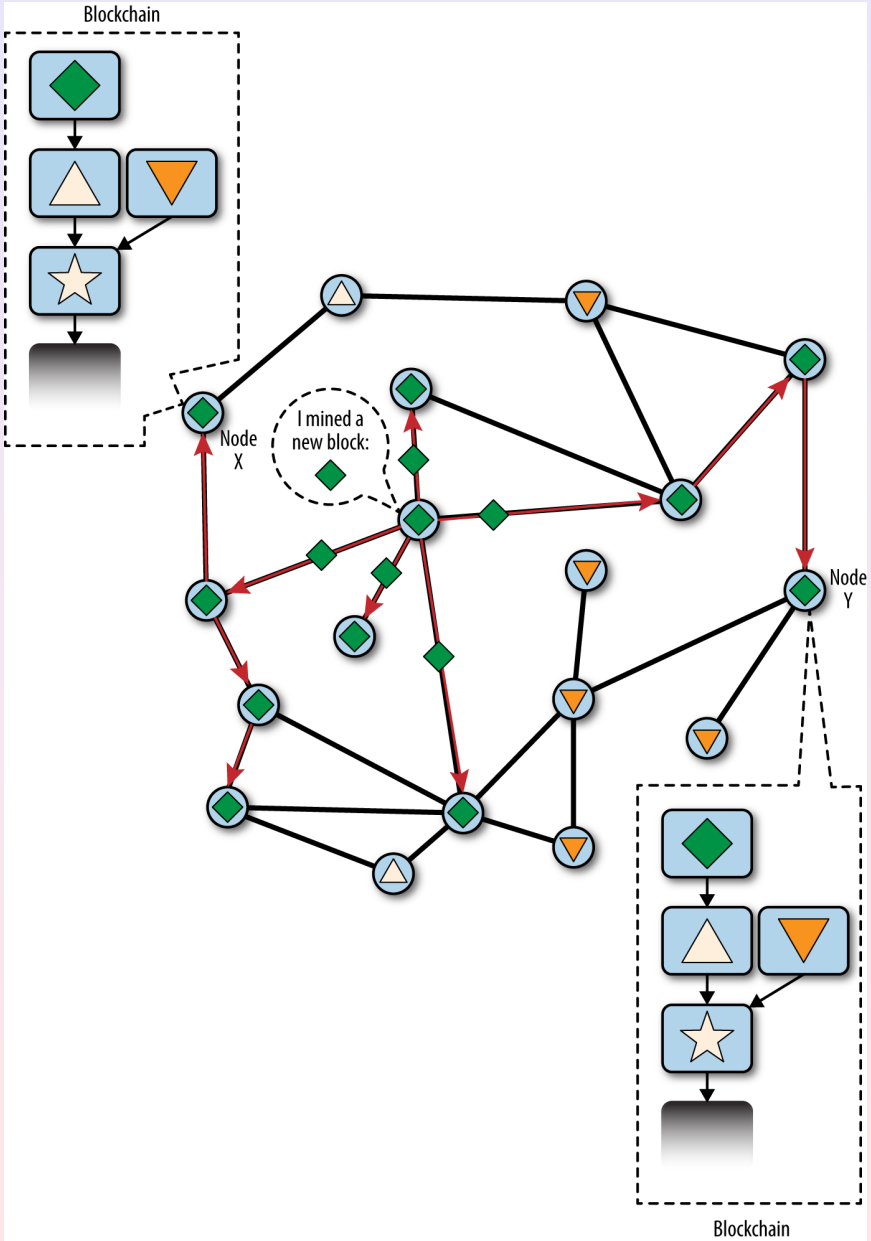
Fork: two nodes expand the blockchain simultaneously



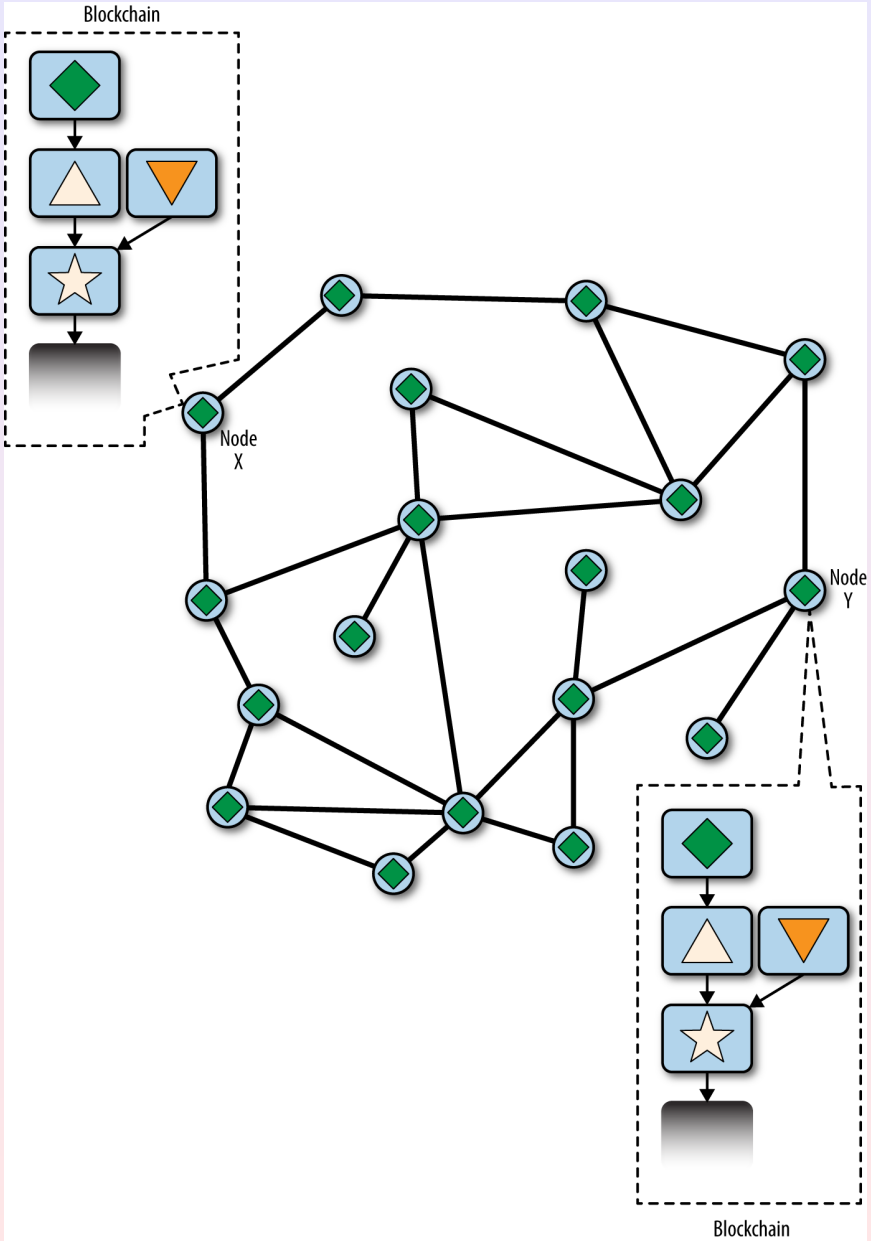
Fork: the network is split



Fork: either chain is expanded further



Fork: the network reconverges

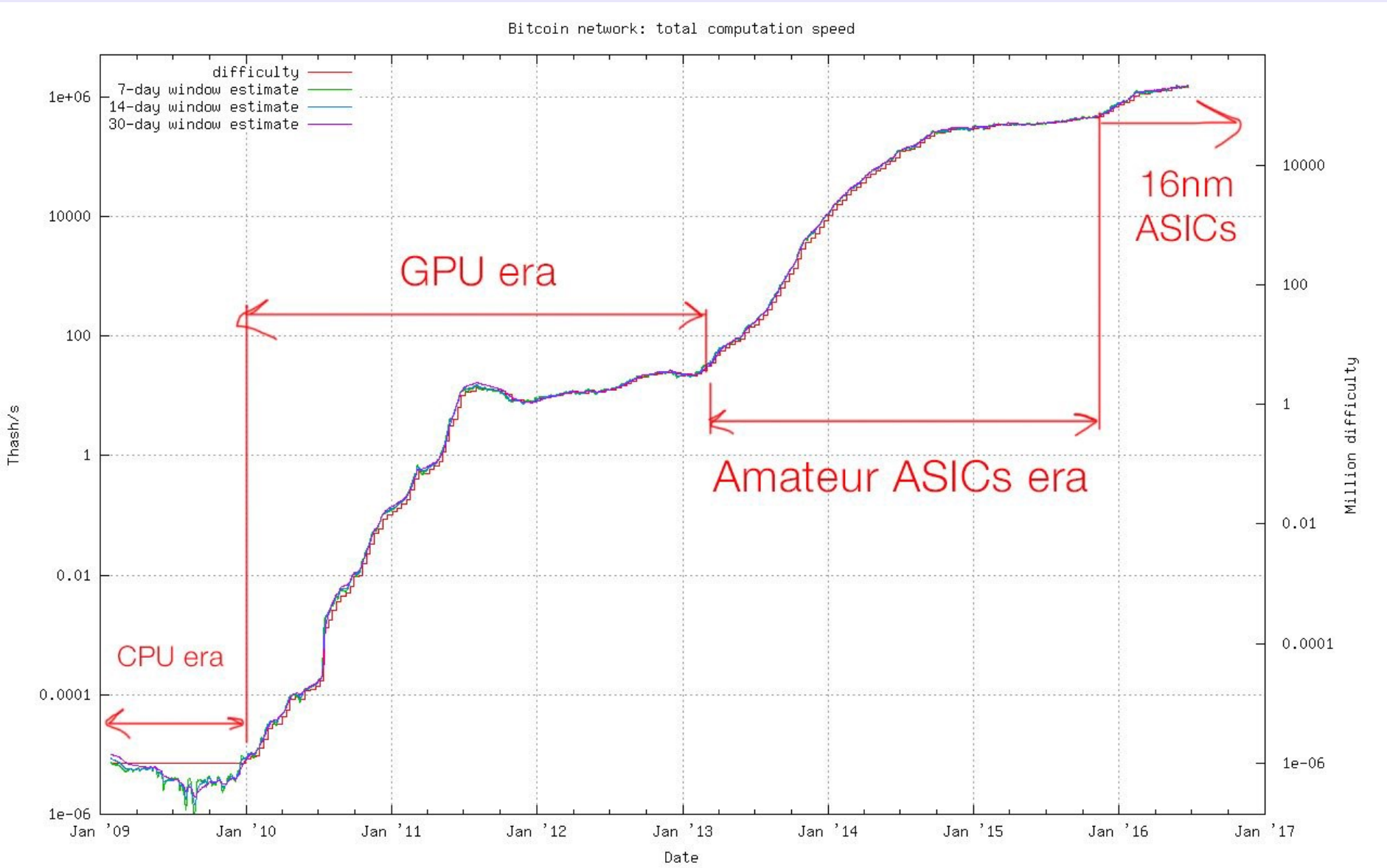


The magic behind PoW

It makes expensive the production of new blocks, in time and cost (electricity)

- who produces invalid blocks sees its blocks rejected by peers and wastes resources
- a single node cannot drive the history, since it must fight against the hashing power of all other nodes together
- forks become unlikely, since the probability of finding a new block at the same time is small

Difficulty over time



PoW costs electricity

2019

Consensus attacks

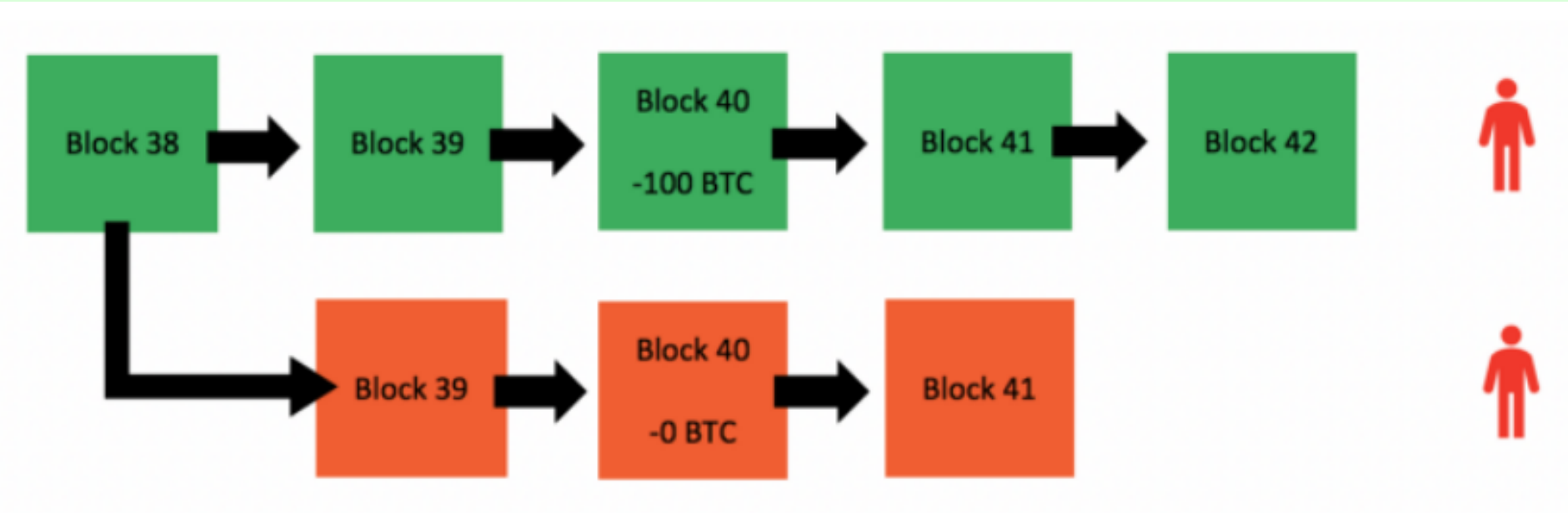
Two main categories

1. history change (for the topmost few blocks)
2. denial-of-service (against specific transactions or accounts)

Possible if the attacker controls a large portion of the total hashing power

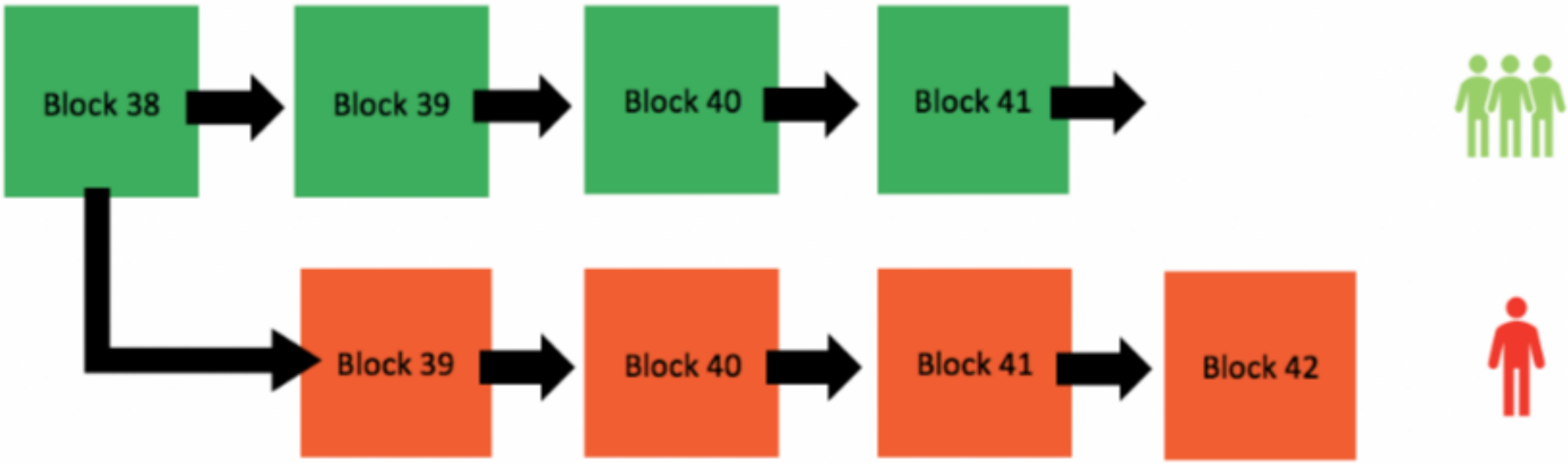
How the 51% attack works

The malicious miner spends 100 BTC on the public chain while it is adding blocks to its private blockchain, without broadcasting them. In its private chain, the spend transaction is missing



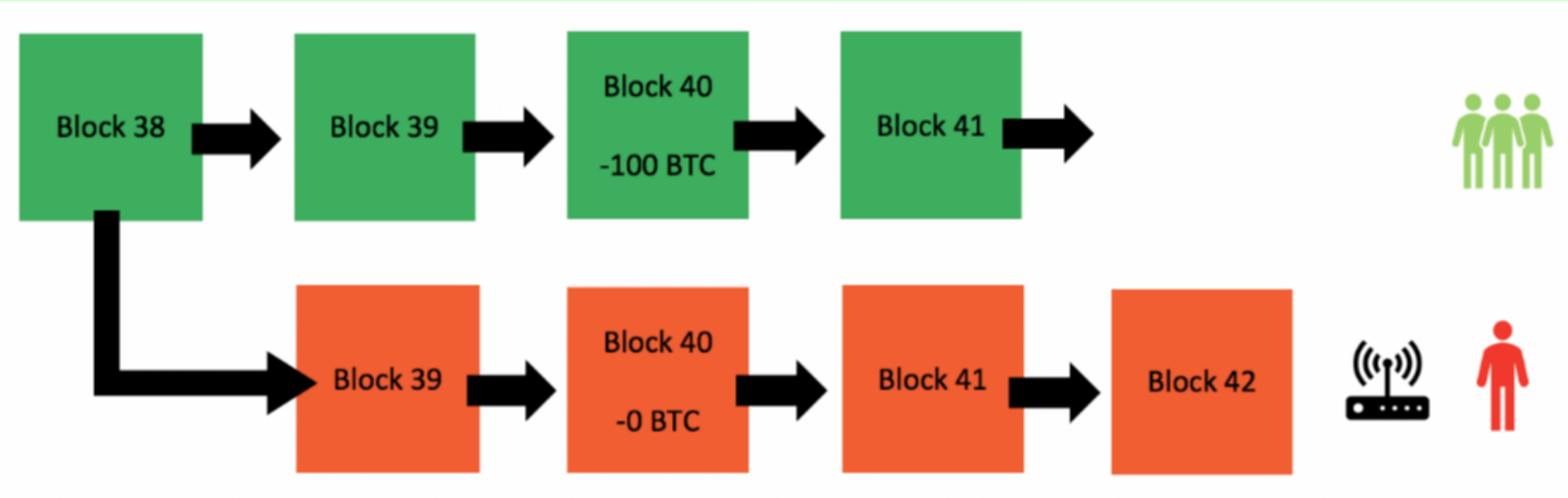
How the 51% attack works

The malicious miner makes its private chain longer, since it has more hashing power than the rest of the miners taken together



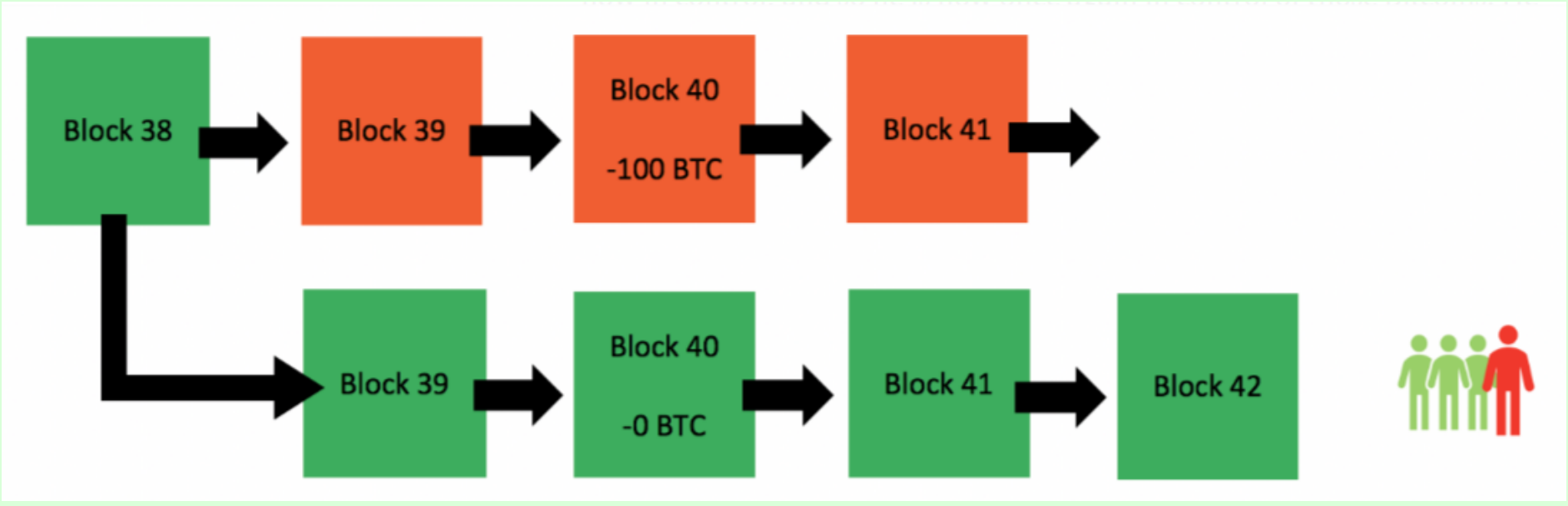
How the 51% attack works

The malicious miner broadcasts its private chain; since it is longer, all other miners switch to this alternative view of the history



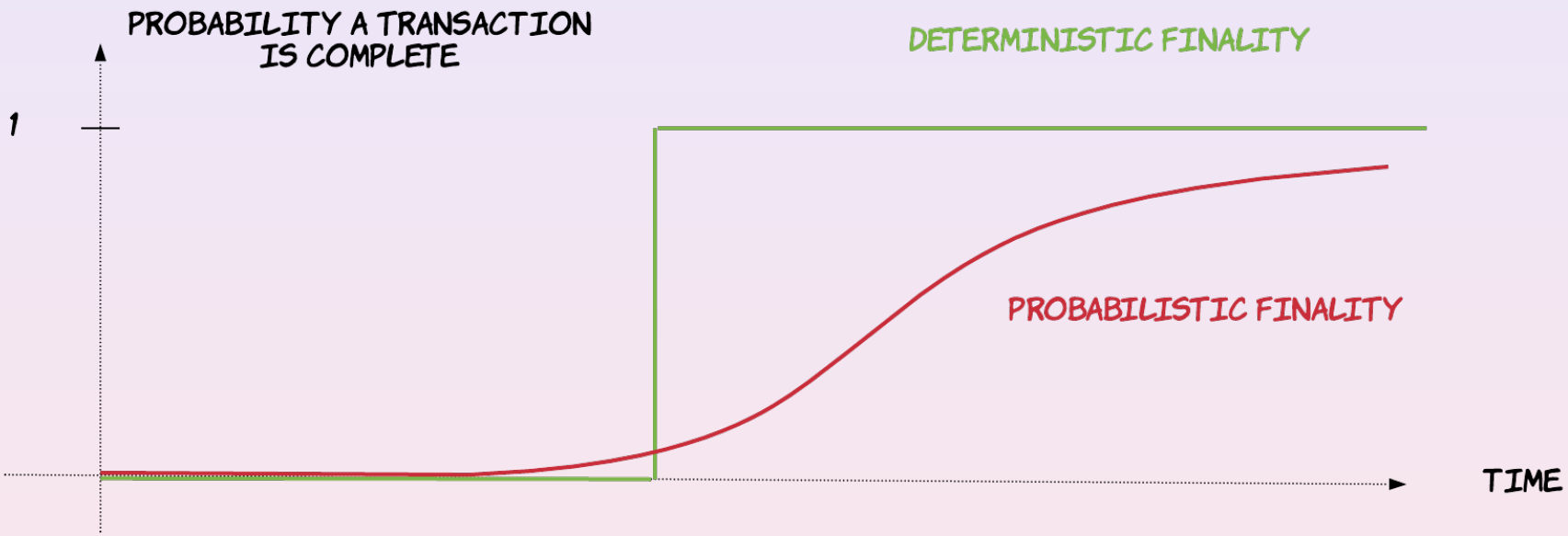
How the 51% attack works

The malicious miner can spend its 100 BTC again! Everybody agrees...



History is written by the victors

Bitcoin has probabilistic finality



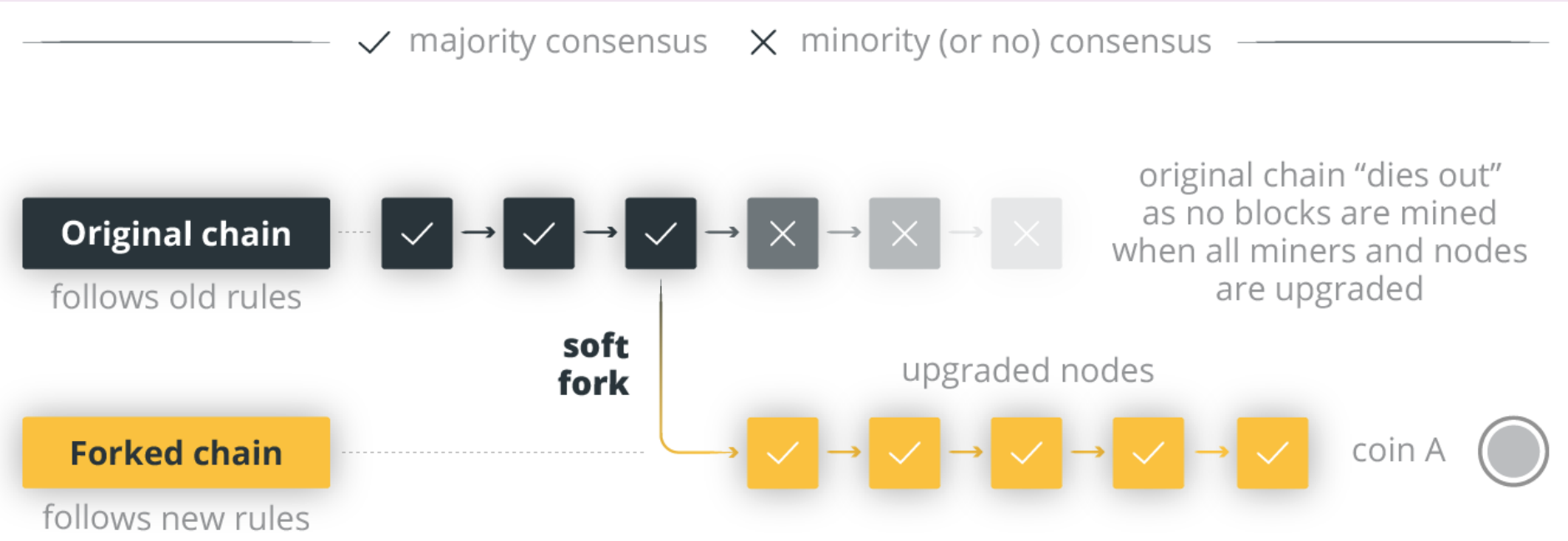
Changing the consensus rules

Why?

- new features
- bug fixes
- protocols exist for voting for/against the upgrade, before the new rules get activated

Soft fork: new rules more restrictive than old rules

- if the next block is mined by a yellow node, it is accepted by all nodes
- if the next block is mined by a black node, it might be rejected by the yellow nodes
- if at least 51% of the hashing power is yellow (upgraded), the network will not split and will follow the new rules of the yellow nodes



Hard fork: new rules less restrictive than old rules

- if the next block is mined by a black node, it is accepted by all nodes
- if the next block is mined by a yellow node, it might be rejected by the black nodes
- if at least 51% of the hashing power is yellow (upgraded), the network splits in two