Bitcoin and Ethereum design demystified

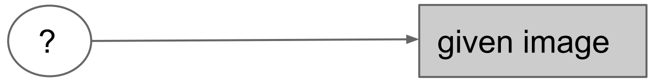
Lecture notes

Alexey Akhunov, March 2016

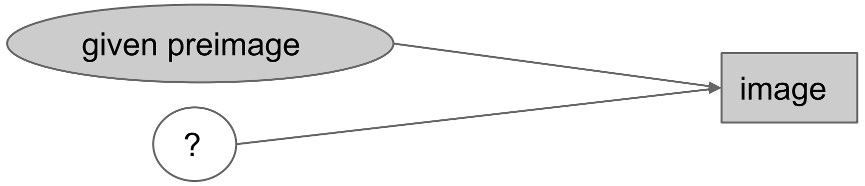
Introductory topics - properties of cryptographic hash functions

Properties of cryptographic hash functions:

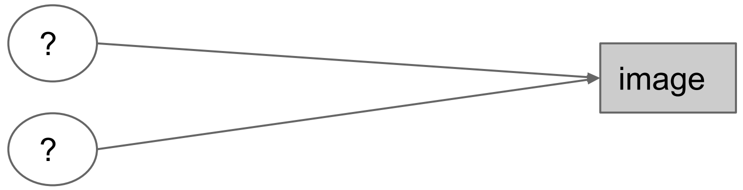
1. **Preimage resistance -** practical impossibility of finding the input (preimage) for a given output (image).



2. **Second preimage resistance -** practical impossibility, for a given preimage, of finding another, different preimage, such that the image for both preimages is the same.



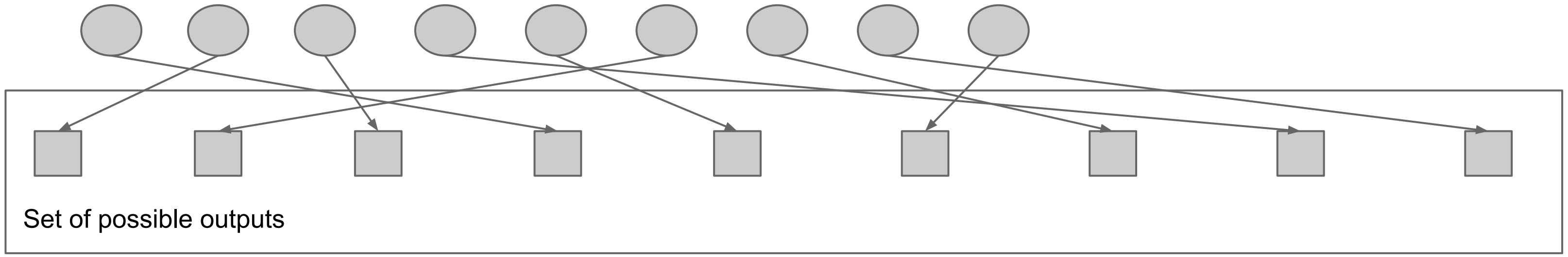
3. **Collision resistance -** practical impossibility of finding two distinct preimages, such that the image for both preimages is the same.



Introductory topics - properties of cryptographic hash functions

It can be easily shown that the third property implies the second, and the second implies the first. Therefore, as a rule, you might often only encounter collision resistance mentioned, even in the situations where only preimage resistance or second preimage resistance is required.

Another property, which turns out to be very useful, and is generally satisfied empirically, but which is not possible to prove theoretically - is the property of “random oracle”. If one generates lots of inputs in an arbitrary way (but using an **efficient algorithm**), and applies a random oracle to all of them, the outputs would be uniformly distributed among the set of all possible outputs:

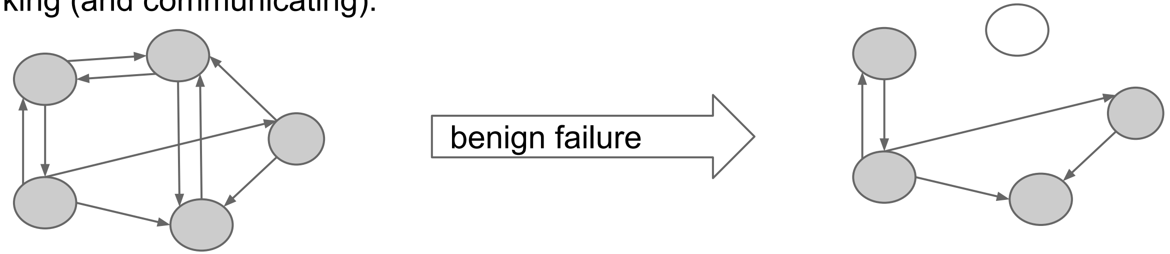


If the distribution was noticeably skewed from uniform, it would make finding collision more efficient by concentrating search on the areas where more inputs map to fewer possible outputs.

Current (since August 2015) U.S. NIST (National Institute of Standards and Technology) standard for cryptographic hash function is SHA3 (Secure Hash Algorithm, version 3), based on the algorithm called Keccak.

Introductory topics - Byzantine fault tolerance

Most of the real distributed systems designed in the past assume that only so-called benign fault tolerance is required. Participants of such systems are expected to sometimes fail (misbehave), but only in a very simple (benign) way - they stop working (and communicating):



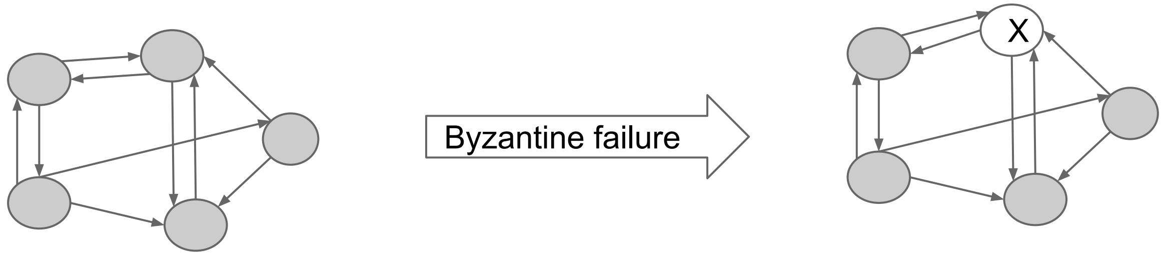
It is known that given such failures, participants can still reach consensus (under certain conditions) about the value of a certain variable (e.g. your account's balance). Consensus is normally understood as a combination of three conditions:

1. Finality - all "honest" and "active" participants will eventually decide on the value of the variable.
2. Agreement - all participants that did decide due to finality (1), decide on the same value.
3. Validity - only a value proposed by one of the participants can be agreed upon (this excludes degenerate cases, like always agreeing about zero, and also agreeing on values that do not make sense)

In the presence of the benign failures, participants can reach consensus given that majority (often called quorum) of them are non-faulty. It comes from the observation that any two majorities necessarily overlap. Most popular consensus algorithm for benign fault tolerance is Paxos (by Leslie Lamport, 1998). Recently, a popular derivative emerged, called "Raft".

Introductory topics - Byzantine fault tolerance

Byzantine failures are harder to deal with, because in this model faulty participant may continue communicating, but they behave in a completely arbitrary (and potentially adversarial) way, i.e. they are expected to drop, delay, modify, repeat messages, as well as create messages not conforming to the protocol, including with the purpose of disturbing or exploiting the correct working of the system. Such participants may not necessarily be known to the others.

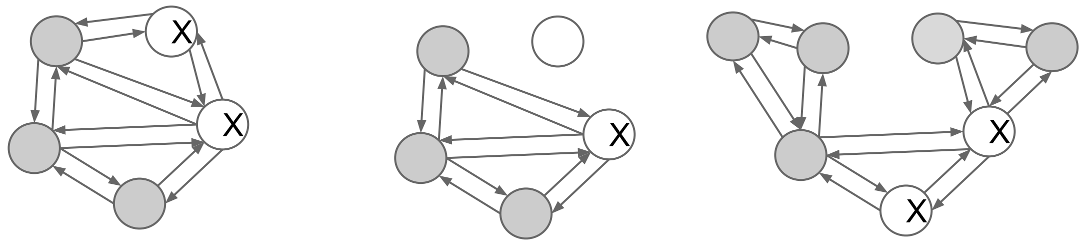


The name comes from the problem stated by Leslie Lamport in 1982 called ‘Byzantine Generals problem’. In fact it was mostly ‘rebranding’ of the problem, which Lamport et al stated in 1980 but did not attract nearly as much attention as Dijkstra’s dining philosophers.

The most well-known result in Byzantine Fault Tolerance currently is by Fischer, Lynch and Patterson (FLP), 1985. They proved that in an **asynchronous** system it is generally not possible to reach consensus even if just a single participant is faulty. Asynchronicity there means that the participants may process messages arbitrarily long time, or stop communicating for some periods of time, and message travel with an unpredictable speed (however they are guaranteed to arrive eventually), and the participants do not have access to a globally synchronized clock.

Introductory topics - Byzantine fault tolerance

In real distributed systems requiring consensus, it is necessary to make assumptions for the system to become quasi-synchronous, for example, by relying on the clock (assuming upper bound on the clock drift), introducing timeouts etc. If asynchronicity is dropped, it turns out that the system can reach consensus if more than ⅔ of participants are active and honest (correctly following the protocol), and are able to reliably communicate to each other. Below are the examples where it is not true:

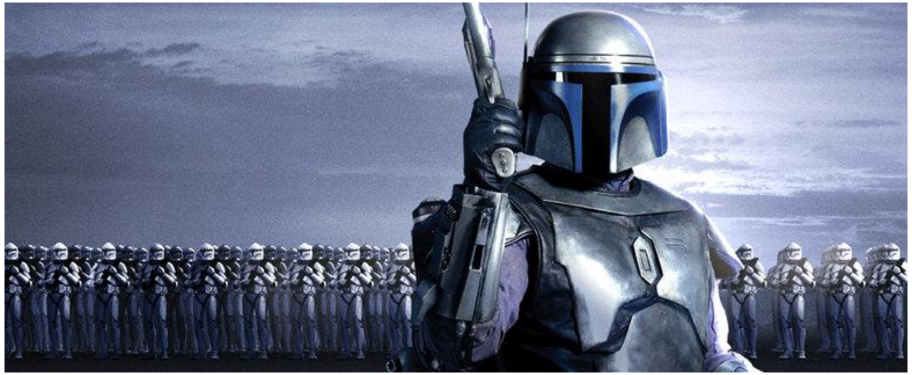


The third example illustrates that it is not only the number of honest participants, but also their connectivity to each other that matters for the possibility of the distributed consensus.

It can also be observed that certain consensus failures only affect finality but not agreement or validity. For example, dishonest nodes may stop honest nodes from changing the value of the variable, but they cannot corrupt the value that was previously agreed upon.

Most famous practical implementation of Byzantine Fault Tolerance in synchronous case is by Castro and Liskov, 1999, and called Practical Byzantine Fault Tolerance (PBFT). Their algorithm requires that all participants are known in advance.

Introductory topics - Sybil attack (aka Attack of Clones)



If participants (users) of the system are not known in advance, it can be subject to Sybil attack (Sybil is a character of 1973 book about treatment of a woman with dissociative identity disorder, having 16 different personalities). If consensus is based by voting, and the eligibility for voting is too permissive, the system can be overwhelmed by an attacker creating many instances of voting users.

The main idea behind Sybil attack protections is associating voting with something having limited supply, like real-world identities, connectivity to existing users, computational resources, virtual currency.

Introductory topics - Digital signature

In cryptography, the goal of digital signature is to attest that two messages have the same origin. It is often called ‘origin authentication’. This cannot be achieved without preserving the message integrity. Unlike message authentication codes, MAC (they serve the same purpose as digital signatures), which can only be verified by the recipient, digital signatures can be verified by anybody (third party). Because of this property, digital signatures can also help achieve non-repudiation, or non-deniability of the signer.

As a rule, digital signatures are based on the usage of asymmetric cryptography, with secret and public keys. Secret key is used to produce the signature, and the public key is used for subsequent verification.

Scheme	**Puzzle**	**Typical key size, bits**	**Underlying algorithm**	**Best publicly known cryptanalysis**
RSA	Factoring of semiprime numbers	3072	RSA	General number field sieve (subexponential)
DSA	Discrete log on multiplicative group of finite fields	3072	El-Gamal (improved by NSA)	Index calculus (subexponential)
ECDSA	Discrete log on group of points on elliptic curves	256	El-Gamal (improved by NSA)	Polard Rho (exponential)

Introductory topics - Digital signature

Most commonly, digital signatures are based on RSA, DSA or ECDSA schemes. Security of RSA relies on the computational hardness of factoring large numbers, and both DSA and ECDSA schemes rely on the computational hardness of discrete logarithm problem. In DSA, discrete logarithm would need to be computed on integers (on a multiplicative group of a finite field), whereas in ECDSA discrete logarithm would need to be computed on a group formed by the point arithmetic on an elliptic curve. ECSDA is preferred to DSA due to existence of index calculus algorithm computing discrete logarithm on integers in subexponential time. As of now, there is no publicly known way of attacking elliptic curves in sub-exponential time (provided that parameters are carefully chosen). Therefore, achieving the same level of security requires much shorter keys and consequently, better performance of encryption/decryption.

Both factoring and discrete logarithm problem can be efficiently solved by a quantum computer of sufficient size, using famous Shor’s algorithm, by Peter Shor, 1995. At the core of it is finding the period of modular exponentiation via quantum Fourier transform. It is interesting to note that longer RSA, DSA, ECDSA keys will require larger number of qubits in the quantum computer to run Shor’s algorithm, but will not increase its runtime.

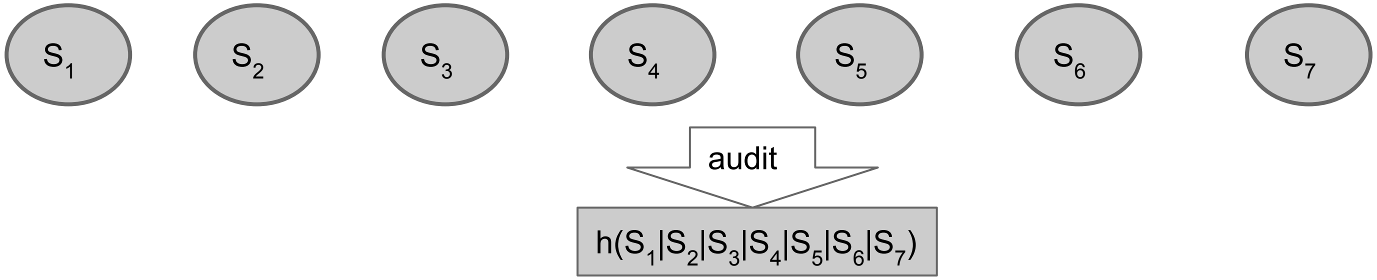
NSA has been actively promoting the use of elliptic curve cryptography since 1995, until August 2015, when it published an advisory mostly related to post-quantum cryptography and largely signalled the end of the agency’s strong commitment to elliptic curves cryptography. This sparked a lot of controversy and conspiracy theories, trying to answer the question: ‘Why NSA is telling the organizations that are still employing RSA not to bother to move to elliptic curves anymore?’

There are some ‘post-quantum’ digital signature algorithms (no known quantum algorithm is known to efficiently break them), such as Lamport signatures and Niederreiter scheme. However, they have serious limitations - in Lamport scheme, each key pair can only be used limited number of times, in Niederreiter - keys are very large.

Introductory topics - Merkle tree

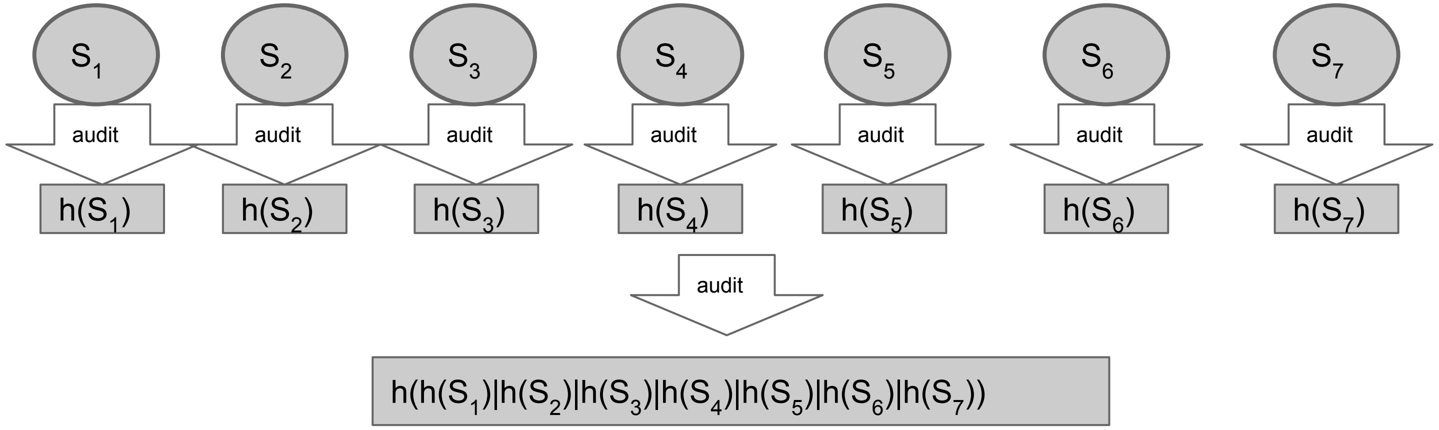
Merkle tree is one of the data structures that are similar in their purpose to cryptographic hash functions. As well as cryptographic hash function, Merkle tree associates a set of arbitrary size (of strings, for instance) with a relatively short code (output of the hash function). An additional property of Merkle tree is the ability to construct a short proof that a given element is indeed contained in the set.

Example: for a gold vault, there is a set of strings, each specifying client account number and amount of gold stored for the client. A trusted auditor inspects all the cells with gold, then computes the cryptographic hash for full set of strings ('|' below is a string concatenation operator), and finally published the result (very short code) in a newspaper. Now, in order for the gold vault to prove to a client Alice, that her string was in the set, it will have to show the full set to Alice, and therefore disclose all other clients' holdings:



Introductory topics - Merkle tree

To fix this problem, the auditor could have applied the hash function to each string individually, then hash the set of the hashes

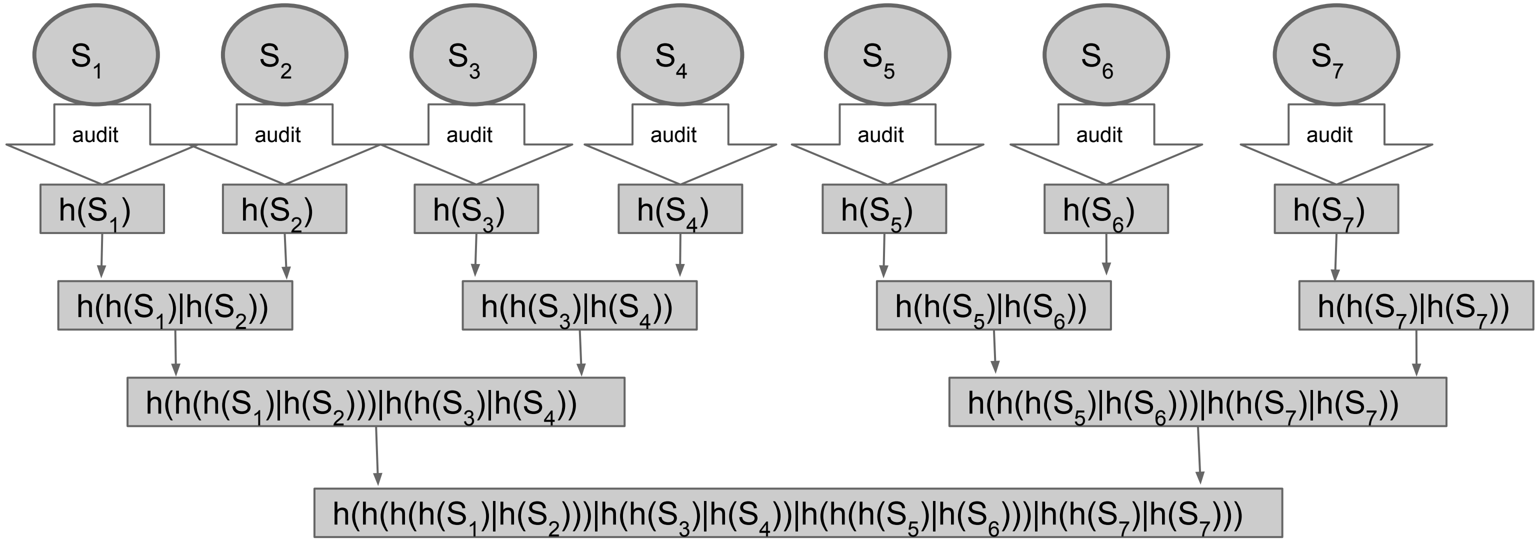


To prove to Alice that her gold is safe, the gold vault now needs to present all the hashes of the strings. Alice can check that the hash of her string is one of them and the resulting hash matches what was published in the newspaper.

Since hash function is preimage resistant, Alice cannot learn anybody else's holdings. But this approach is still not ideal, because of amount of data that needs to be transmitted to the clients like Alice as a proof of their holdings.

Introductory topics - Merkle tree

To further improve the situation, the auditor can build a hash tree (the last value gets used twice if the size of a level is odd):



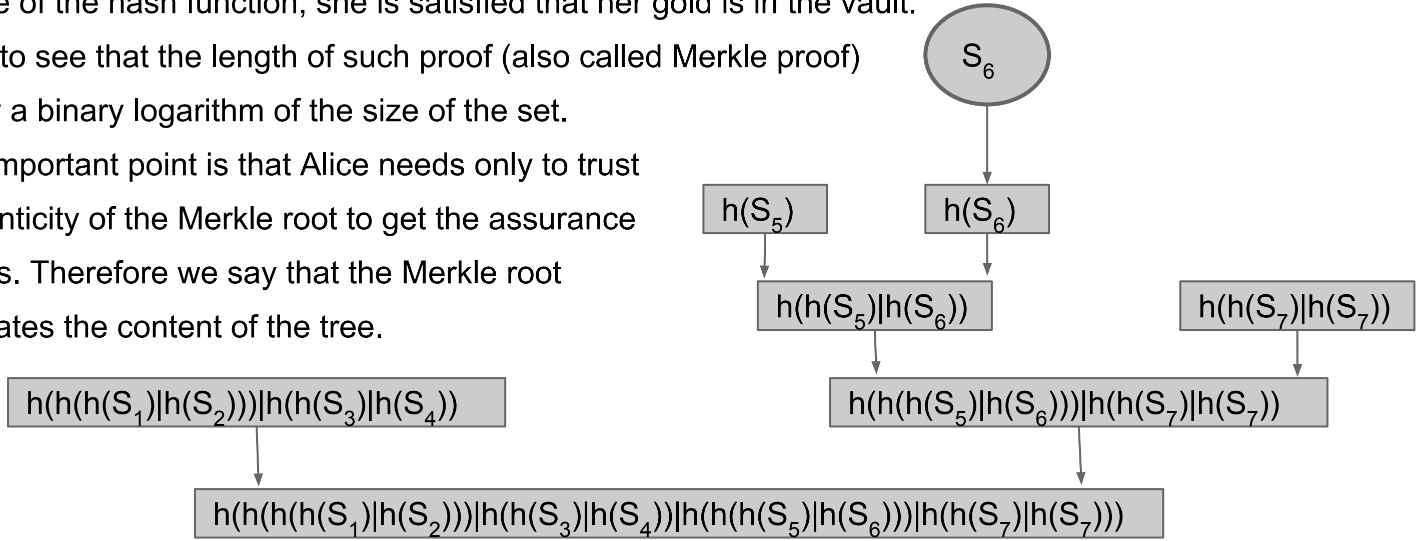
This is a Merkle tree. The root (also called Merkle root) is then published in the newspaper.

Introductory topics - Merkle tree

Now suppose Alice's string is S_6. Then the proof she should be satisfied with consists of the information in the diagram. This is exactly what Alice needs to re-do the auditor's calculations from her string to the root. Given second preimage resistance of the hash function, she is satisfied that her gold is in the vault.

It is easy to see that the length of such proof (also called Merkle proof) is roughly a binary logarithm of the size of the set.

Another important point is that Alice needs only to trust the authenticity of the Merkle root to get the assurance she needs. Therefore we say that the Merkle root authenticates the content of the tree.


S_6
$h(S_5)$
$h(S_6)$
$h(h(S_5)|h(S_6))$
$h(h(S_7)|h(S_7))$
$h(h(h(S_1)|h(S_2)))|h(h(S_3)|h(S_4))$
$h(h(h(S_5)|h(S_6)))|h(h(S_7)|h(S_7))$
$h(h(h(h(S_1)|h(S_2)))|h(h(S_3)|h(S_4))|h(h(h(S_5)|h(S_6)))|h(h(S_7)|h(S_7)))$

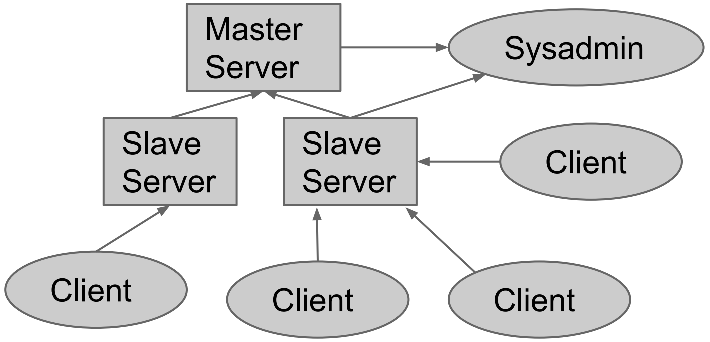
Data structures with such properties are often called 'authenticated data structured', or 'tamper-resistant data structures'.

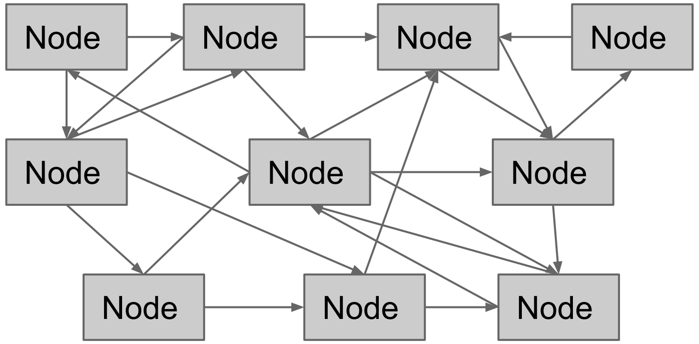
Peer-to-peer network

Bitcoin system has come to existence in 2009, following the 2008 paper published in the Internet by pseudonym Satoshi Nakamoto. It is believed that it might be not a single person, but a group of researchers or hobbyists.

Let's start with defining the goal of Bitcoin for the purpose of the design (with the benefit of the hindsight, of course): **Open system for transfer of purchasing power**.

In order to be open and to stay open (admitting any users without restrictions), the first choice was made in favour of a distributed and decentralised network, or peer-to-peer network. In such a network, all computers are equal - there is no hierarchy, no dedicated server and client roles, no system administrators:

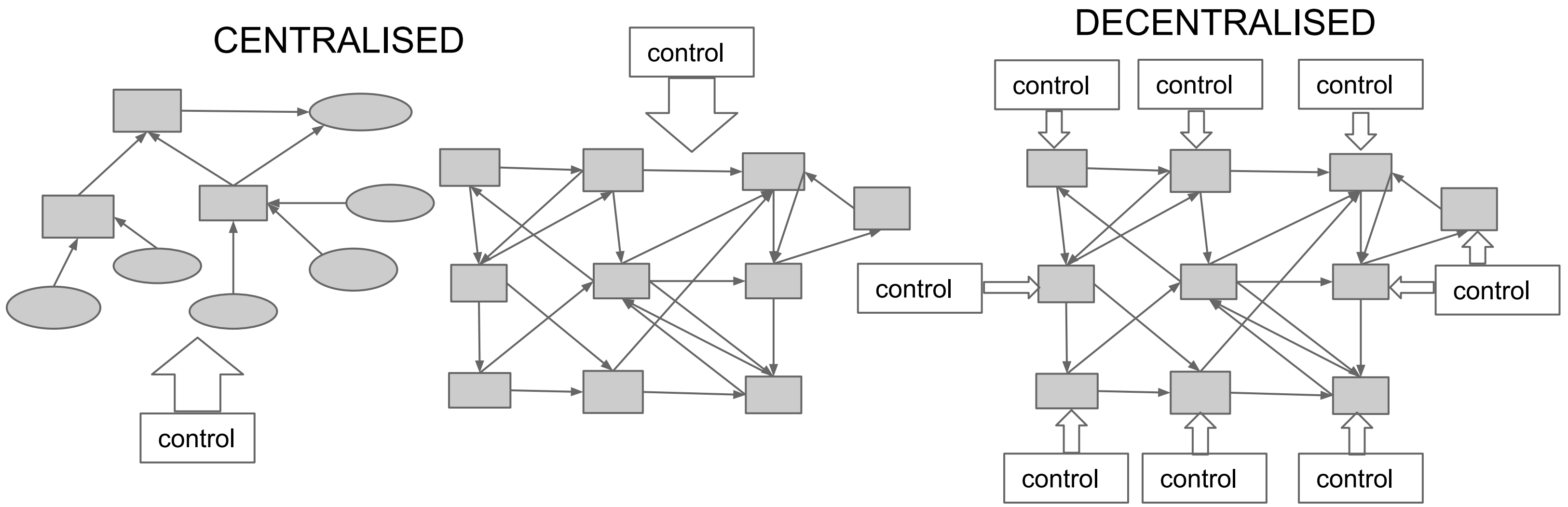




It is relatively easy to see the operational advantages of peer-to-peer networks compared to hierarchical networks: better resiliency, more efficient dissemination of data (recall BitTorrent).

Operational challenges of a decentralised network

Both hierarchical network and peer to peer network can be centralised (to a certain degree).



“Control” in the above diagram can have a broad definition - apart from direct control, it can be someone’s ability to influence most of the participants, or remove crucial functionality from them.

There are operational challenges of decentralised networks - it is harder to perform non-compatible software upgrades, consensus is harder to reach due to lack of control and a priori unknown structure of the network.

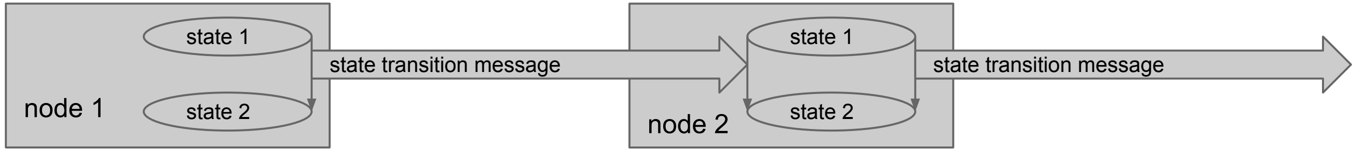
The protocol

At the time of of writing (November 2015), Bitcoin network had around 6100 nodes on average. This number has been steadily going down over the last couple of years: https://getaddr.bitnodes.io/dashboard/?days=365

Nodes connect to each other. As a rule, each node connects to around ten input peers, where it takes the information, and around ten output peers, where the information is sent to. Having connected to the network, nodes start creating and relaying messages according to the protocol, also called Bitcoin protocol. The protocol defines what constitutes a valid message (protocol's syntax), and what the messages mean (protocol's semantics).

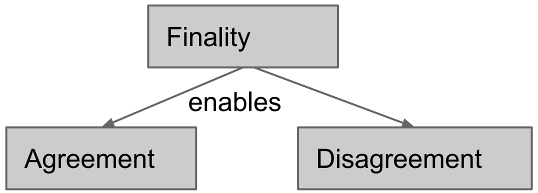
Protocol	
SYNTAX - Structure of a valid message	SEMANTICS - Meaning of a valid message

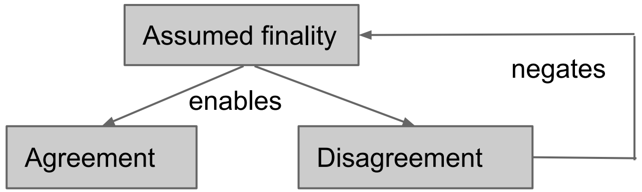
Semantics of such protocols could usually be defined in terms of a replicated state machine (however, Bitcoin protocol does not formally define such semantics, and relies on the reference implementation). In a replicated state machine, each node maintains the state of the system and the protocol helps nodes perform state transitions and thus replicate the state across the network.



Fault tolerance properties

Bitcoin protocol does not implement Byzantine fault tolerance, because it **does not define finality** of any state transition - this is always subjective and up to the participants. Thus, finality is implicitly assumed by some rule (e.g. wait one hour and consider transaction final). This rule cannot be objective, because one might need to wait much longer if the network is overloaded, or is experiencing a partition, which may or may not be noticed by the participants. The simple reason for not providing finality is that achieving finality in a distributed system is hard enough (i.e. committing a distributed transaction), and it is even harder in a decentralized system with potentially Byzantine participants. A lot of financial loses in Bitcoin happen due to wrongly assumed finality.



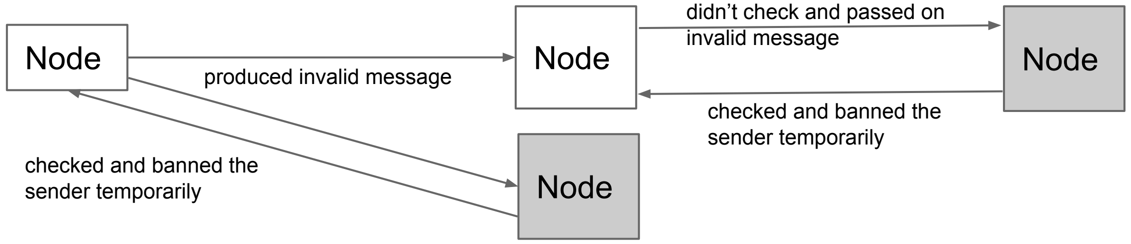


When finality is assumed, **agreement is believed to be achievable**. In practice, disagreements mostly happen due to differences in the implementations used by different nodes, or implementations being dependent on the environment in a subtle ways. Such disagreements are called partitions, or “forks”. They occur when certain state transitions considered valid by some nodes, but invalid but others, and therefore group of nodes (partitions, or branches of the fork) end up maintaining different states of the system. The lack of formal definition of the protocol makes it harder to resolve such partition, because it might not always be clear which of two behaviours is correct, and so-called “bug for bug” compatibility is required between versions. Resolution of such a disagreement leads to “reverted finality” for some participants.

Fault tolerance properties and enforcing validity

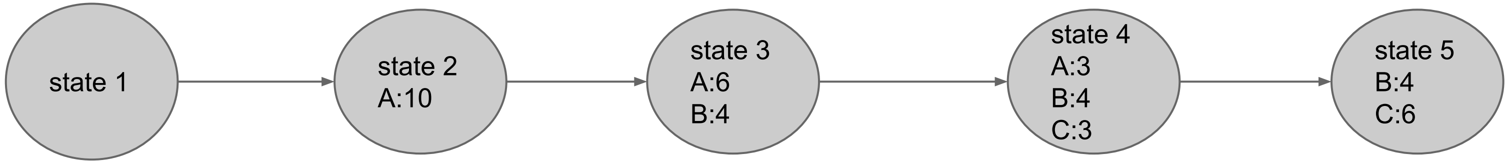
Validity is the only property of consensus that can be achieved, provided finality and agreement. This is due to the fact that all state transition messages are authenticated either by the means of digital signature, or cryptographic hash function. Both can be efficiently verified.

The main function of the nodes in the network to verify all the messages and only relay those that are compliant with the protocol. To reinforce the rationale for doing so, there exists a temporary ban of the peers. If a node caught not complying to the established protocol (by relaying invalid messages), it is added to the temporary blacklist by its peers. Therefore, a non-compliant node (malfunctioning or malicious) will be excluded from the network relatively quickly.

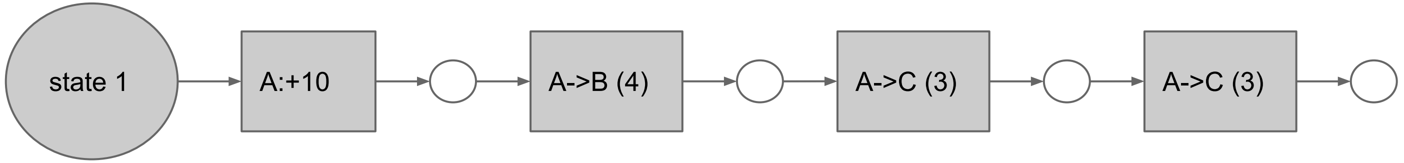

Node
produced invalid message
Node
didn't check and passed on invalid message
Node
checked and banned the sender temporarily
checked and banned the sender temporarily
Node

The public ledger

It follows from the lack of defined finality, that all states have to be kept indefinitely, so that the protocol can in theory roll them back and revert to any of the previous states. One could in theory store the history of all states:



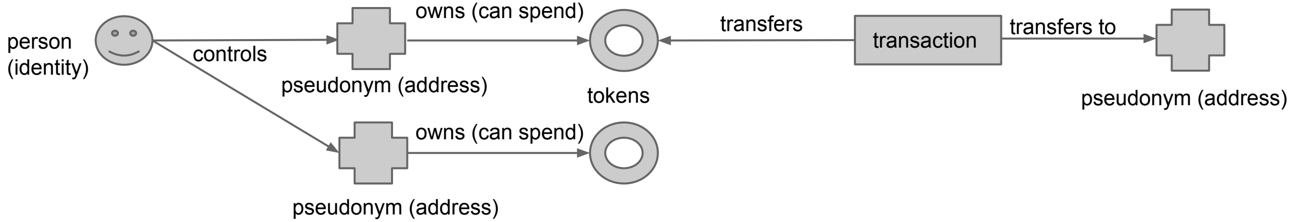
However, that would involve a lot of repetition, especially as the state grows in size. There are multiple ways of making the state storage more efficient. The choice in Bitcoin is to store the initial state (empty) and all state transitions. All intermediate states can to be reconstructed on demand, but are not explicitly stored:



This history of state transitions is called the ledger and a copy of it is stored by all the nodes in the network, so there is a very little chance for this history to be lost. Because it is very easy to obtain a copy of this ledger, simply by joining the network (one needs to download the software, run it on a computer with the Internet connection, and wait for the ledger to be downloaded from other nodes), the ledger is also called the public ledger.

Transfer of purchasing power

In order to support transfer of purchasing power, Bitcoin protocol defines tokens of value, with the same name, “bitcoin”. The tokens are fungible (exchangeable for one another), and divisible up to 100 millionth part (called Satoshi). Bitcoin tokens can be transferred between pseudonyms, also called addresses. One person can have multiple pseudonyms and one pseudonym can be controlled by multiple people (this is like sharing a password to an online bank account). Correspondence between identities and pseudonyms is outside of the protocol and is generally assumed to be private to the person. However, this privacy is increasingly diminished, due to KYC (Know Your Customer) requirements of bitcoin traders and use of deanonymizing techniques (graph analytics - example later).



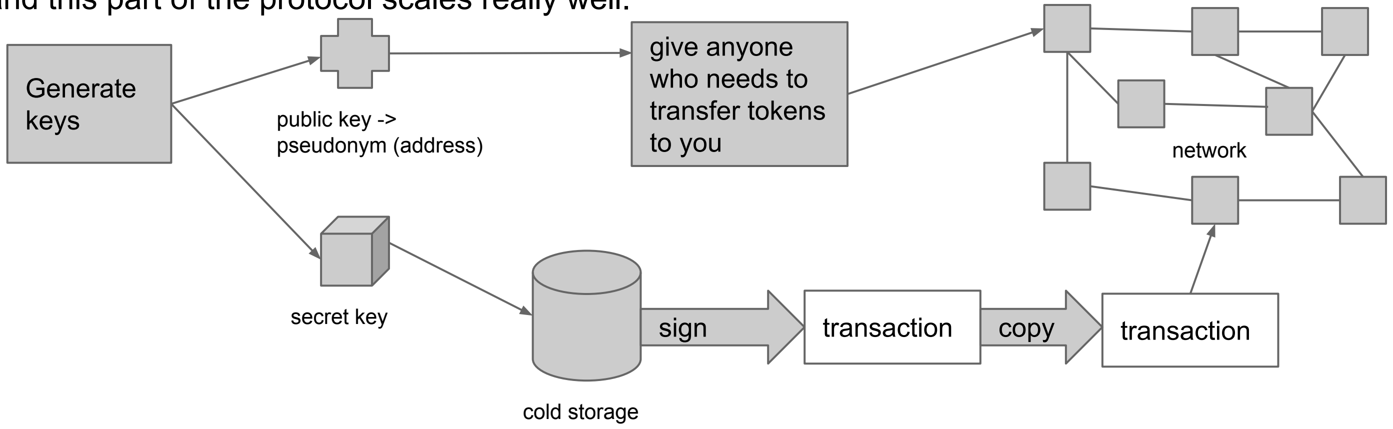
State of the system essentially records how many tokens each pseudonym has. By transferring tokens between pseudonyms in the system’s state, users (persons) are transferring value over the network.

Therefore, in order to receive or transfer tokens of value, one does not need to reveal their identity. They only need to prove that they control the relevant pseudonym. This is accomplished by the means of digital signature.

Creating a state transition (transaction)

A state transition (also called transaction) can be created outside of the network, even on a computer not connected to the Internet. It needs to be well-formed and correctly digitally signed. Since signing requires access to the private (secret) key, very often transactions from important accounts are actually created on a computer disconnected from any networks, and then carried to the network node to be relayed. Such arrangement is referred to as “cold storage”, as opposed to “hot storage”, where private key is stored on the computer which is on-line, and potentially vulnerable to theft.

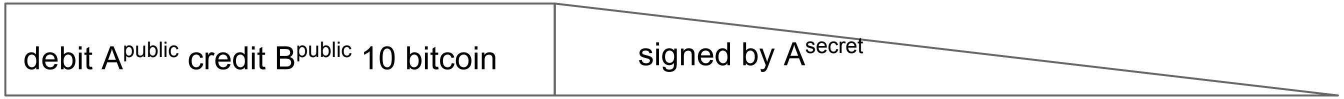
The fact that transactions can be created off-line means that any number of transactions can be created concurrently, and this part of the protocol scales really well.


Generate keys
public key -> pseudonym (address)
give anyone who needs to transfer tokens to you
network
secret key
cold storage
sign
transaction
copy
transaction

Simple transactions and double spending

It is crucial for the value transfer system to ensure that one cannot spend tokens he/she does not own. In Bitcoin terminology this is called “double spending”, which sounds a bit awkward. This little awkwardness is related to a specific way the Bitcoin protocol (but not necessarily other protocols) describes transactions.

Given two counterparties, A and B, each with a pair of cryptographic keys (A^{secret};A^{public}), (B^{secret};B^{public}), one of the simplest and most intuitive ways of describing a transfer of tokens from A to B is as follows:



Now any third party, given this transaction, can verify the signature and include such transaction into the state, provided that A had 10 or more bitcoins prior to that transaction.

After this, counterparty B, even if it did not have any bitcoins before this transaction, can now transfer some tokens (within the limit of 10 bitcoin) to counterparty C, by constructing a similar transaction.

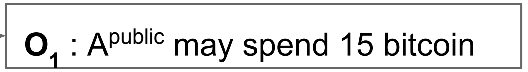
Any third party, seeing transaction B->C, will now need to find all transactions debiting and crediting B (in our case there is only one, A->B), in order to establish whether B had enough funds for the transfer. In a real-life implementation, one will probably keep an associative array (dictionary), storing number of bitcoins for each counterparty:

$$\begin{tikzcd} A^{public} \arrow[r] & \boxed{30} & B^{public} \arrow[r] & \boxed{10} & C^{public} \arrow[r] & \boxed{0} \end{tikzcd}$$

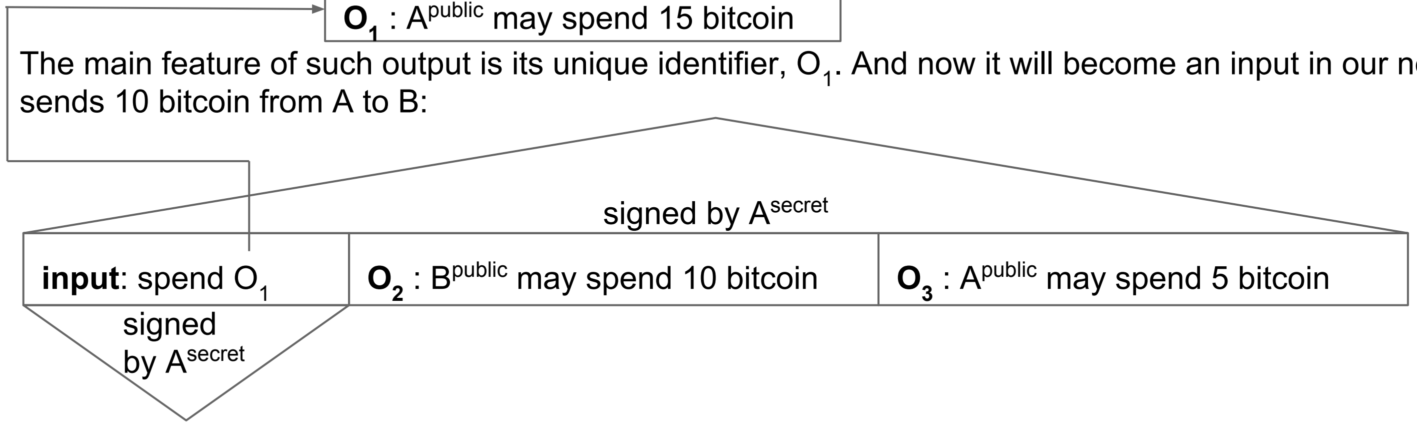
such array would help avoid searching all transactions related to B every time it needs to transfer funds.

Simple transactions and double spending

This is not how transactions are constructed in Bitcoin protocol! Transactions are dealing not with with counterparties directly, but with the “outputs” of previously confirmed transactions (they become “inputs” correspondingly). Our simple example will then change as follows. Counterparty A has previously received 15 bitcoin from counterparty X in a single transaction. That transaction contained an output of this form:



The main feature of such output is its unique identifier, O_1. And now it will become an input in our new transaction that sends 10 bitcoin from A to B:

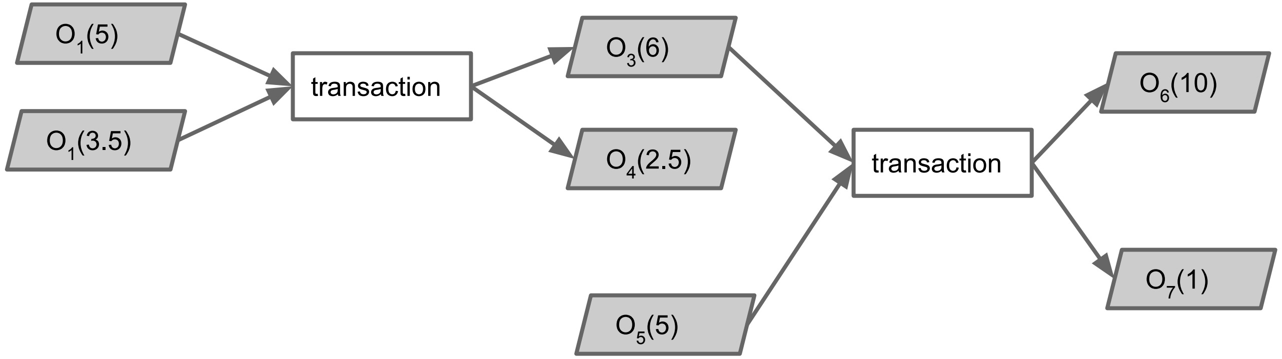

signed by A^{secret}
input: spend O_1
$\mathbf{O_2}$: B^{public} may spend 10 bitcoin
$\mathbf{O_3}$: A^{public} may spend 5 bitcoin
signed by A^{secret}

Two digital signatures are appended to this transaction: one is required to authorize the “access” to the previous output O_1, and the second one ensures that the transaction can not be “re-routed” after creation, by changing outputs O_2 and O_3. Note that is this transactions, the counterparty A sends to itself a “change” of 5 bitcoin. It is a consequence of the rule that all the outputs of any transactions have to be spent entirely, or not spent at all. This rule greatly simplifies the processing. Spending the same output twice is forbidden by the protocol, and therefore the term “double spending”.

More complex transactions

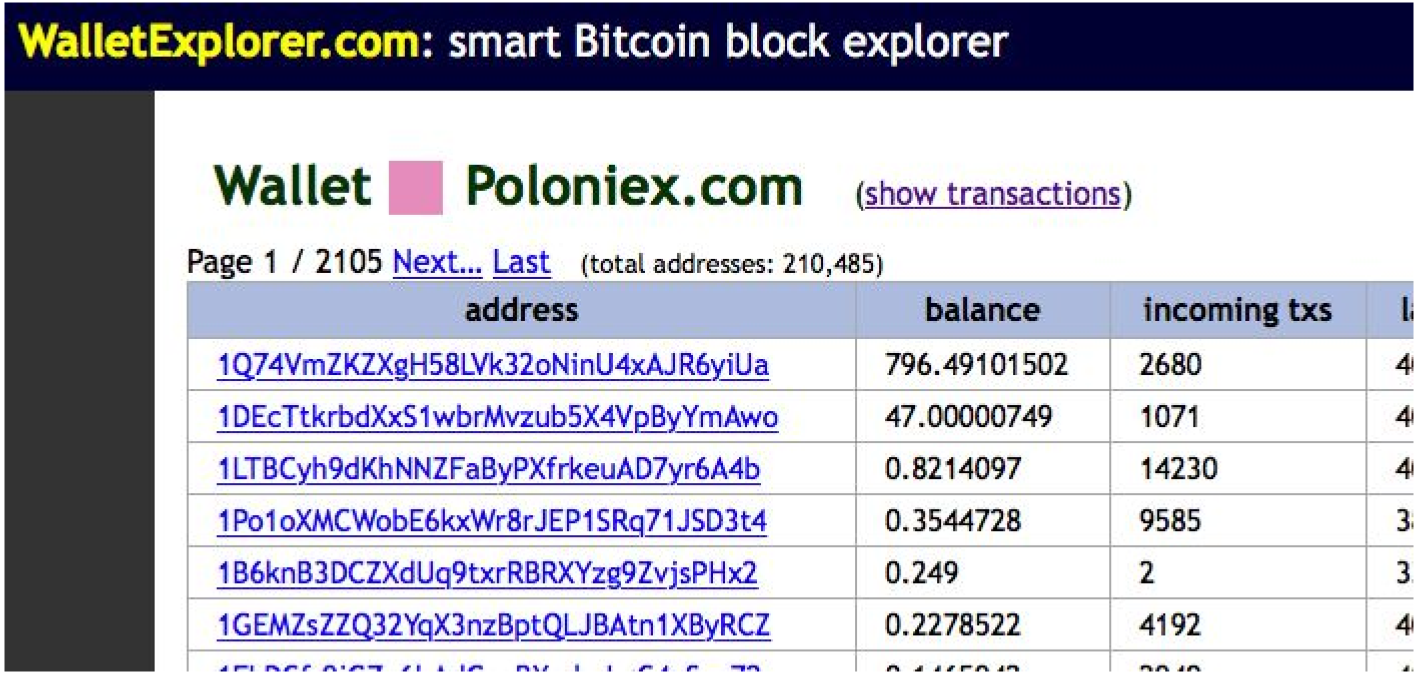
Now, in order to validate such transactions, one needs not find the balances of counterparties, but instead determine whether the previous output (O_1) has been previously spent in another transaction.

The rationale behind doing transactions based on the Unspent Transactions Outputs (UTXO), was to allow safe re-sending of the transaction (if the previous one is believed to get lost or expired), without fear of inadvertently sending the funds twice. There are simple alternative solutions to this, not requiring UTXO. Support for scripting in the transactions (will not be covered here), which was a bit of afterthought, was probably the main reason for choosing UTXO.



UTXO approach has a consequence that most transactions, even the most basic ones (where there is single payer and single payee), there will be more than one input (for gathering/consolidating enough tokens from multiple outputs) and more than one output (for collecting the “change”)

Aside - graph analytics example



WalletExplorer.com: smart Bitcoin block explorer

Wallet Poloniex.com (show transactions)

Page 1 / 2105 Next... Last (total addresses: 210,485)

address	balance	incoming txs	l
1Q74VmZKZXgH58LVk32oNinU4xAJR6yiUa	796.49101502	2680	4
1DEcTtkrbdXxS1wbrMvzub5X4VpByYmAwo	47.00000749	1071	4
1LTBCyh9dKhNNZFaByPXfrkeuAD7yr6A4b	0.8214097	14230	4
1Po1oXMCWobE6kxWr8rJEP1SRq71JSD3t4	0.3544728	9585	3
1B6knB3DCZXdUq9txrRBRXYzg9ZvjsPHx2	0.249	2	3
1GEMZsZZQ32YqX3nzBptQLJBAtn1XByRCZ	0.2278522	4192	4

This website crawls through the graph of transactions trying to figure out who addresses belong to.

Transaction validity

Transactions cannot be in general validated on their own. Some transactions could be mutually exclusive, or only valid in certain order:

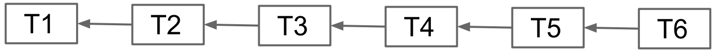
Initial state: {A: 10, B: 0, C: 5}

debit A 10, credit B 10 | debit A 10, credit C 10 | only one can be valid, but not both

debit A 10, credit B 10 | debit B 5, credit C 5 | both valid in that order

debit B 5, credit C 5 | debit A 10, credit B 10 | only second one valid in that order

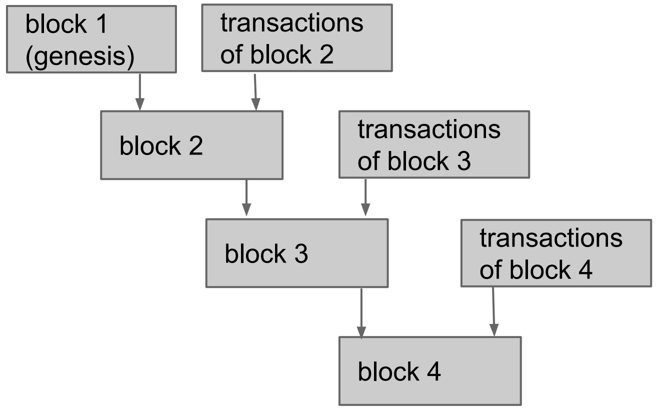
Therefore, to maintain the ledger, it is necessary to arrange transactions in a sequence:



In order for such sequence to be tamper-resistant, each transaction would need to authenticate the previous one, for example, via including a cryptographic hash of the previous transaction. But that would invalidate the premise that many transactions can be created concurrently and the scalability of the system would be impractical.

From transactions to blocks

Since transactions cannot be linked to one another in a chain, they need to be “packaged” into another data structure (container), which enforces their sequence. Therefore, transactions are “packaged” into blocks. Each blocks determines the sequence of transactions it contains and is also connected to the previous block, effectively forming a degenerate Merkle tree:

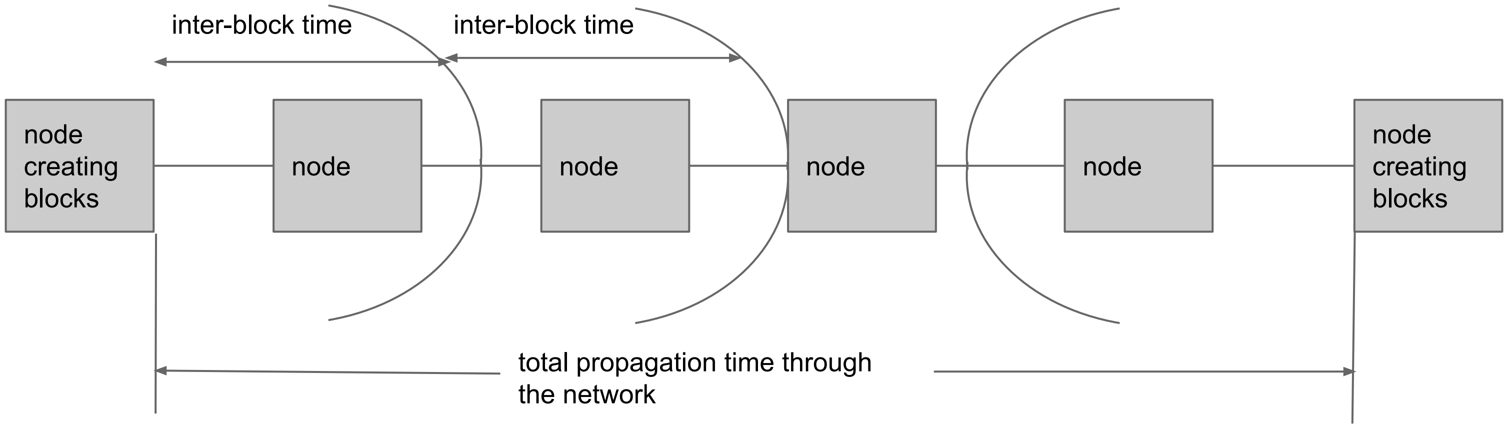


Here, block2 contains the hash of the concatenation of block1’s hash and hash of block1’s transactions, and so on. In fact, it is more like a chain of blocks than a tree. Hence the name “Block-chain”. More specifically, it is a hash-chain of blocks. By the properties of the Merkle tree, the hash of the latest block authenticates the entire block-chain.

Propagation of blocks and inter-block time

After a block is created (with transactions “packaged” inside), it needs to be propagated across the entire network, so that the state transitions of the replicated state machine are performed everywhere. Recall that before propagating any block, a node must verify its compliance to the protocol and also verify each transaction. This takes some time, and as a result (and also given the network latencies), a block can take a few minutes to propagate globally.

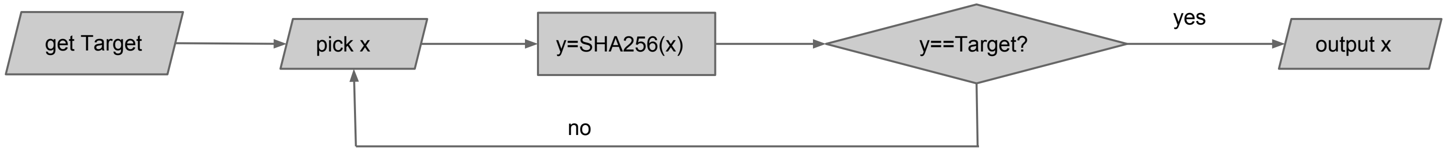
Blocks can be created by anyone - in a truly decentralised system this is an obvious, but not the only choice. Now, if the blocks are created at a faster rate than they can propagate through the network, there will never be a moment in time when a majority of the nodes have seen the same set of blocks and can actually agree on which ones should represent the common chain.



Therefore, a somewhat arbitrary choice of 10 minutes of average inter-block time has been made.

Creation of blocks

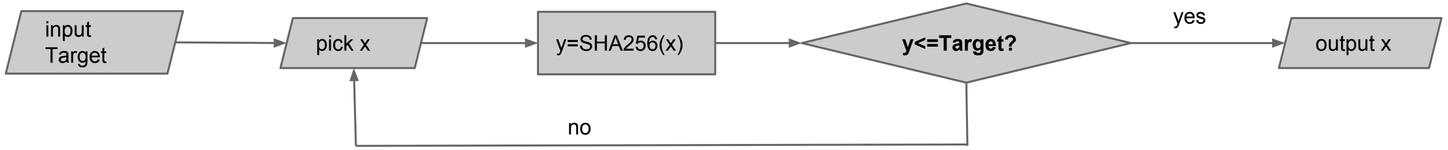
How to ensure that blocks are created every 10 minutes on average, and neither more nor less frequently? In Byzantine settings, one cannot objectively trust any given participants to determine (by sending a message) when 10 minutes have passed (although it is possible to do it subjectively). Instead, Bitcoin design chooses a physical process that must be performed to create a block, and that process should take 10 minutes on average, and cannot be made substantially faster. That chosen physical process is the computation of a partial inverse of a cryptographic hash function. What is it?



If a cryptographic hash function is preimage resistant, the best known way to find an input for a target output is to keep trying (in some order or at random) various inputs and compare the output with the target. The design of preimage resistance aims at making such trial and error process impractically long. In fact, under the random oracle assumption, where all possible outputs are equally likely for every such trial, the probability of “hitting” the target output is 1/N, where N - number of possible outputs. For the function SHA2 (SHA256), used in Bitcoin, with 256-bit output, $N=2^{256}$, and probability of success in one trial is $1/2^{256}$. The probability of success after k attempts is then $p_k=1-(1-1/2^{256})^k$. Solving for k: $k = \ln(1-p_k)/\ln(1-1/2^{256})$. We know that for the extremely small x, $\ln(1+x)$ is well approximated by just x. Therefore: $k = -2^{256}\ln(1-p_k)$. For example, to invert SHA256 with probability of 50%, one would require on average this number of attempts: 160’521’920’371’995’283’198’148’406’451’588’100’866’918’344’094’926’382’540’636’297’893’588’661’078’654

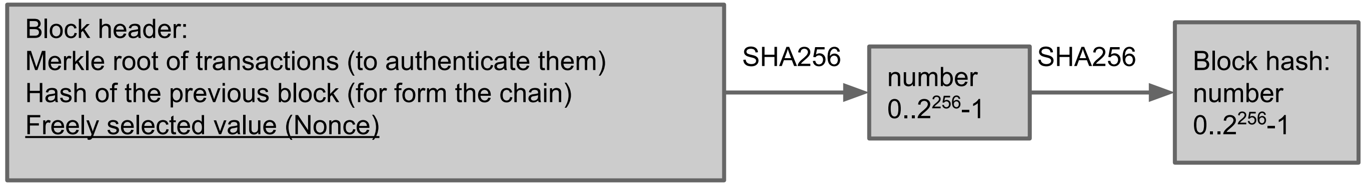
Proof of work

What if we relax the success criteria in that brute force algorithm, and replace “y==Target?” with “y<=Target”



The probability of success in one trial would be Target/2^{256} instead of 1/2^{256}, and by varying the value of “Target”, it is possible to make such partial inverse as difficult or as easy as required. The actual value of “Target” is chosen by the Bitcoin protocol adaptively, based on the historical performance of block creators in solving the partial inverse. If blocks get created more frequently than 10 minutes on average, the value of Target decreases, making the partial inverse more difficult. If, on the other hand, blocks get created less frequently than required, Target increases. This mechanism allows adapting to changing capabilities of block creators.

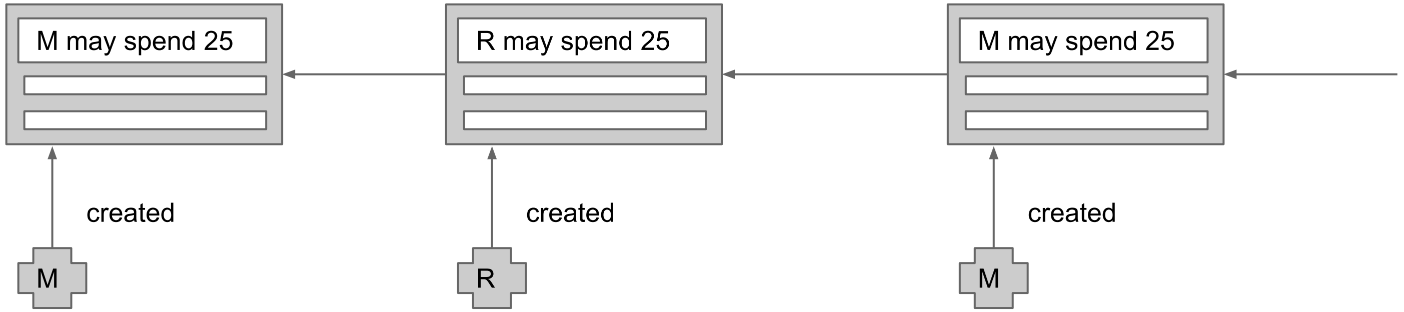
Creator of a block, in order to prove to everyone that the required “physical process” has been performed, finds (via trial and error just described) the value of “Nonce” (in the picture below), such that the hash of the hash of the block header is less or equal to the current “Target”. Since the found “Nonce” is embedded into the block header, such proof is very easy for everyone to verify, but should be very difficult (on average 10 mins) to produce.


Block header:
Merkle root of transactions (to authenticate them)
Hash of the previous block (for form the chain)
Freely selected value (Nonce)
SHA256
number 0..2^{256}-1
SHA256
Block hash: number 0..2^{256}-1

Mining and emission

Proof of work only ensures that blocks are not created too frequently, but it does nothing to ensure that they are created frequently enough. Since all network participants are supposed to be equal and under no obligation to create blocks, they need to be incentivised to do so. This incentivisation is done via emission of bitcoin tokens to the block creators. It is akin to printing money, or using a gold analogy, mining new gold out of the ground. That is why the process of creating blocks is called mining, and the participants involved are called miners.

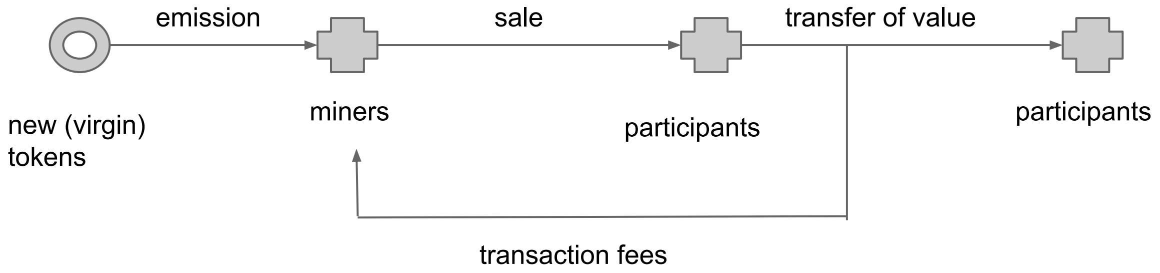
Technically, this happens as follows: the first transaction in the block is always “special” - it is called “coinbase” transaction and it emits certain amount of bitcoin tokens (currently 25) out of nowhere and assigns it to the specified participant (naturally, to the miner who created the block will put his/her address there). Effectively this transaction compensates the miner for the work spent on forming the block.



The coinbase transaction is the only type of transaction where the total value (in bitcoin tokens) of outputs exceeds the total value of inputs (0).

Transaction fees

For all other transactions, the total value of inputs is normally slightly more than total value of outputs. The difference is called transaction fee and is assigned by the protocol to the miner of the block where this transaction was packaged). Apart of from incentivising miners to include transactions into the blocks (rather than discarding them and simply earning the mining reward), transaction fees help prevent DDOS (Distributed Denial of Service) attacks by making them more costly.

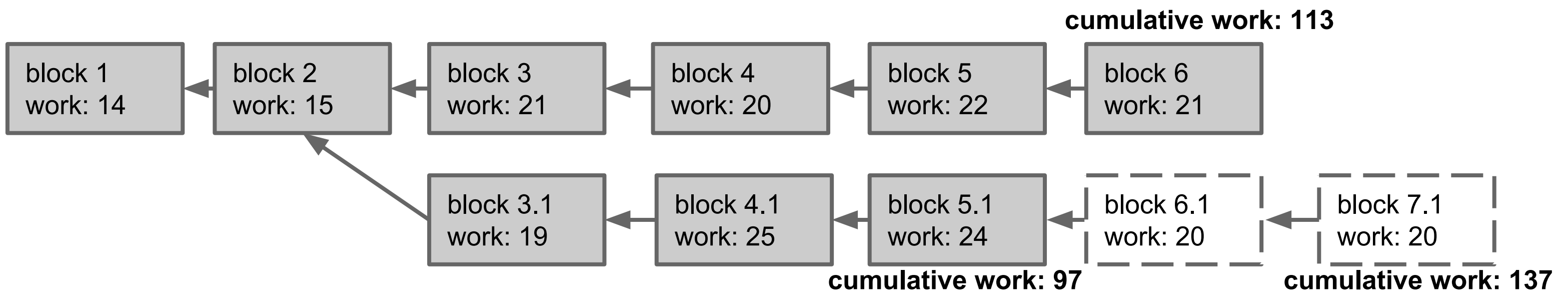


The mining reward is designed to decrease over time (it halves every 210’000 blocks or approximately 4 years), eventually reaching 0 (this is projected to happen around year 2030). The idea that by that time mining will be incentivised only by transaction fees. This decline of mining reward is there to limit the total supply of bitcoin tokens to some predetermined value (21 million).

Mining competition and agreement

When creating new blocks is well compensated, there are naturally lots of participants willing to do it. Due to propagation delays, or an attempt to revert some transactions, or out of other intent, alternative versions of the ledger will arise. In such situation, the protocol must provide the rule that will allow all the participant to eventually agree on one of the version. The most important property of such rule is that all honest participants (those following the protocol) must deterministically decide on the same version.

One way to do it is to always select the version of the blockchain with the greatest cumulative work (in terms of Proof of Work) in it.



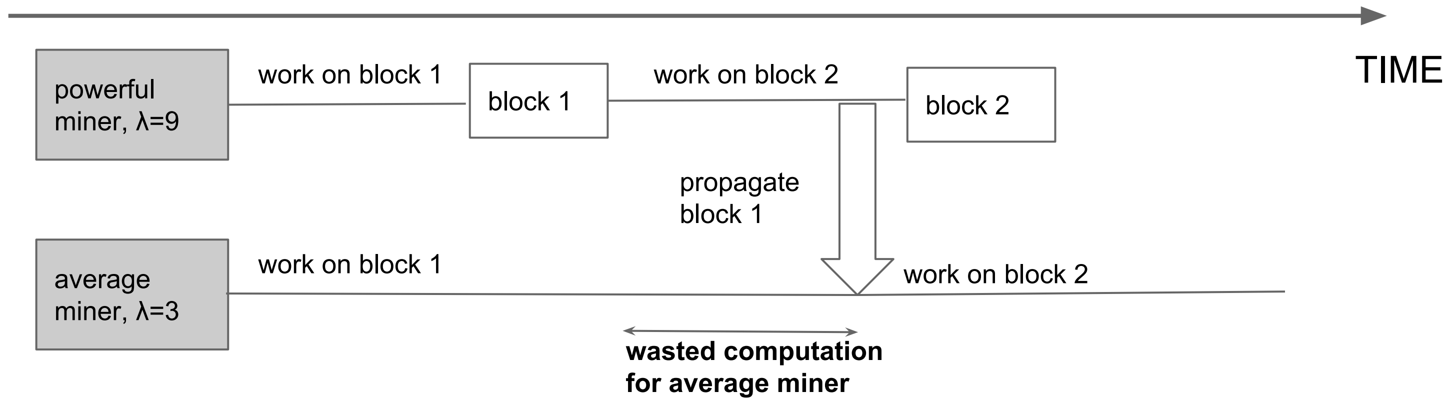
Having received the blocked in the example above, a node must accept the block 6 and all its predecessors and the ledger, because the cumulative work of this version is **113**, which is greater than the cumulative work of the version of the block 5.1: **97**. However, if it will subsequently receive blocks 6.1 and 7.1 with the proof of work 20 each, the node will have to switch to the version of the block 7.1, and effectively revert all the transactions in the blocks 3, 4, 5, and 6. This sort of switch is called chain reorganisation

Mining is a Poisson process

The algorithm used to produce Proof of Work, turns out to be very close to a homogeneous Poisson process. In such a process, the probability of an event (in our case it is finding the right Nonce) is independent of the time of any previous events. This probability only depends on the parameter λ, also called the intensity of the process, and can be interpreted as the expected number of event per unit of time.

The Poisson property means that two miners having the same intensity λ (proportional to the miner's computational power, also called "hashing power"), and trying to compete in producing a Proof of Work, have the same probability of winning (producing the Proof of Work first), regardless of when they have started.

If the Proof of Work process did not have such a property, and always took time proportional to the computational power, it would allow a very powerful miner to gain a superlinear advantage. When it manages to construct a Proof of Work and create a valid block, it does not propagate it immediately. Instead, it uses it to start work on the next block, while the others are still busy competing with the first block. Poisson process eliminates the incentive to do so.



Mining - theory vs practice - selfish mining

It turns out that a large miner, owning ⅓ (or perhaps less) of computing power (in other words, having probability of 0.333… of mining a block), the most profitable strategy becomes to withhold mined blocks temporarily and only announce them when someone else finds a block. It is described in the paper “Majority is not Enough: Bitcoin Mining is Vulnerable” by I. Eyal and E.G. Sirer.

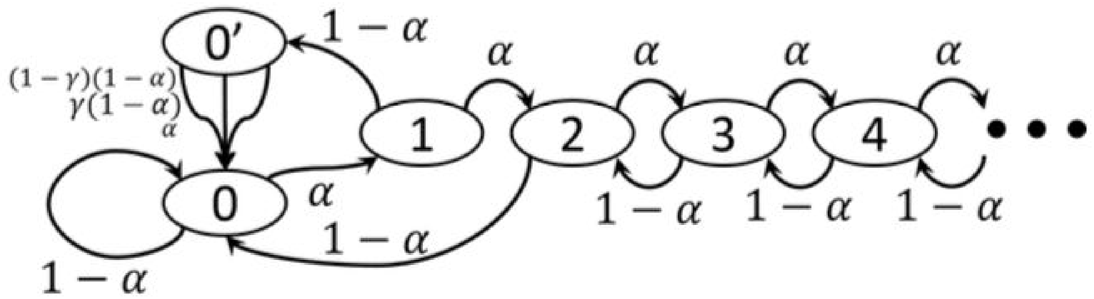
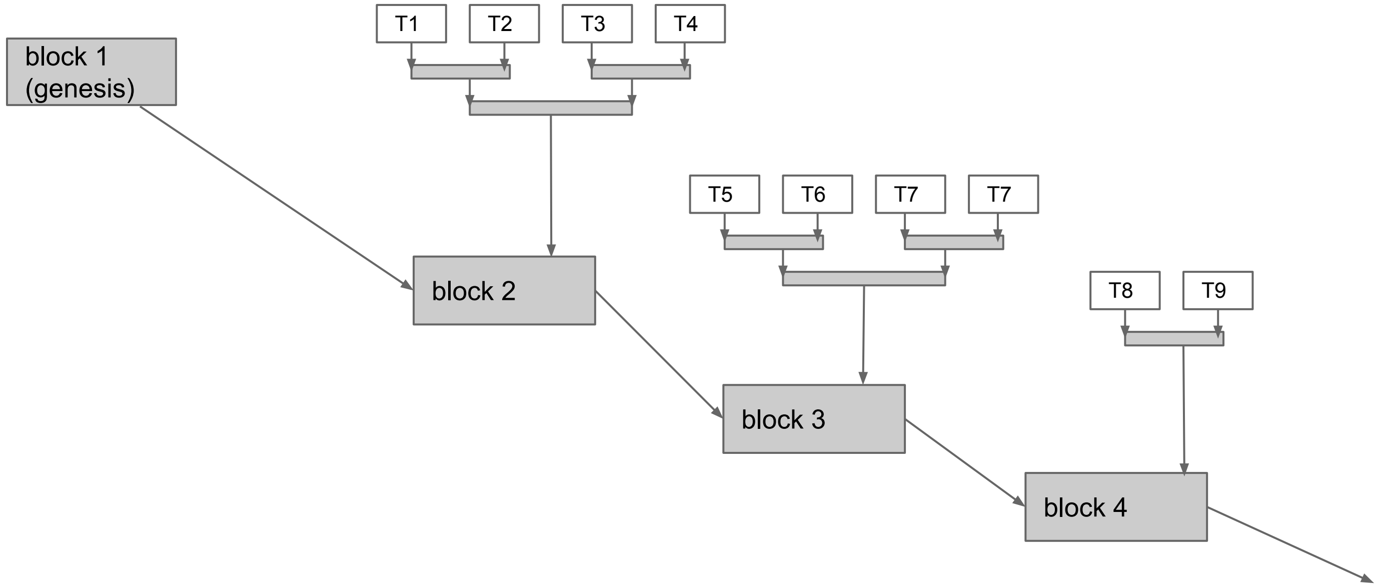


Fig. 1: State machine with transition frequencies.

The same two researchers later found that large mining pools face so-called “Prisoner’s Dilemma” (mining pools are not defined here yet).

Light clients and trees inside the chain

Inside each block, transactions are not just simply arranged in a sequence. There is a Merkle sub-tree of transactions inside every block. It allows producing a short proof that certain transaction is indeed contained in the block. Fully expanded degenerate Merkle tree would look like this (number of transactions in a block can vary and can even be zero):



Main motivation is to support light clients that do not store the full ledger, but only require a proof of specific payment.

Digital bearer asset(s). Bitcoin Script?

This became the main ‘use case’ of bitcoin. Naturally, the question arose about other such assets, beyond bitcoin (alternatives). Here is a typical Bitcoin script (used in 99.9% of bitcoin transactions):

supplied by entity that creates the transaction →

```
<sig>           # push transaction’s signature on the stack
<pubKey>        # push public key of signatory on the stack
--------------
```

specified by entity that created the output that is being spent in this transaction →

```
OP_DUP          # create copy of the top (pubKey)
OP_HASH160      # apply SHA-256|RIPEMD-160 to get address
<pubKeyHash?>   # push expected address (recipient of prev.)
OP_EQUALVERIFY # recipient of prev. trans == signatory?
OP_CHECKSIG     # <sig> is a transaction’s signature?
```

Bitcoin Script - Capabilities

Bitcoin script is by design quite limited. There are no loops, jumps or recursion, but there are `OP_IF`, `OP_ELSE` and `OP_ENDIF`. There are arithmetic operations and cryptographic subroutines. There is a very powerful `OP_CHECKMULTISIG` (transaction have to be signed by n-out of-m specified entities) - allows escrows and is required for micropayment channels. There is `OP_CHECKLOCKTIMEVERIFY` (make transaction invalid until certain time) - required for micropayment channels.

One can also place small amount of arbitrary data into the transaction, after `OP_RETURN` instruction, which makes output un-spendable, because it always fails when reached.

Bitcoin Script - Limitations on expressiveness

Because there are no loops, jumps or recursion, the execution time of a script has a trivial upper bound proportional to the size of the script. The rationale behind these limitations is to prevent miners from sending an ‘execution bomb’ transaction for the others, thereby getting a head-start on mining the next block.

Also, if `OP_RETURN` is present in the script, it will always be executed, and transaction will fail. So everything after it (extra data) can be ignored.

Bitcoin Script - Limitations on context

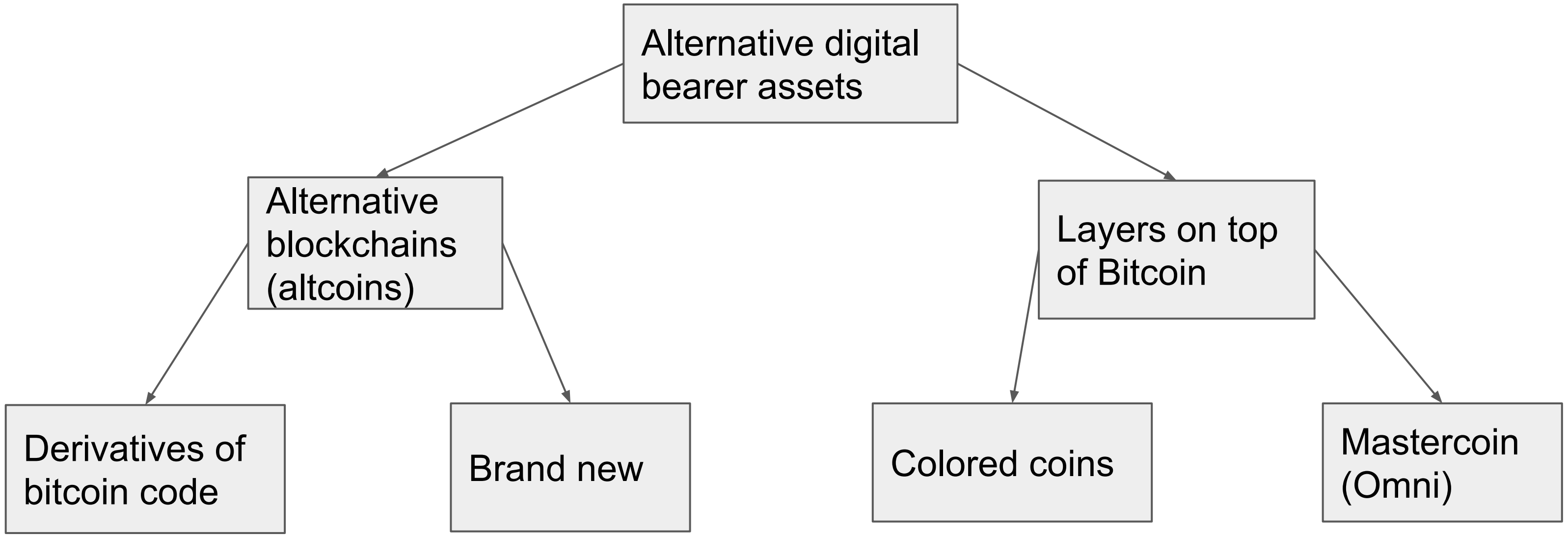
Scripts are running in a limited context, consisting of:

- inputs encoded into the script itself (`OP_PUSHDATA` instructions)
- indirectly - hash of the transaction (to check signature)
- indirectly - timestamps or block number (via `OP_CHECKLOCKTIMEVERIFY`)

Theoretically, some amount of context can be carried between transactions (via `OP_PUSHDATA` inputs), but such context is quite limited in size, needs to be repeated in each transaction, and requires some extra-protocol rules to verify correctness.

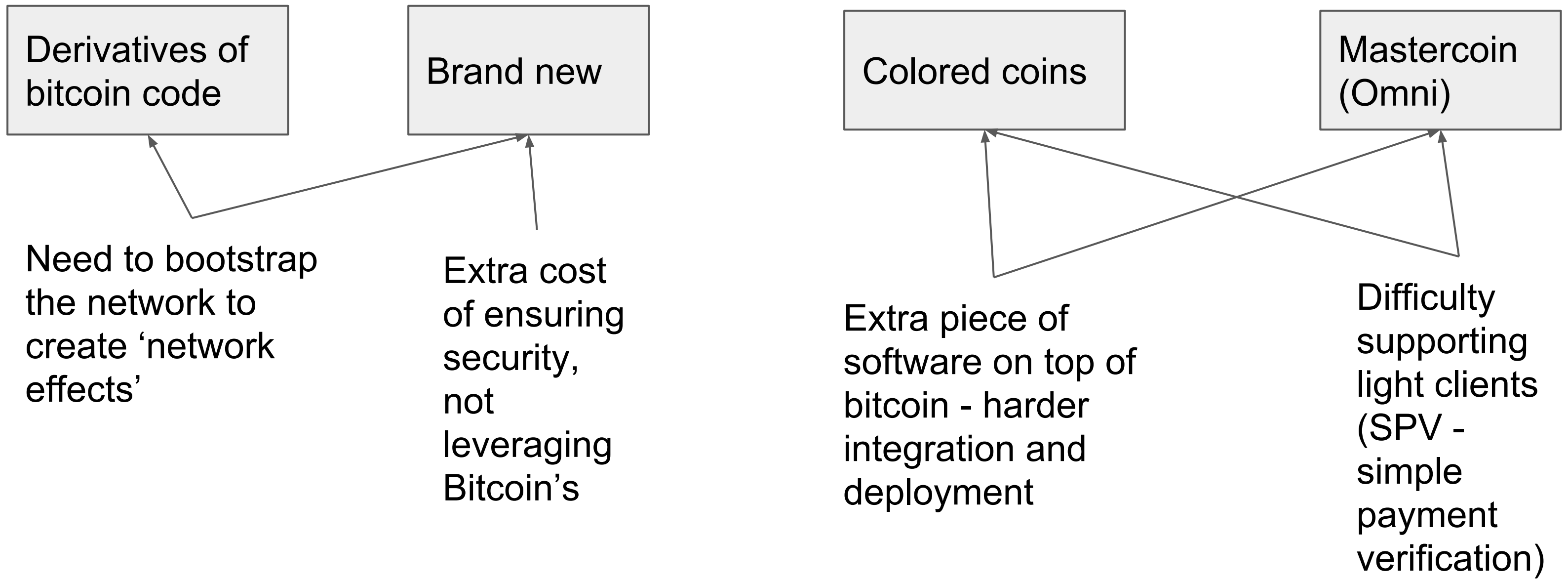
Alternatives?

After realisation that it is not easy (or impossible) to implement generic assets in Bitcoin Script, there were some notable alternative attempts:


Alternative digital bearer assets
Alternative blockchains (altcoins)
Layers on top of Bitcoin
Derivatives of bitcoin code
Brand new
Colored coins
Mastercoin (Omni)

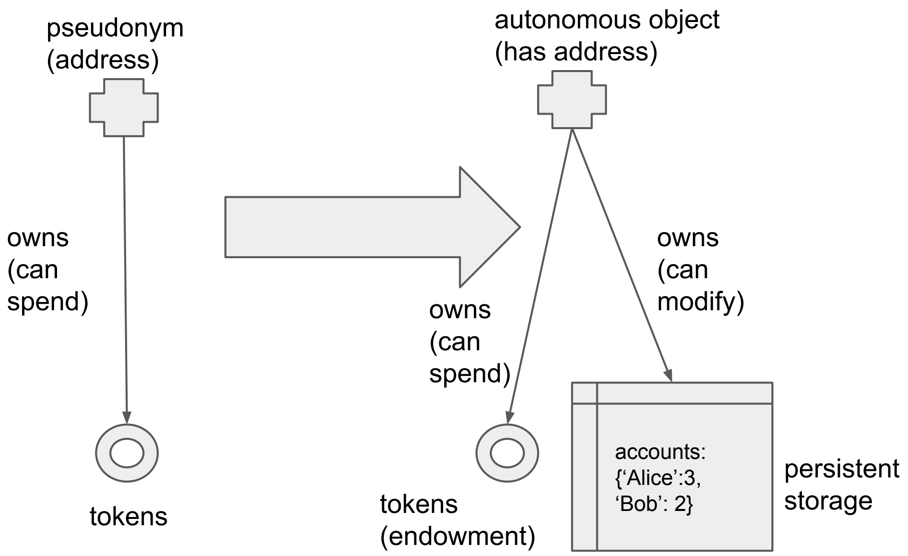
Challenges for the alternatives

All alternatives faced serious challenges:


Derivatives of bitcoin code
Brand new
Colored coins
Mastercoin (Omni)
Need to bootstrap the network to create 'network effects'
Extra cost of ensuring security, not leveraging Bitcoin's
Extra piece of software on top of bitcoin - harder integration and deployment
Difficulty supporting light clients (SPV - simple payment verification)

Generalisation - Step 1. More context

First step in generalising the one-token system to support many assets is to create context - registry of asset holders' accounts in persistent storage

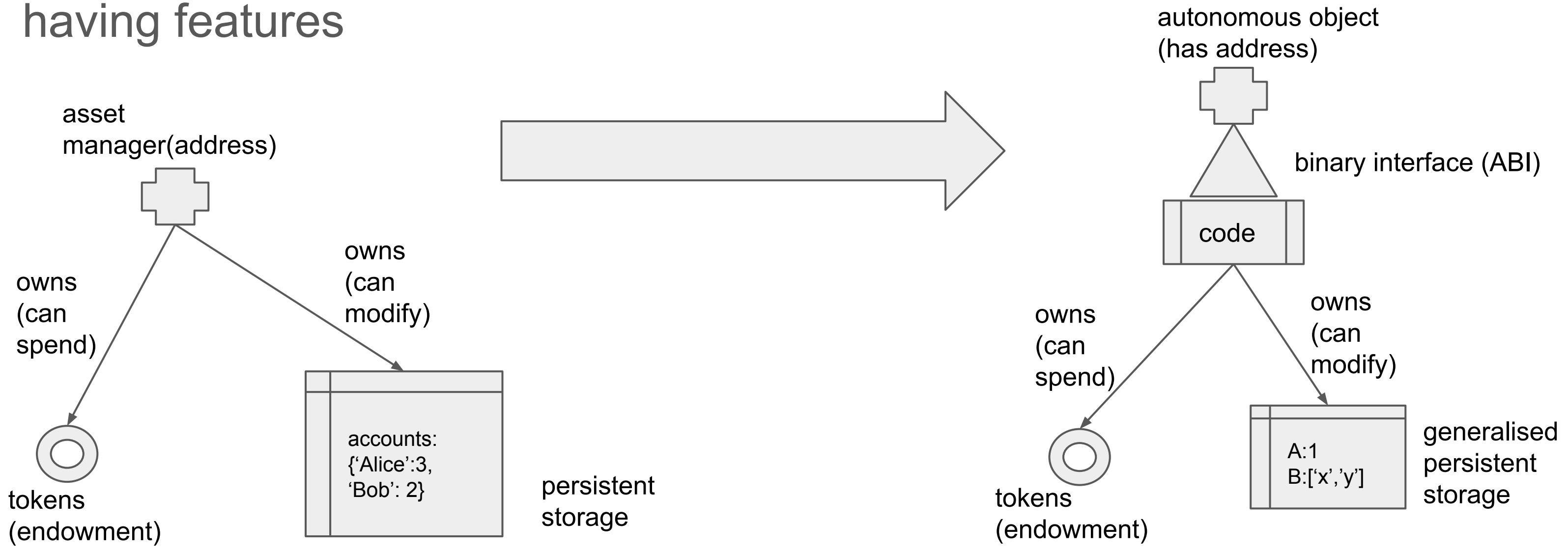


Note 1: Tokens in endowments are not strictly speaking required here, but let's keep them

Note 2: Scripts modifying the persistent storage can still be limited, by adding special new instructions (**features**)

Generalisation - Step 2. More expressiveness

The concern about **features** is that one has to keep adding them. For all possible use-cases. A universally expressive language allows supporting use-cases without having features



What to do about ‘execution bomb’?

In a universally expressive language (Turing-complete), many programs do not have simple (or even computable) upper bounds on the execution time (related to Halting problem, which is undecidable in general).

In practical settings, one may think of imposing a time limit on execution of each script, and render transaction invalid if the time limit is exhausted. That, however, is no good because the validity of that transaction becomes non-deterministic.

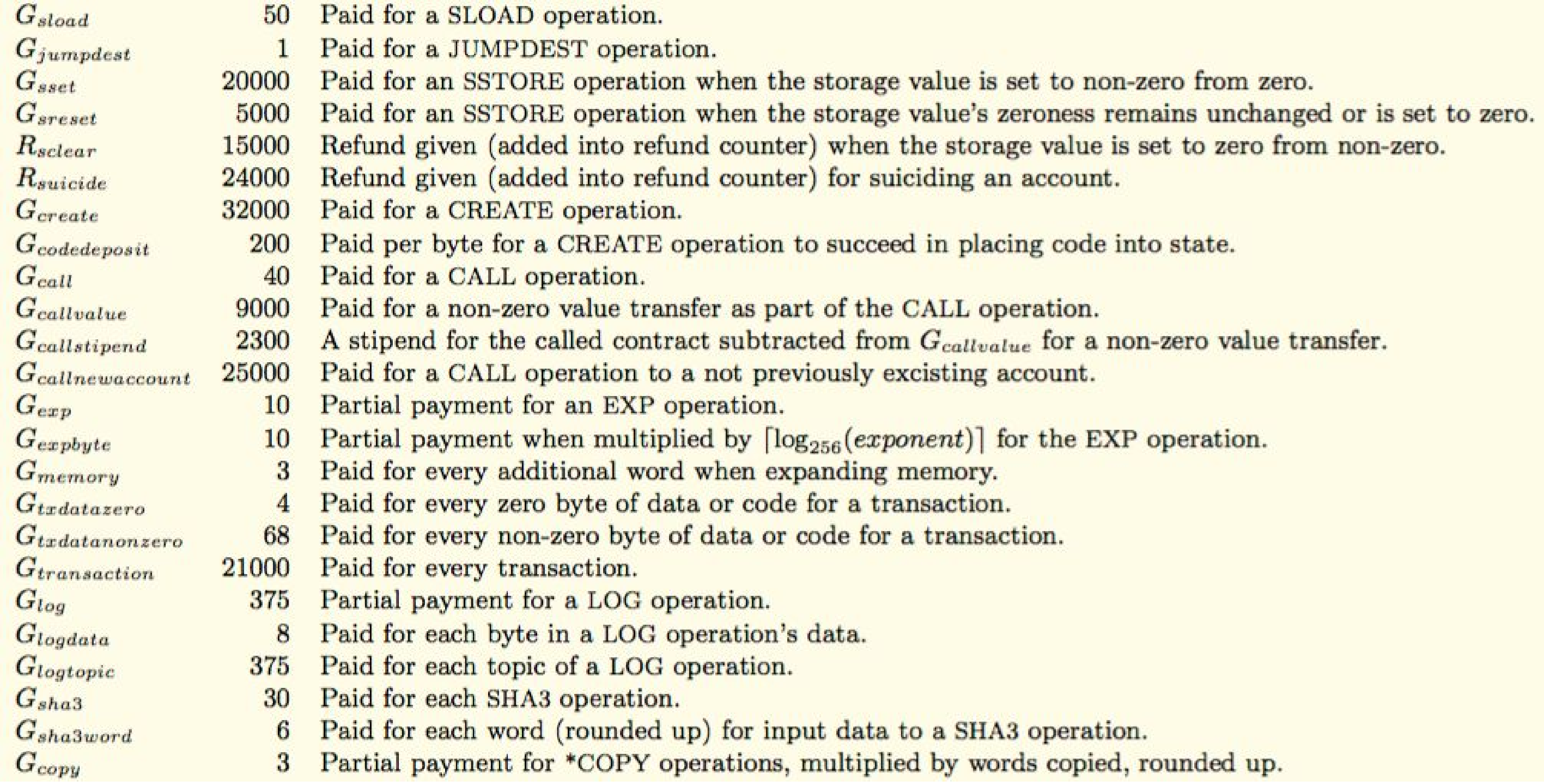
Ethereum solves this problem by introducing a fee for each executed instruction in the script.

Fee schedule

Initial fee schedule (how much is charged per instruction) was published in Ethereum yellow paper (it can easily be found via search engine).

Some modifications were made for the Homestead release (imminent at the time of writing), to make certain things relatively more expensive. Further adjustments of fee schedule are expected in the future.

There is a limit on amount of gas that can be spent in a block - therefore one should call Ethereum language quasi-Turing complete rather than Turing complete. The limit is not hard-coded and can be dynamically changes via the protocol.



G_{sload}	50	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{suicide}$	24000	Refund given (added into refund counter) for suiciding an account.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	40	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{callnewaccount}$	25000	Paid for a CALL operation to a not previously excisting account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	10	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.

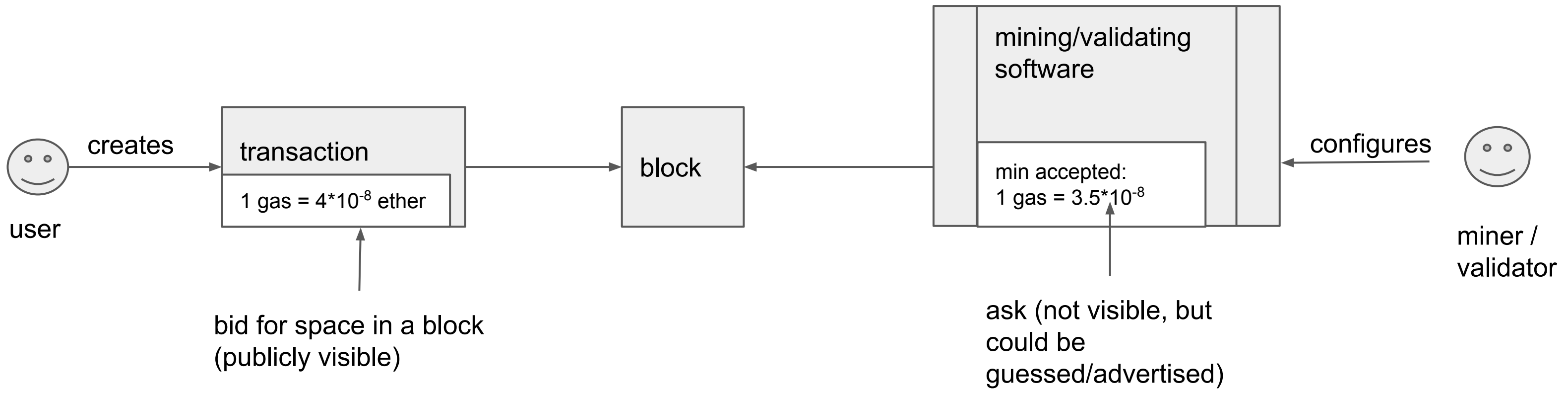
What these fees are paid in?

Although Ethereum has a built-in token, ‘ether’, the fees for executing of instructions are not denominated in units of ether. Instead, they are specified in units of ‘gas’ (there were proposals to rename it to ‘mana’), which is supposed to have free-floating exchange rate from and into ether.

The reason for such an intermediate unit is a anticipation that it would be desirable to keep the price of ‘gas’ relative to non-cryptocurrencies relatively stable, regardless of how volatile the price of ether might be (at that point it was clear that cryptocurrencies tend to have very high price volatility).

How would that free-floating rate work?

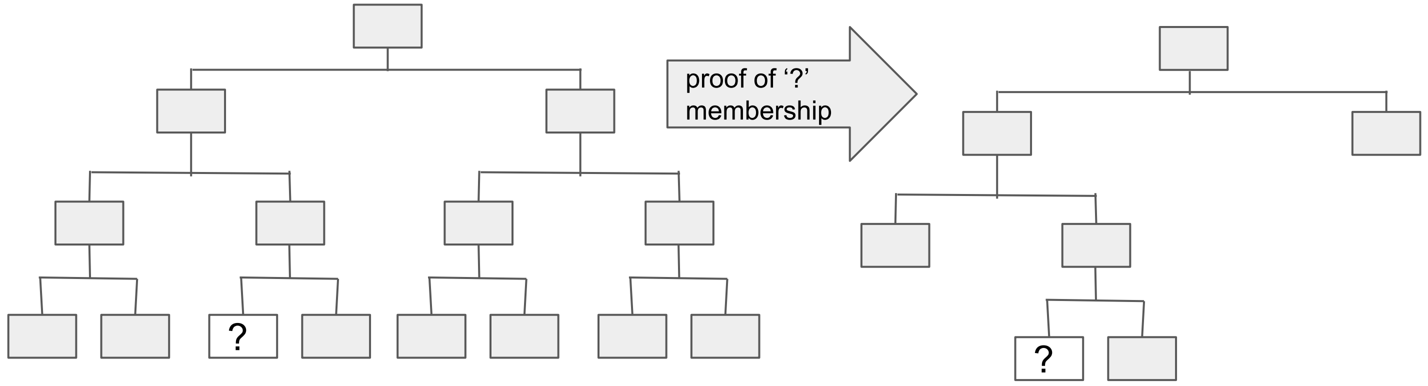
How would this ‘gas’/’ether’ rate be established? Who are the market participants and how they go about to discover the price?


creates
transaction
1 gas = 4*10^{-8} ether
block
mining/validating software
min accepted:
1 gas = 3.5*10^{-8}
configures
user
miner / validator
bid for space in a block (publicly visible)
ask (not visible, but could be guessed/advertised)

Miner/validator who gets to create a block, picks bids for space in the block.

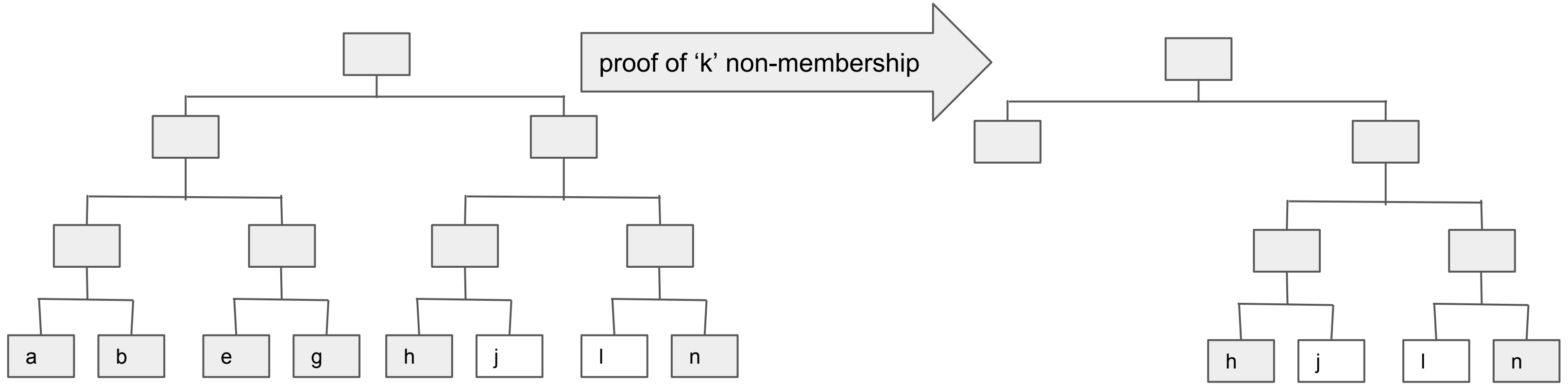
Persistent storage - data structures

Data on a blockchain has to be stored in authenticated, tamper resistant data structures. In Bitcoin, we have seen Merkle tree, representing a ‘sort of’ authenticated set. Its main feature is a short proof of membership:



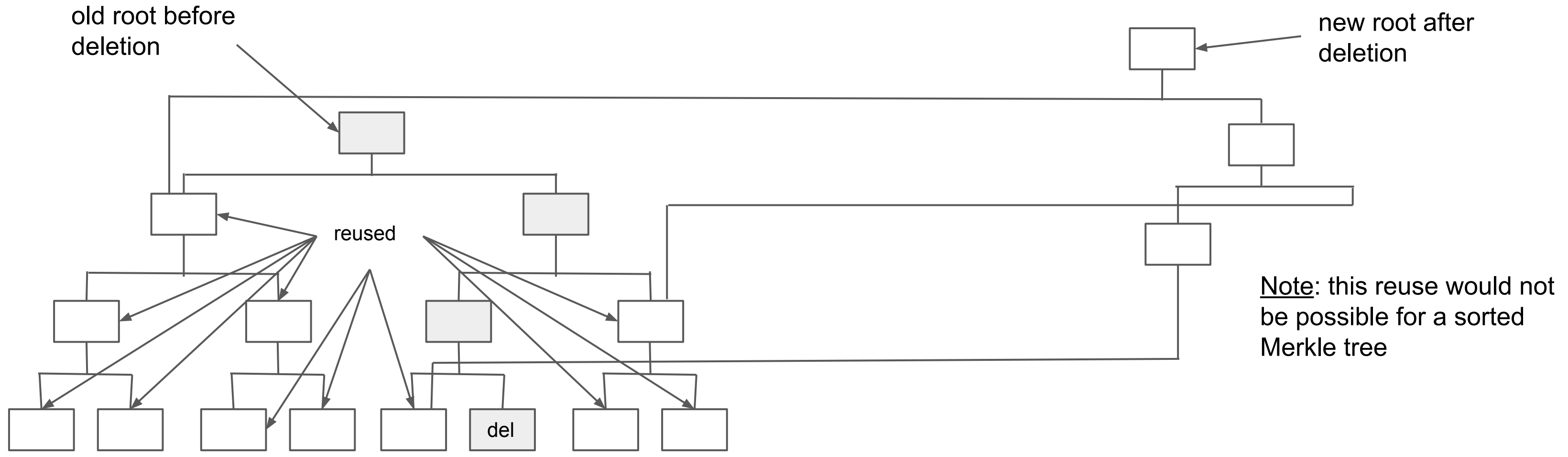
What about proof of non-membership?

How would we prove non-membership? In an ordinary Merkle tree, such proof consists of entire tree, which is no good. Another alternative would be a ‘sorted Merkle tree’:



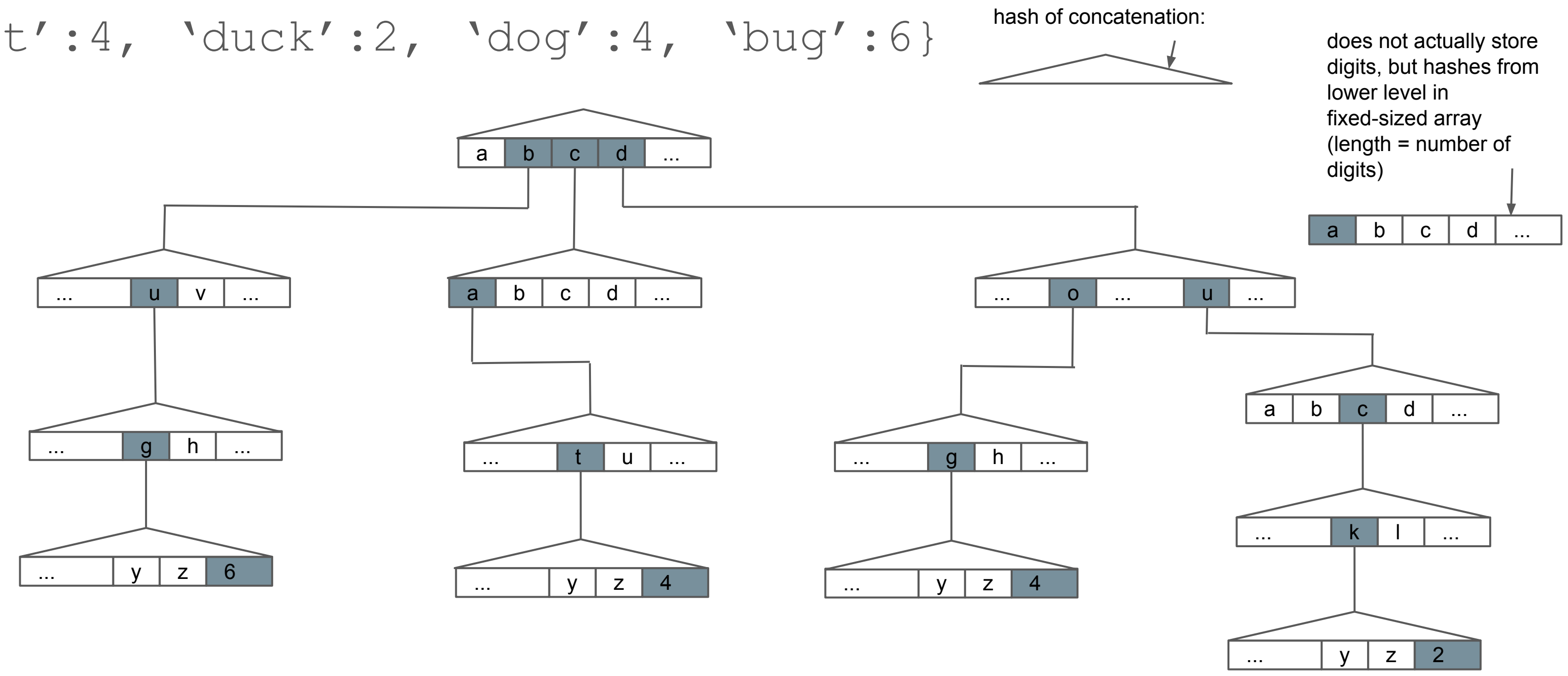
Encoding modifications

Going from ordinary to sorted Merkle tree introduces one feature (non-membership proof), but loses another - compact encoding of modifications.


old root before deletion
new root after deletion
reused
del
Note: this reuse would not be possible for a sorted Merkle tree

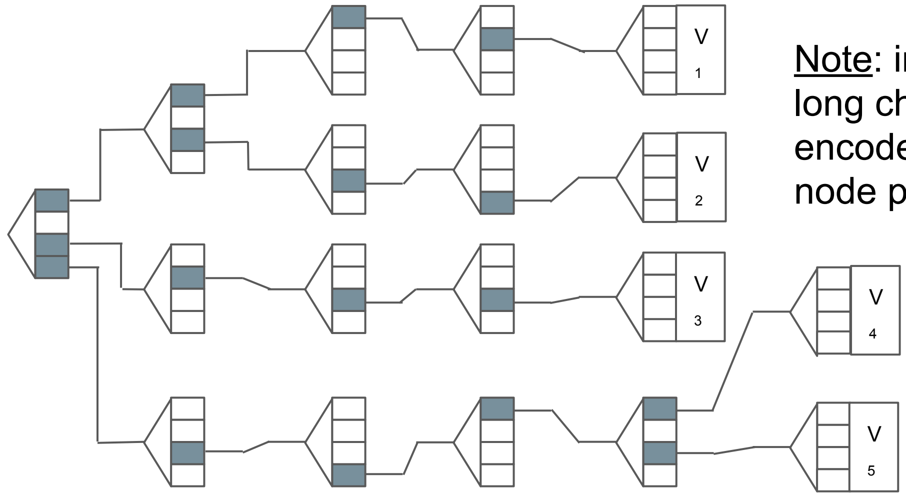
Merkle radix tree (trie)

`{'cat':4, 'duck':2, 'dog':4, 'bug':6}`



Merkle radix tree (trie)

Merkle radix tree supports compact encoding of modifications, short proofs of membership and non-memberships. However, if the keys are relatively long, the tree tends to be quite space-inefficient, because it looks like this:

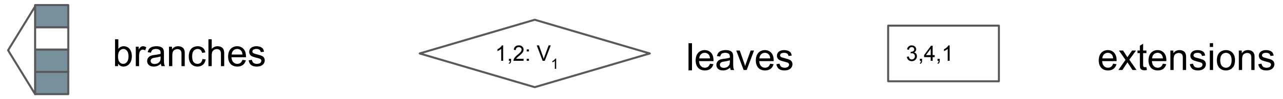


Note: inefficiency comes from the fact that long chains of digits for the keys are encoded by the chains of tree nodes, 1 node per digit

Merkle Patricia tree (trie)

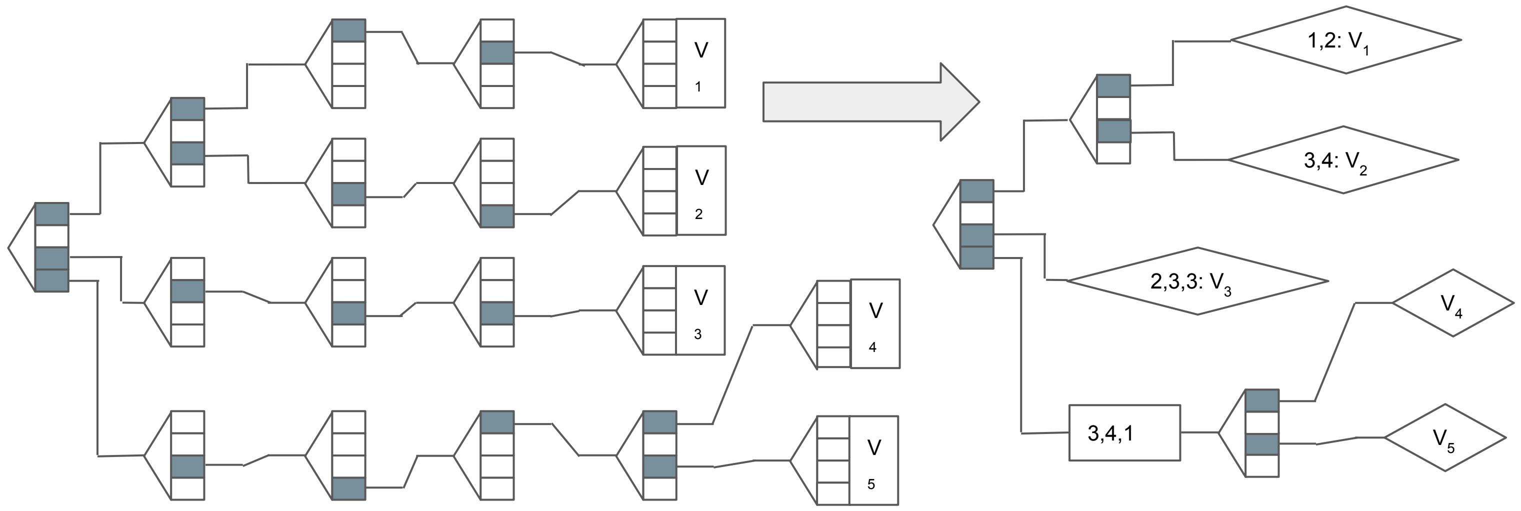
PATRICIA stands for “Practical Algorithm To Retrieve Information Coded In Alphanumeric” and was invented in 1968 independently by Donald Morrison and Gernot Gwehenberger.

Patricia tree in Ethereum’s version addresses the inefficiency of radix tree by having 3 types of nodes:



Merkle radix tree (trie)

If we transform the radix tree from earlier to Patricia tree, it will look like this:



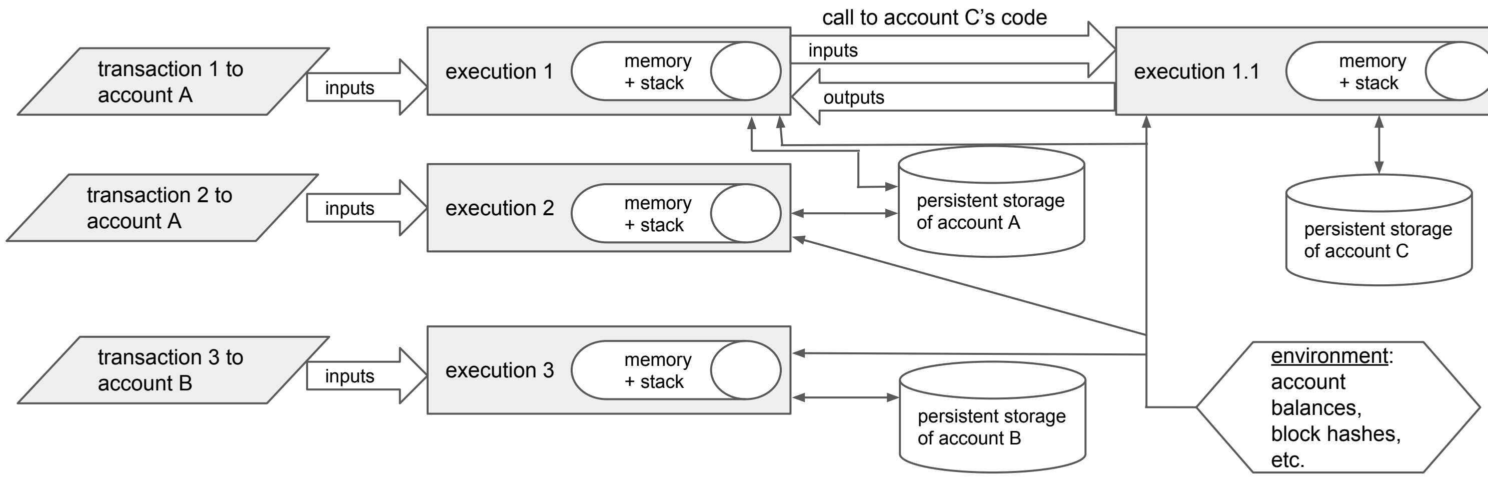
Secure Merkle Patricia tree (trie)

Merkle Patricia tree has a further problem when used as a back-bone of permanent storage for autonomous objects. If the keys in the dictionary are very close to each other lexicographically, the resulting tree will become unbalanced, and most operations will be performed in time linear rather than logarithmic to the size of the trie. This might be used as an attack vector too.

To fix that, instead of using keys themselves, Patricia trees in Ethereum use secure hashes of the keys, whereby relying on the random oracle property to minimize the chance of unbalanced trees. And preimage resistance property makes constructing deliberately unbalanced trees computationally hard.

Architecture of Ethereum Virtual Machine

Each Ethereum node embeds a stack-based, 256-bit (32-byte) virtual machine. It does not follow von Neumann architecture, in that the code is not accessible as data. Memory and stack are private to one autonomous objects’s execution:


call to account C’s code
transaction 1 to account A
inputs
execution 1
memory + stack
inputs
outputs
execution 1.1
memory + stack
transaction 2 to account A
inputs
execution 2
memory + stack
persistent storage of account A
persistent storage of account C
transaction 3 to account B
inputs
execution 3
memory + stack
persistent storage of account B
environment: account balances, block hashes, etc.

EVM word size

Memory and storage addresses are 256 bit (32 bytes) integers.

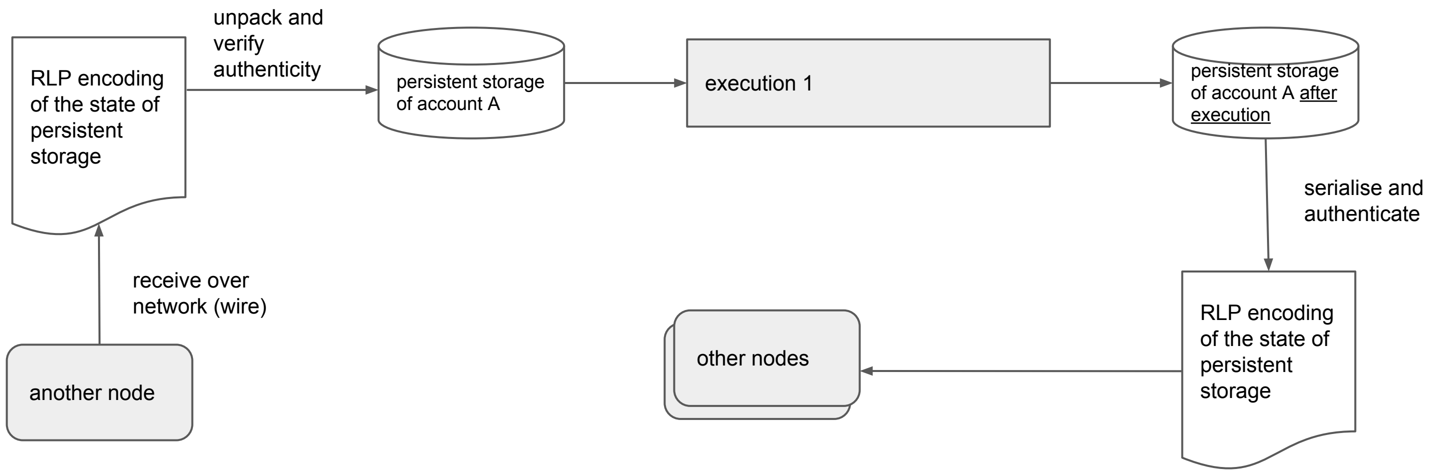
Most memory, and storage operations take and return 256 bit (32 bytes) Big Endian integers.

Output of Keccak-256 (very closed to SHA3), used as the cryptographic hash function of choice is 256 bit.

However, minimal addressable unit of memory is byte, not a chunk of 32 bytes.

Wire-level serialization

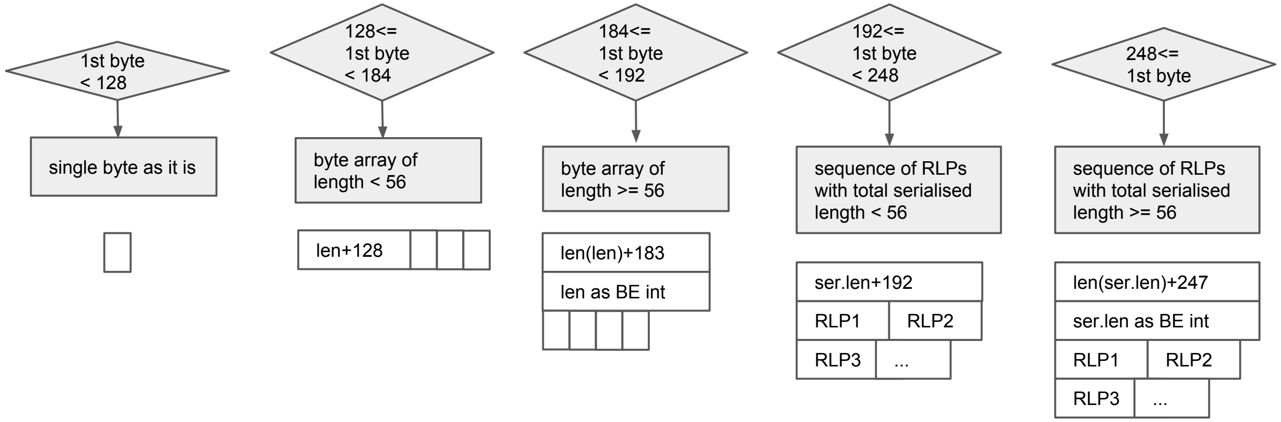
Ethereum specifies the standard of representation of data structures on the wire (when used inside the inter-node protocol) and for use in authenticated data structures (mostly Merkle Patricia tree).



unpack and verify authenticity
RLP encoding of the state of persistent storage
persistent storage of account A
execution 1
persistent storage of account A after execution
serialise and authenticate
receive over network (wire)
another node
other nodes
RLP encoding of the state of persistent storage

RLP - Recursive Length Prefix encoding

RLP is able to encode two types of structures: byte arrays of arbitrary size and sequences of other RLP-encoded structures. Interpretation of RLP stream:



RLP and encoding rules for Patricia define most of the protocol’s syntax.

Programming Ethereum

The lowest level of programming is EVM instructions. When one sends a transaction to an empty address with some data, the data are interpreted as EVM machine code. For example, a ‘Hello, world!’ program in EVM machine code (and assembly) would look like this:

```
602480600b6000396000f360207
f48656c6c6f20776f726c642100
000000000000000000000000000
00000000000f3
```

```
PUSH1 0x24
DUP1
PUSH1 0xB
PUSH1 0x0
CODECOPY
PUSH1 0x0
RETURN
PUSH1 0x20
PUSH32
0x48656C6C6F20776F726C6421000
00000000000000000000000000000
00000000
RETURN
```

```
.code:
 PUSH #[$00000000…00000000]
 DUP1
 PUSH [$00000000…00000000]
 PUSH 0
 CODECOPY
 PUSH 0
 RETURN
.data:
 0:
 .code:
   PUSH 20
   PUSH "Hello world!"
   RETURN
```

One level up - LLL

It stands for Lisp-Like Low-Level Language. It was the language used in initial Proof of Concepts and then as a cross-compilation target for higher level languages. Created by Gavin Wood as a build-in language of the first Ethereum C++ client, AlethZero. The same ‘Hello, World!’ program in LLL:

```
;INIT
{
    ;BODY
    (return 0x0 (lll
    {
        (return "Hello world!" 32)
    } 0x0) )
}
```

Other languages

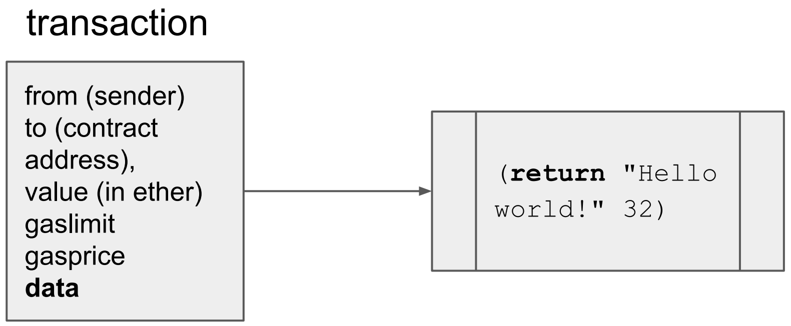
Mutan has C-inspired syntax, similarly low-level as LLL. Created by Jeffrey Wilcke as a built-in language of the first Go-based Ethereum client, ‘Ethereal’. Currently deprecated in favour of Solidity.

Serpent has a Python-inspired syntax. Created by Vitalik Buterin as a part of Python implementation of Ethereum client. Compiles into LLL.

Solidity has a Javascript and C++ inspired syntax and is under heavy development. Created by Christian Reitwiessner. Compiles directly into EVM code.

Higher level languages and ABI

On the low level of EVM machine code or LLL, contract does not have any ‘functions’ (like in Serpent) or ‘methods’ (like in Solidity). It only has one entry point which takes transaction data (txdata) as input. How do programs in Serpent and Solidity manage to have multiple entry points?



```
(return "Hello
world!" 32)
```

Higher level languages and ABI

Let’s take a simple function in Serpent and look how it compiles into LLL:

```
def double(x):
    return(x * 2)
```

```
(return 0
  (lll
    (with '__funid
      (div (calldataload 0)
        26959946667150639794667015087019630673637144422540572481103610249216
      )
      (unless (iszero (eq (get '__funid) 4008276486))
        (seq
          (set 'x (calldataload 4))
          (seq
            (set '_temp_521 (mul (get 'x) 2))
            (return (ref '_temp_521) 32)
          ) ) ) ) 0 ) )
```

26959946667150639794667015087019630673637144422540572481103610249216 is 2^{224}, 4008276486 - first 4 bytes of hash of “`double(int256)`” . First 4 bytes of txdata act as a “function selector”.

Contract ABI

ABI - Application Binary Interface, formally specifies how array of bytes (taken from transaction ‘data’) translated into a call of a specific function (potentially overloaded), and encodes all the arguments. It also encodes the return values of the functions into a byte array. ABI introduces types that are allowed in the function arguments and return values. Here is a list of currently supported types:

primitive types

```
uint8, uint16, ..., uint256
int8, int16, ..., int256
address
bool

fixed8x8, fixed8x16, ...,
fixed128x128, fixed120x136

ufixed8x8, ufixed8x16, ...,
ufixed128x128, ufixed120x136

bytes1, bytes2, ..., bytes32
```

fixed-sized array of any fixed-length type

```
<type>[M]
```

non-fixed-sized types

```
bytes
string (assumed UTF-8)
<type>[] (<type> is fixed length)
```

Other topics...

From White paper to Yellow paper to implementations (C++, Go, Python).

Logging and Transaction receipts (and their use in asynchronous programming)

EtHash and GHOST - current Proof Of Work consensus.

Casper - Proof Of Stake algorithm planned for Serenity release.

Sharding and Asynchronous programming for solving Scalability challenge.

Spin-offs: ConsenSys, slock.it, EthCore