Réunion Island, 13/4/2023

An Overview of Blockchain Technology

fausto.spoto@univr.it

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

Introduction

The mainstream view of blockchain

How a 26-Year-Old College Dropout Makes $15,000 a Month With Bitcoin and Cryptocurrency Without Breaking a Sweat

By Marc Thomson **Published on February 16, 2021**, FinanceIndex.co

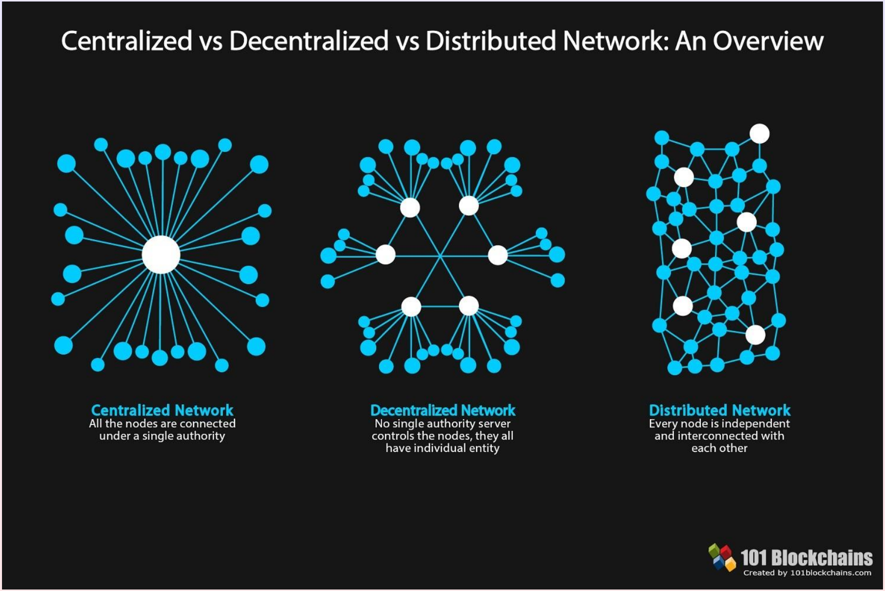
Meet the bitcoin investors who got insanely rich off crypto

By Suzy Weiss January 13, 2021 | 7:23pm | Updated

History

- 1988 proof of work (Dwork & Naor)
- 1991 a cryptographically secure chain of blocks (Haber & Stornetta)
- 199x smart contracts (Szabo)
- 2008 Bitcoin (Nakamoto)
- 2012 proof of stake (Peercoin)
- 2013 Ethereum (Buterin & Wood)
- 2014 proof of space (Burstcoin/Signum)
- 2014 Tendermint generic proof of stake engine (Kwon)
- 2022 Ethereum 2.0 moves to proof of stake

Distributed network

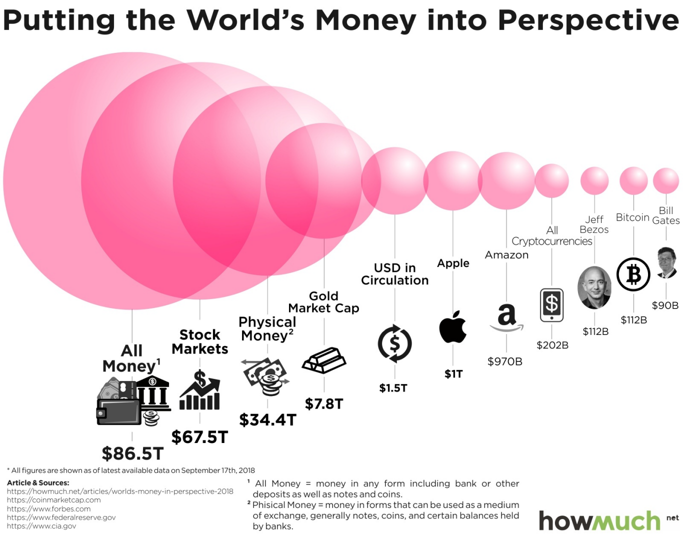


Cryptocurrencies

☆	1	**Bitcoin** BTC	€43,439.39	▲1.78%	▲11.96%	€811,654,732,050	€45,070,079,455 1,034,656 BTC	18,632,831 BTC
☆	2	**Ethereum** ETH	€1,590.21	▲1.60%	▲9.73%	€182,490,495,147	€21,692,262,865 13,638,149 ETH	114,733,656 ETH
☆	3	**Binance Coin** BNB	€215.69	▲39.28%	▲106.89%	€32,922,693,592	€9,229,311,959 43,320,613 BNB	154,532,785 BNB
☆	4	**Tether** USDT	€0.8232	▼0.04%	▼0.10%	€27,608,853,121	€86,517,646,829 105,076,438,643 USDT	33,531,193,546 USDT
☆	5	**Polkadot** DOT	€26.79	▲3.52%	▲28.14%	€24,343,111,959	€2,126,956,641 79,458,749 DOT	909,408,867 DOT
☆	6	**Cardano** ADA	€0.7630	▼1.70%	▲0.49%	€23,662,518,973	€3,853,459,801 5,066,692,562 ADA	31,112,484,646 ADA
☆	7	**XRP** XRP	€0.4433	▼0.87%	▼7.31%	€20,123,554,344	€4,449,941,491 10,040,237,796 XRP	45,404,028,640 XRP
☆	8	**Litecoin** LTC	€191.53	▼0.24%	▲26.86%	€12,730,944,343	€6,268,248,742 32,751,917 LTC	66,519,829 LTC
☆	9	**Chainlink** LINK	€28.05	▲4.58%	▲23.29%	€11,386,121,071	€1,672,730,305 59,793,604 LINK	407,009,556 LINK
☆	10	**Bitcoin Cash** BCH	€591.11	▲0.26%	▲34.87%	€11,009,352,859	€3,793,957,459 6,430,233 BCH	18,659,331 BCH

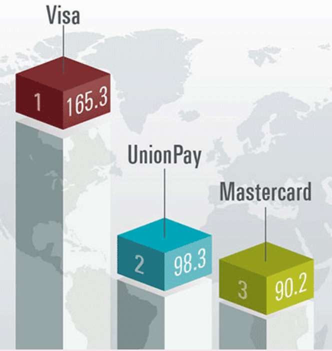
Bitcoin chart

Bitcoin capitalization (2018)



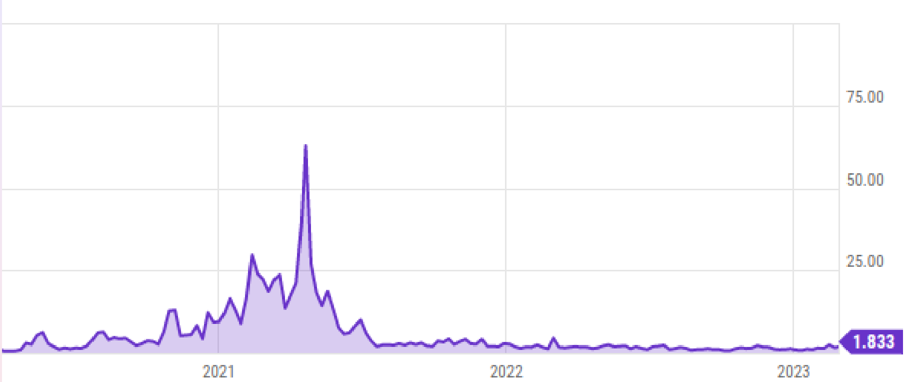
source: HowMuch.net, a financial literacy website

Credit cards transactions (billions, 2018) vs Bitcoin



Visa: around 451,639,000 transactions per day
UnionPay: around 268,579,000 transactions per day
Mastercard: around 246,448,000 transactions per day
Bitcoin: around 300,000 transactions per day

Bitcoin transaction fees



Independent from the transacted value

Credit cards transaction fees

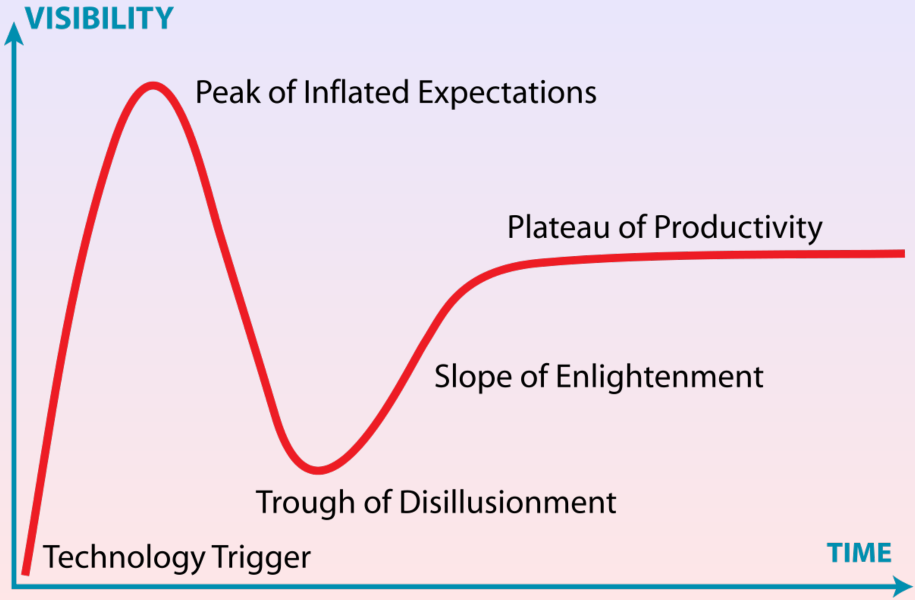
Average credit card interchange fees

Payment network	Interchange fee range
Visa	1.15% + $0.05 to 2.40% + $0.10
Mastercard	1.15% + $0.05 to 2.50% + $0.10
Discover	1.40% + $0.05 to 2.40% + $0.10
American Express	1.43% + $0.10 to 3.15% +$0.10

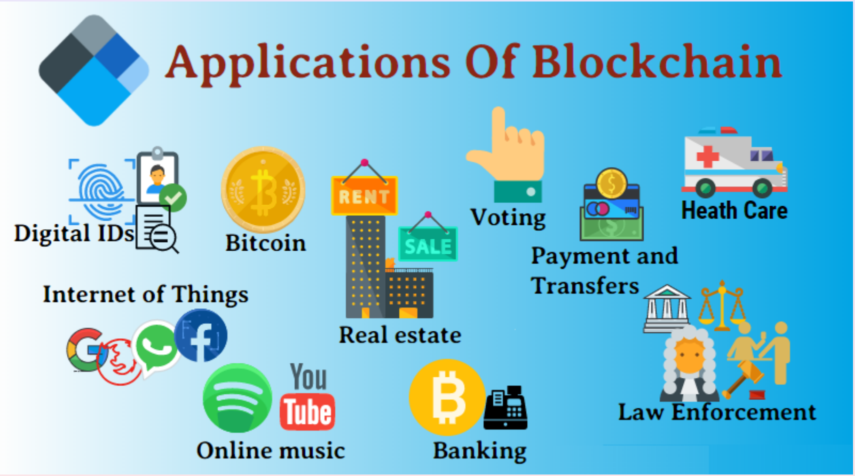
Sources: Visa USA Interchange Reimbursement Fees published on April 13, 2019, Mastercard 2019-2020 U.S. Region Interchange Program and Rates, and Wells Fargo Payment Network

Proportional to the transacted value

The hype cycle



Beyond the hype



Bitcoin

The internet of money

What we expect from money

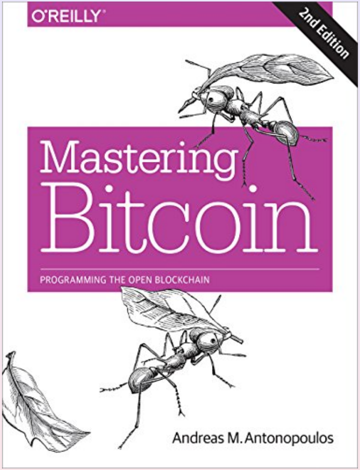
- money should be protected from counterfeiting (*legality*)
- money should not be spent twice (*uniqueness*)
- no one can claim that my money belongs to him (*ownership*)
- money should be untained (*fungibility*)
- money should be movable (*liquidity*)

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)

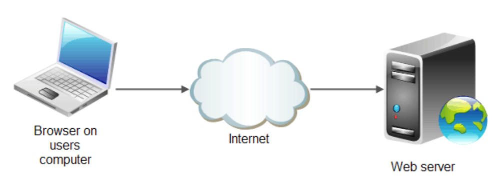
"Bitcoin: A Peer-to-Peer Electronic Cash System" by Satoshi Nakamoto, 2008

The best reference



https://github.com/bitcoinbook/bitcoinbook

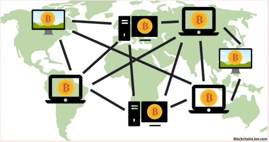
Bitcoin as a web service



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

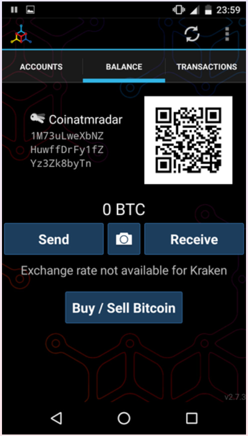
Users interact through a browser (wallet) to ask to transfer balances

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



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. Its creation is a local operation that does not do anything on the network: fully 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 (!!!!!!!)

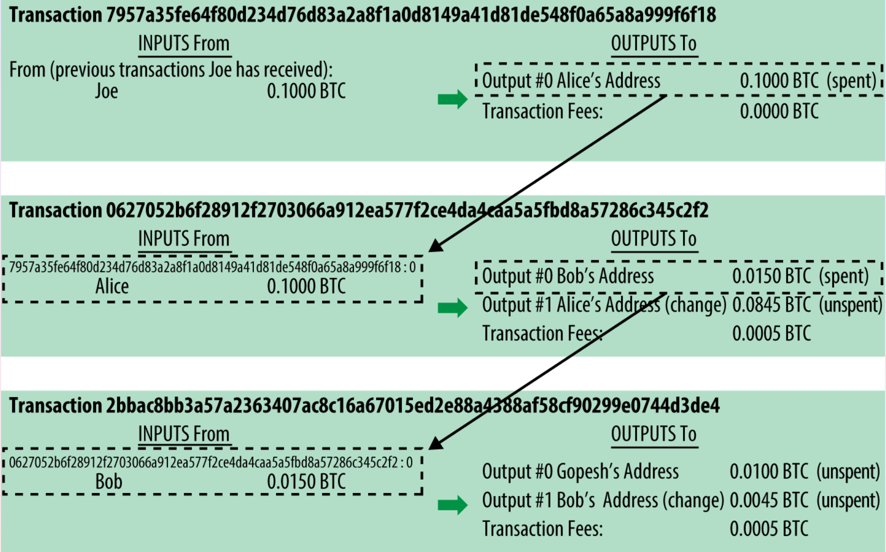
Alice charges her wallet with a transaction

- she asks a friend to send bitcoins to her address
- meets a bitcoin seller in person
- earns bitcoin by working
- uses a bitcoin ATM
- uses a bitcoin currency exchange company

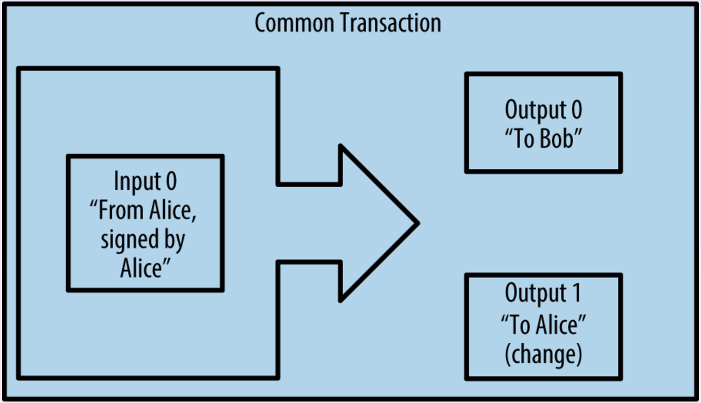
What is the price?

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

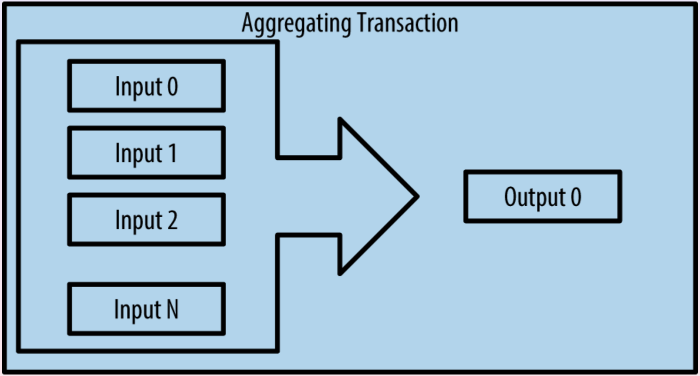
Transactions form a chain, outputs can be change



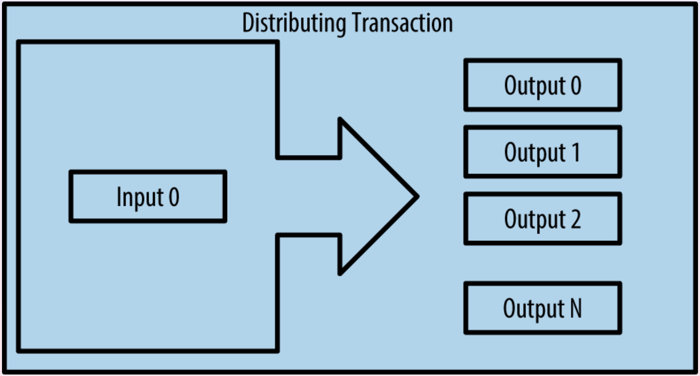
Typical transaction: pay somebody and gets the change



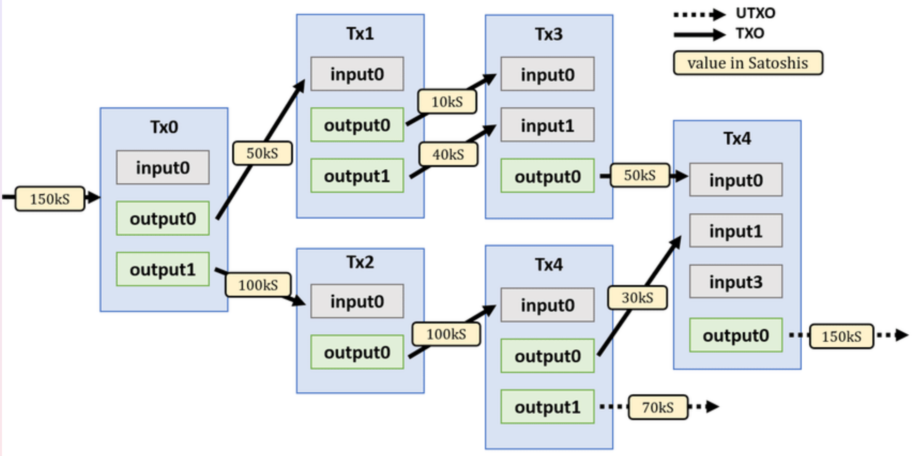
Typical transaction: aggregate small notes into a larger one



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 and signs to prove she's their owner
   - any strategy can be applied here
3. the wallet specifies an output for the destination address of the transaction and the *amount* ≥ 0 sent to that output
4. the wallet specifies a second output, normally Alice's address itself, and the *change* ≥ 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
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, depending 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:

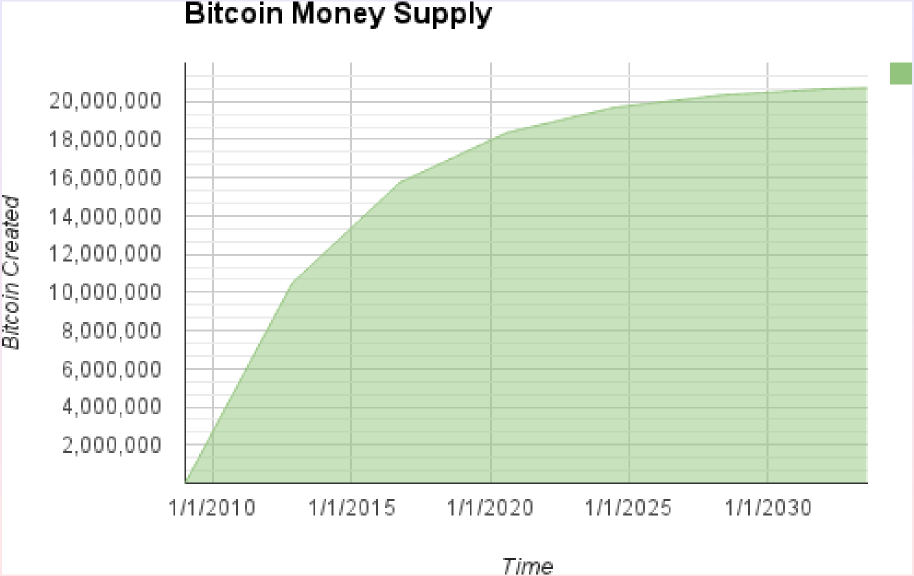
- the fees $\phi_1 \cdots \phi_n$ of the n transactions in the block go to μ
- some amount of money ι is created out of thin air and goes to μ

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

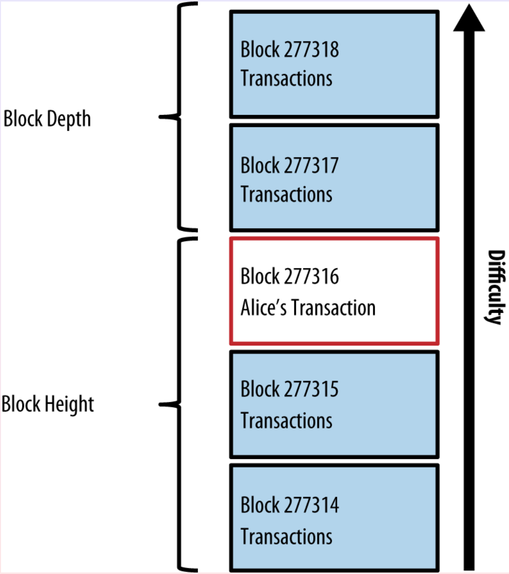
Bitcoin supply over the years



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: different miners can use different strategies
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
   - if no other miner has been faster, it forwards the new block to all its peers

Block's height, depth and confirmations



The 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 `vout`th 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*

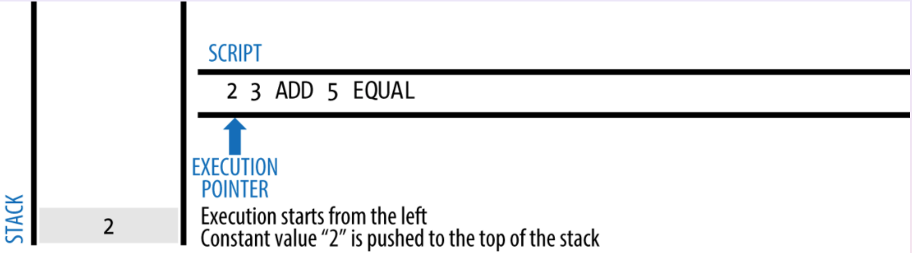
Script validity

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`

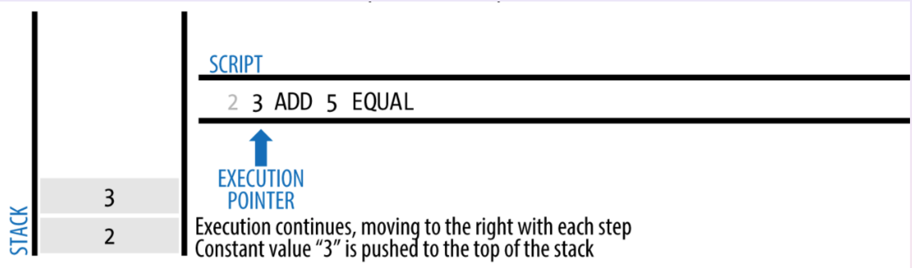
Execution proceeds left-to-right

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

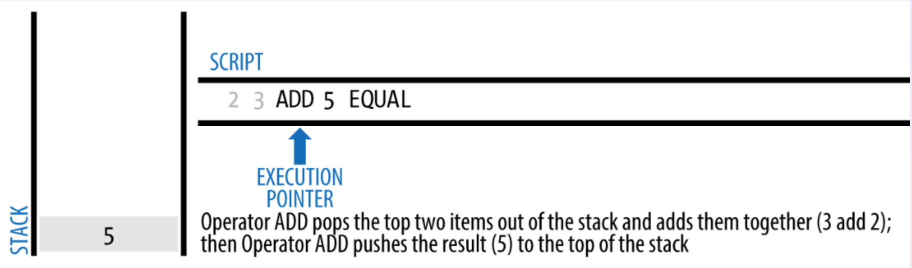
2 3 ADD 5 EQUAL



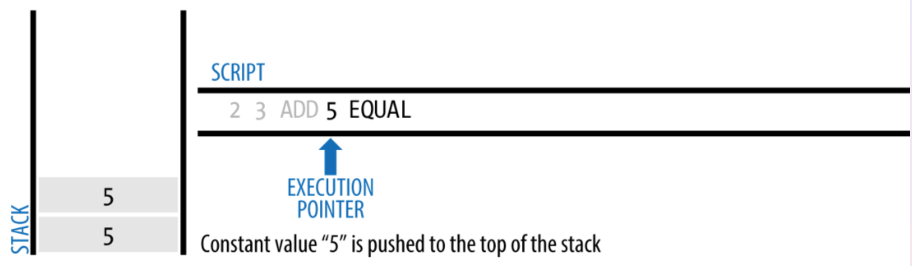
2 3 ADD 5 EQUAL

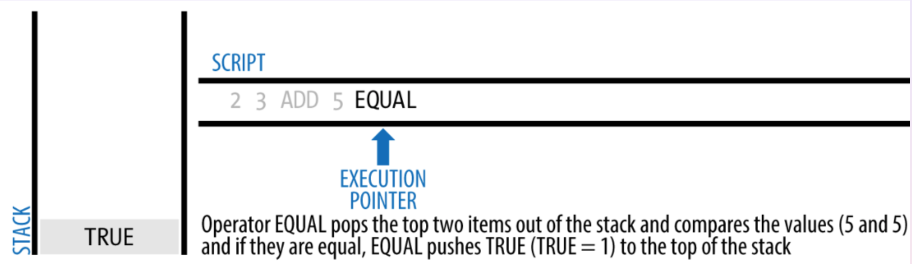


2 3 ADD 5 EQUAL



2 3 ADD 5 EQUAL





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"

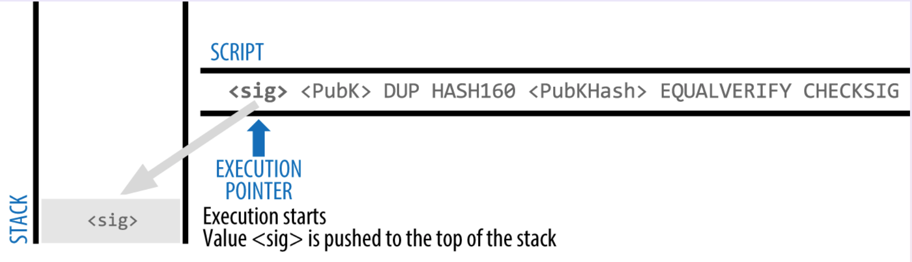
```
tx = {
  "vins": [ { "txid": H,   "vout": .....,   "unlock": <sig> <PubK>}, ..... ],
  "vouts": .....
}
```

`unlock lock`

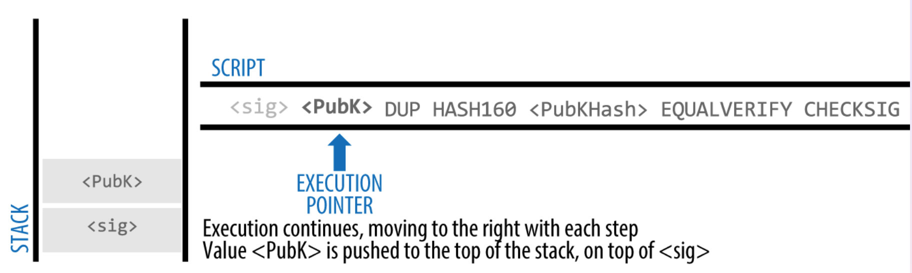
```
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG
```

The bitcoin `address` is often referred to as `PublicKHash`

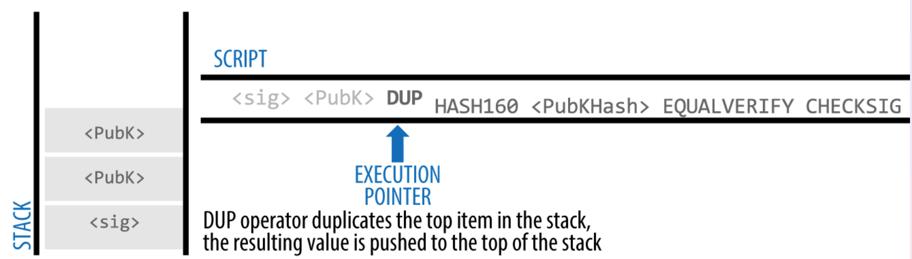
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



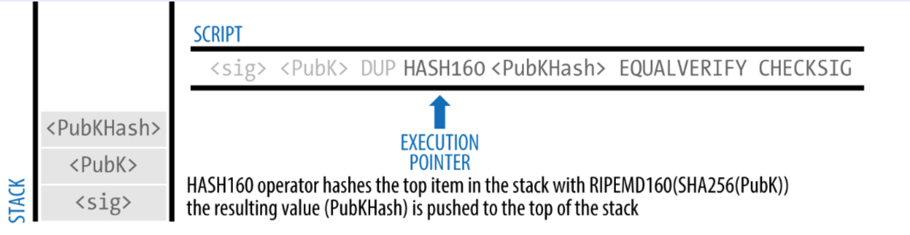
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



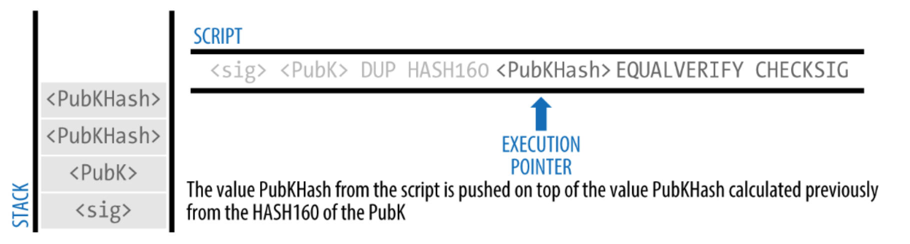
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



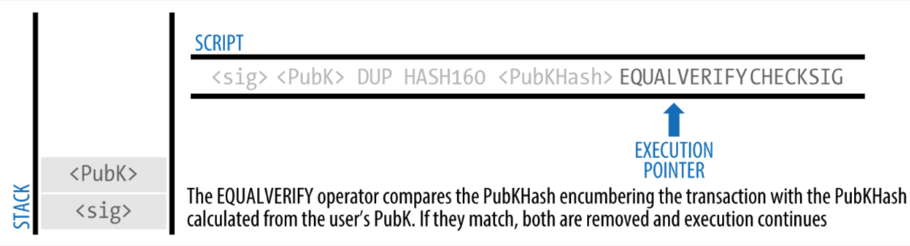
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



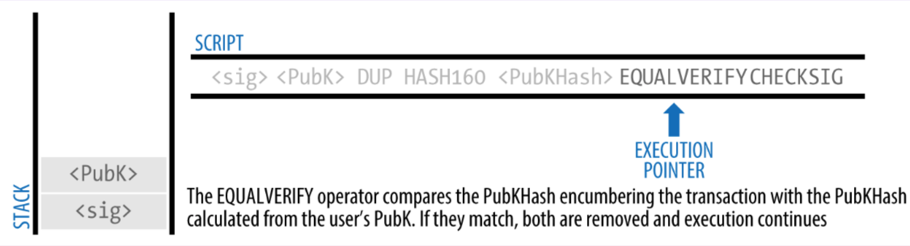
<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



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

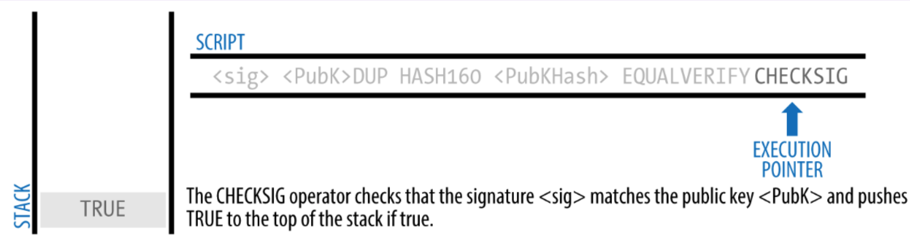


<sig> <PubK> DUP HASH160 <address> EQUALVERIFY CHECKSIG



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

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

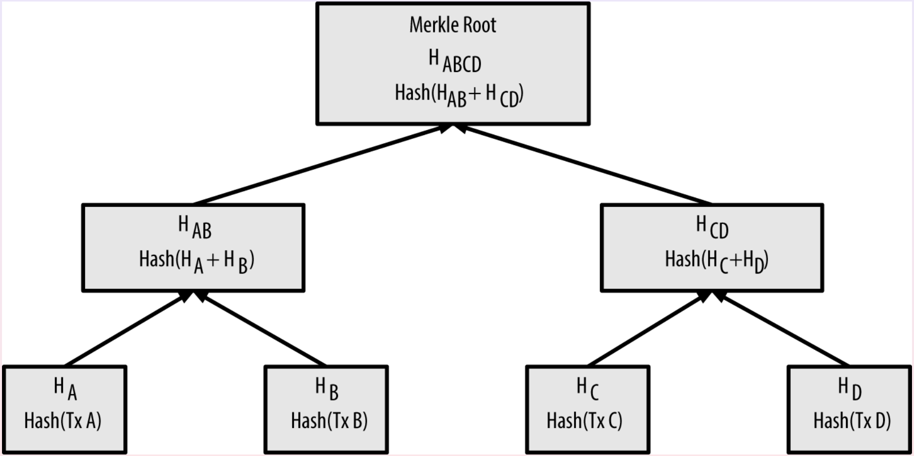


This script gives proof of ownership!

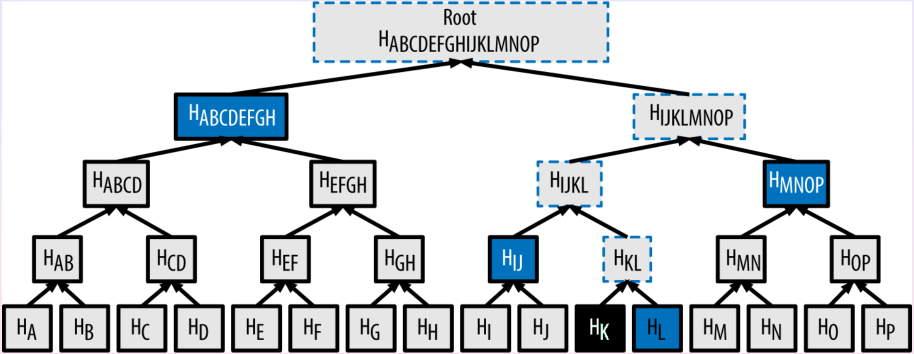
Properties of Bitcoin's transactions

- Script programs are only used to check the validity of the transactions
- Script programs never modify the state of the system
- Transactions undo is very easy (they just move money around)

Block headers contain the hash of all transactions in the block (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*)

Proof of work

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 can only mine correct 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)
- ...

But what about fairness and progress?

How to kill a dictator

Without proof of work

A single node dictates the history of the blockchain if it is faster than *each* other node

How to kill a dictator

Without proof of work

A single node dictates the history of the blockchain if it is faster than *each* other node

With proof of work

A single node dictates the history of the blockchain if it is faster than *the sum of all* other nodes

Proof of work (PoW)

Add the following consensus rule

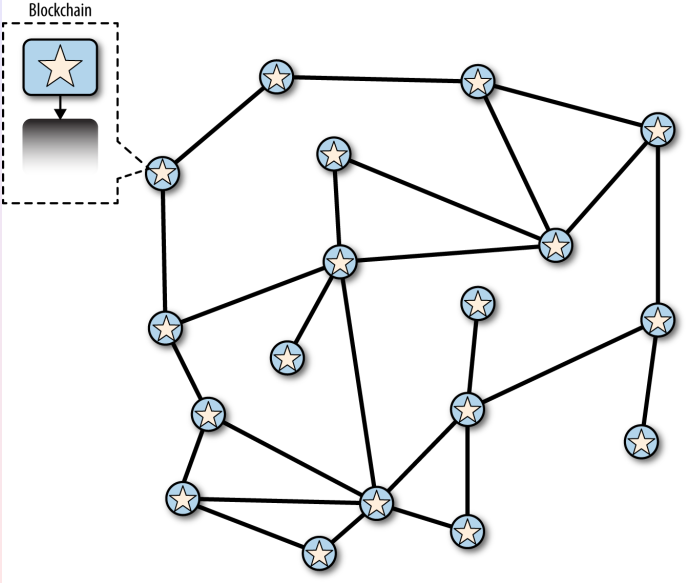
The hash of valid blocks is smaller than a given constant *difficulty*

Miners must work hard now

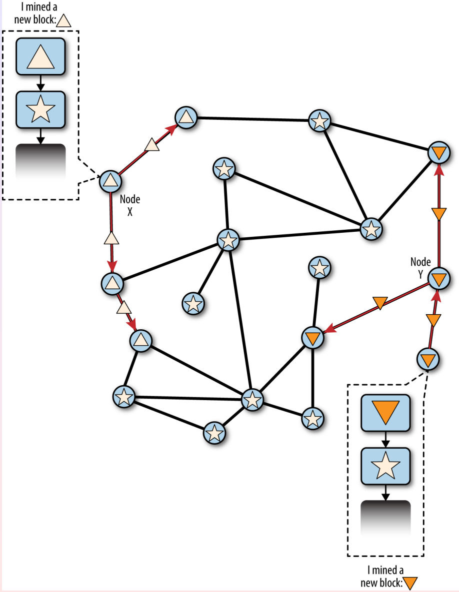
1. build a new block
2. set the nonce field of its header to a random value
3. compute the hash h of the header
4. if $h < \textit{difficulty}$ stop
5. otherwise, go back to step 2 and try again

- 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, GPU, ASIC

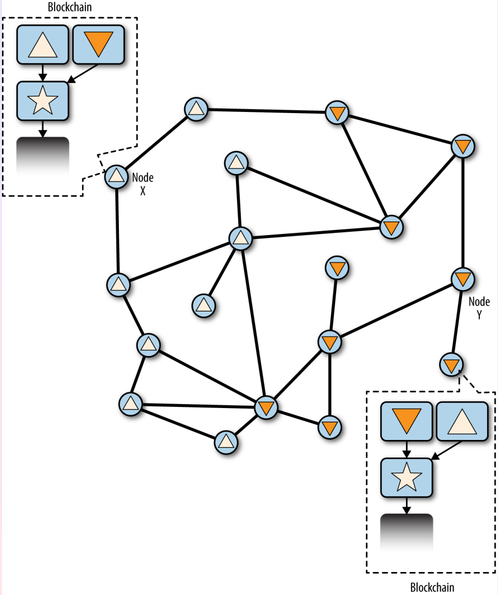
Fork: all nodes start with the same vision



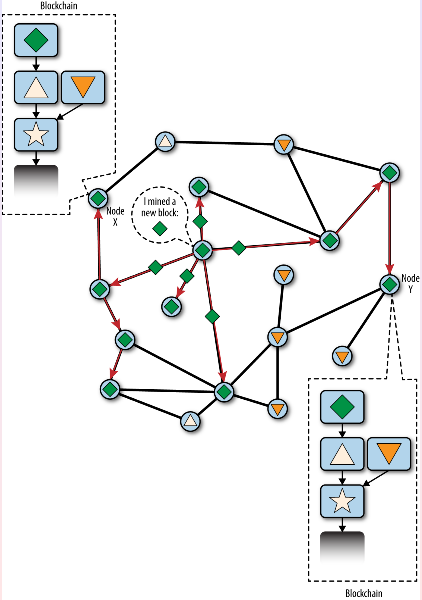
Fork: two nodes expand the blockchain simultaneously



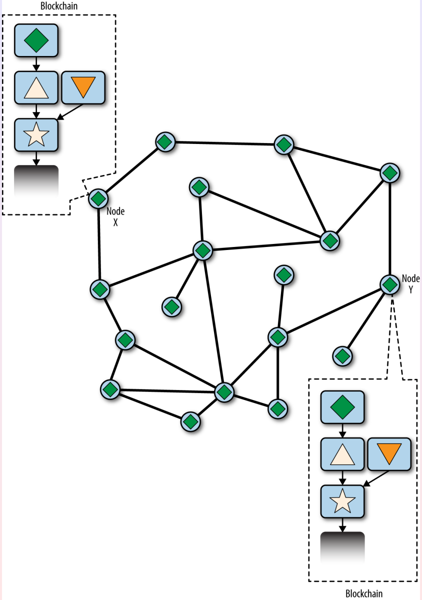
Fork: the network is split



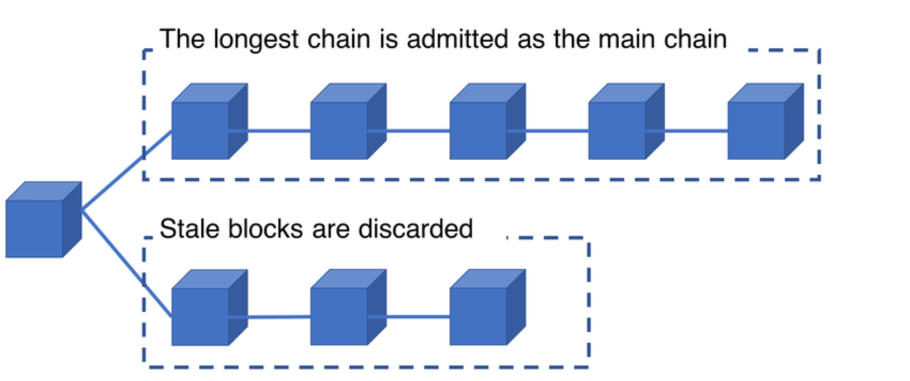
Fork: either chain is expanded further



Fork: the network reconverges



State update in case of history change



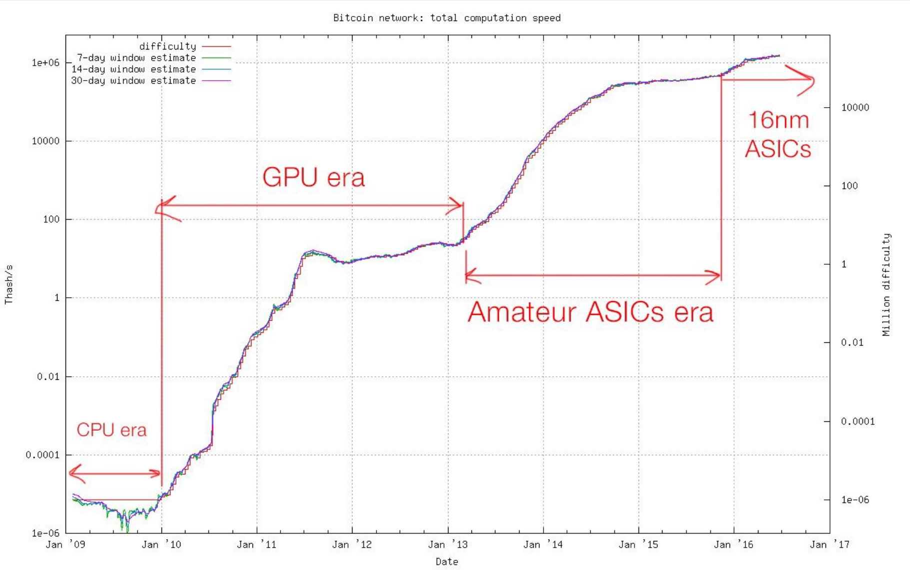
1. the transactions in the discarded chain must be undone (easy in Bitcoin)
2. the transactions in the longest chain must be done (easy)

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 two nodes finding a new block at the same time is small

Difficulty over time



PoW costs electricity

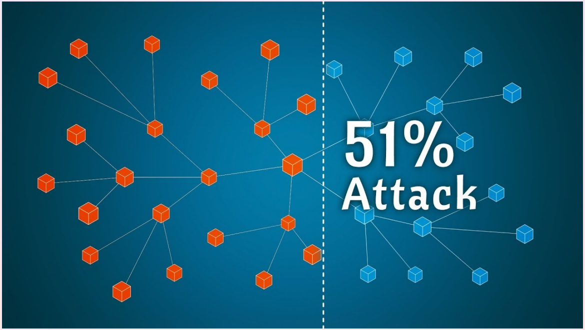
2019

Consensus attacks

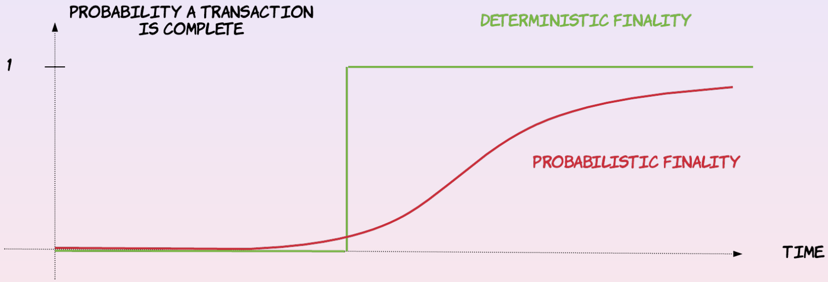
Two main categories

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



Bitcoin has probabilistic finality



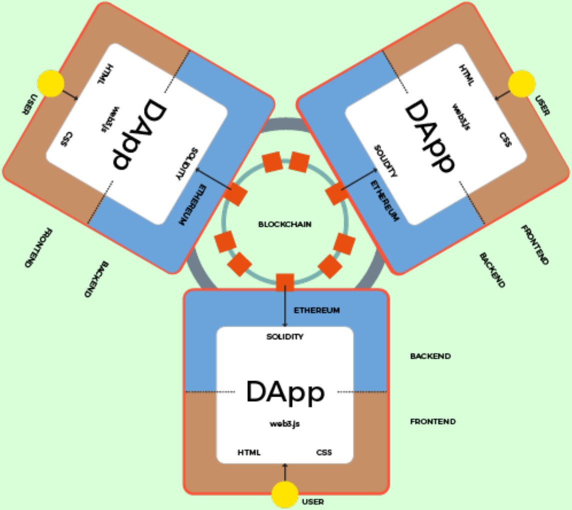
Ethereum

The world computer

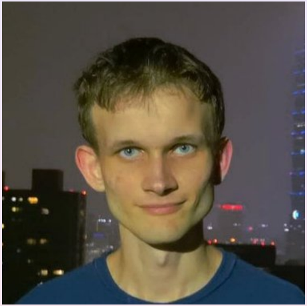
An open source, globally decentralized computing infrastructure that executes programs called smart contracts, written in a Turing-complete programming language, translated into bytecode and run on a virtual machine. It uses a blockchain to synchronize and store the system's singleton state changes (key/value tuples), along with a cryptocurrency called ether to meter and constrain execution resource costs. It enables developers to build decentralized applications with built-in economic functions

DApps

DApps = smart contracts (Solidity) + web3 frontend (JavaScript. . .)



People behind Ethereum

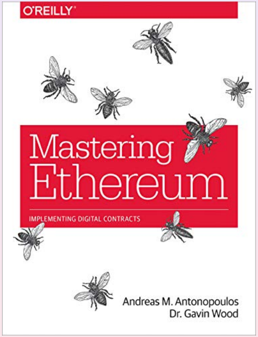


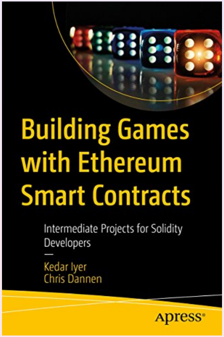
Vitalik Buterin

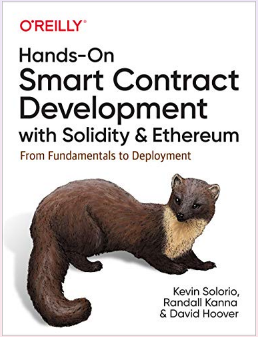
Gavin Wood

Suggested references

Yellow Paper:
`https://ethereum.github.io/yellowpaper/paper.pdf`







The leftmost: `https://github.com/ethereumbook/ethereumbook`

Deterministic (infinite) state machine

A very abstract view of blockchain

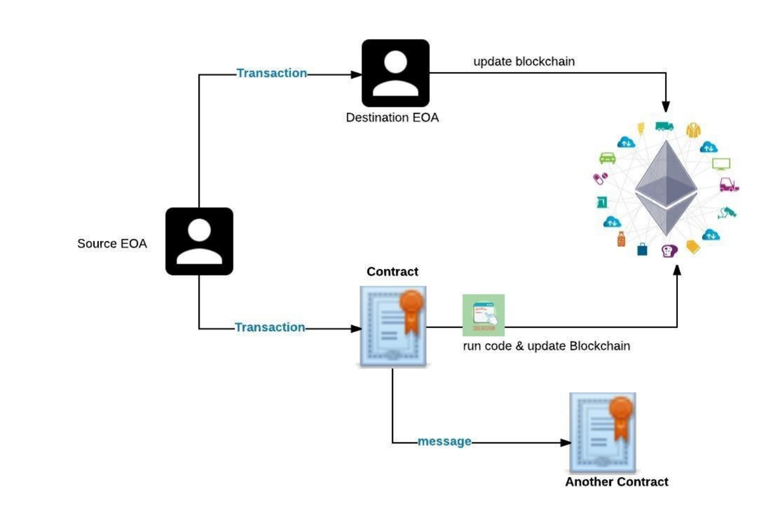
A blockchain is a distributed ledger of transaction requests, aggregated in blocks

- Bitcoin: transaction requests require a change of the set of UTXOs
- Ethereum: transaction requests require a change of a map *key* $\rightarrow$ *value*

The change must be deterministic otherwise consensus cannot be reached!

Externally owned accounts (EOA) and contracts

EOAs have keys, contracts have code, both have an address



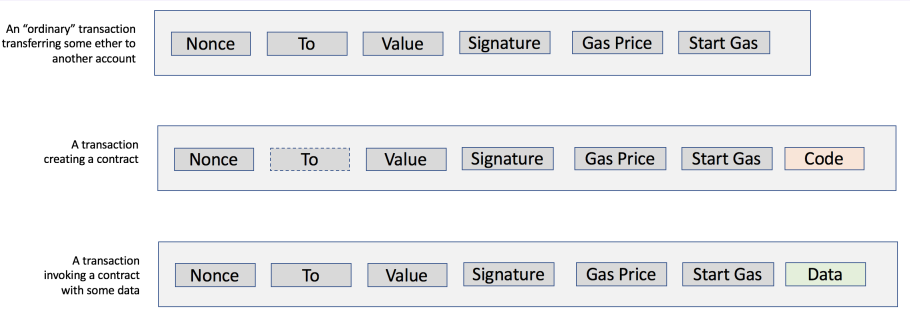
Ethereum transactions

A transaction is a signed message originated by an EOA, transmitted by the Ethereum network, and recorded on the Ethereum blockchain:

- nonce: sequence number per each originating EOA
- gas price: maximum willing to pay
- gas limit: maximum willing to consume
- to: recipient (destination address)
- value: ether sent to destination
- data: generic payload (method name, parameters, contract code...)
- signature: ECDSA signature of the originating EOA

The address of the originating EOA is implied by the signature

Many kinds of transactions



The nonce

The nonce of an EOA

A scalar value equal to the number of transactions sent from the EOA

Wallets keep track of nonces

They increase it and attach to each transaction they create per originating EOA

Nodes check nonces

They count the number n of transactions originated by the EOA. If the nonce is smaller than $n+1$, the transaction is rejected. If the nonce is greater than $n+1$, the transaction is delayed and not yet executed:

- this guarantees transaction ordering
- and avoids transaction replaying

Why Ethereum's transactions have a nonce?

Bitcoin's transactions can only transform UTXOs into TXOs

It is not possible to spend a UTXO again, inside the same history: it would be against the consensus rules

⇒ Executing a valid transaction today makes it invalid tomorrow

Ethereum's transactions can induce any state change or fail

A valid (syntactically correct) transaction can always be executed in Ethereum

⇒ Executing a valid transaction today doesn't make it invalid tomorrow (without a nonce)

The state of Ethereum

The state of Ethereum is a global singleton map $\sigma : key \rightarrow value$

The state is not in blockchain!

Each node keeps and maintains its own copy in a private database

API of the state

1. `value=get(address)`
2. `put(address, value)`

Encoding data into the state

Store the balance of an EOA

```
put(address_of_EOA, balance)
```

Read the balance of an EOA

```
get(address_of_EOA)
```

Encoding data into the state

EOA installs a smart contract

$$\texttt{put}(\underbrace{\texttt{hash(address\_of\_EOA, nonce\_of\_EOA)}}_{\textit{address of the new smart contract}}, \texttt{bytecode\_of\_smart\_contract})$$

A smart contract writes *v* into its *n*th instance variable (field)

$$\texttt{put}(\underbrace{\texttt{hash(address\_of\_smart\_contract, n)}}_{\textit{address of the } n-th \textit{ field}}, \texttt{v})$$

A smart contract reads from its *n*th instance variable (field)

$$\texttt{get}(\underbrace{\texttt{hash(address\_of\_smart\_contract, n)}}_{\textit{address of the } n-th \textit{ field}})$$

The hash of the state

In Bitcoin, the header of a block contains the hash of the transactions in the block

That is, the head of the Merkle tree of transactions. Miners must execute the transactions to validate them, since invalid transactions would make the whole block invalid, which would make the miner lose money

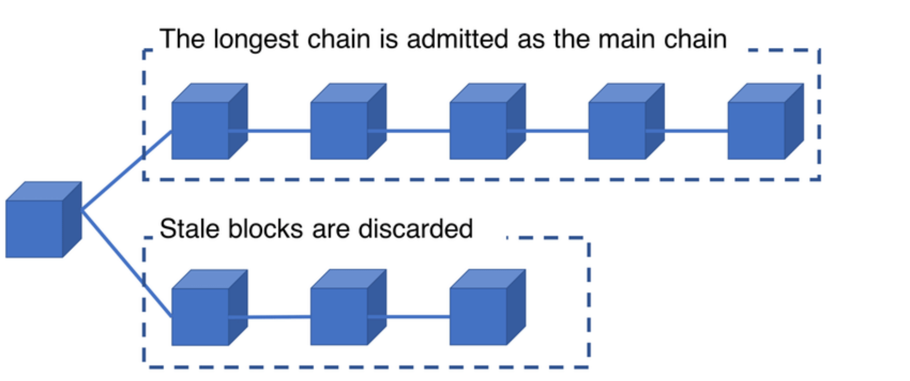
In Ethereum, the header of a block contains the hash of the state at the end of the execution of the transactions in the block

Since syntactically correct transactions are always valid, this obliges the miners to execute the transactions

API of the state

1. `value=get(address)`
2. `put(address, value)`
3. `h=get_hash()`

The state of a node must change across different views



1. the transactions in the discarded chain must be undone (hard in Ethereum)
2. the transactions in the longest chain must be done (easy)

Ethereum = Bitcoin + Git

Undo of state updates, up to the state at the end of an old block

`checkout(old_block.header.state_hash)`

(possible through a Merkle-Patricia trie)

The final API of the state

1. `value=get(address)`
2. `put(address, value)`
3. `h=get_hash()`
4. `checkout(h)`

Smart contracts

A smart contract is...

- a computer program (not *smart* nor *a contract*)
- immutable
- deterministic
- operating on restricted data
- running on a decentralized world computer

In Ethereum:

- compiled into EVM bytecode
- installed in blockchain
- has no keys
- its installer gets no automatic privileges
- runs after a transaction initiated by an EOA
    - or a chain of transactions initiated by an EOA
    - no parallelism, no background processing
- transactions are atomic

Gavin Wood's Solidity

There are many programming languages for Ethereum smart contracts, but Solidity is the de facto standard:

- imperative
- vaguely object-oriented
- in continuous evolution
- non-strongly-typed
- unorthogonal features

- ✓ sequence
- ✓ conditional
- ✓ repetition

$\Rightarrow$ Turing complete (bug or feature?)

A simple Solidity example

```
// Version of Solidity compiler this program was written for
pragma solidity ^0.6.0;

// Our first contract is a faucet!
contract Faucet {
    // Accept any incoming amount
    receive () external payable {}

    // Give out ether to anyone who asks
    function withdraw(uint withdraw_amount) public {

        // Limit withdrawal amount
        require(withdraw_amount <= 1000000000000000000);

        // Send the amount to the address that requested it
        msg.sender.transfer(withdraw_amount);
    }
}
```

Basic Solidity types

`bool`

with constants `true` and `false` and usual operators

`int`, `uint`

signed or unsigned, with usual operators, in increments of 8 bit size: `uint8`, `uint16`, `int24`.... Without specification, they stand for `int256` and `uint256`, respectively

`fixed`$M \times N$, `ufixed`$M \times N$

fixed point arithmetic, signed or unsigned, M bits, N decimals after the point: currently not implemented

Basic Solidity types

`bytes`N

fixed-size array of bytes, of length N

`bytes` or `string`

variable-sized arrays of bytes

Arrays

`uint32[][5]` is a fixed size array of five dynamic arrays of 32 bits unsigned integers

Enumerations

```
enum NAME { A, B, ... }
```

Basic Solidity types

Structures

```
struct pair {
  int16 x;
  uint8 y;
}
```

Mappings

```
mapping(address => uint256) balances;
```

A field of type mapping spreads its values into the state through hashing:

`balances[k]=v` executes `put( hash(balances,k) , v)`

(where `hash(balances,k)` is the *address of* `balances[k]`)

- ⇒ mappings default to 0
- ⇒ there is no `containsKey` (you need a sentinel value)
- ⇒ it is not possible to compute the key set or value set of a mapping
- ⇒ it is not possible to iterate on a mapping

A simple Ponzi scheme

```
pragma solidity ^0.4.21;

contract SimplePonzi {
    address public currentInvestor;
    uint public currentInvestment = 0;

    function () payable external {
        uint minimumInvestment = currentInvestment * 11 / 10;
        require(msg.value > minimumInvestment);
        address previousInvestor = currentInvestor;
        currentInvestor = msg.sender;
        currentInvestment = msg.value;
        // for malicious investors it will return false but not fail
        previousInvestor.send(msg.value);
    }
}
```

The first investment will be burned to address `0x0`

Gas consumption

Each bytecode instruction and transaction type has a gas cost

- it is possible to compute in advance the gas cost of simple functions (use `estimateGas` in the web3 library for instance)
- the result is wrong in the presence of loops or recursion!
  - obvious, since gas cost computation is harder than complexity analysis which can be used to decide termination of programs
    - ⇒ an algorithm for computing gas costs in advance cannot exist
- in general, it is important to know which operations (might) cost much gas, and avoid them
  - loops over unbounded dynamic arrays
  - calls to unknown contracts

A gradual Ponzi scheme

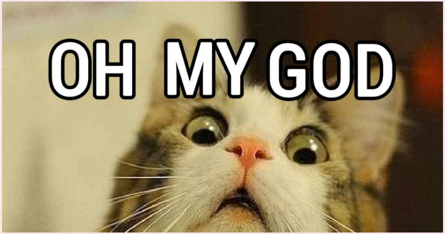
```
pragma solidity ^0.4.21;

contract GradualPonzi {
    address[] public investors; // dynamic array
    mapping (address => uint) public balances; // map
    uint public constant MINIMUM_INVESTMENT = 1e15;

    constructor () public {
        investors.push(msg.sender);
    }

    function () public payable {
        require(msg.value >= MINIMUM_INVESTMENT);
        uint eachInvestorGets = msg.value / investors.length;
        for (uint i = 0; i < investors.length; i++)
            balances[investors[i]] += eachInvestorGets;
        investors.push(msg.sender);
    }

    function withdraw() public {
        uint payout = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(payout);
    }
}
```

Type `address`: Solidity is not strongly-typed

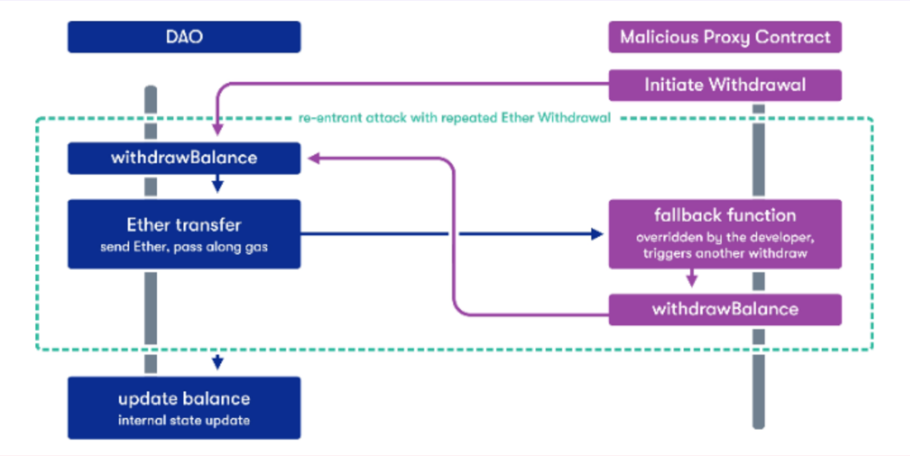
1. casts are not checked
2. parameter types are just Christmas decorations

A function declaring a formal parameter of type `address` or explicitly `C` can actually receive any value, of any type, also completely unrelated to `C`. No run-time error occurs. Callers can inject malicious code through such parameters!

Never talk to strangers!



The DAO attack (2016): they talked to strangers...



The DAO attack (2016)

The most famous reentrancy exploit

- the DAO was a contract for autonomous decentralized organizations
- the attacker used reentrancy to steal 50M$ equivalent of ETH
- the Ethereum team decided to make the consensus rules more restrictive in order to make such transactions illegal and get some of that money back
- some node maintainers didn't accept the change and continued operating with the old rules and another chain id, leading to a network hard fork known as Ethereum Classic

Security best practice

- Minimalism: the simpler, the better
- Code reuse: DRY, use well-known libraries
- Study: be aware of well-known issues and solutions
- Readability: simpler audit
- Test: try corner cases
- Analysis: static or dynamic, still in infancy

Considering the importance of security for smart contracts, it is questionable to have invented Solidity (hard, new, weakly-typed, complex low-level semantics) for writing such delicate pieces of software

Tendermint

Proof of. . .

Who decides the next block?

- Proof of work [PoW] (who works harder and is lucky)
- proof of stake [PoS] (who commits more money)
- proof of space (who commits more disk space)
- proof of authority (who has more authority)
- . . .

PoS is a variant of Practical Byzantine Fault Tolerance (BFT)

Miguel Castro and Barbara Liskov. *Practical Byzantine Fault Tolerance and Proactive Recovery*. ACM Trans. Comput. Syst., 20(4):398–461, November 2002

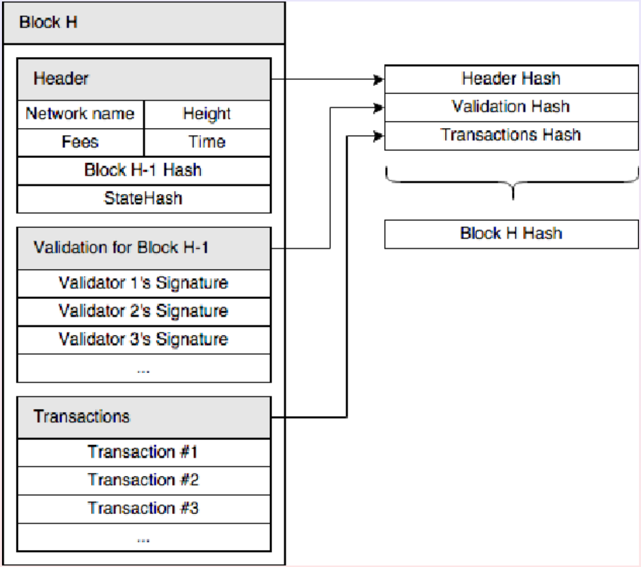
Tendermint (now Ignite): `ignite.com`

Jae Kwon. *Tendermint: Consensus without Mining*, 2014.
`https://tendermint.com/static/docs/tendermint.pdf`

- a dynamic set V of validators decides the next block
- V might be different for each block
  - but deterministically computed from the previous history
- at each height H, each validator $v \in V$:
  1. identifies (deterministically) a validator $p \in V$ that is expected to aggregate some transactions and that proposes a next block b
  2. if v considers b valid, it pre-votes b
  3. v counts how many validators pre-voted b
  4. if v counted at least $\frac{2}{3}$ pre-votes, v pre-commits b
  5. v counts how many validators pre-committed b
  6. if v counted at least $\frac{2}{3}$ pre-commits, v commits b and increases H
  7. v goes back to step 1

Tendermint is BFT. If step 1 or rewards are based on stakes, then it is PoS

Inside a Tendermint block



Proof of stake: can we trust it?

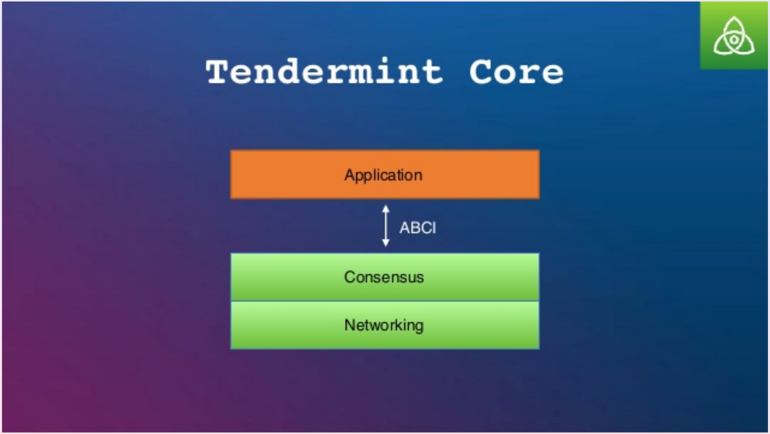
Yes we can: Ethereum successfully moved from PoW to PoS

- it's a special case, whose coin is very valuable: validators are a serious form of investment

No we can't: all new blockchain projects use PoS nowadays

- validators have no interest in being validators (the coin has no value)
- validators are afraid of having a machine always connected and open to the internet
- validators find it expensive to maintain and update their machine
- validators lose cryptocurrency if a blackout or network failure isolate their machine
- please ask this question: "How many validators your blockchain project has, where are they and who maintains such machines" (spoiler: very few, in the same room, all maintained by a single person)

A layered implementation in Golang



ABCI: Application BlockChain Interface

The ABCI

https://docs.tendermint.com/master/spec/abci/abci.html

`checkTx`: called before entering the mempool and to verify blocks

⇒ only transactions that satisfy `checkTx` are added in blocks

- must not modify the state of the application

`beginBlock`: called at the beginning of a block; receives information about the validator set of the previous block and which of them signed the previous block

`deliverTx`: called for each transaction added to a block: it executes the transaction by modifying the state of the application

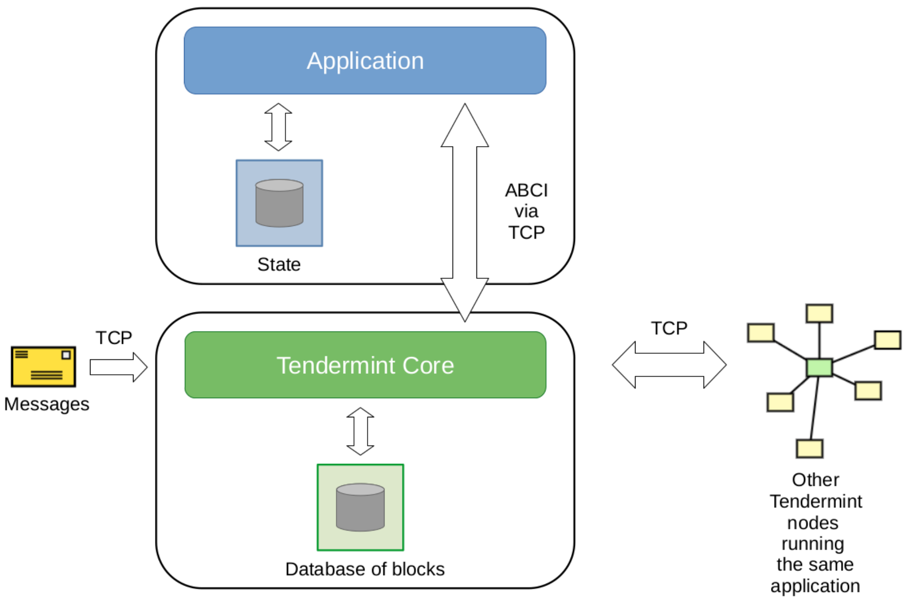
`endBlock`: called at the end of a block; provides information about the validator set for the next block

`commit`: called when a block is being committed; provides the hash of the state of the application

---

`query`: called when the user wants to read data from the blockchain

The database of blocks and the application state



The application state

It must have a function to compute its hash

Only that hash is reported in blockchain, for consensus

It must allow transactional, atomic updates

Between `beginBlock` and `commit`

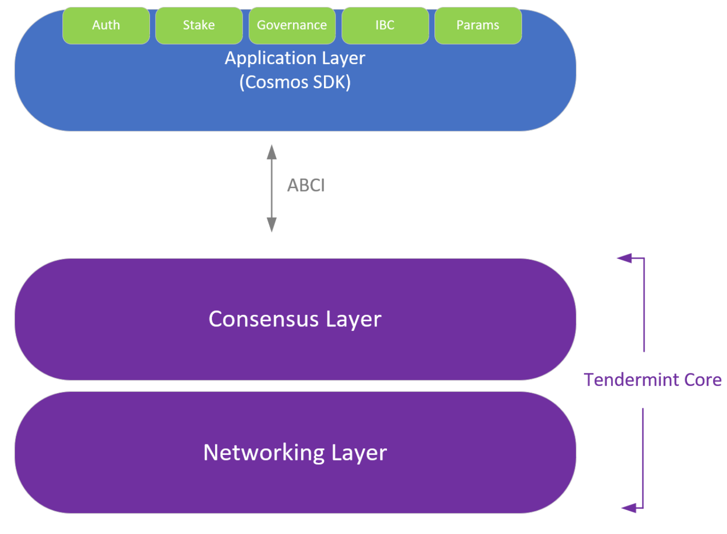
The API of the state

Tendermint enjoys finality: there are no forks

⇒ one never needs to come back in time to the state of a previous block

1. get data
2. put data
3. `h=get_hash()`
4. ~~`checkout(h)`~~ ⇒ big opportunity for garbage collection!

Cosmos: a Tendermint application in Golang



Hotmoka + Takamaka

Hotmoka (Fausto Spoto, 2019–2021): `www.hotmoka.io`

Hotmoka

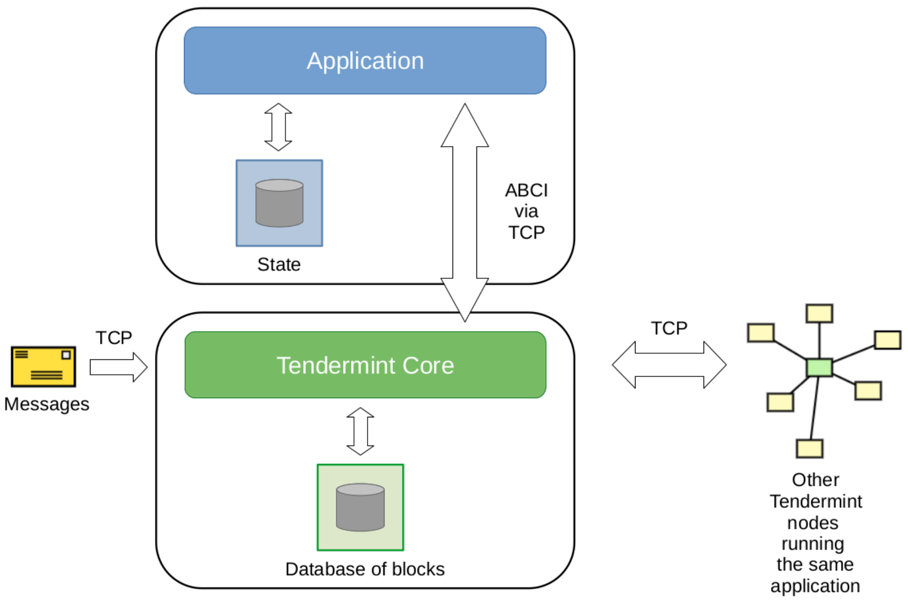
An open-source implementation of a network of nodes:

- nodes of a blockchain
- IoT devices
- computers in the cloud

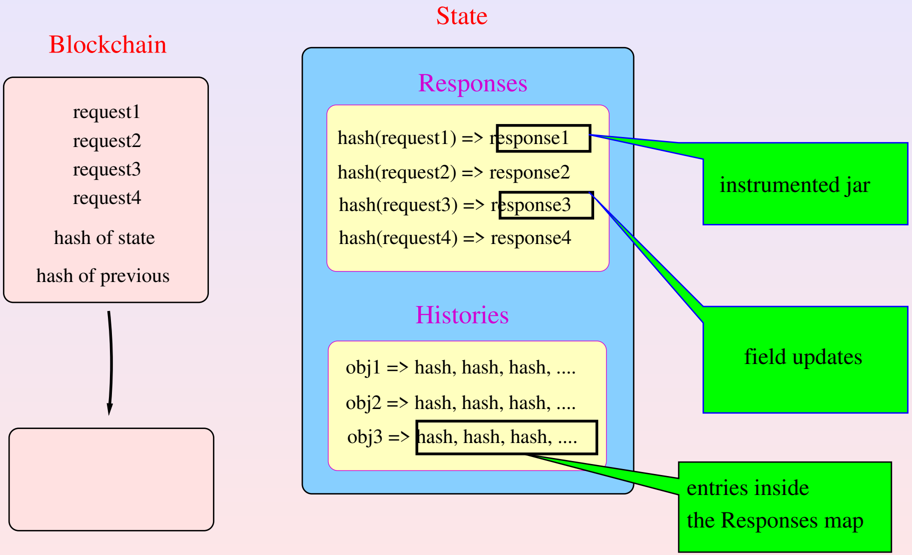
Requests are OO-based

- install code in the node
- create an object
- call a method of an object
- methods are implemented in Takamaka (subset of Java)

Hotmoka nodes can be Tendermint applications



An OO state (hash is sha256)



The state contains actual Java objects

manifest: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#0

machine-independent memory address of an object

```
moka state 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#0 --url panarea.hotmoka.io
```

```
class io.takamaka.code.governance.Manifest (from jar installed at 02dfd29348abaa44f7205251...)
  allowsSelfCharged:boolean = false
  allowsUnsignedFaucet:boolean = true
  chainId:java.lang.String = "chain-ASdWiN"
  gamete:io.takamaka.code.lang.Account = 4f7d7ca1fbea152d8f323c21e1abcfa1d979c7c4ea667d8457381a26b08a2d71#0
  gasStation:io.takamaka.code.governance.GasStation = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b8...
  maxCumulativeSizeOfDependencies:long = 10000000
  maxDependencies:int = 20
  maxErrorLength:int = 300
  signature:java.lang.String = "ed25519"
  skipsVerification:boolean = false
  validators:io.takamaka.code.governance.Validators = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b8...
  versions:io.takamaka.code.governance.Versions = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4...
^ balance:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
^ balanceRed:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
^ nonce:java.math.BigInteger = 227 (inherited from io.takamaka.code.lang.ExternallyOwnedAccount)
^ publicKey:java.lang.String = "" (inherited from io.takamaka.code.lang.ExternallyOwnedAccount)
```

How can Hotmoka identify updates to fields of objects?

The original code

```
public class C {
  public int i;
  public void foo() {
    i = 42;
  }
}
```

No way to know if `i` changed its value during the execution of `foo()`

How can Hotmoka identify updates to fields of objects?

The instrumented code

```
public class C extends Storage {
  public int i, old_i; // aliased at method start
  public void foo() {
    i = 42;
  }
}
```

`i` changed its value during the execution of `foo()` iff at the end `i`≠`old_i`

How can Hotmoka enforce gas limits?

The original code

```
public class C {
  public void foo() {
    while (...) {
      ...
    }
  }
}
```

This loop might run for very long or even forever

How can Hotmoka enforce gas limits?

The instrumented code

```
static long counter;
public class C {
  public void foo() {
    while (...) {
      if (counter++ >= gaslimit)
        throw new OutOfGasError();
      ...
    }
  }
}
```

Actual gas costs are more fine-grained

Verification and instrumentation of jars in state

Each jar that gets installed in a Hotmoka node undergoes two processes:

1. Verification: absence of frequent errors
   - objects stored in state extend `Storage`
   - non-deterministic or non-terminating library code is not used
   - no synchronization
   - no native code
   - no *dangerous* bytecodes
   - no finalizers
   - no static fields (mostly)
   - code annotations are used correctly
   - ...
2. Instrumentation
   - fields of `Storage` classes get duplicated
   - gas metering is weaved into the code
   - code annotations get implemented *by magic*
   - `caller()` is given semantics
   - ...

The Takamaka programming language

Takamaka is the subset of Java that passes the verification of a Hotmoka node. It uses code annotations to implement contract-based aspects:

- `@FromContract` annotates something that can only be called by a contract, not by any other code; hence, it has a `caller()`
- `@Payable` annotates something whose execution requires to pay some cryptocurrency units
- `@View` annotates something whose execution can be run for free, without paying for its gas: it must not generate any update at its end (*pure* code)

Takamaka comes equipped with a support library (`io-takamaka-code`) that defines such annotations and other typical classes that are useful for programming smart contracts (tokens, NFTs, DAOs)

An example of a Takamaka smart contract

```java
import static io.takamaka.code.lang.Takamaka.require;
import java.math.BigInteger;
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Payable;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.lang.View;

public class SimplePonzi extends Contract {
  private final BigInteger _10 = BigInteger.valueOf(10L), _11 = BigInteger.valueOf(11L);
  private PayableContract currentInvestor;
  private BigInteger currentInvestment = BigInteger.ZERO;

  public @Payable @FromContract(PayableContract.class) void invest(BigInteger amount) {
    BigInteger minimum = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimum) >= 0, () -> "you must invest at least " + minimum);

    if (currentInvestor != null)
      currentInvestor.receive(amount); // no risk of reentrancy

    currentInvestor = (PayableContract) caller();
    currentInvestment = amount;
  }

  public @View BigInteger getCurrentInvestment() {
    return currentInvestment;
  }
}
```

An insurance smart contract in Takamaka

The contract allows one to insure specific days of the year

If it rains on those days, one will get an indemnization larger than the cost of the insurance

- much larger in summer
- just a bit larger in winter

The contract provides the following functionalities:

- construction, upon specification of the oracle:

```
@FromContract @Payable Insurance(BigInteger amount, Contract oracle)
```

- purchase of an insurance for specific days:

```
@FromContract(PayableContract.class) @Payable void buy
(long amount, int day, int month, int year, int duration)
```

- notification of rain and indemnization:

```
@FromContract void itRains()
```

An insurance contract

```
public class Insurance extends Contract {
  public final static long MIN = 1_000, MAX = 1_000_000_000;
  private final Contract oracle;
  private final StorageSet<InsuredDay> insuredDays = new StorageTreeSet<>();

  public @FromContract @Payable Insurance(BigInteger amount, Contract oracle) {
    this.oracle = oracle;
  }

  // inner class
  private static class InsuredDay extends Storage { /* not shown */ }

  public @FromContract(PayableContract.class) @Payable void buy
    (long amount, int day, int month, int year, int duration) { /* shown later */ }

  public @FromContract void itRains() { /* shown later */ }
}
```

Buy an insurance

```
public @FromContract(PayableContract.class) @Payable void buy
    (long amount, int day, int month, int year, int duration) {

  require(duration >= 1, "you must insure at least one day");
  require(duration <= 7, "you cannot insure more than a week");
  require(amount >= MIN * duration,
    () -> "we insure a single day for at least " + MIN + " units of coin");
  require(amount <= MAX * duration,
    () -> "we insure a single day for up to " + MAX + " units of coin");

  // if the date is wrong, this generates an exception
  LocalDate start = LocalDate.of(year, month, day);

  PayableContract payer = (PayableContract) caller();
  for (int offset = 0; offset < duration; offset++)
    insuredDays.add(new InsuredDay
                    (payer, amount / duration, start.plusDays(offset)));
}
```

Pay the indemnization

```
public @FromContract void itRains() {
  require(caller() == oracle, "only the oracle can call this method");

  // pay who insured today
  insuredDays.stream()
    .filter(InsuredDay::isToday)
    .forEachOrdered(insuredDay ->
          insuredDay.payer.receive(insuredDay.indemnization()));

  // clean-up the set of insured days
  insuredDays.stream()
    .filter(InsuredDay::isTodayOrBefore)
    .forEachOrdered(insuredDays::remove);
}
```

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

```
public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long
amount, int day, int month, int year, int duration)
```

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

```
public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long
amount, int day, int month, int year, int duration)
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

```
public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long
amount, int day, int month, int year, int duration)
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io
```

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

```
public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long
amount, int day, int month, int year, int duration)
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io

Do you really want to spend up to 852500 gas units to install the jar [Y/N] Y
total gas consumed: 852500
  for CPU: 255
  for RAM: 1326
  for storage: 381762
  for penalty: 469157     !!!!!!!
io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException:
it/univr/insurance/Insurance.java method buy:
@Payable can only be applied to a @FromContract method or constructor
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io

Do you really want to spend up to 852500 gas units to install the jar [Y/N] Y
total gas consumed: 853700
  for CPU: 255
  for RAM: 1326
  for storage: 382362
  for penalty: 469757     !!!!!!!
io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException:
it/univr/insurance/Insurance.java:95:
illegal call to non-white-listed method java.util.stream.Stream.forEach
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
```

Off-chain verification

Using the blockchain as a debugger is very expensive...

`moka verify <jar> --libs dependencies`

We verify the jar off-chain, to find all errors

```
moka verify
jar
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
```

Off-chain verification

Using the blockchain as a debugger is very expensive...

`moka verify <jar> --libs dependencies`

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
--libs dependencies
```

Off-chain verification

Using the blockchain as a debugger is very expensive...

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
--libs io-takamaka-code-1.0.0.jar
```

Off-chain verification

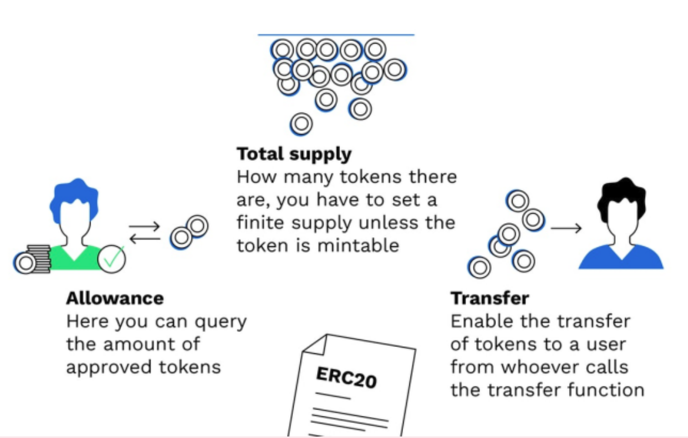
Using the blockchain as a debugger is very expensive. . .

`moka verify <jar> --libs dependencies`

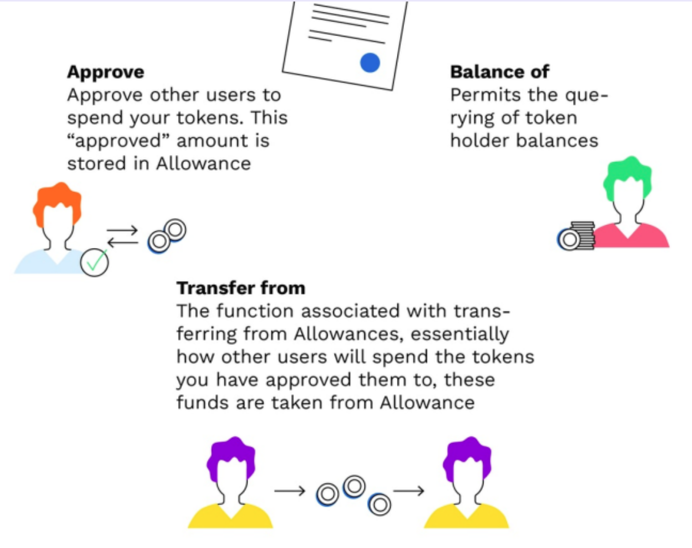
We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
--libs io-takamaka-code-1.0.0.jar


it/univr/insurance/Insurance.java method buy:
  @Payable can only be applied to a @FromContract method or constructor
it/univr/insurance/Insurance.java:46:
  caller() can only be used inside a @FromContract method or constructor
it/univr/insurance/Insurance.java:95:
  illegal call to non-white-listed method java.util.stream.Stream.forEach
it/univr/insurance/Insurance.java:99:
  illegal call to non-white-listed method java.util.stream.Stream.forEach
```

ERC20 Tokens



ERC20 Tokens

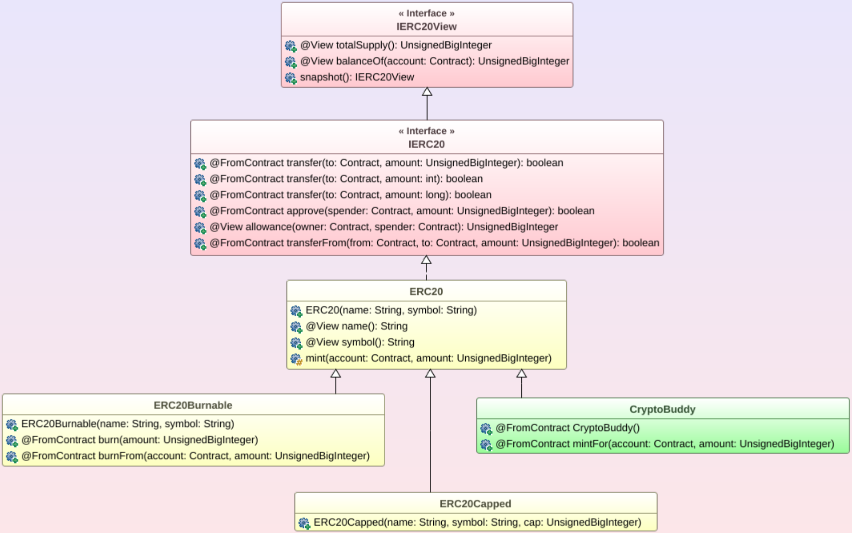


The OpenZeppelin reference implementation

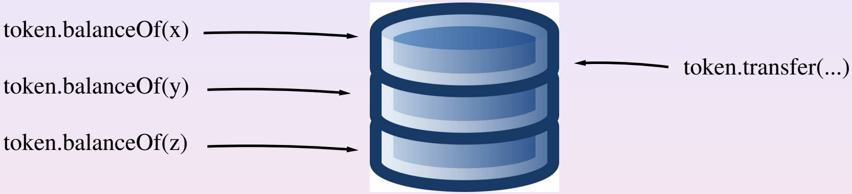
We follow (in part) the implementation by OpenZeppelin

https://docs.openzeppelin.com/contracts/2.x/api/token/erc20

The hierarchy of the implementation

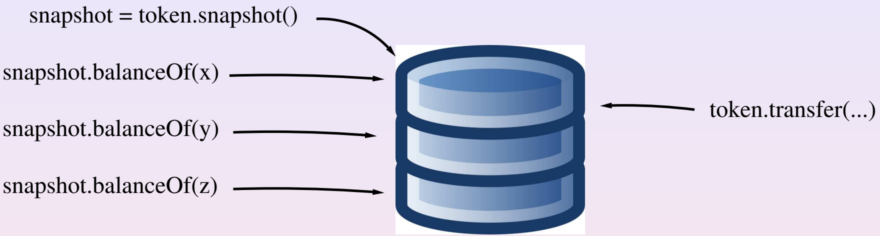


Inconsistent view



Between a call to `balanceOf` and the next, the state of the token might change in the database because other users might call the transfer functions, concurrently

Consistent view



- `snapshot()` works in $O(1)$
- all calls to `balanceOf` refer to the same, consistent state of the token (possibly not the latest)
- impossible in Solidity, where maps cannot be cloned

References

- Fausto Spoto: *A Java Framework for Smart Contracts*. Financial Cryptography Workshops 2019: 122-137
- Fausto Spoto: *Enforcing Determinism of Java Smart Contracts*. Financial Cryptography Workshops 2020: 568-583
- Luca Olivieri, Fausto Spoto, Fabio Tagliaferro: *On-Chain Smart Contract Verification over Tendermint*. Financial Cryptography Workshops 2021: 333-347
- Marco Crosara, Luca Olivieri, Fausto Spoto, Fabio Tagliaferro: *Re-engineering ERC-20 Smart Contracts with Efficient Snapshots for the Java Virtual Machine*. BCCA 2021: 187-194, to appear in Cluster Computing
- Andrea Benini, Mauro Gambini, Sara Migliorini, Fausto Spoto: *Power and Pitfalls of Generic Smart Contracts*. BCCA 2021: 179-186, to appear in Cluster Computing

Proof of space

- proof of work is too expensive and polluting
- proof of stake is complex and makes it hard to have many really independent validators

Proof of space

In 2014, Burstcoin (later Signum) implemented a mining algorithm where miners must solve a puzzle to gain the right to mine a new block:

- the puzzle is too hard to be computed for each new block
- the puzzle becomes very simple if some information is precomputed and stored on disk
- the CPU of the miners remains largely idle: no electricity cost
- the more precomputation, the more disk is committed, the higher the probability of solving the puzzle and mining a new block

Mokamint (`www.mokamint.io`)

Signum is monolithic, non-commented, Java 5, undocumented code

The idea of Mokamint (Fausto Spoto 2023, work in progress)

- Tendermint: a generic blockchain engine based on proof of stake
- Mokamint: a generic blockchain engine based on proof of space

Later attach Hotmoka on top of Mokamint, as an application instance