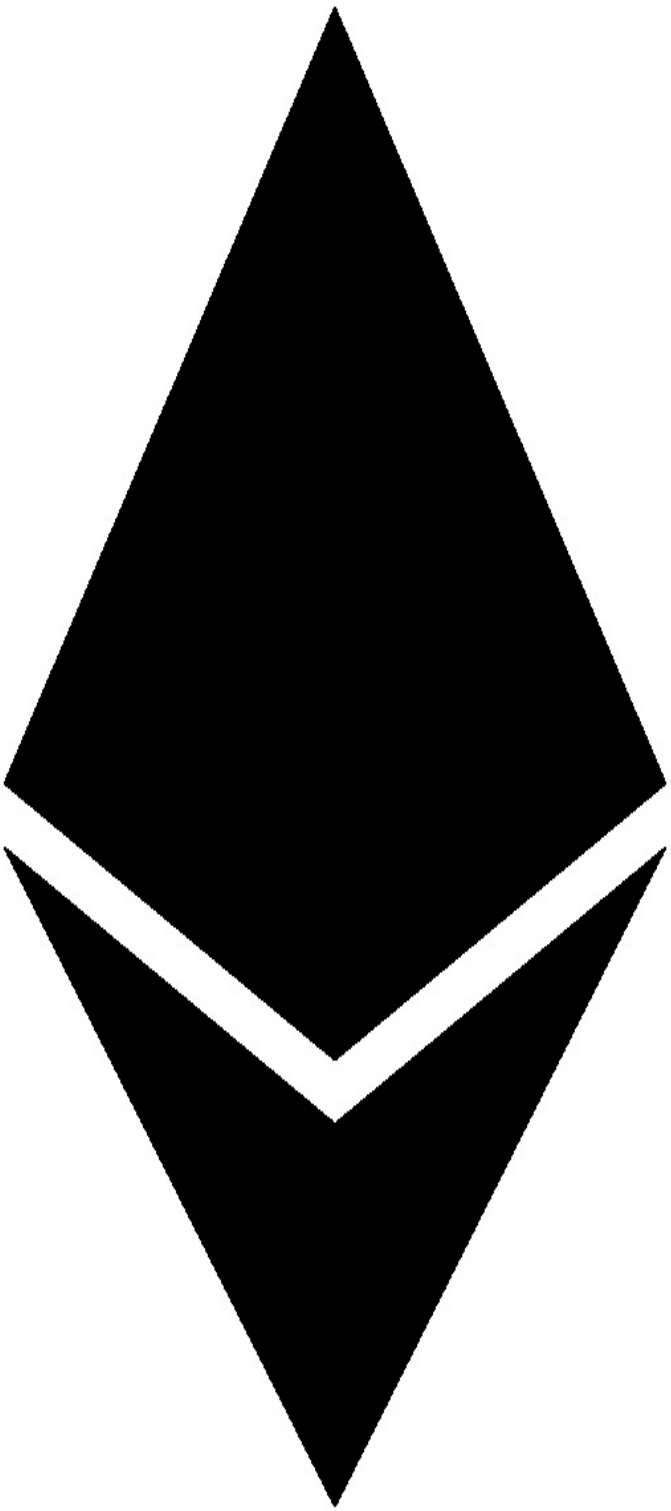


Table of Contents

This guide is still under heavy development.

Into the ether

Continuing the experiment ...

When the grand experiment that is Bitcoin began, the anonymous wizard desired to test two parameters:

1. **a trustless, decentralized database enjoying security enforced by the austere relentlessness of cryptography**
2. **a robust transaction system capable of sending value across the world without intermediaries.**

Yet the past five years years have painfully demonstrated a ***third*** **missing feature: a sufficiently powerful Turing-complete scripting language**. Up until this point, most innovation in advanced applications such as domain and identity registration, user-issued currencies, smart property, smart contracts, and decentralized exchange has been highly fragmented, and implementing any of these technologies has required creating an entire meta-protocol layer or even a specialized blockchain. Theoretically, however, each and every one of these innovations and more can potentially be made hundreds of times easier to implement, and easier to scale, if only there was a stronger foundational layer with a powerful scripting language for all of these protocols to build upon. And this need is what we seek to satisfy.

Ethereum is a modular, stateful, Turing-complete contract scripting system married to a blockchain and developed with a philosophy of simplicity, universal accessibility and generalization. Our goal is to provide a platform for decentralized applications - an android of the cryptocurrency world, where all efforts can share a common set of APIs, trustless interactions and no compromises.

... together

During the past year there have been a number of developers and researchers who have either worked for Ethereum or working on ideas as volunteers and happen to spend lots of time interacting with the Ethereum community, and this set of people has coalesced into a group dedicated to building out the vision set by Vitalik Buterin in the Ethereum Whitepaper.

We ask for the community to join us as volunteers, developers, investors and evangelists seeking to enable a fundamentally different paradigm for the Internet and the relationships it provides.

Thank *you*.

Our philosophy at Ethereum is that every developer no matter what their level of technical ability should be able to create amazing things on our open-source, blockchain based platform.

With this in mind we started this guide to help developers, from seasoned full stack developer to a complete beginner to learn how to setup the Ethereum clients, build DAPPs and in the process learn the skills necessary to build decentralized applications on top of Ethereum.

This is a knowledge remix

Just as in the Bitcoin ecosystem, in the beginning, there was quite a bit of knowledge fragmentatation - one blog post here, a forum post tutorial there, a misleading article there and so on.

This book represents a mashup of wiki pages, tutorials and other relevant information that so far has been scattered without an easy way to find and access it.

The aim of this work

This guide tries to create an **easy to access ethereum knowledge base** that can help people from all around the world to **start experimenting with blockchain technologies**.

Hopefully through this work, we will get closer to a reality in which we all realize that as long as we experiment with blockchain related technologies, we enrich the decentralized ecosystem in our own unique ways - contributing to something bigger than any individual project or initiative.

Special thanks

Through their work, these people make this guide possible.

Vitalik Buterin

Inventor of Ethereum

Gavin Wood

C++/IDE/Whisper/Swarm Lead

Jeffrey Wilcke

Go/Mist Project Lead

(the list will get bigger soon)

...

...

...

In the end, we'd also like to thank ***you*** for being part of this and for your support in making this happen.

Open Collaboration Welcomed

This book lives on GitHub. You can find everything listed in this book also in this repository.

If you want to jump straight in and contribute directy from GitHub, feel free to do so and enjoy the warm fuzzy feeling for being part of something truly world-changing.

If you want to be part of the GitBook contributors team please get in touch with us at hello@ethereum.builders and we'll get back to you as soon as humanly possible.

If you want to join our friendly GitBook Gitter chatroom to say hello, contribute or give feedback click the button bellow.

See you there!

The Why, How and What

Design philosophy

These are the core tenets that sparked the work on Ethereum's architeture and imlementation.

Simplicity

The Ethereum protocol should be as simple as possible, even at the cost of some data storage inefficiency or time inefficiency. An average programmer should ideally be able to follow and implement the entire specification, so as to eventually help minimize the influence that any specific individual or elite group can have on the protocol and furthering the vision of Ethereum as a protocol that is open to all. Optimizations which add complexity should not be included unless they provide very substantial benefit.

Universality

It is a fundamental part of Ethereum’s design philosophy that Ethereum does not have “features”. Instead, Ethereum provides an internal Turing-complete scripting language which you can use to construct any smart contract or transaction type that can be mathematically defined. Want to invent your own financial derivative? With Ethereum, you can. Want to make your own currency? Set it up as an Ethereum contract. Want to set up a full-scale Daemon or Skynet? You may need to have a few thousand interlocking contracts, and be sure to feed them generously, to do that, but nothing is stopping you.

Modularity

Different parts of the Ethereum protocol should be designed to be as modular and separable as possible. Over the course of development, it should be easy to make a small protocol modification in one place and have the application stack continue to function without any further modification. Innovations such as Dagger, Patricia trees and RLP should be implemented as separate libraries and made to be feature-complete even if Ethereum does not require certain features so as to make them usable in other protocols as well. Ethereum development should be maximally done so as to benefit the entire cryptocurrency ecosystem, not just itself.

Non-Discrimination

The protocol should not attempt to actively restrict or prevent specific categories of usage, and all regulatory mechanisms in the protocol should be designed to directly regulate the harm, not attempt to oppose specific undesirable applications. You can even run an infinite loop script on top of Ethereum for as long as you are willing to keep paying the per-computational-step transaction fee for it.

Agility

Details of the Ethereum protocol are not set in stone. Although we will be extremely judicious about making modifications to high-level constructs such as the C-like language and the address system, computational tests later on in the development process may lead us to discover that certain modifications to the algorithm or scripting language will substantially improve scalability or security. If any such opportunities are found, we will make use of them.

Although Ethereum borrows many ideas that have already been tried and tested for half a decade in older cryptocurrencies like Bitcoin, there are a number of places in which Ethereum diverges from the most common way of handling certain protocol features, and there are also many situations in which Ethereum has been forced to develop completely new economic approaches because it offers functionality that is not offered by other existing systems. The purpose of this document will be to detail all of the finer potentially nonobvious or in some cases controversial decisions that were made in the process of building the Ethereum protocol, as well as showing the risks involved in both our approach and possible alternatives.

Principles

The Ethereum protocol design process follows a number of principles:

1. **Sandwich complexity model**: we believe that the bottom level architecture of Ethereum should be as simple as possible, and the interfaces to Ethereum (including high level programming languages for developers and the user interface for users) should be as easy to understand as possible. Where complexity is inevitable, it should be pushed into the "middle layers" of the protocol, that are not part of the core consensus but are also not seen by end users - high-level-language compilers, argument serialization and deserialization scripts, storage data structure models, the leveldb storage interface and the wire protocol, etc. However, this preference is not absolute.
2. **Freedom**: users should not be restricted in what they use the Ethereum protocol for, and we should not attempt to preferentially favor or disfavor certain kinds of Ethereum contracts or transactions based on the nature of their purpose. This is similar to the guiding principle behind the concept of "net neutrality". One example of this principle *not* being followed is the situation in the Bitcoin transaction protocol where use of the blockchain for "off-label" purposes (eg. data storage, meta-protocols) is discouraged, and in some cases explicit quasi-protocol changes (eg. OP_RETURN restriction to 40 bytes) are made to attempt to attack applications using the blockchain in "unauthorized" ways. In Ethereum, we instead strongly favor the approach of setting up transaction fees in such a way as to be roughly incentive-compatible, such that users that use the blockchain in bloat-producing ways internalize the cost of their activities (ie. Pigovian taxation).
3. **Generalization**: protocol features and opcodes in Ethereum should embody maximally low-level concepts, so that they can be combined in arbitrary ways including ways that may not seem useful today but which may become useful later, and so that a bundle of low-level concepts can be made more efficient by stripping out some of its functionality when it is not necessary. An example of this principle being followed is our choice of a LOG opcode as a way of feeding information to (particularly light client) dapps, as opposed to simply logging all transactions and messages as was internally suggested earlier - the concept of "message" is really the agglomeration of multiple concepts, including "function call" and "event interesting to outside watchers", and it is worth separating the two.
4. **We Have No Features**: as a corollary to generalization, we often refuse to build in even very common high-level use cases as intrinsic parts of the protocol, with the understanding that if people really want to do it they can always create a sub-protocol (eg. ether-backed subcurrency, bitcoin/litecoin/dogecoin sidechain, etc) inside of a contract. An example of this is the lack of a Bitcoin-like "locktime" feature in Ethereum, as such a feature can be simulated via a protocol where users send "signed data packets" and those data packets can be fed into a specialized contract that processes them and performs some corresponding function if the data packet is in some contract-specific sense valid.
5. **Non-risk-aversion**: we are okay with higher degrees of risk if a risk-increasing change provides very substantial benefits (eg. generalized state transitions, 50x faster block times, consensus efficiency, etc)

These principles are all involved in guiding Ethereum development, but they are not absolute; in some cases, desire to reduce development time or not to try too many radical things at once has led us to delay certain changes, even some that are obviously beneficial, to a future release (eg. Ethereum 1.1).

Blockchain-level protocol

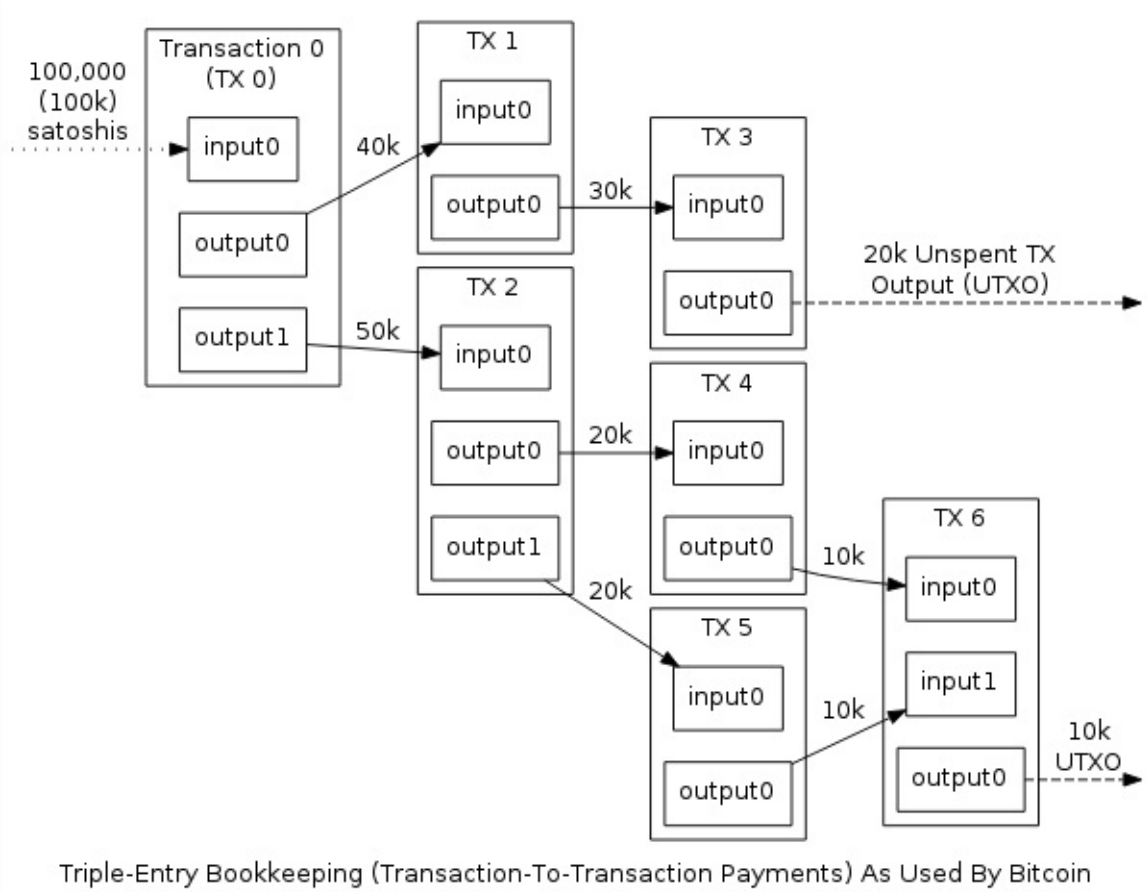
This section provides a description of some of the blockchain-level protocol changes made in Ethereum, including how blocks and transactions work, how data is serialized and stored, and the mechanisms behind accounts.

Accounts and not UTXOs

Bitcoin, along with many of its derivatives, stores data about users' balances in a structure based on *unspent transaction outputs* (UTXOs): the entire state of the system consists of a set of "unspent outputs" (think, "coins"), such that each coin has an owner and a value, and a transaction spends one or more coins and creates one or more new coins, subject to the validity constraints:

1. Every referenced input must be valid and not yet spent
2. The transaction must have a signature matching the owner of the input for every input
3. The total value of the inputs must equal or exceed the total value of the outputs

A user's "balance" in the system is thus the total value of the set of coins for which the user has a private key capable of producing a valid signature.



Triple-Entry Bookkeeping (Transaction-To-Transaction Payments) As Used By Bitcoin

(Image from https://bitcoin.org/en/developer-guide)

Ethereum jettisons this scheme in favor of a simpler approach: the state stores a list of accounts where each account has a balance, as well as Ethereum-specific data (code and internal storage), and a transaction is valid if the sending account has enough balance to pay for it, in which case the sending account is debited and the receiving account is credited with the value. If the receiving account has code, the code runs, and internal storage may also be changed, or the code may even create additional messages to other accounts which lead to further debits and credits.

The benefits of UTXOs are:

1. **Higher degree of privacy**: if a user uses a new address for each transaction that they receive then it will often be difficult to link accounts to each other. This applies greatly to currency, but less to arbitrary dapps, as arbitrary dapps often necessarily involve keeping track of complex bundled state of users and there may not exist such an easy user state partitioning scheme as in currency.
2. **Potential scalability paradigms**: UTXOs are more theoretically compatible with certain kinds of scalability paradigms,

as we can rely on only the owner of some coins maintaining a Merkle proof of ownership, and even if everyone including the owner decides to forget that data then only the owner is harmed. In an account paradigm, everyone losing the portion of a Merkle tree corresponding to an account would make it impossible to process messages that affect that account at all in any way, including sending to it. However, non-UTXO-dependent scalability paradigms do exist.

The benefits of accounts are:

1. **Large space savings**: for example, if an account has 5 UTXO, then switching from a UTXO model to an account model would reduce the space requirements from (20 + 32 + 8) * 5 = 300 bytes (20 for the address, 32 for the txid and 8 for the value) to 20 + 8 + 2 = 30 bytes (20 for the address, 8 for the value, 2 for a nonce(see below)). In reality savings are not nearly this massive because accounts need to be stored in a Patricia tree (see below) but they are nevertheless large. Additionally, transactions can be smaller (eg. 100 bytes in Ethereum vs. 200-250 bytes in Bitcoin) because every transaction need only make one reference and one signature and produces one output.
2. **Greater fungibility**: because there is no blockchain-level concept of the source of a specific set of coins, it becomes less practical, both technically and legally, to institute a redlist/blacklisting scheme and to draw a distinction between coins depending on where they come from.
3. **Simplicity**: easier to code and understand, especially once more complex scripts become involved. Although it is possible to shoehorn arbitrary decentralized applications into a UTXO paradigm, essentially by giving scripts the ability to restrict what kinds of UTXO a given UTXO can be spent to, and requiring spends to include Merkle tree proofs of change-of-application-state-root that scripts evaluate, such a paradigm is much more complicated and ugly than just using accounts.
4. **Constant light client reference**: light clients can at any point access all data related to an account by scanning down the state tree in a specific direction. In a UTXO paradigm, the references change with each transaction, a particularly burdensome problem for long-running dapps that try to use the above mentioned state-root-in-UTXO propagation mechanism.

We have decided that, particularly because we are dealing with dapps containing arbitrary state and code, the benefits of accounts massively outweigh the alternatives. Additionally, in the spirit of the We Have No Features principle, we note that if people really do care about privacy then mixers and coinjoin can be built via signed-data-packet protocols inside of contracts.

One weakness of the account paradigm is that in order to prevent replay attacks, every transaction must have a "nonce", such that the account keeps track of the nonces used and only accepts a transaction if its nonce is 1 after the last nonce used. This means that even no-longer-used accounts can never be pruned from the account state. A simple solution to this problem is to require transactions to contain a block number, making them un-replayable after some period of time, and reset nonces once every period. Miners or other users will need to "ping" unused accounts in order to delete them from the state, as it would be too expensive to do a full sweep as part of the blockchain protocol itself. We did not go with this mechanism only to speed up development for 1.0; 1.1 and beyond will likely use such a system.

Merkle Patricia Trees

The Merkle Patricia tree/trie, previously envisioned by Alan Reiner and implemented in the Ripple protocol, is the primary data structure of Ethereum, and is used to store all account state, as well as transactions and receipts in each block. The MPT is a combination of a Merkle tree and Patricia tree, taking the elements of both to create a structure that has both of the following properties:

1. Every unique set of key/value pairs maps uniquely to a root hash, and it is not possible to spoof membership of a key/value pair in a trie (unless an attacker has ~2^128 computing power)
2. It is possible to change, add or delete key/value pairs in logarithmic time

This gives us a way of providing an efficient, easily updateable, "fingerprint" of our entire state tree. The Ethereum MPT is formally described here: https://github.com/ethereum/wiki/wiki/Patricia-Tree

Specific design decisions in the MPT include:

1. **Having two classes of nodes**, kv nodes and diverge nodes (see MPT spec for more details). The presence of kv nodes increases efficiency because if a tree is sparse in a particular area the kv node will serve as a "shortcut" removing the need to have a tree of depth 64.
2. **Making diverge nodes hexary and not binary**: this was done to improve lookup efficiency. We now recognize that this choice was suboptimal, as the lookup efficiency of a hexary tree can be simulated in a binary paradigm by storing nodes batched. However, because the trie construction is so easy to implement incorrectly and end up with at the very least state root mismatches, we have decided to table such a reorganization until 1.1.
3. **No distinction between empty value and non-membership**: this was done for simplicity, and because it works well with Ethereum's default that values that are unset (eg. balances) generally mean zero and the empty string is used to represent zero. However, we do note that it sacrifices some generality and is thus slightly suboptimal.
4. **Distinction between terminating and non-terminating nodes**: technically, the "is this node terminating" flag is unnecessary, as all tries in Ethereum are used to store static key lengths, but we added it anyway to increase generality, hoping that the Ethereum MPT implementations will be used as-is by other cryptographic protocols.
5. **Using sha3(k) as the key in the "secure tree"** (used in the state and account storage tries): this makes it much more difficult to DDoS the trie by setting up maximally unfavorable chains of diverge nodes 64 levels deep and repeatedly calling SLOAD and SSTORE on them. Note that this makes it more difficult to enumerate the tree; if you want to have enumeration capability in your client, the simplest approach is to maintain a database mapping `sha3(k) -> k`.

RLP

RLP ("recursive length prefix") encoding is the main serialization format used in Ethereum, and is used everywhere - for blocks, transactions, account state data and wire protocol messages. RLP is formally described here: https://github.com/ethereum/wiki/wiki/RLP

RLP is intended to be a highly minimalistic serialization format; its sole purpose is to store nested arrays of bytes. Unlike protobuf, BSON and other existing solutions, RLP does not attempt to define any specific data types such as booleans, floats, doubles or even integers; instead, it simply exists to store structure, in the form of nested arrays, and leaves it up to the protocol to determine the meaning of the arrays. Key/value maps are also not explicitly supported; the semi-official suggestion for supporting key/value maps is to represent such maps as `[[k1, v1], [k2, v2], ...]` where `k1, k2...` are sorted using the standard ordering for strings.

The alternative to RLP would have been using an existing algorithm such as protobuf or BSON; however, we prefer RLP because of (1) simplicity of implementation, and (2) guaranteed absolute byte-perfect consistency. Key/value maps in many languages don't have an explicit ordering, and floating point formats have many special cases, potentially leading to the same data leading to different encodings and thus different hashes. By developing a protocol in-house we can be assured that it is designed with these goals in mind (this is a general principle that applies also to other parts of the code, eg. the VM). Note that bencode, used by BitTorrent, may have provided a passable alternative for RLP, although its use of decimal encoding for lengths makes it slightly suboptimal compared to the binary RLP.

Compression algorithm

The wire protocol and the database both use a custom compression algorithm to store data. The algorithm can best be described as run-length-encoding zeroes and leaving other values as they are, with the exception of a few special cases for common values like `sha3('')`. For example:

```
>>> compress('horse')
'horse'
>>> compress('donkey dragon 1231231243')
'donkey dragon 1231231243'
>>> compress('\xf8\xaf\xf8\xab\xa0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
'\xf8\xaf\xf8\xab\xa0\xfe\x9e\xbe{b\xd5\xcd\x8d\x87\x97'
```

```
>>> compress("\xc5\xd2F\x01\x86\xf7#<\x92~}\xb2\xdc\xc7\x03\xc0\xe5\x00\xb6S\xca\x82';{\xfa\xd8\x04]\x85\xa4p")
'\xfe\x01'
```

Before the compression algorithm existed, many parts of the Ethereum protocol had a number of special cases; for example, `sha3` was often overridden so that `sha3('') = ''`, as that would save 64 bytes from not needing to store code or storage in accounts. However, a change was made recently where all of these special cases were removed, making Ethereum data structures much bulkier by default, instead adding the data saving functionality to a layer outside the blockchain protocol by putting it on the wire protocol and seamlessly inserting it into users' database implementations. This adds modularity, simplifying the consensus layer, and also allows continued upgrades to the compression algorithm to be deployed relatively easily (eg. via network protocol versions).

Trie Usage

Warning: this section assumes knowledge of how bloom filters work. For an introduction, see http://en.wikipedia.org/wiki/Bloom_filter

Every block header in the Ethereum blockchain contains pointers to three tries: the *state trie*, representing the entire state after accessing the block, the *transaction trie*, representing all transactions in the block keyed by index (ie. key 0: the first transaction to execute, key 1: the second transaction, etc), and the *receipt tree*, representing the "receipts" corresponding to each transaction. A receipt for a transaction is an RLP-encoded data structure:

```
[ medstate, gas_used, logbloom, logs ]
```

Where:

- `medstate` is the state trie root after processing the transaction
- `gas_used` is the amount of gas used after processing the transaction
- `logs` is a list of items of the form `[address, [topic1, topic2...], data]` that are produced by the `LOG0` ... `LOG4` opcodes during the execution of the transaction (including by the main call and sub-calls). `address` is the address of the contract that produced the log, the topics are up to 4 32-byte values, and the data is an arbitrarily sized byte array.
- `logbloom` is a bloom filter made up of the addresses and topics of all logs in the transaction.

There is also a bloom in the block header, which is the OR of all of the blooms for the transactions in the block. The purpose of this construction is to make the Ethereum protocol light-client friendly in as many ways as possible. In Ethereum, a light client can be viewed as a client that downloads block headers by default, and verifies only a small portion of what needs to be verified, using a DHT as a database for trie nodes in place of its local hard drive. Some use cases include:

- A light client wants to know the state of an account (nonce, balance, code or storage index) at a particular time. The light client can simply recursively download trie nodes from the state root until it gets to the desired value.
- A light client wants to check that a transaction was confirmed. The light client can simply ask the network for the index and block number of that transaction, and recursively download transaction trie nodes to check for availability.
- Light clients want to collectively validate a block. Each light client `C[i]` chooses one transaction index `i` with transaction `T[i]` (with corresponding receipt `R[i]`) and does the following:
  - Initiate the state with state root `R[i-1].medstate` and `R[i-1].gas_used` (if `i = 0` use the parent endstate and 0 `gas_used`)
  - Process transaction `T[i]`
  - Check that the resulting state root is `R[i].medstate` and the gas_used is `R[i].gas_used`
  - Check that the set of logs and bloom produced matches `R[i].logs` and `R[i].logbloom`
  - Checks that the bloom is a subset of the block header-level bloom (this detects block header-level blooms with false negatives); then pick a few random indices of the block header-level bloom where that bloom contains a 1

and ask other nodes for a transaction-level bloom that contains a 1 at that index, rejecting the block if no response is given (this detects block header-level blooms with false positives)

- Light clients want to "watch" for events that are logged. The protocol here is the following:
  - A light client gets all block headers, checks for block headers that contain bloom filters that match one of a desired list of addresses or topics that the light client is interested in
  - Upon finding a potentially matching block header, the light client downloads all transaction receipts, checks them for transactions whose bloom filters match
  - Upon finding a potentially matching transaction, the light client checks its actual log RLP, and sees if it actually matches

The first three light client protocols require a logarithmic amount of data access and computation; the fourth requires ~O(sqrt(N)) since bloom filters are only a two-level structure, although this can be improved to O(log(N)) if the light client is willing to rely on multiple providers to point to "interesting" transaction indices and decommission providers if they are revealed to have missed a transaction. The first protocol is useful to simply check up on state, and the second in consumer-merchant scenarios to check that a transaction was validated. The third protocol allows Ethereum light clients to collectively validate blocks with a very low degree of trust. In Bitcoin, for example, a miner can create a block that gives the miner an excessive amount of transaction fees, and there would be no way for light nodes to detect this themselves, or upon seeing an honest full node detect it verify a proof of invalidity. In Ethereum, if a block is invalid, it must contain an invalid state transition at some index, and so a light client that happens to be verifying that index can see that something is wrong, either because the proof step does not check out, or because data is unavailable, and that client can then raise the alarm.

The fourth protocol is useful in cases where a dapp wants to keep track of some kind of events that need to be efficiently verifiable, but which do not need to be part of the permanent state; an example is a decentralized exchange logging trades or a wallet logging transactions (note that the light client protocol will need to be augmented with header-level coinbase and uncle checks for this to work fully with mining accounts). In Bitcoin terminology, `LOG` can be viewed as a pure "proof of publication" opcode.

Uncle incentivization

The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013, and is the first serious attempt at solving the issues preventing much faster block times. The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted ("stale") and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

As described by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor (in Ethereum jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it.

To solve the second issue of centralization bias, we adopt a different strategy: we provide block rewards to stales: a stale block receives 7/8 (87.5%) of its base reward, and the nephew that includes the stale block receives 1/32 (3.125%) of the base reward as an inclusion bounty. Transaction fees, however, are not awarded to uncles or nephews.

In Ethereum, stale block can only be included as an uncle by up to the seventh-generation descendant of one of its direct siblings, and not any block with a more distant relation. This was done for several reasons. First, unlimited GHOST would

include too many complications into the calculation of which uncles for a given block are valid. Second, unlimited uncle incentivization as used in Ethereum removes the incentive for a miner to mine on the main chain and not the chain of a public attacker. Finally, calculations show that restricting to seven levels provides most of the desired effect without many of the negative consequences.

- A simulator that measures centralization risks is available at https://github.com/ethereum/economic-modeling/blob/master/ghost.py
- A high-level discussion can be found at https://blog.ethereum.org/2014/07/11/toward-a-12-second-block-time/

Design decisions in our block time algorithm include:

- **12 second block time**: 12 seconds was chosen as a time that is as fast as possible, but is at the same time substantially longer than network latency. A 2013 paper by Decker and Wattenhofer in Zurich measures Bitcoin network latency, and determines that 12.6 seconds is the time it takes for a new block to propagate to 95% of nodes; however, the paper also points out that the bulk of the propagation time is proportional to block size, and thus in a faster currency we can expect the propagation time to be drastically reduced. The constant portion of the propagation interval is about 2 seconds; however, for safety we assume that blocks take 12 seconds to propagate in our analysis.
- **7 block ancestor limit**: this is part of a design goal of wanting to make block history very quickly "forgettable" after a small number of blocks, and 7 blocks has been proven to provide most of the desired effect
- **1 block descendant limit** (eg. `c(c(p(p(p(head)))))`, where c = child and p = parent, is invalid): this is part of a design goal of simplicity, and the simulator above shows that it does not pose large centralization risks.
- **Uncle validity requirements**: uncles have to be valid headers, not valid blocks. This is done for simplicity, and to maintain the model of a blockchain as being a linear data structure (and not a block-DAG, as in Sompolinsky and Zohar's newer models). Requiring uncles to be valid blocks is also a valid approach.
- **Reward distribution**: 7/8 of the base mining reward to the uncle, 1/32 to the nephew, 0% of transaction fees to either. This will make uncle incentivization ineffective from a centralization perspective if fees dominate; however, this is one of the reasons why Ethereum is meant to continue issuing ether for as long as we continue using PoW.

Difficulty Update Algorithm

The difficulty in Ethereum is currently updated according to the following rule:

```
diff(genesis) = 2^32

diff(block) = diff.block.parent + floor(diff.block.parent / 1024) *
    1 if block.timestamp - block.parent.timestamp < 9 else
    -1 if block.timestamp - block.parent.timestamp >= 9
```

The design goals behind the difficulty update rule are:

- **Fast updating**: the time between blocks should readjust quickly given increasing or decreasing hashpower
- **Low volatility**: the difficulty should not bounce excessively if the hashpower is constant
- **Simplicity**: the algorithm should be relatively simple to implement
- **Low memory**: the algorithm should not rely on more than a few blocks of history, and should include as few "memory variables" as possible. Assume that the last ten blocks, plus all memory variables placed in the block headers of the last ten blocks, are all that is available for the algorithm to work with
- **Non-exploitability**: the algorithm should not excessively encourage miners to fiddle with timestamps, or mining pools to repeatedly add and remove hashpower, in an attempt to maximize their revenue

We have already determined that our current algorithm is highly suboptimal on low volatility and non-exploitability, and at the very least we plan to switch the timestamps compares to be the parent and grandparent, so that miners only have the incentive to modify timestamps if they are mining two blocks in a row. Another more powerful formula with simulations is located at https://github.com/ethereum/economic-modeling/blob/master/diffadjust/blkdiff.py (the simulator uses Bitcoin

mining power, but uses the per-day average for the entire day; it at one point simulates a 95% crash in a single day).

Gas and Fees

Whereas all transactions in Bitcoin are roughly the same, and thus their cost to the network can be modeled to a single unit, transactions in Ethereum are more complex, and so a transaction fee system needs to take into account many ingredients, including cost of bandwidth, cost of storage and cost of computation. Of particular importance is the fact that the Ethereum programming language is Turing-complete, and so transactions may use bandwidth, storage and computation in arbitrary quantities, and the latter may end up being used in quantities that due to the halting problem cannot even be reliably predicted ahead of time. Preventing denial-of-service attacks via infinite loops is a key objective.

The basic mechanism behind transaction fees is as follows:

- Every transaction must specify a quantity of "gas" that it is willing to consume (called `startgas`), and the fee that it is willing to pay per unit gas (`gasprice`). At the start of execution, `startgas * gasprice` ether are removed from the transaction sender's account.
- All operations during transaction execution, including database reads and writes, messages, and every computational step taken by the virtual machine consumes a certain quantity of gas.
- If a transaction execution processes fully, consuming less gas than its specified limit, say with `gas_rem` gas remaining, then the transaction executes normally, and at the end of the execution the transaction sender receives a refund of `gas_rem * gasprice` and the miner of the block receives a reward of `(startgas - gas_rem) * gasprice`.
- If a transaction "runs out of gas" mid-execution, then all execution reverts, but the transaction is nevertheless valid, and the only effect of the transaction is to transfer the entire sum `startgas * gasprice` to the miner.
- When a contract sends a message to the other contract, it also has the option to set a gas limit specifically on the sub-execution arising out of that message. If the sub-execution runs out of gas, then the sub-execution is reverted, but the gas is nevertheless consumed.

Each of the above components is necessary. For example:

- If transactions did not need to specify a gas limit, then a malicious user could send a transaction that makes a multi-billion round loop, and no one would be able to process it since processing such a transaction would take longer than a block interval, but miners would not be able to tell beforehand, leading to a denial-of-service vulnerability.
- The alternative to strict gas-counting, time-limiting, does not work because it is too highly subjective (some machines are faster than others, and even among identical machines close-calls will always exist)
- The entire value `startgas * gasprice` has to be taken out at the start as a deposit so that there arise no situations where an account "bankrupts" itself mid-execution and becomes unable to pay for its gas costs. Note that balance checking is not sufficient, because an account can send its balance somewhere else.
- If execution did not revert in the event of an insufficient gas error, then contracts would need to take strong and difficult security measures to prevent themselves from being exploited by transactions or messages that provide only enough gas halfway through, thereby leading to some of the changes in a contract execution being executed but not others.
- If sub-limits did not exist, then hostile accounts could enact a denial-of-service attack against other contracts by entering into agreements with them, and then inserting an infinite loop at the beginning of computation so that any attempts by the victim contract to compensate the attack contract or send a message to it would starve the entire transaction execution.
- Requiring transaction senders to pay for gas instead of contracts substantially increases developer usability. Very early versions of Ethereum had contracts pay for gas, but this led to the rather ugly problem that every contract had to implement "guard" code that would make sure that every incoming message compensated the contract with enough ether to pay for the gas that it consumed.

Note the following particular features in gas costs:

- 21000 gas is charged for any transaction as a "base fee". This covers the cost of an elliptic curve operation to recover the sender address from the signature as well as the disk and bandwidth space of storing the transaction.

- A transaction can include an unlimited amount of "data", and there exist opcodes in the virtual machine which allow the contract receiving a transaction to access this data. The gas fee for data is 1 gas per zero byte and 5 gas per nonzero byte. This formula arose because we saw that most transaction data in contracts written by users was organized into a series of 32-byte arguments, most of which had many leading zero bytes, and given that such constructions seem inefficient but are actually efficient due to compression algorithms, we wanted to encourage their use in place of more complicated mechanisms which would try to tightly pack arguments according to the expected number of bytes, leading to very substantial complexity increase at compiler level. This is an exception to the sandwich complexity model, but a justified one due to the ratio of cost to benefit.
- The cost of the SSTORE opcode, which sets values in account storage, is either: (i) 20000 gas when changing a zero value to a nonzero value, (ii) 5000 gas when changing a zero value to a zero value or a nonzero value to a nonzero value, or (iii) 5000 gas when changing a nonzero value to a zero value, plus a 20000 gas refund to be given at the end of successful transaction execution (ie. NOT an execution leading to an out-of-gas exception). Refunds are capped at 50% of the total gas spent by a transaction. This provides a small incentive to clear storage, as we noticed that lacking such an incentive many contracts would leave storage unused, leading to quickly increasing bloat, providing most of the benefits of "charging rent" for storage without the cost of losing the assurance that a contract once placed will continue to exist forever. The delayed refund mechanism is necessary to prevent denial-of-service attacks where the attacker sends a transaction with a low amount of gas that repeatedly clears a large number of storage slots as part of a long-running loop, and then runs out of gas, consuming a large amount of verifiers' computing power without actually clearing storage or spending a lot of gas. The 50% cap is needed to ensure that a miner given a transaction with some quantity of gas can still determine an upper bound on the computational time to execute the transaction.
- There is no gas cost to data in messages provided by contracts. This is because there is no need to actually "copy" any data during a message call, as the call data can simply be viewed as a pointer to the parent contract's memory which will not change while the child execution is in progress.
- Memory is an infinitely expandable array. However, there is a gas cost of 1 per 32 bytes of memory expansion, rounding up.
- Some opcodes, whose computation time is highly argument-dependent, have variable gas costs. For example, the gas cost of EXP is 10 + 10 per byte in the exponent (ie. x^0 = 1 gas, x^1 ... x^255 = 2 gas, x^256 ... x^65535 = 3 gas, etc), and the gas cost of the copy opcodes (CALLDATACOPY, CODECOPY, EXTCODECOPY) is 1 + 1 per 32 bytes copies, rounding up (LOG also has a similar rule). The memory expansion gas cost is not sufficient to cover this, as it opens up a quadratic attack (50000 rounds of CALLDATACOPY of 50000 gas ~= 50000^2 computing effort, but only ~50000 gas before the variable gas cost was introduced)
- The CALL opcode (and CALLCODE for symmetry) costs an additional 9000 gas if the value is nonzero. This is because any value transfer causes significant bloat to history storage for an archival node. Note that the actual fee *charged* is 6700; on top of this we add a mandatory 2300 gas minimum that is automatically given to the recipient. This is in order to ensure that wallets that receive transactions to at least have enough gas to make a log of the transaction.

The other important part of the gas mechanism is the economics of the gas price itself. The default approach, used in Bitcoin, is to have purely voluntary fees, relying on miners to act as the gatekeepers and set dynamic minimums; the equivalent in Ethereum would be allowing transaction senders to set arbitrary gas costs. This approach has been received very favorably in the Bitcoin community particularly because it is "market-based", allowing supply and demand between miners and transaction senders determine the price. The problem with this line of reasoning is, however, that transaction processing is not a market; although it is intuitively attractive to construe transaction processing as a service that the miner is offering to the sender, in reality every transaction that a miner includes will need to be processed by every node in the network, so the vast majority of the cost of transaction processing is borne by third parties and not the miner that is making the decision of whether or not to include it. Hence, tragedy-of-the-commons problems are very likely to occur.

Currently, due to a lack of clear information about how miners will behave in reality, we are going with a fairly simple approach: a voting system. Miners have the right to set the gas limit for the current block to be within ~0.0975% (1/1024) of the gas limit of the last block, and so the resulting gas limit should be the median of miners' preferences. The hope is that in the future we will be able to soft-fork this into a more precise algorithm.

Virtual Machine

The Ethereum virtual machine is the engine in which transaction code gets executed, and is the core differentiating feature between Ethereum and other systems. Note that the *virtual machine* should be considered separately from the *contract and message model* - for example, the SIGNEXTEND opcode is a feature of the VM, but the fact that contracts can call other contracts and specify gas limits to sub-calls is part of the contract and message model. Design goals in the EVM include:

- **Simplicity**: as few and as low-level opcodes as possible, as few data types as possible and as few virtual-machine-level constructs as possible
- **Total determinism**: there should be absolutely no room for ambiguity in any part of the VM specification, and the results should be completely deterministic. Additionally, there should be a precise concept of computational step which can be measured so as to compute gas consumption.
- **Space savings**: EVM assembly should be as compact as possible (eg. the 4000 byte base size of default C programs is NOT acceptable)
- **Specialization to expected applications**: the ability to handle 20-byte addresses and custom cryptography with 32-byte values, modular arithmetic used in custom cryptography, read block and transaction data, interact with state, etc
- **Simple security**: it should be easy to come up with a gas cost model for operations that makes the VM non-exploitable
- **Optimization-friendliness**: it should be easy to apply optimizations so that JIT-compiled and otherwise sped-up versions of the VM can be built.

Some particular design decisions that were made:

- **Temporary/permanent storage distinction** - a distinction exists between temporary storage, which exists within each instance of the VM and disappears when VM execution finishes, and permanent storage, which exists on the blockchain state level on a per-account basis. For example, suppose the following tree of execution takes place (using S for permanent storage and M for temporary): (i) A calls B, (ii) B sets `B.S[0] = 5`, `B.M[0] = 9`, (iii) B calls C, (iv) C calls B. At this point, if B tries to read `B.S[0]`, it will receive the value stored in B earlier, 5, but is B tries to read `B.M[0]` it will receive 0 because it is a new instance of the virtual machine with fresh temporary storage. If B now sets `B.M[0] = 13` and `B.S[0] = 17` in this inner call, and then both this inner call and C's call terminate, bringing the execution back to B's outer call, then B reading M will see `B.M[0] = 9` (since the last time this value was set was in the same VM execution instance) and `B.S[0] = 17`. If B's outer call ends and A calls B again, then B will ses `B.M[0] = 0` and `B.S[0] = 17`. The purpose of this distinction is to (1) provide each execution instance with its own memory that is not subject to corruption by recursive calls, making secure programming easier, and (2) to provide a form of memory which can be manipulated very quickly, as storage updates are necessarily slow due to the need to modify the trie.
- **Stack/memory model** - the decision was made early on to have three types of computational state (aside from the program counter which points to the next instruction): stack (a standard LIFO stack of 32-byte values), memory (an infinitely expandable temporary byte array) and storage (permanent storage). On the temporary storage side, the alternative to stack and memory is a memory-only paradigm, or some hybrid of registers and memory (not very different, as registers basically are a kind of memory). In such a case, every instruction would have three arguments, eg. `ADD R1 R2 R3: M[R1] = M[R2] + M[R3]`. The stack paradigm was chosen for the obvious reason that it makes the code four times smaller.
- **32 byte word size** - the alternative is 4 or 8 byte words, as in most other architectures, or unlimited, as in Bitcoin. 4 or 8 byte words are too restrictive to store addresses and big values for crypto computations, and unlimited values are too hard to make a secure gas model around. 32 bytes is ideal because it is just large enough to store 32 byte values common in many crypto implementations, as well as addresses (and provides the ability to pack address and value into a single storage index as an optimization), but not so large as to be extremely inefficient.
- **Having our own VM at all** - the alternative is reusing Java, or some Lisp dialect, or Lua. We decided that having a specialized VM was appropriate because (i) our VM spec is much simpler than many other virtual machines, because other virtual machines have to pay a much lower cost for complexity, whereas in our case every additional unit of complexity is a step toward high barriers of entry creating development centralization and potential for security flaws including consensus failures, (ii) it allows us to specialize the VM much more, eg. by having a 32 byte word size, (iii) it allows us not to have a very complex external dependency which may lead to installation difficulties, and (iv) a full security review of Ethereum specific to our particular security needs would necessitate a security review of the external VM anyway, so the effort savings are not that large.

- **Using a variable extendable memory size** - we deemed a fixed memory size unnecessarily restrictive if the size is small and unnecessarily expensive if the size is large, and noted that if statements for memory access are necessary in any case to check for out-of-bounds access, so fixed size would not even make execution more efficient.
- **Not having a stack size limit** - no particular justification either way; note that limits are not strictly necessary in many cases as the combination of gas costs and a block-level gas limit will always act as a ceiling on the consumption of every resource.
- **Having a 1024 call depth limit** - many programing languages break at high stack depths much more quickly than they break at high levels of memory usage or computational load, so the implied limit from the block gas limit may not be sufficient.
- **No types** - done for simplicity. Instead, signed and unsigned opcodes for DIV, SDIV, MOD, SMOD are used instead (it turns out that for ADD and MUL the behavior of signed and unsigned opcodes is equivalent), and the transformations for fixed point arithmetic (high-depth fixed-point arithmetic is another benefit of 32-byte words) are in all cases simple, eg. at 32 bits of depth, `a * b -> (a * b) / 2^32`, `a / b -> a * 2^32 / b`, and +, - and * are unchanged from integer cases.

The function and purpose of some opcodes in the VM is obvious, however other opcodes are less so. Some particular justifications are given below:

- **ADDMOD**, **MULMOD**: in most cases, `addmod(a, b, c) = a * b % c`. However, in the specific case of many classes of elliptic curve cryptography, 32-byte modular arithmetic is used, and doing `a * b % c` directly is therefore actually doing `((a * b) % 2^256) % c`, which gives a completely different result. A formula that calculates `a * b % c` with 32-byte values in 32 bytes of space is rather nontrivial and bulky.
- **SIGNEXTEND**: the purpose of SIGNEXTEND is to facilitate typecasting from a larger signed integer to a smaller signed integer. Small signed integers are useful because JIT-compiled virtual machines may in the future be able to detect long-running chunks of code that deals primarily with 32-byte integers and speed it up considerably.
- **SHA3**: SHA3 is very highly applicable in Ethereum code as secure infinite-sized hash maps that use storage will likely need to use a secure hash function so as to prevent malicious collisions, as well as for verifying Merkle trees and even verifying Ethereum-like data structures. A key point is that its companions `SHA256`, `ECRECOVER` and `RIPEMD160` are included not as opcodes but as pseudo-contracts. The purpose of this is to place them into a separate category so that, if/when we come up with a proper "native extensions" system later, more such contracts can be added without filling up the opcode space.
- **ORIGIN**: the primary use of the ORIGIN opcode, which provides the sender of a transaction, is to allow contracts to make refund payments for gas.
- **COINBASE**: the primary uses of the COINBASE opcode are to (i) allow sub-currencies to contribute to network security if they so choose, and (ii) open up the use of miners as a decentralized economic set for sub-consensus-based applications like Schellingcoin.
- **PREVHASH**: used as a semi-secure source of randomness, and to allow contracts to evaluate Merkle tree proofs of state in the previous block without requiring a highly complex recursive "Ethereum light client in Ethereum" construction.
- **EXTCODESIZE**, **EXTCODECOPY**: the primary uses here are to allow contracts to check the code of other contracts against a template, or even simulating them, before interacting with them. See http://lesswrong.com/lw/aq9/decision_theories_a_less_wrong_primer/ for applications.
- **JUMPDEST**: JIT-compiled virtual machines become much easier to implement when jump destinations are restricted to a few indices (specifically, the computational complexity of a variable-destination jump is roughly O(log(number of valid jump destinations)), although static jumps are always constant-time). Hence, we need (i) a restriction on valid variable jump destinations, and (ii) an incentive to use static over dynamic jumps. To meet both goals, we have the rules that (i) jumps that are immediately preceded by a push can jump anywhere but another jump, and (ii) other jumps can only jump to a JUMPDEST. The restriction against jumping on jumps is needed so that the question of whether a jump is dynamic or static can be determined by simply looking at the previous operation in the code. The lack of a need for JUMPDEST operations for static jumps is the incentive to use them. The prohibition against jumping into push data also speeds up JIT VM compilation and execution.
- **LOG**: LOG is meant to log events, see trie usage section above.
- **CALLCODE**: the purpose of this is to allow contracts to call "functions" in the form of code stored in other contracts,

with a separate stack and memory, but using the contract's own storage. This makes it much easier to scalably implement "standard libraries" of code on the blockchain.

- **SUICIDE**: an opcode which allows a contract to quickly delete itself if it is no longer needed. The fact that SUICIDES are processed at the end of transaction execution, and not immediately, is motivated by the fact that having the ability to revert suicides that were already executed would substantially increase the complexity of the cache that would be required in an efficient VM implementation.
- **PC**: although theoretically not necessary, as all instances of the PC opcode can be replaced by simply putting in the actual program counter at that index as a push, using PC in code allows for the creation of position-independent code (ie. compiled functions which can be copy/pasted into other contracts, and do not break if they end up at different indices).

What can you do with it

The purpose of this page is to serve as an introduction to the basics of Ethereum that you will need to understand from a development standpoint, in order to produce contracts and decentralized applications. For a general introduction to Ethereum, see the white paper, and for a full technical spec see the yellow papers, although those are not prerequisites for this page; that is to say, this page is meant as an alternative introduction to Ethereum specifically targeted toward application developers.

Introduction

Ethereum is a platform that is intended to allow people to easily write decentralized applications (Đapps) using blockchain technology. A decentralized application is an application which serves some specific purpose to its users, but which has the important property that the application itself does not depend on any specific party existing. Rather than serving as a front-end for selling or providing a specific party's services, a Đapp is a tool for people and organizations on different sides of an interaction use to come together without any centralized intermediary.

Even necessary "intermediary" functions that are typically the domain of centralized providers, such as filtering, identity management, escrow and dispute resolution, are either handled directly by the network or left open for anyone to participate, using tools like internal token systems and reputation systems to ensure that users get access to high-quality services. Early examples of Đapps include BitTorrent for file sharing and Bitcoin for currency. Ethereum takes the primary developments used by BitTorrent and Bitcoin, the peer to peer network and the blockchain, and generalizes them in order to allow developers to use these technologies for any purpose.

The Ethereum blockchain can be alternately described as a blockchain with a built-in programming language, or as a consensus-based globally executed virtual machine. The part of the protocol that actually handles internal state and computation is referred to as the Ethereum Virtual Machine (EVM). From a practical standpoint, the EVM can be thought of as a large decentralized computer containing millions of objects, called "accounts", which have the ability to maintain an internal database, execute code and talk to each other.

There are two types of accounts:

1. **Externally owned account (EOAs)**: an account controlled by a private key, and if you own the private key associated with the EOA you have the ability to send ether and messages from it.
2. **Contract**: an account that has its own code, and is controlled by code.

By default, the Ethereum execution environment is lifeless; nothing happens and the state of every account remains the same. However, any user can trigger an action by sending a transaction from an externally owned account, setting Ethereum's wheels in motion. If the destination of the transaction is another EOA, then the transaction may transfer some ether but otherwise does nothing. However, if the destination is a contract, then the contract in turn activates, and automatically runs its code.

The code has the ability to read/write to its own internal storage (a database mapping 32-byte keys to 32-byte values), read the storage of the received message, and send messages to other contracts, triggering their execution in turn. Once execution stops, and all sub-executions triggered by a message sent by a contract stop (this all happens in a deterministic and synchronous order, ie. a sub-call completes fully before the parent call goes any further), the execution environment halts once again, until woken by the next transaction.

Contracts generally serve four purposes:

1. Maintain a data store representing something which is useful to either other contracts or to the outside world; one example of this is a contract that simulates a currency, and another is a contract that records membership in a particular organization.

2. Serve as a sort of externally owned account with a more complicated access policy; this is called a "forwarding contract" and typically involves simply resending incoming messages to some desired destination only if certain conditions are met; for example, one can have a forwarding contract that waits until two out of a given three private keys have confirmed a particular message before resending it (ie. multisig). More complex forwarding contracts have different conditions based on the nature of the message sent; the simplest use case for this functionality is a withdrawal limit that is overrideable via some more complicated access procedure.

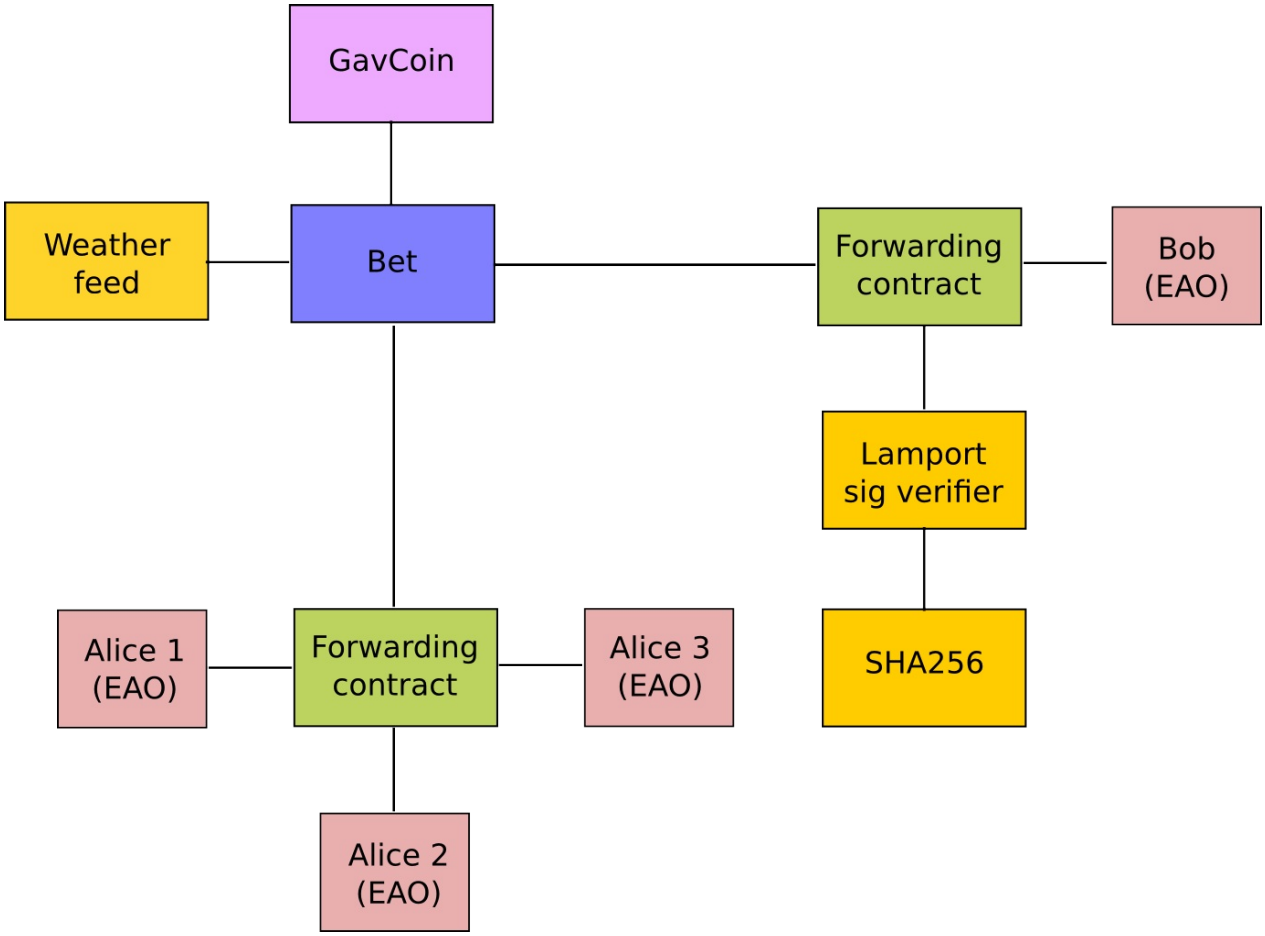
3. Manage an ongoing contract or relationship between multiple users. Examples of this include a financial contract, an escrow with some particular set of mediators, or some kind of insurance. One can also have an open contract that one party leaves open for any other party to engage with at any time; one example of this is a contract that automatically pays a bounty to whoever submits a valid solution to some mathematical problem, or proves that it is providing some computational resource.

4. Provide functions to other contracts; essentially serving as a software library.

Contracts interact with each other through an activity that is alternately called either "calling" or "sending messages". A "message" is an object containing some quantity of ether (a special internal currency used in Ethereum with the primary purpose of paying transaction fees), a byte-array of data of any size, the addresses of a sender and a recipient. When a contract receives a message it has the option of returning some data, which the original sender of the message can then immediately use. In this way, sending a message is exactly like calling a function.

Because contracts can play such different roles, we expect that contracts will be interacting with each other. As an example, consider a situation where Alice and Bob are betting 100 GavCoin that the temperature in San Francisco will not exceed 35ºC at any point in the next year. However, Alice is very security-conscious, and as her primary account uses a forwarding contract which only sends messages with the approval of two out of three private keys. Bob is paranoid about quantum cryptography, so he uses a forwarding contract which passes along only messages that have been signed with Lamport signatures alongside traditional ECDSA (but because he's old fashioned, he prefers to use a version of Lamport sigs based on SHA256, which is not supported in Ethereum directly).

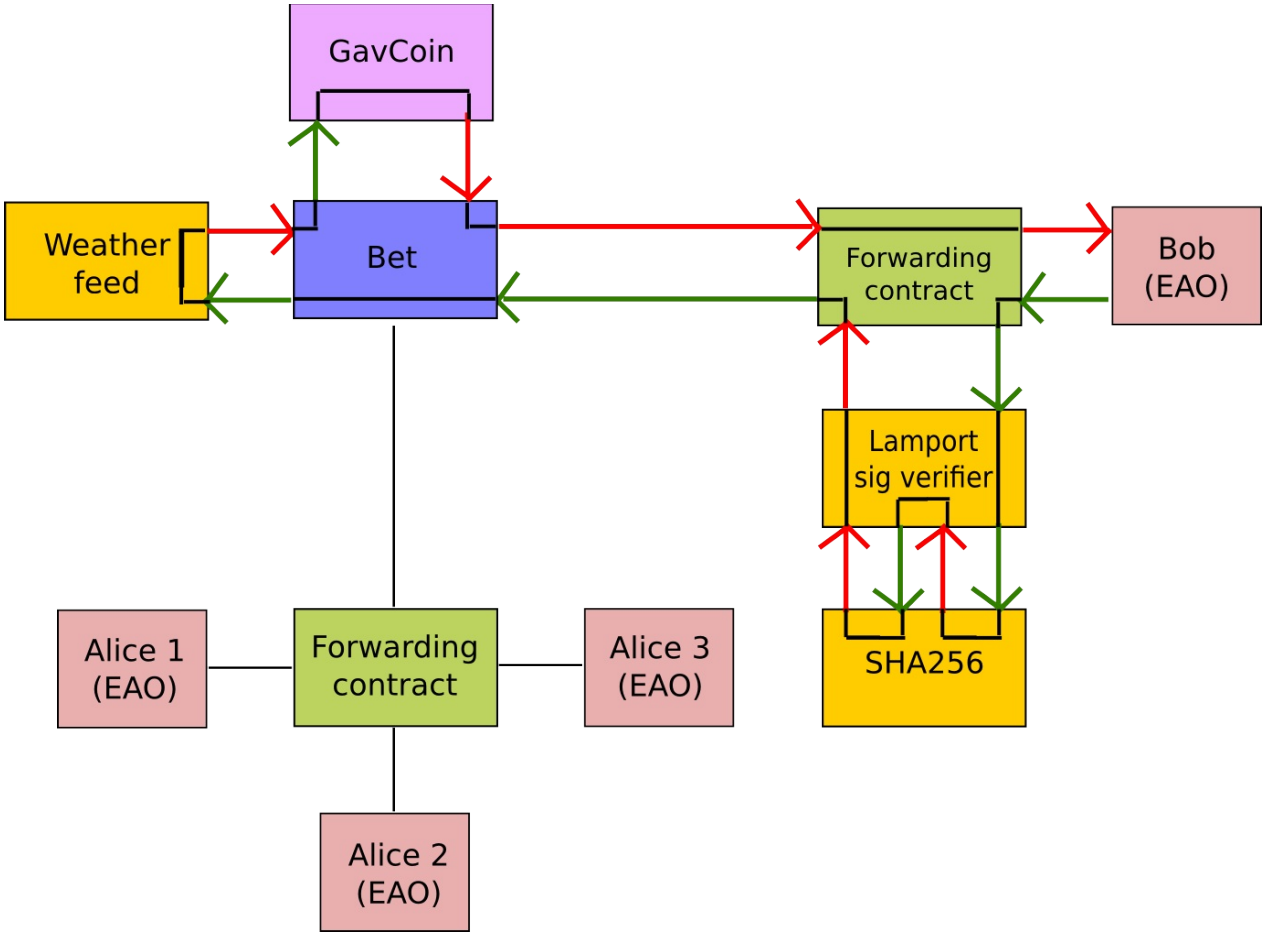
The betting contract itself needs to fetch data about the San Francisco weather from some contract, and it also needs to talk to the GavCoin contract when it wants to actually send the GavCoin to either Alice or Bob (or, more precisely, Alice or Bob's forwarding contract). We can show the relationships between the accounts thus:



When Bob wants to finalize the bet, the following steps happen:

1. A transaction is sent, triggering a message from Bob's EOA to Bob's forwarding contract.
2. Bob's forwarding contract sends the hash of the message and the Lamport signature to a contract which functions as a Lamport signature verification library.
3. The Lamport signature verification library sees that Bob wants a SHA256-based Lamport sig, so it calls the SHA256 library many times as needed to verify the signature.
4. Once the Lamport signature verification library returns 1, signifying that the signature has been verified, it sends a message to the contract representing the bet.
5. The bet contract checks the contract providing the San Francisco temperature to see what the temperature is.
6. The bet contract sees that the response to the messages shows that the temperature is above 35ºC, so it sends a message to the GavCoin contract to move the GavCoin from its account to Bob's forwarding contract.

Note that the GavCoin is all "stored" as entries in the GavCoin contract's database; the word "account" in the context of step 6 simply means that there is a data entry in the GavCoin contract storage with a key for the bet contract's address and a value for its balance. After receiving this message, the GavCoin contract decreases this value by some amount and increases the value in the entry corresponding to Bob's forwarding contract's address. We can see these steps in the following diagram:



State Machine

Computation in the EVM is done using a stack-based bytecode language that is like a cross between Bitcoin Script, traditional assembly and Lisp (the Lisp part being due to the recursive message-sending functionality). A program in EVM is a sequence of opcodes, like this:

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0 CALLDATALOAD SSTORE
```

The purpose of this particular contract is to serve as a name registry; anyone can send a message containing 64 bytes of data, 32 for the key and 32 for the value. The contract checks if the key has already been registered in storage, and if it has not been then the contract registers the value at that key.

During execution, an infinitely expandable byte-array called "memory", the "program counter" pointing to the current instruction, and a stack of 32-byte values is maintained. At the start of execution, memory and stack are empty and the PC is zero. Now, let us suppose the contract with this code is being accessed for the first time, and a message is sent in with 123 wei (10^{18} wei = 1 ether) and 64 bytes of data where the first 32 bytes encode the number 54 and the second 32 bytes encode the number 2020202020.

Thus, the state at the start is:

```
PC: 0 STACK: [] MEM: [], STORAGE: {}
```

The instruction at position 0 is PUSH1, which pushes a one-byte value onto the stack and jumps two steps in the code. Thus, we have:

```
PC: 2 STACK: [0] MEM: [], STORAGE: {}
```

The instruction at position 2 is CALLDATALOAD, which pops one value from the stack, loads the 32 bytes of message data starting from that index, and pushes that one the stack. Recall that the first 32 bytes here encode 54.

```
PC: 3 STACK: [54] MEM: [], STORAGE: {}
```

SLOAD pops one from the stack, and pushes the value in contract storage at that index. Since the contract is used for the first time, it has nothing there, so zero.

```
PC: 4 STACK: [0] MEM: [], STORAGE: {}
```

NOT pops one value and pushes 1 if the value is zero, else 0

```
PC: 5 STACK: [1] MEM: [], STORAGE: {}
```

Next, we PUSH1 9.

```
PC: 7 STACK: [1, 9] MEM: [], STORAGE: {}
```

The JUMPI instruction pops 2 values and jumps to the instruction designated by the first only if the second is nonzero. Here, the second is nonzero, so we jump. If the value in storage index 54 had not been zero, then the second value from top on the stack would have been 0 (due to NOT), so we would not have jumped, and we would have advanced to the STOP instruction which would have led to us stopping execution.

```
PC: 9 STACK: [] MEM: [], STORAGE: {}
```

Here, we PUSH1 32.

```
PC: 11 STACK: [32] MEM: [], STORAGE: {}
```

Now, we CALLDATALOAD again, popping 32 and pushing the bytes in message data starting from byte 32 until byte 63.

```
PC: 13 STACK: [2020202020] MEM: [], STORAGE: {}
```

Next, we PUSH1 0.

```
PC: 14 STACK: [2020202020, 0] MEM: [], STORAGE: {}
```

Now, we load message data bytes 0-31 again (loading message data is just as cheap as loading memory, so we don't bother to save it in memory)

```
PC: 16 STACK: [2020202020, 54] MEM: [], STORAGE: {}
```

Finally, we SSTORE to save the value 2020202020 in storage at index 54.

```
PC: 17 STACK: [] MEM: [], STORAGE: {54: 2020202020}
```

At index 17, there is no instruction, so we stop. If there was anything left in the stack or memory, it would be deleted, but the storage will stay and be available next time someone sends a message. Thus, if the sender of this message sends the same message again (or perhaps someone else tries to reregister 54 to 3030303030), the next time the `JUMPI` at position 7 would not process, and execution would STOP early at position 8.

Fortunately, you do not have to program in low-level assembly; a high-level language exists, especially designed for writing contracts, known as Solidity exists to make it much easier for you to write contracts (there are several others, too, including LLL, Serpent and Mutan, which you may find easier to learn or use depending on your experience). Any code you write in these languages gets compiled into EVM, and to create the contracts you send the transaction containing the EVM bytecode.

There are two types of transactions: a sending transaction and a contract creating transaction. A sending transaction is a standard transaction, containing a receiving address, an ether amount, a data bytearray and some other parameters, and a signature from the private key associated with the sender account. A contract creating transaction looks like a standard transaction, except the receiving address is blank. When a contract creating transaction makes its way into the blockchain, the data bytearray in the transaction is interpreted as EVM code, and the value returned by that EVM execution is taken to be the code of the new contract; hence, you can have a transaction do certain things during initialization. The address of the new contract is deterministically calculated based on the sending address and the number of times that the sending account has made a transaction before (this value, called the account nonce, is also kept for unrelated security reasons). Thus, the full code that you need to put onto the blockchain to produce the above name registry is as follows:

```
PUSH1 16 DUP PUSH1 12 PUSH1 0 CODECOPY PUSH1 0 RETURN STOP PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP PUSH1 32 C
```

The key opcodes are CODECOPY, copying the 16 bytes of code starting from byte 12 into memory starting at index 0, and RETURN, returning memory bytes 0-16, ie. code byes 12-28 (feel free to "run" the execution manually on paper to verify that those parts of the code and memory actually get copied and returned). Code bytes 12-28 are, of course, the actual code as we saw above.

Gas

One important aspect of the way the EVM works is that every single operation that is executed inside the EVM is actually simultaneously executed by every full node. This is a necessary component of the Ethereum 1.0 consensus model, and has the benefit that any contract on the EVM can call any other contract at almost zero cost, but also has the drawback that computational steps on the EVM are very expensive. Roughly, a good heuristic to use is that you will not be able to do anything on the EVM that you cannot do on a smartphone from 1999. Acceptable uses of the EVM include running business logic ("if this then that") and verifying signatures and other cryptographic objects; at the upper limit of this are applications that verify parts of other blockchains (eg. a decentralized ether-to-bitcoin exchange); unacceptable uses include using the EVM as a file storage, email or text messaging system, anything to do with graphical interfaces, and applications best suited for cloud computing like genetic algorithms, graph analysis or machine learning.

In order to prevent deliberate attacks and abuse, the Ethereum protocol charges a fee per computational step. The fee is market-based, though mandatory in practice; a floating limit on the number of operations that can be contained in a block forces even miners who can afford to include transactions at close to no cost to charge a fee commensurate with the cost of the transaction to the entire network; see the whitepaper section on fees for more details on the economic underpinnings of our fee and block operation limit system.

The way the fee works is as follows. Every transaction must contain, alongside its other data, a `GASPRICE` and `STARTGAS` value. `STARTGAS` is the amount of "gas" that the transaction assigns itself, and `GASPRICE` is the fee that the transaction pays per unit of gas; thus, when a transaction is sent, the first thing that is done during evaluation is subtracting `STARTGAS * GASPRICE` wei plus the transaction's value from the sending account balance. `GASPRICE` is set by the transaction sender, but miners will likely refuse to process transactions whose `GASPRICE` is too low.

Gas can be roughly thought of as a counter of computational steps, and is something that exists during transaction execution but not outside of it. When transaction execution starts, the gas remaining is set to `STARTGAS - 21000 - 68 * TXDATALEN` where `TXDATALEN` is the number of bytes in transaction data (note: zero bytes are charged only 4 gas due to the greater compressibility of long strings of zero bytes). Every computational step, a certain amount (usually 1, sometimes more depending on the operation) of gas is subtracted from the total. If gas goes down to zero, then all execution reverts, but the transaction is still valid and the sender still has to pay for gas. If transaction execution finishes with `N >= 0` gas remaining, then the sending account is refunded with `N * GASPRICE` wei.

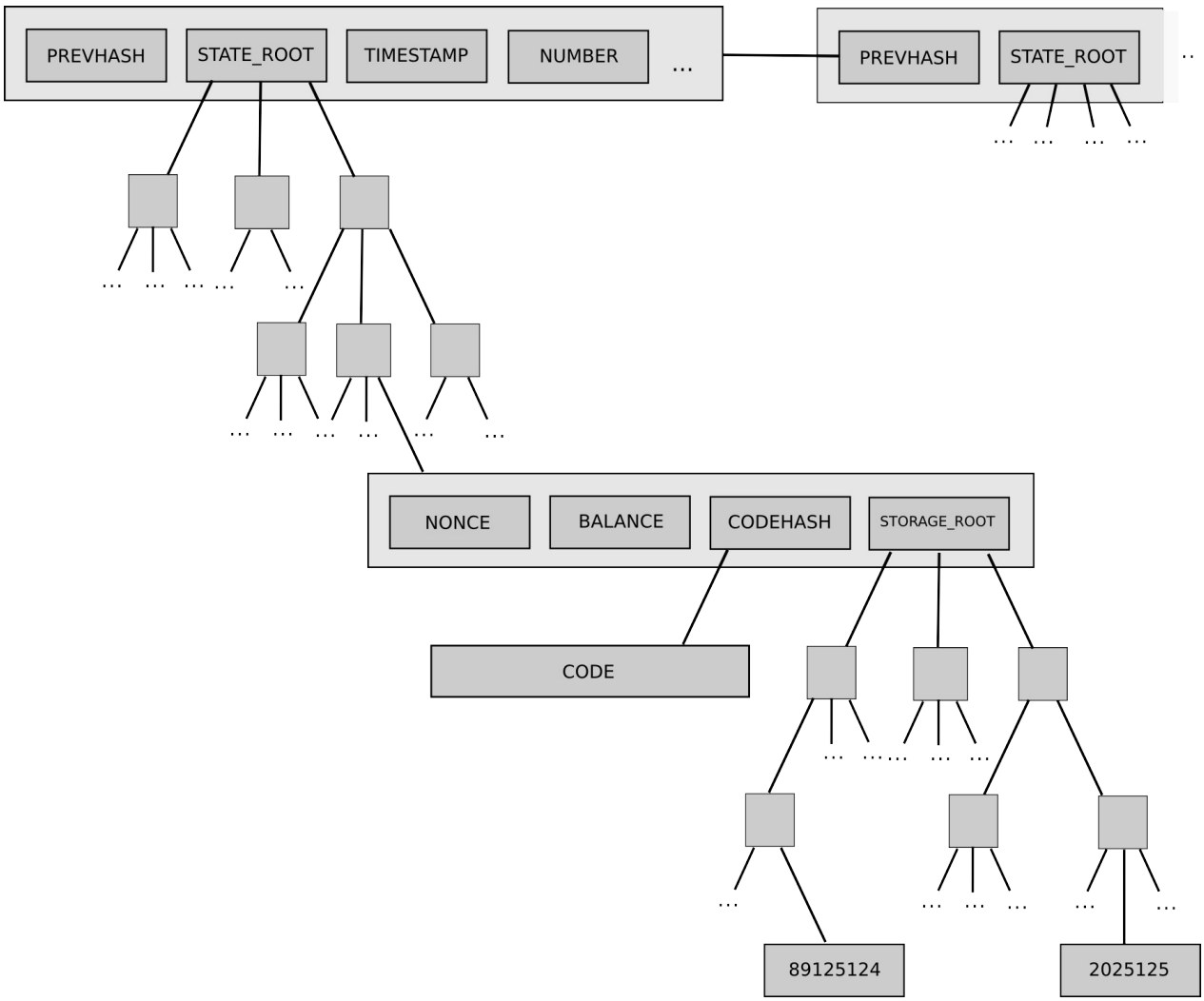
During contract execution, when a contract sends a message, that message call itself comes with a gas limit, and the sub-execution works the same way (namely, it can either run out of gas and revert or execute successfully and return a value). If sub-execution runs out of gas, the parent execution continues; thus, it is perfectly "safe" for a contract to call another contract if you set a gas limit on the sub-execution. If sub-execution has some gas remaining, then that gas is returned to the parent execution to continue using.

Virtual machine opcodes

A complete listing of the opcodes in the EVM can be found in the yellow paper. Note that high-level languages will often have their own wrappers for these opcodes, sometimes with very different interfaces.

Basics of the Ethereum Blockchain

The Ethereum blockchain (or "ledger") is the decentralized, massively replicated database in which the current state of all accounts is stored. The blockchain uses a database called a Patricia tree (or "trie") to store all accounts; this is essentially a specialized kind of Merkle tree that acts as a generic key/value store. Like a standard Merkle tree, a Patricia tree has a "root hash" that can be used to refer to the entire tree, and the contents of the tree cannot be modified without changing the root hash. For each account, the tree stores a 4-tuple containing `[account_nonce, ether_balance, code_hash, storage_root]`, where `account_nonce` is the number of transactions sent from the account (kept to prevent replay attacks), `ether_balance` is the balance of the account, `code_hash` the hash of the code if the account is a contract and "" otherwise, and `storage_root` is the root of yet another Patricia tree which stores the storage data.



Every minute, a miner produces a new block (the concept of mining in Ethereum is exactly the same as in Bitcoin; see any Bitcoin tutorial for more info on this), and that block contains a list of transactions that happened since the last block and the root hash of the Patricia tree representing the new state ("state tree") after applying those transactions and giving the miner an ether reward for creating the block.

Because of the way the Patricia tree works, if few changes are made then most parts of the tree will be exactly the same as in the last block; hence, there is no need to store data twice as nodes in the new tree will simply be able to point back to the same memory address that stores the nodes of the old tree in places where the new tree and the old tree are exactly the same. If a thousand pieces of data are changed between block `N` and block `N + 1`, even if the total size of the tree is many gigabytes, the amount of new data that needs to be stored for block `N + 1` is at most a few hundred kilobytes and often substantially less (especially if multiple changes happen inside the same contract). Every block contains the hash of the previous block (this is what makes the block set a "chain") as well as ancillary data like the block number, timestamp, address of the miner and gas limit.

Graphical Interfaces

A contract by itself is a powerful thing, but it is not a complete Đapp. A Đapp, rather, is defined as a combination of a contract and a graphical interface for using that contract (note: this is only true for now; future versions of Ethereum will include whisper, a protocol for allowing nodes in a Đapp to send direct peer-to-peer messages to each other without the blockchain). Right now, the interface is implemented as an HTML/CSS/JS webpage, with a special Javascript API in the form of the `eth` object for working with the Ethereum blockchain. The key parts of the Javascript API are as follows:

- `eth.transact(from, ethervalue, to, data, gaslimit, gasprice)` - sends a transaction to the desired address from the

desired address (note: `from` must be a private key and `to` must be an address in hex form) with the desired parameters

- `(string).pad(n)` - converts a number, encoded as a string, to binary form `n` bytes long
- `eth.gasPrice` - returns the current gas price
- `eth.secretToAddress(key)` - converts a private key into an address
- `eth.storageAt(acct, index)` - returns the desired account's storage entry at the desired index
- `eth.key` - the user's private key
- `eth.watch(acct, index, f)` - calls `f` when the given storage entry of the given account changes

You do not need any special source file or library to use the `eth` object; however, your Đapp will only work when opened in an Ethereum client, not a regular web browser. For an example of the Javascript API being used in practice, see the source code of this webpage.

Fine Points To Keep Track Of

See https://github.com/ethereum/wiki/wiki/Subtleties

Installing Ethereum

Building on Ubuntu

Below are the build instructions for the latest versions of Ubuntu. The best supported platform as of December 2014 is Ubuntu 14.04, 64 bit, with at least 2 GB RAM. All our tests are done with this version. Community contributions for other versions are welcome!

Building for Ubuntu 14.04, 64 bit

Install dependencies

Before you can build the source, you need several tools and dependencies for the application to get started.

First, update your repositories. Not all packages are provided in the main Ubuntu repository, those you'll get from the Ethereum PPA and the LLVM archive:

```
sudo apt-get -y update
sudo apt-get -y install language-pack-en-base
sudo dpkg-reconfigure locales
sudo apt-get -y install software-properties-common
wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add -
sudo add-apt-repository "deb http://llvm.org/apt/trusty/ llvm-toolchain-trusty-3.5-binaries main"
sudo add-apt-repository -y ppa:ethereum/ethereum-qt
sudo add-apt-repository -y ppa:ethereum/ethereum
sudo add-apt-repository -y ppa:ethereum/ethereum-dev
sudo apt-get -y update
sudo apt-get -y upgrade
```

You'll need to install the following develop packages if you want to build the full suite, including the GUI components:

```
sudo apt-get -y install build-essential g++-4.8 git cmake libboost-all-dev automake unzip libgmp-dev libtool libleveldb
```

Optional: If you're only wanting to build in headless mode (i.e. no GUI), you won't need any of the Qt-specific dependencies, so instead of the last step, just do:

```
sudo apt-get -y install build-essential g++-4.8 git cmake libboost-all-dev automake unzip libgmp-dev libtool libleveldb
```

Choose your source

First grab/unpack the sources. If you want to build the latest version, clone the repository:

```
git clone https://github.com/ethereum/cpp-ethereum
cd cpp-ethereum
git checkout develop
```

If you have a prepackaged source distribution, then simply unpack:

```
tar xjf cpp-ethereum-<version>.tar.bz2
cd cpp-ethereum-<version>
```

Build

Create and configure the build environment and the build inside the cpp-ethereum directory:

```
mkdir build
cd build
cmake ..
```

Optional: You can configure the build in several ways by passing parameters to the cmake command above:

```
cmake .. -DCMAKE_BUILD_TYPE=Release -DBUNDLE=minimal # Compile minimal amount, just enough for a node, enable compiler
cmake .. -DCMAKE_BUILD_TYPE=Debug -DBUNDLE=user -DFATDB=1 # Compile enough for normal usage and with support for the fu
cmake .. -DBUNDLE=full -DGUI=0 # builds only the commandline-clients; no need to bother with the Qt dependencies.
```

Now, build the source tree:

```
make -j2
```

You can increase `2` to `4` or even `8` to make it build considerably faster on decent machines.

Start your client

Command line interface client (see [[Using Ethereum CLI Client]]):

```
cd ~/cpp-ethereum/build/eth
./eth
```

Once done, you might then [[Configure a Server]] and start a [[Local Test Net]].

Optional: To test that the JSON RPC interface is working, launch with `eth -j` instead, and check the coinbase with cURL:

```
curl -X POST --data '{"jsonrpc": "2.0","method": "eth_coinbase","params": null,"id": 1}' http://localhost:8080
```

Start AlethZero to run the experimental Ethereum GUI client (also see [[Using AlethZero]]).

```
cd ~/cpp-ethereum/build/alethzero
./alethzero
```

If you're updating from a previous version and you find you get errors when running, delete your old block chain before restarting:

```
rm -rf ~/.ethereum
```

Optional: To enable support for the common Ethereum unit tests (CEUT), clone the tests repository into the same path as cpp-ethereum and checkout the develop branch:

```
git clone https://github.com/ethereum/tests
cd tests
git checkout develop
cd ..
```

Building and installing a cpp-ethereum node server on Ubuntu 14.04, 64 bit

Copy and paste the following into a terminal:

```
wget http://opensecrecy.com/setupbuildeth.sh && source ./setupbuildeth.sh BRANCH NODE_IP NODE_NAME && rm -f setupbuilde
```

- `BRANCH` should be substituted for either `master` or `develop`, depending on whether you want a stable or bleeding-edge version.
- `NODE_IP` should be substituted for the 4-digit, dot-deliminated IP address of the node. For example `1.2.3.4` or `192.168.1.69`.
- `NODE_NAME` should be substituted for the name of the node, quoted if it contains spaces. Avoid using symbols. e.g. `"Gavs Server Node"` or `Release_Node_1`.

Wait for it to reboot and you'll be running a node.

Instructions for other versions

As of December 2014, we have changed the build system. The following instructions are untested. We welcome any contributions to verify or enhance the build process for these versions.

Ubuntu 14.04 with manual Qt installation

Web Install Qt version >= 5.4

Qt is for the GUI. It's not needed for the headless build. You can download it from their website, or use:

```
# For 32-bit:
wget http://download.qt-project.org/official_releases/online_installers/qt-opensource-linux-x86-online.run
# For 64-bit:
wget http://download.qt-project.org/official_releases/online_installers/qt-opensource-linux-x64-online.run
```

Run with:

```
chmod +x qt-opensource-linux-???-online.run
./qt-opensource-linux-???-online.run
```

When the installer asks you for the desired version, make sure to install at minimum version 5.4 and not any older version. Note the installation directory

To prepare the environment for the new Qt installation (and prevent it using any previous Qt install), we'll tell cmake we prefer this version:

```
export CMAKE_PREFIX_PATH=<Qt installation directory>/Qt/5.*/gcc
```

Wheezy 13.04 and Saucy 13.10

Here the necessary packages are almost the same, however since some dependencies are not available in these releases, the installation is a little different:

```
sudo apt-get install build-essential g++-4.8 git cmake libboost1.53-all-dev # for build
sudo apt-get install libncurses5-dev automake libtool unzip libgmp-dev libleveldb-dev yasm libminiupnpc-dev libreadline
sudo apt-get install libcurl4-openssl-dev # for json-rpc serving client
```

Precise 12.04

Pretty problematic building/installation, with plenty of issues. Highly recommended to upgrade to the 14.04 (which is also a LTS version). Still, an intense review (aiming to be complete) of the possible issues can be found in the "Compatibility Info and Build Tips" of this wiki.

Other advises: For this release we need some backports. See http://www.swiftsoftwaregroup.com/upgrade-gcc-4-7-ubuntu-12-04/ for GCC 4.7, which is required. You will need these PPAs and these libleveldb libraries:

```
sudo apt-add-repository ppa:ubuntu-sdk-team/ppa && sudo apt-add-repository ppa:apokluda/boost1.53
wget http://launchpadlibrarian.net/148520969/libleveldb-dev_1.13.0-1_amd64.deb && wget http://launchpadlibrarian.net/14
```

More info in the "Compatibility Info and Build Tips" of this wiki.

Option 1: Install from PPA

For the latest development snapshot, both `ppa:ethereum/ethereum` and `ppa:ethereum/ethereum-dev` are needed. If you want the stable version from the last PoC release, simply omit the `-dev` one.

**Warning: The `ethereum-qt` PPA will upgrade your system-wide Qt5 installation, from 5.2 on Trusty and 5.3 on Utopic, to 5.4.*

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:ethereum/ethereum-qt
sudo add-apt-repository ppa:ethereum/ethereum
sudo add-apt-repository ppa:ethereum/ethereum-dev
sudo apt-get update
sudo apt-get install ethereum
```

Run `mist` for the GUI or `geth` for the CLI.

You can alternatively install only the CLI or GUI, with `apt-get install geth` or `apt-get install mist` respectively.

Option 2: Automatic installation

Note Outdated, please use the PPA.

This Mist install script will install everything required from a fresh Ubuntu 14.04 installation and start running Mist.

```
wget https://gist.githubusercontent.com/tgerring/d4ab3f1672ed91a53c6c/raw/677a3dd9c6db099eee620657bf7fb1e664173ee1/mist
chmod +x install
./install
```

Option 3: Manual build from source

Installing Go

Verify that Go is installed by running `go version` and checking the version. If not, see Installing Go

Prerequisites

Mist depends on the following external libraries to be installed:

- GMP
- Readline
- Qt 5.4.

First install GMP & Readline from repositories:

```
sudo apt-get install -y libgmp3-dev libreadline6-dev
```

Second, follow the instructions for Building Qt

Installing Mist

At last you will now be finished with all the prerequisites. The following commands will build the Ethereum Mist GUI client for you:

```
go get -u github.com/ethereum/go-ethereum/cmd/mist
```

(You may need to run "sudo apt-get install mercurial" first)

Note: Mist does not automatically look in the right location for its GUI assets. For this reason you have to launch it from its build directory

```
cd $GOPATH/src/github.com/ethereum/go-ethereum/cmd/mist && mist
```

or supply an absolute `-asset_path` option:

```
mist -asset_path $GOPATH/src/github.com/ethereum/go-ethereum/cmd/mist/assets
```

To eliminate the need to remember this cumbersome command, you can create the following a file in $GOPATH/bin :

```
#!/bin/bash
cd $GOPATH/src/github.com/ethereum/go-ethereum/cmd/mist && mist
```

Name the file 'misted' and make it executable:

```
chmod +x $GOPATH/bin/misted
```

Now mist can be run with the command `misted`

Building on Mac OS

Although Mavericks is required for compiling the app, the app has been tested to run on 10.8, 10.9 and 10.10.

Build with Homebrew

For the stable branch:

```
brew tap ethereum/ethereum
brew install cpp-ethereum --build-from-source
brew linkapps cpp-ethereum
```

For the development branch:

```
brew reinstall cpp-ethereum --devel --build-from-source
brew linkapps cpp-ethereum
```

Add the `--with-gui` option to also build AlethZero and the Mix IDE.

Then `open /Applications/AlethZero.app`, `eth` (CLI) or `neth` (ncurses interface)

For options and patches, see: https://github.com/ethereum/homebrew-ethereum

Manual Build

Prerequisites

- Latest Xcode on Mavericks (at least Xcode 5.1+).
- Homebrew package manager http://brew.sh *or* MacPorts package manager http://macports.org
- XQuartz http://xquartz.macosforge.org/landing/

Install dependencies

Using homebrew:

```
brew install boost --c++11 # this takes a while
brew install boost-python --c++11
brew install cmake qt5 cryptopp miniupnpc leveldb gmp libmicrohttpd libjson-rpc-cpp
```

Or, using MacPorts:

```
port install boost +universal # this takes a while
port install cmake qt5-mac libcryptopp miniupnpc leveldb gmp
(probably missing libmicrohttpd and libjson-rpc-cpp)
```

Clone source repo and create build folder

```
git clone https://github.com/ethereum/cpp-ethereum.git
cd cpp-ethereum
mkdir -p build # create build folder ('build' ignored by git)
```

Compile

```
cd build
cmake ..
make -j6
make install
```

This will also install the cli tool and libs into /usr/local.

XCode Instructions

From the project root:

```
mkdir build_xc
cd build_xc
cmake -G Xcode ..
```

This will generate an xcode project file along with some configs for you: ethereum.xcodeproj . Open this file in XCode and you should be able to build the project

Troubleshooting

Linking errors can usually be resolved by ensuring correct paths:

```
export LIBRARY_PATH=/opt/local/lib # this is MacPorts's default
```

Make sure your `macdeployqt` can be found. Ethereum expects it to be in `/usr/local/opt/qt5/bin/macdeployqt` so symlink it if it's not:

```
ln -s `which macdeployqt` /usr/local/opt/qt5/bin/macdeployqt
```

If you're not planning to develop, make sure you are building from the master branch, not from develop or any others. These branches can often be broken or have other requirements.

Option 1: Homebrew tap

Assumptions:

- Brew is installed. If not, run:

  ```
  ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
  ```

  and follow the prompts.

```
brew tap ethereum/ethereum
brew install ethereum
```

Install the latest develop branch with `--devel`.

Add `--with-gui` to brew the `mist` browser.

Then run `mist` (GUI) or `geth` (CLI)

For options and patches, see: https://github.com/ethereum/homebrew-ethereum

Option 2: Manual installation

Assumptions:

- Go 1.2+ is installed. If not refer to this page.
- Brew is installed. If not, run `ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"` and follow the prompts

Install Qt 5.4.x:

`brew install qt5`

Now configure some path for go-qml to properly build:

```
export PKG_CONFIG_PATH=/usr/local/opt/qt5/lib/pkgconfig
export CGO_CPPFLAGS="-I/usr/local/opt/qt5/include/QtCore"
export LD_LIBRARY_PATH=/usr/local/opt/qt5/lib
```

Compile **Mist**

```
go get github.com/ethereum/go-ethereum/cmd/mist
```

This should start Mist and connect you to the Ethereum network. However, if you want a traditional `*.app` file we'll need to use our build tool.

```
git clone git@github.com:ethereum/go-build.git
cd go-build/osx && python build.py
```

This will save a `Mist.app` in the **osx** directory.

Option 3: Building from source easy guide

WARNING: Guide is outdated but remains for references

We now have a simple tutorial to install from source: http://forum.ethereum.org/discussion/905/go-ethereum-cli-ethereal-simple-build-guide-for-osx

Building on Windows

With CMake and Visual Studio Express 12.0 (VS2013)

Initial configuration

Ensure you have installed:

- CMake 3.0+ - The cross-platform, open-source make system.
- Visual Studio Express 2013 for Windows Desktop - VS2013 for Windows Desktop is a minimum as cpp-ethereum uses languages features not supported in 2012. The Professional edition also works. (tested 10/2014)
- Windows 7 SDK or Windows 8 SDK - for files like winsock2.h and gdi32.lib
- Git for Windows 1.9.0+ - This for retrieving source from Github.
- 7-Zip - Ensure that 7z.exe has been added to the PATH.

Get the source

Execute the following step to download the latest source code

```
C:\> git clone https://github.com/ethereum/cpp-ethereum.git
```

Retrieve dependecies

Execute the following commands within a command shell to download all dependencies:

1. Generate solution with CMake

```
cd extdep
C:\cpp-ethereum\extdep> cmake . -G "Visual Studio 12"
```

2. MSBuild

```
C:\cpp-ethereum\extdep> msbuild project.sln /p:Configuration=Release
```

Note: The 'Debug'-configuration is also supported.

Build Ethereum project

Execute the following commands within a command shell to build the project:

3. Generate solution with CMake

First generate the ethereum solution and visual studio project files

```
cd ..
mkdir build
cd build
C:\cpp-ethereum\build> cmake .. -G "Visual Studio 12"
```

Note: In-source build (running cmake in root directory) is not currently supported.

4. Compile with MSBuild

```
C:\cpp-ethereum\build> msbuild ethereum.sln /p:Configuration=Release
```

Note: The 'Debug'-configuration is also supported.

And it should now be possible to compile and debug by opening the generated Ethereum solution with Visual Studio 2013.

Fresh builds

After setting up the dependencies and running your first successful build, unless there are changes to any of the dependencies, you can make a fresh build from the latest code much faster by just fetching the new code and rebuilding.

1. In the directory `C:\cpp-ethereum>`, run the command `git pull`.
2. Re-run steps 3 & 4.

Troubleshooting

Need to create a wiki page with instructions for windows Go

The Ethereum APIs

Ethereum JSON RPC

Overview

JSON is a lightweight data-interchange format. It can represent numbers, strings, ordered sequences of values, and collections of name/value pairs.

JSON-RPC is a stateless, light-weight remote procedure call (RPC) protocol. Primarily this specification defines several data structures and the rules around their processing. It is transport agnostic in that the concepts can be used within the same process, over sockets, over HTTP, or in many various message passing environments. It uses JSON (RFC 4627) as data format.

JavaScript API

To talk to an ethereum node from inside an JavaScript application use the ethereum.js library, which gives an convenient interface for the RPC methods. See the JavaScript API for more.

JSON-RPC Endpoint

Default JSON-RPC endpoints:

```
C++: http://localhost:8080
Go: http://localhost:8545
```

Go

You can start the HTTP JSON-RPC with the `--rpc` flag

```
geth --rpc
```

change the default port (8545) and listing address (localhost) with:

```
geth --rpc --rpcaddr <ip> --rpcport <portnumber>
```

If accessing the RPC from a browser, CORS will need to be enabled with the appropriate domain set. Otherwise, JavaScript calls are limit by the same-origin policy and requests will fail:

```
geth --rpc --rpccorsdomain "http://localhost:3000"
```

The JSON RPC can also be started from the geth console using the `admin.startRPC(addr, port)` command.

C++

You can start it by running `eth` application with `-j` option:

```
./eth -j
```

You can also specify JSON-RPC port (default is 8080):

```
./eth -j --json-rpc-port 8079
```

JSON-RPC support

	cpp-ethereum	go-ethereum
JSON-RPC 1.0	✓	
JSON-RPC 2.0	✓	✓
Batch requests	✓	✓
HTTP	✓	✓

Output HEX values

At present there are two key datatypes that are passed over JSON: unformatted byte arrays and quantities. Both are passed with a hex encoding, however with different requirements to formatting:

When encoding **QUANTITIES** ("integers", "numbers"): encode as hex, prefix with "0x", the most compact representation (slight exception: zero should be represented as "0x0"). Examples:

- 0x41 (65 in decimal)
- 0x400 (1024 in decimal)
- WRONG: 0x (should always have at least one digit - zero is "0x0")
- WRONG: 0x0400 (no leading zeroes allowed)
- WRONG: ff (must be prefixed 0x)

When encoding **UNFORMATTED DATA** ("byte arrays", account addresses, hashes, bytecode arrays): encode as hex, prefix with "0x", two hex digits per byte. Examples:

- 0x41 (size 1, "A")
- 0x004200 (size 3, "\0B\0")
- 0x (size 0, "")
- WRONG: 0xf0f0f (must be even number of digits)
- WRONG: 004200 (must be prefixed 0x)

Currently cpp-ethereum and go-ethereum provides JSON-RPC communication only over http.

The default block parameter

The following methods have a extra default block parameter:

- eth_getBalance
- eth_getCode
- eth_getTransactionCount
- eth_getStorageAt
- eth_call

When requests are made that act on the state of ethereum, the last default block parameter determines the height of the block.

The following options are possible for the defaultBlock parameter:

- `HEX String` - an integer block number
- `String "latest"` - for the latest minded block
- `String "earliest"` for the earliest/genesis block
- `String "pending"` - for the pending state/transactions

JSON-RPC methods

- web3_clientVersion
- web3_sha3
- net_version
- net_peerCount
- net_listening
- eth_protocolVersion
- eth_coinbase
- eth_mining
- eth_gasPrice
- eth_accounts
- eth_blockNumber
- eth_getBalance
- eth_getStorageAt
- eth_getTransactionCount
- eth_getBlockTransactionCountByHash
- eth_getBlockTransactionCountByNumber
- eth_getUncleCountByBlockHash
- eth_getUncleCountByBlockNumber
- eth_getCode
- eth_sendTransaction
- eth_call
- eth_getBlockByHash
- eth_getBlockByNumber
- eth_getTransactionByHash
- eth_getTransactionByBlockHashAndIndex
- eth_getTransactionByBlockNumberAndIndex
- eth_getUncleByBlockHashAndIndex
- eth_getUncleByBlockNumberAndIndex
- eth_getCompilers
- eth_compileLLL
- eth_compileSolidity
- eth_compileSerpent
- eth_newFilter
- eth_newBlockFilter
- eth_uninstallFilter
- eth_getFilterChanges
- eth_getFilterLogs
- eth_getLogs
- eth_getWork
- eth_submitWork
- db_putString
- db_getString
- db_putHex
- db_getHex
- shh_post
- shh_version

- shh_newIdentity
- shh_hasIdentity
- shh_newGroup
- shh_addToGroup
- shh_newFilter
- shh_uninstallFilter
- shh_getFilterChanges
- shh_getMessages

API

---

web3_clientVersion

Returns the current client version.

Parameters

none

Returns

`String` - The current client version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc":"2.0",
  "result": "Mist/v0.9.3/darwin/go1.4.1"
}
```

---

web3_sha3

Returns SHA3 of the given data.

Parameters

1. `String` - the data to convert into a SHA3 hash

```
params: [
  '0x68656c6c6f20776f726c64'
]
```

Returns

`String` - The SHA3 result of the given string.

Example

```
// Request
```

```
curl -X POST --data '{"jsonrpc":"2.0","method":"web3_sha3","params":["0x68656c6c6f20776f726c64"],"id":64}'

// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0x47173285a8d7341e5e972fc677286384f802f8ef42a5ec5f03bbfa254cb01fad"
}
```

net_version

Returns the current network protocol version.

Parameters

none

Returns

`String` - The current network protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_version","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "59"
}
```

net_listening

Returns `true` if client is actively listening for network connections.

Parameters

none

Returns

`Boolean` - `true` when listening, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_listening","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc":"2.0",
  "result":true
}
```

net_peerCount

Returns number of peers currenly connected to the client.

Parameters

none

Returns

`HEX String` - integer of the number of connected peers.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"net_peerCount","params":[],"id":74}'

// Result
{
  "id":74,
  "jsonrpc": "2.0",
  "result": "0x2" // 2
}
```

---

eth_protocolVersion

Returns the current ethereum protocol version.

Parameters

none

Returns

`String` - The current ethereum protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_protocolVersion","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "54"
}
```

---

eth_coinbase

Returns the client coinbase address.

Parameters

none

Returns

`HEX String` - the current coinbase address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_coinbase","params":[],"id":64}'

// Result
{
  "id":64,
  "jsonrpc": "2.0",
  "result": "0x407d73d8a49eeb85d32cf465507dd71d507100c1"
}
```

---

eth_mining

Returns `true` if client is actively mining new blocks.

Parameters

none

Returns

`Boolean` - returns `true` of the client is mining, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_mining","params":[],"id":71}'

// Result
{
  "id":71,
  "jsonrpc": "2.0",
  "result": true
}
```

---

eth_gasPrice

Returns the current price per gas in wei.

Parameters

none

Returns

`HEX String` - integer of the current gas price in wei.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_gasPrice","params":[],"id":73}'

// Result
{
  "id":73,
```

```
  "jsonrpc": "2.0",
  "result": "0x09184e72a000" // 10000000000000
}
```

eth_accounts

Returns a list of addresses owned by client.

Parameters

none

Returns

`Array of HEX Strings` - addresses owned by the client.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_accounts","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["0x407d73d8a49eeb85d32cf465507dd71d507100c1"]
}
```

eth_blockNumber

Returns the number of most recent block.

Parameters

none

Returns

`HEX String` - integer of the current block number the client is on.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_blockNumber","params":[],"id":83}'

// Result
{
  "id":83,
  "jsonrpc": "2.0",
  "result": "0x4b7" // 1207
}
```

eth_getBalance

Returns the balance of the account of given address.

Parameters

1. `HEX String` - address to check for balance.
2. `HEX String|String` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest'
]
```

Returns

`HEX String` - integer of the current balance in wei.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBalance","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1",

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0234c8a3397aab58" // 158972490234375000
}
```

eth_getStorageAt

Returns the value from a storage position at a given address.

Parameters

1. `HEX String` - address of the storage.
2. `HEX String` - integer of the position in the storage.
3. `HEX String|String` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   '0x0', // storage position at 0
   '0x2' // state at block number 2
]
```

Returns

`HEX String` - the value at this storage position.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getStorageAt","params":["0x407d73d8a49eeb85d32cf465507dd71d507100c1

// Result
{
  "id":1,
```

```
  "jsonrpc": "2.0",
  "result": "0x03"
}
```

eth_getTransactionCount

Returns the number of transactions *send* from a address.

Parameters

1. `HEX String` - address.
2. `HEX String|String` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1',
   'latest' // state at the latest block
]
```

Returns

`HEX String` - integer of the number of transactions send from this address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionCount","params":["0x407d73d8a49eeb85d32cf465507dd71d5

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getBlockTransactionCountByHash

Returns the number of transactions in a block from a block matching the given block hash.

Parameters

1. `HEX String` - hash of a block

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1'
]
```

Returns

`HEX String` - integer of the number of transactions in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByHash","params":["0x407d73d8a49eeb85d32cf4

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xb" // 11
}
```

eth_getBlockTransactionCountByNumber

Returns the number of transactions in a block from a block matching the given block number.

Parameters

1. `HEX String` - integer of a block number, or the string "latest", "earliest" or "pending", see the default block parameter

```
params: [
   '0xe8', // 232
]
```

Returns

`HEX String` - integer of the number of transactions in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockTransactionCountByNumber","params":["0xe8"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xa" // 10
}
```

eth_getUncleCountByBlockHash

Returns the number of uncles in a block from a block matching the given block hash.

Parameters

1. `HEX String` - hash of a block

```
params: [
   '0x407d73d8a49eeb85d32cf465507dd71d507100c1'
]
```

Returns

`HEX String` - integer of the number of uncles in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockHash","params":["0x407d73d8a49eeb85d32cf465507d

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getUncleCountByBlockNumber

Returns the number of uncles in a block from a block matching the given block number.

Parameters

1. `HEX String` - integer of a block number, or the string "latest", "earliest" or "pending", see the default block parameter

```
params: [
   '0xe8', // 232
]
```

Returns

`HEX String` - integer of the number of uncles in this block.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleCountByBlockNumber","params":["0xe8"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_getCode

Returns code at a given address.

Parameters

1. address as hex string
2. `HEX String|String` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

```
params: [
   '0xd5677cf67b5aa051bb40496e68ad359eb97cfbf8',
   '0x2'  // 2
]
```

Returns

`HEX String` - the code from the given address.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getCode","params":["0xd5677cf67b5aa051bb40496e68ad359eb97cfbf8", "0

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f25b600060078202905091905056"
}
```

eth_sendTransaction

Creates new message call transaction or a contract creation, if the data field contains code.

Parameters

1. `Object` - The transaction object
   - `from`: `HEX String` - The address the transaction is send from.
   - `to`: `HEX String` - (optional when creating new contract) The address the transaction is directed to.
   - `gas`: `HEX String` - (optional, default: To-Be-Determined) Integer of the gas provided for the transaction execution. It will return unused gas.
   - `gasPrice`: `HEX String` - (optional, default: To-Be-Determined) Integer of the gasPrice used for each payed gas
   - `value`: `HEX String` - (optional) Integer of the value send with this transaction
   - `data`: `HEX String` - (optional) The compiled code of a contract

```
params: [{
  "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
  "to": "0xd46e8dd67c5d32be8058bb8eb970870f072445675",
  "gas": "0x76c0", // 30400,
  "gasPrice": "0x9184e72a000", // 10000000000000
  "value": "0x9184e72a", // 2441406250
  "data": "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"
}]
```

Returns

`HEX String` - the address of the newly created contract, or the transaction hash.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_sendTransaction","params":[{see above}],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x7f9fade1c0d57a7af66ab4ead7c2eb7b11a91385"
}
```

eth_call

Executes a new message call immediately without creating a transaction on the block chain.

Parameters

1. `Object` - The transaction call object
   - `from`: `HEX String` - The address the transaction is send from.
   - `to`: `HEX String` - The address the transaction is directed to.
   - `gas`: `HEX String` - (optional) Integer of the gas provided for the transaction execution. It will return unused gas.
   - `gasPrice`: `HEX String` - (optional) Integer of the gasPrice used for each payed gas
   - `value`: `HEX String` - (optional) Integer of the value send with this transaction
   - `data`: `HEX String` - (optional) The compiled code of a contract
2. `HEX String|String` - integer block number, or the string `"latest"`, `"earliest"` or `"pending"`, see the default block parameter

See: eth_sendTransaction Parameters

Returns

`HEX String` - the return value of executed contract.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_call","params":[{see above}],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x0"
}
```

---

eth_getBlockByHash

Returns information about a block by hash.

Parameters

1. `HEX String` - Hash of a block.
2. `Boolean` - If `true` it returns the full transaction objects, if `false` only the hashes of the transactions.

```
params: [
   '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
   true
]
```

Returns

`Object` - A block object, or `null` when no transaction was found:

- `number`: `HEX String` - integer of the block number.
- `hash`: `HEX String` - 32-byte hash of the block.
- `parentHash`: `HEX String` - 32-byte hash of the paretn block.

- `nonce` : `HEX String` - 8-byte hash of the generated proof-of-work.
- `sha3Uncles` : `HEX String` - SHA3 of all uncle (?).
- `logsBloom` : `HEX String` - bloom (?).
- `transactionsRoot` : `HEX String` - (?).
- `stateRoot` : `HEX String` - (?).
- `miner` : `HEX String` - the address of the miner, who minded this block.
- `difficulty` : `HEX String` - integer of the difficulty for this block.
- `totalDifficulty` : `HEX String` - integer of the total difficulty up until this block (?).
- `extraData` : `HEX String` - byte array of extra data add to this block (?),
- `size` : `HEX String` - integer the size of this block in bytes.
- `gasLimit` : `HEX String` - integer of the maximum gas allowed in this block (?).
- `minGasPrice` : `HEX String` - the minimal (?).
- `gasUsed` : `HEX String` - the total used gas by all transactions in this block (?).
- `timestamp` : `HEX String` - the unix timestamp when the block was mined (?).
- `transactions` : `Array` - Array of transaction objects, or transaction hashes depending on the last parameter.
- `uncles` : `Array` - Array of uncle hashes.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByHash","params":["0xe670ec64341771606e55d6b4ca35a1a6b75ee3

// Result
{
"id":1,
"jsonrpc":"2.0",
"result": {
    "number": "0x1b4", // 436
    "hash": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "parentHash": "0x9646252be9520f6e71339a8df9c55e4d7619deeb018d2a3f2d21fc165dde5eb5",
    "nonce": "0xe04d296d2460cfb8472af2c5fd05b5a214109c25688d3704aed5484f9a7792f2",
    "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
    "logsBloom": "0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331",
    "transactionsRoot": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421",
    "stateRoot": "0xd5855eb08b3387c0af375e9cdb6acfc05eb8f519e419b874b6ff2ffda7ed1dff",
    "miner": "0x4e65fda2159562a496f9f3522f89122a3088497a",
    "difficulty": "0x027f07", // 163591
    "totalDifficulty":  "0x027f07", // 163591
    "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000",
    "size":  "0x027f07", // 163591
    "gasLimit": "0x9f759", // 653145
    "minGasPrice": "0x9f759", // 653145
    "gasUsed": "0x9f759", // 653145
    "timestamp": "0x54e34e8e" // 1424182926
    "transactions": [{...},{ ... }]
    "uncles": ["0x1606e5...", "0xd5145a9..."]
  }
}
```

eth_getBlockByNumber

Returns information about a block by block number.

Parameters

1. `HEX String` - integer of a block number.
2. `Boolean` - If `true` it returns the full transaction objects, if `false` only the hashes of the transactions.

```
params: [
```

```
    '0x1b4', // 436
    true
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getBlockByNumber","params":["0x1b4", true],"id":1}'
```

Result see eth_getBlockByHash

---

eth_getTransactionByHash

Returns the information about a transaction requested by transaction hash.

Parameters

1. `HEX String` - hash of a transaction

```
params: [
   '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331'
]
```

Returns

`Object` - A transaction object, or `null` when no transaction was found:

- `hash`: `HEX String` - 32-byte hash of the transaction.
- `nonce`: `HEX String` - integer of the transaction nonce (?).
- `blockHash`: `HEX String` - 32-byte hash of the block where this transaction was in. `null` when the transaction is pending.
- `blockNumber`: `HEX String` - integer of the block number where this transaction was in. `null` when the transaction is pending.
- `transactionIndex`: `HEX String` - integer of the transactions index position in the block.
- `from`: `HEX String` - 20-byte address of the sender.
- `to`: `HEX String` - 20-byte address of the receiver. `null` when its a contract creation transaction.
- `value`: `HEX String` - integer of the value transfered in wei.
- `gasPrice`: `HEX String` - integer of the price payed per gas in wei.
- `gas`: `HEX String` - integer of the gas used.
- `input`: `HEX String` - the data send along with the transaction.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByHash","params":["0xc6ef2fc5426d6ad6fd9e2a26abeab0aa

// Result
{
"id":1,
"jsonrpc":"2.0",
"result": {
```

```
    "hash":"0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
    "nonce":"0x",
    "blockHash": "0xbeab0aa2411b7ab17f30a99d3cb9c6ef2fc5426d6ad6fd9e2a26a6aed1d1055b",
    "blockNumber": "0x15df", // 5599
    "transactionIndex":  "0x1", // 1
    "from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1",
    "to":"0x85h43d8a49eeb85d32cf465507dd71d507100c1",
    "value":"0x7f110" // 520464
    "gas": "0x7f110" // 520464
    "gasPrice":"0x09184e72a000",
    "input":"0x603880600c6000396000f300603880600c6000396000f3603880600c6000396000f360",
  }
}
```

eth_getTransactionByBlockHashAndIndex

Returns information about a transaction by block hash and transaction index position.

Parameters

1. `HEX String` - hash of a block.
2. `HEX String` - integer of the transaction index position.

```
params: [
   '0xe670ec64341771606e55d6b4ca35a1a6b75ee3d5145a99d05921026d1527331',
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockHashAndIndex","params":[0xc6ef2fc5426d6ad6fd9e
```

Result see eth_getTransactionByHash

eth_getTransactionByBlockNumberAndIndex

Returns information about a transaction by block number and transaction index position.

Parameters

1. `HEX String` - integer of a block number.
2. `HEX String` - integer of the transaction index position.

```
params: [
   '0x29c', // 668
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getTransactionByBlockNumberAndIndex","params":["0x29c", "0x0"],"id'
```

Result see eth_getTransactionByHash

---

eth_getUncleByBlockHashAndIndex

Returns information about a uncle of a block by hash and uncle index position.

Parameters

1. `HEX String` - hash a block.
2. `HEX String` - integer of the uncle index position.

```
params: [
   '0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b',
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockHashAndIndex","params":["0xc6ef2fc5426d6ad6fd9e2a26a
```

Result see eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

---

eth_getUncleByBlockNumberAndIndex

Returns information about a uncle of a block by number and uncle index position.

Parameters

1. `HEX String` - integer a block number.
2. `HEX String` - integer of the uncle index position.

```
params: [
   '0x29c', // 668
   '0x0' // 0
]
```

Returns

See eth_getBlockByHash

Note: An uncle doesn't contain individual transactions.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getUncleByBlockNumberAndIndex","params":["0x29c", "0x0"],"id":1}'
```

Result see eth_getBlockByHash

eth_getCompilers

Returns a list of available compilers in the client.

Parameters

none

Returns

`Array` - Array of available compilers.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getCompilers","params":[],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": ["solidity", "lll", "serpent"]
}
```

eth_compileSolidity

Returns compiled solidity code.

Parameters

1. `String` - The source code.

```
params: [
   "contract test { function multiply(uint a) returns(uint d) {   return a * 7;   } }",
]
```

Returns

`HEX String` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["contract test { function multiply(uint

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b806000526020
}
```

eth_compileLLL

Returns compiled LLL code.

Parameters

1. `String` - The source code.

```
params: [
   "(?)",
]
```

Returns

`HEX String` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["(?)"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b806000526020
}
```

eth_compileSerpent

Returns compiled serpent code.

Parameters

1. `String` - The source code.

```
params: [
   "(?)",
]
```

Returns

`HEX String` - The compiled source code.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_compileSolidity","params":["(?)"],"id":1}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b806000526020
}
```

eth_newFilter

Creates a filter object, based on filter options, to notify when the state changes (logs). To check if the state has changed, call eth_getFilterChanges.

Parameters

1. `Object` - The filter options:
   - `fromBlock`: `HEX String|String` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
   - `toBlock`: `HEX String|String` - (optional, default: `"latest"`) Integer block number, or `"latest"` for the last mined block or `"pending"`, `"earliest"` for not yet mined transactions.
   - `address`: `HEX String|Array` - (optional) Contract address or a list of addresses from which logs should originate.
   - `topics`: `Array` - (optional) Array of `HEX Strings` topics.

```
params: [{
  "fromBlock": "0x1",
  "toBlock": "0x2",
  "address": "0x01231f12a01231f12a01ff231f12a0123101231f12a",
  "topics": ['0x1234fa1234']
}]
```

Returns

`HEX String` - The integer of a filter id.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newFilter","params":[{"topics":["0x12341234"]}],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0x1" // 1
}
```

eth_newBlockFilter

Creates a filter object, based on an option string, to notify when state changes (logs). To check if the state has changed, call eth_getFilterChanges.

Parameters

1. `String` - The string `"latest"` for notifications about new block and `"pending"` for notifications about pending transactions.

```
params: ["pending"]
```

Returns

`HEX String` - The integer of a filter id.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_newBlockFilter","params":["pending"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":  "2.0",
  "result": "0x1" // 1
}
```

eth_uninstallFilter

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with eth_getFilterChanges for a period of time.

Parameters

1. `HEX String` - The filter id.

```
params: [
  "0xb" // 11
]
```

Returns

`Boolean` - `true` if the filter was successfully uninstalled, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_uninstallFilter","params":["0xb"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

eth_getFilterChanges

Polling method for a filter, which returns an array of logs which occurred since last poll.

Parameters

1. `HEX String` - Integer of the filter id.

```
params: [
  "0x16" // 22
]
```

Returns

`Array` - Array of log objects, or an empty array (if nothing has changed since last poll).

For filters created with `eth_newBlockFilter` log objects are `null` .

For filters created with `eth_newFilter` logs are objects with following params:

- `logIndex` : `HEX String` - integer of the log index position in the block.
- `transactionIndex` : `HEX String` - integer of the transactions index position log was created from.
- `transactionHash` : `HEX String` - hash of the transactions this log was created from.
- `blockHash` : `HEX String` - 32-byte hash of the block where this log was in. `null` when the log is pending.
- `blockNumber` : `HEX String` - integer of the block number where this log was in. `null` when the log is pending.
- `address` : `HEX String` - address from which this log originated.
- `data` : `HEX String` - contains the non-indexed arguments of the log.
- `topics` : `Array` - Array of 0 to 4 `HEX Strings` of indexed log arguments. (In *solidity*: The first topic is the *hash* of the signature of the event (e.g. `Deposit(address,bytes32,uint256)`), except you declared the event with the `anonymous` specifier.)

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterChanges","params":["0x16"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
    "hash": "0x5785ac562ff41e2dcfdf8216c5785ac562ff41e2dcfdf829c5a142f1fccd7d",
    "logIndex": "0x1", // 1
    "blockNumber":"0x1b4" // 436
    "blockHash": "0x8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcfdf829c5a142f1fccd7d",
    "transactionHash":  "0xdf829c5a142f1fccd7d8216c5785ac562ff41e2dcfdf5785ac562ff41e2dcf",
    "transactionIndex": "0x0", // 0
    "address": "0x16c5785ac562ff41e2dcfdf829c5a142f1fccd7d",
    "data":"0x000000000000000000000000000000000000000000000000000000000000000000",
    "topics": ["0x59ebeb90bc63057b6515673c3ecf9438e5058bca0f92585014eced636878c9a5"]
    },{
      ...
    }]
}
```

eth_getFilterLogs

Returns an array of all logs matching filter with given id.

Parameters

1. `HEX String` - The filter id.

```
params: [
  "0x16" // 22
]
```

Returns

See eth_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getFilterLogs","params":["0x16"],"id":74}'
```

Result see eth_getFilterChanges

---

eth_getLogs

Returns an array of all logs matching a given filter object.

Parameters

1. `Object` - the filter object, see eth_newFilter parameters.

```
params: [{
  "topics": ["0x12341234"]
}]
```

Returns

See eth_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getLogs","params":[{"topics":["0x12341234"]}],"id":74}'
```

Result see eth_getFilterChanges

---

eth_getWork

Returns the hash of the current block, the seedHash, and the boundary condition to be met ("target").

Parameters

none

Returns

`Array` - Arrway with the following properties:

1. `HEX String` - current block header hash without the nonce (the first part of the proof-of-work pair) (?).
2. `HEX String` - the seed hash used for the DAG.
3. `HEX String` - the boundary condition ("target"), 2^256 / difficulty.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"eth_getWork","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [
      "0x1234567890abcdef1234567890abcdef",
      "0x5EED0000000000000000000000000000",
      "0xd1ff1c01710000000000000000000000"
    ]
}
```

eth_submitWork

Used for submitting a proof-of-work solution.

Parameters

1. `HEX String` - The nonce found (64 bits)
2. `HEX String` - The header's hash (256 bits)
3. `HEX String` - The mix digest (256 bits)

```
params: [
  "0x0000000000000001",
  "0x1234567890abcdef1234567890abcdef",
  "0xD1GE5700000000000000000000000000"
]
```

Returns

`Boolean` - returns `true` if the provided solution is valid, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0", "method":"eth_submitWork", "params":["0x0000000000000001", "0x1234567890abcdef12

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

db_putString

Stores a string in the local database.

Parameters

1. `String` - Database name.
2. `String` - Key name.
3. `String` - String to store.

```
params: [
  "testDB",
  "myKey",
  "myString"
]
```

Returns

`Boolean` - returns `true` if the value was stored, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_putString","params":["testDB","myKey","myString"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

---

db_getString

Returns string from the local database.

Parameters

1. `String` - Database name.
2. `String` - Key name.

```
params: [
  "testDB",
  "myKey",
]
```

Returns

`String` - The previously stored string.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_getString","params":["testDB","myKey"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
```

```
  "result": "myString"
}
```

db_putHex

Stores binary data in the local database.

Parameters

1. `String` - Database name.
2. `String` - Key name.
3. `HEX String` - HEX string to store.

```
params: [
  "testDB",
  "myKey",
  "0x68656c6c6f20776f726c64"
]
```

Returns

`Boolean` - returns `true` if the value was stored, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_putHex","params":["testDB","myKey","0x68656c6c6f20776f726c64"],"id":

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

db_getHex

Returns binary data from the local database.

Parameters

1. `String` - Database name.
2. `String` - Key name.

```
params: [
  "testDB",
  "myKey",
]
```

Returns

`HEX String` - The previously stored HEX string.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_getHex","params":["testDB","myKey"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": "0x68656c6c6f20776f726c64"
}
```

shh_version

Returns the current whisper protocol version.

Parameters

none

Returns

`String` - The current whisper protocol version

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_version","params":[],"id":67}'

// Result
{
  "id":67,
  "jsonrpc": "2.0",
  "result": "2"
}
```

shh_post

Sends a whisper message.

Parameters

1. `Object` - The whisper post object:
   - `from`：`HEX String` - (optional) The identity of the sender.
   - `to`：`HEX String` - (optional) The identity of the receiver. When present whisper will encrypt the message so that only the receiver can decrypt it.
   - `topics`：`Array` - Array of `HEX String` topics, so that receiver can identify messages.
   - `payload`：`HEX String` - The payload of the message.
   - `priority`：`HEX String` - The integer of the priority in a rang from ... (?).
   - `ttl`：`HEX String` - integer of the time to live in seconds.

```
params: [{
  from: "0x0487bc1d6bf3aa84d8a2c24865f83cd7bcfd11ccd8cb18d9dead364e...",
  to: "0x742d636c69656e776869737065bb722d63686174...",
  topics: ["0x776869737065722d636861742d636c69656e74", "0x4d5a695276454c39425154466b61693532"],
  payload: "0x7b2274797065223a226d6...",
  priority: "0x64",
  ttl: "0x64",
```

```
}]
```

Returns

`Boolean` - returns `true` if the message was send, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"db_getString","params":[{"from":"0xc931d93e97ab07fe42d923478ba2465f2..'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

shh_newIdentinty

Creates new whisper identity in the client.

Parameters

none

Returns

`HEX String` - the address of the new identiy.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newIdentinty","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc931d93e97ab07fe42d923478ba2465f283f440fd6cabea4dd7a2c807108f651b7135d1d6ca9007d5b68aa497e4619ac10aa3b27
}
```

shh_hasIdentity

Checks if the client hold the private keys for a given identity.

Parameters

1. `HEX String` - The identity address to check.

```
params: [
  "0xc931d93e97ab07fe42d9234..."
]
```

Returns

`Boolean` - returns `true` if the client holds the privatekey for that identity, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_hasIdentity","params":["0xc931d93e97ab07fe42d923478ba2465f283..."],

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

shh_newGroup

(?)

Parameters

none

Returns

`HEX String` - the address of the new group.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newIdentinty","params":[],"id":73}'

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": "0xc65f283f440fd6cabea4dd7a2c807108f651b7135d1d6ca90931d93e97ab07fe42d923478ba2407d5b68aa497e4619ac10aa3b27
}
```

shh_addToGroup

(?)

Parameters

1. `HEX String` - The identity address to add to a group (?).

```
params: [
  "0xc931d93e97ab07fe42d9234..."
]
```

Returns

`Boolean` - returns `true` if the identity was successfully added to the group, otherwise `false` (?).

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_hasIdentity","params":["0xc931d93e97ab07fe42d923478ba2465f283..."],

// Result
{
  "id":1,
  "jsonrpc": "2.0",
  "result": true
}
```

shh_newFilter

Creates filter to notify, when client receives whisper message matching the filter options.

Parameters

1. `Object` - The filter options:
   - `to`: `HEX String` - (optional) Identity of the receiver. When present it will try to decrypt any incoming message if the client holds the private key to this identity.
   - `topics`: `Array` - Array of `HEX String` topics which the message has to have.

```
params: [{
   "topics": ['0x12341234bf4b564f'],
   "to": "0x2341234bf4b2341234bf4b564f..."
}]
```

Returns

`HEX String` - Integer of the newly created filter.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_newFilter","params":[{"topics": ['0x12341234bf4b564f'],"to": "0x234

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": "0x7" // 7
}
```

shh_uninstallFilter

Uninstalls a filter with given id. Should always be called when watch is no longer needed. Additonally Filters timeout when they aren't requested with shh_getFilterChanges for a period of time.

Parameters

1. `HEX String` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

`Boolean` - `true` if the filter was successfully uninstalled, otherwise `false`.

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_uninstallFilter","params":["0x7"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": true
}
```

---

shh_getFilterChanges

Polling method for whisper filters.

Parameters

`. HEX String` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

`Array` - Array of messages received since last poll:

- `hash`: `HEX String` - The hash of the message.
- `from`: `HEX String` - The sender of the message, if a sender was specified.
- `to`: `HEX String` - The receiver of the message, if a receiver was specified.
- `expiry`: `HEX String` - Integer of the time in seconds when this message should expire (?).
- `ttl`: `HEX String` - Integer of the time the message should float in the system in seconds (?).
- `sent`: `HEX String` - Integer of the unix timestamp when the message was sent.
- `topics`: `Array` - Array of `HEX String` topics the message contained.
- `payload`: `HEX String` - The payload of the message.
- `workProved`: `HEX String` - Integer of the work this message required before it was send (?).

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_getFilterChanges","params":["0x7"],"id":73}'

// Result
{
  "id":1,
  "jsonrpc":"2.0",
  "result": [{
```

```
    "hash": "0x33eb2da77bf3527e28f8bf493650b1879b08c4f2a362beae4ba2f71bafcd91f9",
    "from": "0x3ec052fc33..",
    "to": "0x87gdf76g8d7fgdfg...",
    "expiry": "0x54caa50a", // 1422566666
    "sent": "0x54ca9ea2", // 1422565026
    "ttl": "0x64" // 100
    "topics": ["0x6578616d"],
    "payload": "0x7b2274797065223a226d657373616765222c2263686...",
    "workProved": "0x0"
    }]
}
```

---

shh_getMessages

Get all messages matching a filter, which are still existing in the node.

Parameters

1. `HEX String` - The filter id.

```
params: [
  "0x7" // 7
]
```

Returns

See shh_getFilterChanges

Example

```
// Request
curl -X POST --data '{"jsonrpc":"2.0","method":"shh_getMessages","params":["0x7"],"id":73}'
```

Result see shh_getFilterChanges

The Ethereum Javascript API

Introduction

To make your Đapp work with on Ethereum, you can use the `web3` object provided by the ethereum.js library. ethereumjs communicates to a local node through RPC calls which can be found here. ethereum.js works with AlethZero and Mist, and also in an external browser if one of the former nodes are running locally.

`web3` contains the `eth` object - `web3.eth` (for specifically Ethereum blockchain interactions) and the `shh` object - `web3.shh` (for Whisper interaction). Over time we'll introduce other objects for each of the other web3 protocols.

Using callbacks

As this API is designed to work with a local RPC node and all its functions are by default use synchronous HTTP requests.

If you want to make asynchronous request, you can pass an optional callback as the last parameter to most functions. All callbacks are using an error first callback style:

```
web3.eth.getBlock(48, function(error, result){
    if(!error)
        console.log(result)
    else
        console.error(error);
})
```

A note on big numbers in JavaScript

You will always get a BigNumber object for balance values as JavaScript is not able to handle big numbers correctly. Look at the following examples:

```
"1010101003243253453464564564564564564"
// "10101010032432534534646456456456456456456"
1010101003243253453464564564564564564
// 1.0101010032432535e+38
```

Ethereum.js depends on the BigNumber Library and adds it automatically.

```
var balance = new BigNumber('131242344353464564564574574567456');
// or var balance = web3.eth.getBalance(someAddress);

balance.plus(21).toString(10); // toString(10) converts it to a number string
// "131242344353464564564574574567477"
```

The next example wouldn't work as we have more than 20 floating points, therefore it is recommended to keep you balance always in *wei* and only transform it to other units when presenting to the user:

```
var balance = new BigNumber('13124.234435346456466666457455567456');

balance.plus(21).toString(10); // toString(10) converts it to a number string, but can only show max 20 floating points
// "13145.234435346456466666457455567456" // you number would be cut after the 20 floating point
```

API

- web3
  - version
    - api -> string e.g. '0.2.0' (the ethereum.js version)
    - client -> string e.g. 'AlethZero/1.0.0' (the client ID)
    - network -> number e.g. 58 (the network version)
    - ethereum -> number e.g. 60 (the ethereum protocol version)
    - whisper -> number e.g. 20 (the shh version)
  - port -> number e.g. 8080 (not available yet)
  - setProvider(provider)
  - reset()
  - sha3(string) -> hexString
  - toHex(stringOrNumber) -> textString
  - toAscii(hexString) -> textString
  - fromAscii(textString, [padding]) -> hexString
  - toDecimal(hexString) -> number
  - fromDecimal(number) -> hexString
  - fromWei(numberStringOrBigNumber, unit) -> string|BigNumber (depending on the input)
  - toWei(numberStringOrBigNumber, unit) -> string|BigNumber (depending on the input)
  - toBigNumber(numberOrHexString) -> BigNumber
  - isAddress(hexString) -> boolean
  - net
    - listening -> boolean
    - peerCount -> Integer
  - eth
    - coinbase -> hexString
    - gasPrice -> BigNumber
    - accounts -> array of hexStrings
    - mining -> boolean
    - register(hexString)
    - unRegister(hexString)
    - defaultBlock -> Integer
    - blockNumber -> Integer
    - getBalance(address) -> BigNumber
    - getStorageAt(address, position) -> hexString
    - getCode(address) -> hexString
    - getBlock(hash/number) -> headerObject
    - getBlockTransactionCount(hash/number) -> Integer
    - getUncle(hash/number) -> headerObject
    - getBlockUncleCount(hash/number) -> Integer
    - getTransaction(hash) -> transactionObject
    - getTransactionFromBlock(hashOrNumber, indexNumber) -> transactionObject
    - getTransactionCount(address) -> Integer
    - sendTransaction(object)
    - contract(abiArray) -> contractObject
    - call(object) -> hexString
    - filter(array (, options))
      - watch(callback)

Usage

web3

The `web3` object provides all methods.

Example

```
var web3 = require('web3')
```

---

web3.version.api

```
web3.version.api
```

Returns

The ethereum js api version.

Example

```
var version = web3.version.api;
console.log(api); // "0.2.0"
```

---

web3.version.client

```
web3.version.client
```

Returns

The client/node version.

Example

```
var version = web3.version.client;
console.log(version); // "Mist/v0.9.3/darwin/go1.4.1"
```

web3.version.network

```
web3.version.network
```

Returns

The network protocol version.

Example

```
var version = web3.version.network;
console.log(version); // 54
```

web3.version.ethereum

```
web3.version.ethereum
```

Returns

The ethereum protocol version.

Example

```
var version = web3.version.ethereum;
console.log(version); // 60
```

web3.version.whisper

```
web3.version.whisper
```

Returns

The whisper protocol version.

Example

```
var version = web3.version.whisper;
console.log(version); // 20
```

web3.setProvider

```
web3.setProvider(providor)
```

Returns

undefined

Should be called to set provider.

Example

```
web3.setProvider(new web3.providers.HttpProvider('http://localhost:8545')); // 8080 for cpp/AZ, 8545 for go/mist
// or
web3.setProvider(new web3.providers.QtSyncProvider());
```

web3.reset

```
web3.reset()
```

Returns

undefined

Should be called to reset state of web3. Resets everything except manager. Uninstalls all filters. Stops polling.

Example

```
web3.reset();
```

web3.sha3

```
web3.sha3(string [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The SHA3 of the given data.

Example

```
var str = web3.sha3("Some ASCII string to be hashed");
console.log(str); // "0x536f6d6520415343494920737472696e6720746f206265206861736864"

var hash = web3.sha3(str);
console.log(hash); // "0xb21dbc7a5eb6042d91f8f584af266f1a512ac89520f43562c6c1e37eab6eb0c4"
```

---

web3.toHex

```
web3.toHex(mixed);
```

Returns

The hex string of `mixed`.

Following input values are possible:

- string
- number or string of number
- BigNumber instance -> will be converted to a hex representation of a number
- hex string
- array (will be stringified before)
- object (will be stringified before)

Example

```
var str = web3.toHex({test: 'test'});
console.log(str); // '0x7b2274657374223a2274657374227d'
```

---

web3.toAscii

```
web3.toAscii(hexString);
```

Returns

An ASCII string made from the data `hexString`.

Example

```
var str = web3.toAscii("0x657468657265756d0000000000000000000000000000000000000000000000000000000000000000");
console.log(str); // ethereum
```

---

web3.fromAscii

```
web3.fromAscii(string[, padding]);
```

Returns

data of the ASCII string `string` and padded to `padding` bytes (default to 32) and left-aligned.

Example

```
var str = web3.fromAscii('ethereum');
console.log(str); // "0x657468657265756d0000000000000000000000000000000000000000000000"

var str2 = web3.fromAscii('ethereum', 32);
console.log(str2); // "0x657468657265756d0000000000000000000000000000000000000000000000"
```

---

web3.toDecimal

```
web3.toDecimal(hexString);
```

Returns

The decimal string representing the data `hexString` (when interpreted as a big-endian integer).

Example

```
var value = web3.toDecimal('0x15');
console.log(value === "21"); // true
```

---

web3.fromDecimal

```
web3.fromDecimal(number);
```

Returns

The HEX string representing the decimal integer `number`.

Example

```
var value = web3.fromDecimal('21');
console.log(value === "0x15"); // true
```

---

web3.fromWei

```
web3.fromWei([String,BigNumber] number, [String] unit)
```

Returns

either a number string, or a BigNumber object, depending on the given `number` parameter.

Converts a number of wei into the following ethereum units:

- kwei/ada

- mwei/babbage
- gwei/shannon
- szabo
- finney
- ether
- kether/grand/einstein
- mether
- gether
- tether

Example

```
var value = web3.fromWei('21000000000000', 'finney');
console.log(value); // "0.021"
```

---

web3.toWei

```
web3.toWei([String,BigNumber] number, [String] unit)
```

Returns

either a number string, or a BigNumber object, depending on the given `number` parameter.

Converts a ethereum unit into wei. Possible units are:

- kwei/ada
- mwei/babbage
- gwei/shannon
- szabo
- finney
- ether
- kether/grand/einstein
- mether
- gether
- tether

Example

```
var value = web3.toWei('1', 'ether');
console.log(value); // "1000000000000000000"
```

---

web3.toBigNumber

```
web3.toBigNumber(numberOrHexString);
```

Returns

a BigNumber object representing the given value

See the note on BigNumber.

Example

```
var value = web3.toBigNumber('200000000000000000000001');
console.log(value); // instanceOf BigNumber
console.log(value.toNumber()); // 2.0000000000000002e+23
console.log(value.toString(10)); // '200000000000000000000001'
```

web3.net.listening

```
web3.net.listening
```

Returns

`true` if and only if the client is actively listening for network connections, otherwise `false`.

Example

```
var listening = web3.net.listening;
console.log(listening); // true of false
```

web3.net.peerCount

```
web3.net.peerCount
```

Returns

The number of peers currently connected to the client.

Example

```
var peerCount = web3.net.peerCount;
console.log(peerCount); // 4
```

web3.eth

Contains the ethereum blockchain related methods.

Example

```
var eth = web3.eth;
```

web3.eth.coinbase

```
web3.eth.coinbase
```

The coinbase is the address were the mining rewards go into.

Returns

The coinbase address of the client.

Example

```
var coinbase = web3.eth.coinbase;
console.log(coinbase); // "0x407d73d8a49eeb85d32cf465507dd71d507100c1"
```

---

web3.eth.mining

```
web3.eth.mining
```

Returns

`true` if the client is mining, otherwise `false`.

Example

```
var mining = web3.eth.mining;
console.log(mining); // true or false
```

---

web3.eth.gasPrice

```
web3.eth.gasPrice
```

Returns -> a BigNumber object of the current gas price in wei.

The gas price is determined by the x latest blocks median gas price.

See the note on BigNumber.

Example

```
var gasPrice = web3.eth.gasPrice;
console.log(gasPrice.toString(10)); // "10000000000000"
```

---

web3.eth.accounts

```
web3.eth.accounts
```

Returns

An array of the addresses owned by client.

Example

```
var accounts = web3.eth.accounts;
console.log(accounts); // ["0x407d73d8a49eeb85d32cf465507dd71d507100c1"]
```

---

web3.eth.register

```
web3.eth.register(addressHexString)
```

Returns

?

Registers the given address to be included in `web3.eth.accounts`. This allows non-private-key owned accounts to be associated as an owned account (e.g., contract wallets).

Example

```
web3.eth.register("0x407d73d8a49eeb85d32cf465507dd71d507100ca")
```

---

web3.eth.unRegister

```
 web3.eth.unRegister(addressHexString [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

?

Unregisters a given address.

Example

```
web3.eth.unregister("0x407d73d8a49eeb85d32cf465507dd71d507100ca")
```

---

web3.eth.defaultBlock

```
web3.eth.defaultBlock
```

Returns

The default block number/age to use when querying a state.

This default block is used for the following methods (optionally you can overwrite the defaultBlock by passing it as the last parameter):

- web3.eth.getBalance()
- web3.eth.getCode()
- web3.eth.getTransactionCount()
- web3.eth.getStorageAt()
- web3.eth.call()

default block can be

- a block number
- `'latest'`, which would be the latest minded block
- `'pending'`, which would the currently minded block including pending transactions

Default is `latest`

Example

```
var defaultBlock = web3.eth.defaultBlock;
console.log(defaultBlock); // 'latest'

// set the default block
web3.eth.defaultBlock = 231;
```

---

web3.eth.blockNumber

```
web3.eth.blockNumber
```

Returns

The number of the most recent block.

Example

```
var number = web3.eth.blockNumber;
console.log(number); // 2744
```

---

web3.eth.getBalance

```
web3.eth.getBalance(addressHexString [, defaultBlock] [, callback])
```

- If you pass an optional defaultBlock it will not use the default web.eth.defaultBlock.
- If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

a BigNumber object of the current balance for the given address in wei.

See the note on BigNumber.

Example

```
var balance = web3.eth.getBalance("0x407d73d8a49eeb85d32cf465507dd71d507100c1");
console.log(balance); // instanceof BigNumber
console.log(balance.toString(10)); // '1000000000000'
console.log(balance.toNumber()); // 1000000000000
```

web3.eth.getStorageAt

```
web3.eth.getStorageAt(addressHexString, position [, defaultBlock] [, callback])
```

- If you pass an optional defaultBlock it will not use the default web.eth.defaultBlock.
- If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The value in storage at position `position` of the address `addressHexString`.

Example

```
var state = web3.eth.getStorageAt("0x407d73d8a49eeb85d32cf465507dd71d507100c1", 0);
console.log(state); // "0x03"
```

web3.eth.getCode

```
web3.eth.getCode(addressHexString [, defaultBlock] [, callback])
```

- If you pass an optional defaultBlock it will not use the default web.eth.defaultBlock.
- If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The data at given address `addressHexString`.

Example

```
var code = web3.eth.getCode("0xd5677cf67b5aa051bb40496e68ad359eb97cfbf8");
console.log(code); // "0x600160008035811a818181146012578301005b601b6001356025565b8060005260206000f25b600060078202905091
```

web3.eth.getBlock

```
web3.eth.getBlock(hashHexStringOrBlockNumber[, returnTransactionObjects] [, callback])
```

If the `returnTransactionObjects` parameter is `true` it returns all the transactions as objects in the `transactions` property, if `false` it only includes an array with transaction hashes. **Default is** `false`

If you pass an optional callback the HTTP request is made asynchronous.

Returns

a block object with number or hash `hashHexStringOrBlockNumber`:

- `number` (integer): The number of this block.
- `hash` (32-byte hash): The block hash (i.e. the SHA3 of the RLP-encoded dump of the block's header).
- `parentHash` (32-byte hash): The parent block's hash (i.e. the SHA3 of the RLP-encoded dump of the parent block's header).
- `nonce`: the nonce of the block.
- `sha3Uncles` (32-byte hash): The SHA3 of the RLP-encoded dump of the uncles portion of the block.
- `logsBloom` (32-byte hash): The bloom filter of this block.
- `stateRoot` (32-byte hash): The root of the state trie.
- `transactionsRoot` (32-byte hash): The root of the block's transactions trie.
- `miner` (20-byte address): The address of the account that was rewarded for mining this block (né the coinbase address).
- `difficulty` (BigNumber): The PoW difficulty of this block.
- `totalDifficulty` (BigNumber): The total difficulty of the entire chain up and including this block.
- `size` (integer) the size in bytes of the block
- `minGasPrice` (BigNumber): The minimum price, in Wei per GAS, that the miner accepted for any transactions in this block.
- `gasLimit` (integer): The gas limit of this block.
- `gasUsed` (integer): The amount of gas used in this block.
- `timestamp` (integer): The timestamp of this block (an integer).
- `extraData` (byte array): Any extra data this block contains.
- `children` (array of 32-byte hashes): The hashes of any children this block has.
- `transactions` (array) transaction objects or hashes, depending on the `returnTransactionObjects` parameter
- `uncles` (array) hashes of uncles

Example

```
var info = web3.eth.block(3150);
console.log(info);
 /*{
  "number": 3150,
  "difficulty": 125000,
  "totalDifficulty": 124334345000,
  "transactionsRoot": "0x56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421"
  "hash": "0x452e84d0d0599d779a4e466d0b70d50ca316499a8489604d5b9df436ccfeee",
  "parentHash": "0xf9de6948d835ed5257229b5103f9421a7f70c3ccc65fd31c0db85324e05702f5",
  "nonce": "0xed0c6a53777b15880fb359dfd9368d002c959243a19c35ca1ae2e97f9bf78a53",
  "sha3Uncles": "0x1dcc4de8dec75d7aab85b567b6ccd41ad312451b948a7413f0a142fd40d49347",
  "logsBloom": "0xe84d0d0599d779a4e466d0b70d50ca316499a8489604d5b9df436ccfeee",
  "stateRoot": "0xd878fcee309af964e8e70c66ec25b1e7de9eca9c4f49e90efddeea8f77a37e43",
  "extraData": "0x0000000000000000000000000000000000000000000000000000000000000000",
  "minGasPrice": instanceof BigNumber,
  "gasLimit": 125000,
  "gasUsed": 34345,
  "miner": "0x407d73d8a49eeb85d32cf465507dd71d507100c1",
  "timestamp": 1416585210,
  "transactions": [{...}, {...}],
  "uncles": ["0xkj423h4kj23...", "0xlj4234k2..."]
} */
```

web3.eth.getBlockTransactionCount

```
web3.eth.getBlockTransactionCount(hashStringOrBlockNumber [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The number of transactions in a given block `hashStringOrBlockNumber`.

Example

```
var number = web3.eth.getBlockTransactionCount("0x407d73d8a49eeb85d32cf465507dd71d507100c1");
console.log(number); // 1
```

---

web3.eth.getUncle

```
web3.eth.getUncle(blockHashStringOrNumber, uncleNumber[, returnTransactionObjects] [, callback])
```

If the `returnTransactionObjects` parameter is `true` it returns all the transactions as objects in the `transactions` property, if `false` it only includes an array with transaction hashes. **Default is** `false`

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns the uncle with number `uncleNumber` from the block with number or hash `blockHashStringOrNumber`.

Return value see web3.eth.getBlock()

Example

```
var blockNumber = 500;
var indexOfUncle = 0

var uncle = web3.eth.getUncle(blockNumber, indexOfUncle);
console.log(uncle); // see web3.eth.getBlock
```

---

web3.eth.getTransaction

```
web3.eth.getTransaction(transactionHash [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

a transaction object its hash `transactionHash`:

- `status`: "mined" or "pending"
- `hash` (32-byte hash): The hash of the transaction.
- `nonce` (32-byte hash): The transaction nonce.

- `blockHash` (32-byte hash): the blocks hash, where this transaction appeared, `null` when pending
- `blockNumber` (integer): the blocks number, where this transaction appeared, `null` when pending
- `transactionIndex` (integer): the index a t which this transaction appeared in the block
- `to` (20-byte address): The address to which the transaction was sent. This may be the null address (address 0) if it was a contract-creation transaction.
- `from` (20-byte address): The cryptographically verified address from which the transaction was sent .
- `gas` (integer): The amount of GAS supplied for this transaction to happen.
- `gasPrice` (BigNumber): The price offered to the miner to purchase this amount of GAS, in Wei per GAS.
- `value` (BigNumber): The amount of ETH to be transferred to the recipient with the transaction in wei.
- `input` (byte array): The binary data that formed the input to the transaction, either the input data if it was a message call or the contract initialisation if it was a contract creation.

Example

```
var blockNumber = 668;
var indexOfTransaction = 0

var transaction = web3.eth.getTransaction(blockNumber, indexOfTransaction);
console.log(transaction);
/*
{
    "hash":"0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
    "nonce":"0x",
    "blockHash": "0x6fd9e2a26abeab0aa2411c6ef2fc5426d6adb7ab17f30a99d3cb96aed1d1055b",
    "blockNumber": 5599
    "transactionIndex":  1
    "from":"0x407d73d8a49eeb85d32cf465507dd71d507100c1",
    "gas":520464,
    "gasPrice": instanceof BigNumber,
    "hash":"0xc6ef2fc5426d6ad6fd9e2a26abeab0aa2411b7ab17f30a99d3cb96aed1d1055b",
    "input":"0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b806000526020
    "to":"0x0000000000000000000000000000000000000000",
    "value": instanceof BigNumber
}
*/
```

web3.eth.getTransactionFromBlock

```
getTransactionFromBlock(hashStringOrNumber, indexNumber [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

a transaction object by block number or hash `hashStringOrNumber` with transaction index `indexNumber`:

```
var transaction = web3.eth.getTransactionFromBlock('0x4534534534', 2);
console.log(transaction); // see web3.eth.getTransaction
```

web3.eth.getTransactionCount

```
web3.eth.getTransactionCount(addressHexString [, defaultBlock] [, callback])
```

- If you pass an optional defaultBlock it will not use the default web.eth.defaultBlock.
- If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The number of transactions send from the given address `addressHexString`.

Example

```
var number = web3.eth.getTransactionCount("0x407d73d8a49eeb85d32cf465507dd71d507100c1");
console.log(number); // 1
```

web3.eth.sendTransaction

```
web3.eth.sendTransaction(transactionObject [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Creates a new message-call transaction.

- `transactionObject`, an anonymous object specifying the parameters of the transaction.
  - `from` (hexString), the address for the sending account;
  - `to` (hexString), the destination address of the message, left undefined for a contract-creation transaction
  - `value (number|hexString|BigNumber)`, (optional) the value transferred for the transaction in Wei, also the endowment if it's a contract-creation transaction;
  - `gas (number|hexString|BigNumber)`, (optional, default: To-Be-Determined) the amount of gas to purchase for the transaction (unused gas is refunded), defaults to the most gas your ether balance allows; and
  - `gasPrice (number|hexString|BigNumber)`, (optional, default: To-Be-Determined) the price of gas for this transaction in wei, defaults to the mean network gasPrice.
  - `data (hexString)`, (optional) either a byte string containing the associated data of the message, or in the case of a contract-creation transaction, the initialisation code;
  - `code`, (optional) a synonym for `data`;
- `callback`, the callback function, called on completion of the transaction. If the transaction was a contract-creation transaction, it is passed with a single argument; the address of the new account.

Example

```
// compile solidity source code
var source = "" +
"contract test {\n" +
"   function multiply(uint a) returns(uint d) {\n" +
"       return a * 7;\n" +
"   }\n" +
"}\n";

var compiled = web3.eth.solidity(source);

web3.eth.sendTransaction({data: compiled}, function(err, address) {
  if (!err) console.log(address); // "0x7f9fade1c0d57a7af66ab4ead7c2eb7b11a91385"
});
```

web3.eth.contract

```
web3.eth.contract(abiArray)
```

Returns

a contract object, which can be used to initiate use contracts on an address.

This method creates a contract object for a solidity contract. It has the same methods available as solidity contract description itself.

You can read more about events here.

Example

```
// contract abi
var abi = [{
     name: 'myMethod',
     type: 'function',
     inputs: [{ name: 'a', type: 'string' }],
     outputs: [{name: 'd', type: 'string' }]
}, {
     name: 'myEvent',
     type: 'event',
     inputs: [{name: 'a', type: 'int', indexed: true},{name: 'b', type: 'bool', indexed: false]
}];

// creation of contract object
var MyContract = web3.eth.contract(abi);

// initiate contract for an address
var myContractInstance = new MyContract('0x43gg423k4h4234235345j3453');

myContractInstance.myMethod('this is test string param for call'); // myMethod call (implicit, default)
myContractInstance.call().myMethod('this is test string param for call'); // myMethod call (explicit)
myContractInstance.sendTransaction().myMethod('this is test string param for transact'); // myMethod sendTransaction

var filter = myContractInstance.myEvent({a: 5});
filter.watch(function (result) {
    console.log(result);
    /*
    {
        address: '0x0123123121',
        topics: "0x1234567890123456789012345678901234567890123456789012", "0x0000000000000000000000000000000000000000000000000000000000000000",
        data: "0x0000000000000000000000000000000000000000000000000000000000000001",
        ...
    }
    */
});
```

Contract Events

You can use events like filters and they have the same methods. Though the first parameter for an event are the indexed arguments as an object:

```
var MyContract = web3.eth.contract(abi);
var myContractInstance = new MyContract('0x43gg423k4h4234235345j3453');

// watch for an event
var myEvent = myContractInstance.MyEvent({some: 'args'}, filterObject);
myEvent.watch(function(result){
   ...
});

var myResults = myEvent.get();

...
```

```
myEvent.stopWatching();
```

web3.eth.call

```
web3.eth.call(callObject [, defaultBlock] [, callback])
```

- If you pass an optional defaultBlock it will not use the default web.eth.defaultBlock.
- If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The data which matches the call.

Executes a new message-call immediately without creating a transaction on the block chain. `callObject` is an object specifying the parameters of the transaction.

Example

```
var options = {
    to: "0xc4abd0339eb8d57087278718986382264244252f",
    data: "0xc6888fa1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000003"
};
var result = web3.eth.call(options);
console.log(result); // "0x0000000000000000000000000000000000000000000000000000000000000015"
```

web3.eth.filter

```
// can be 'latest' or 'pending'
web3.eth.filter(filterString)
// object is a log filter
web3.eth.filter(options)
// object is an event object
web3.eth.filter(eventObject [, eventArguments [, options]])
// an array of events object belonging to specific contract objects (not implemented yet)
web3.eth.filter(eventArray [, options])
// object is a contract object (not implemented yet)
web3.eth.filter(contractObject [, options])
```

- `filterString` : `'latest'` or `'pending'` to watch for changes in the latest block or pending transactions respectively
- `options`
  - `fromBlock` : The number of the earliest block (`latest` may be given to mean the most recent and `pending` currently mining, block).
  - `toBlock` : The number of the latest block (`latest` may be given to mean the most recent and `pending` currently mining, block).
  - `address` : An address or a list of addresses to restrict log entries by requiring them to be made from a particular account.
  - `topics` : An array of values which must each appear in the log entries.
- `eventArguments` is an object with keys of one or more indexed arguments for the event(s) and values of either one (directly) or more (in an array) e.g. {'a': 1, 'b': [myFirstAddress, mySecondAddress]}.

Returns

a filter object with the following methods:

Filter Methods:

- `filter.get()` : Returns all of the log entries that fit the filter.
- `filter.watch(callback)` : Watches for state changes that fit the filter and calls the callback.
- `filter.stopWatching()` : Stops the watch and uninstalls the filter in the node. Should always be called once it is done.

Callback return values

If its a log filter it returns a list of log entries; each includes the following fields:

- `status` : "pending" or "mined"
- `address` : The address of the account whose execution of the message resulted in the log entry being made.
- `topics` : The topic(s) of the log.
- `data` : The associated data of the log.
- `blockNumber` : The block number from at which this event happened.
- `blockHash` : The block hash from at which this event happened.
- `transactionHash` : The transaction hash from at which this event happened.
- `transactionIndex` : The transaction index from at which this event happened.
- `logIndex` : The log index

If its an event filter it returns a filter object with the return values of event:

- `status` : "pending" or "mined"
- `args` : The arguments coming from the event
- `topics` : The topic(s) of the log.
- `blockNumber` : The block number from at which this event happened.
- `blockHash` : The block hash from at which this event happened.
- `transactionHash` : The transaction hash from at which this event happened.
- `transactionIndex` : The transaction index from at which this event happened.
- `logIndex` : The log index

Example

```
var filter = web3.eth.filter('pending');

filter.watch(function (log) {
  console.log(log); //  {"address":"0x0000000000000000000000000000000000000000","data":"0x00000000000000000000000000000000
});

var myResults = filter.get();

...

filter.stopWatching();
```

web3.eth.getCompilers

```
web3.eth.getCompilers([callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

An array of available compilers.

Example

```
var number = web3.eth.getCompilers();
console.log(number); // ["lll", "solidity", "serpent"]
```

web3.eth.compile.solidity

```
web3.eth.compile.solidity(sourceString [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The compiled solidity code as HEX string.

Compiles the solidity source code `sourceString` and returns the output data.

Example

```
var source = "" +
    "contract test {\n" +
    "   function multiply(uint a) returns(uint d) {\n" +
    "       return a * 7;\n" +
    "   }\n" +
    "}\n";
var code = web3.eth.compile.solidity(source);
console.log(code); // "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b80
```

web3.eth.compile.lll

```
web3. eth.compile.lll(sourceString [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The compiled lll code as HEX string.

Compiles the LLL source code `sourceString` and returns the output data.

Example

```
var source = "...";

var code = web3.eth.compile.lll(source);
console.log(code); // "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b80
```

web3.eth.compile.serpent

```
web3.eth.compile.serpent(sourceString [, callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Returns

The compiled serpent code as HEX string.

Compiles the serpent source code `sourceString` and returns the output data.

```
var source = "...";

var code = web3.eth.compile.serpent(source);
console.log(code); // "0x603880600c6000396000f3006001600060e060020a600035048063c6888fa114601857005b6021600435602b565b80
```

web3.eth.flush

Example

```
// TODO:
```

web3.db.putString

This method should be called, when we want to store a string in a local leveldb database. First

Example

param is db name, second is the key, and third is the string value.

```
var result = web3.db.putString('testDB', 'key', 'myString')
console.log(result); // true
```

web3.db.getString

This method should be called, when we want to get string from local leveldb database. First

Example

param is db name and second is the key of string value.

```
var value = web3.db.getString('testDB', 'key');
console.log(value); // "myString"
```

web3.db.putHex

This method should be called, when we want to store HEX in a local leveldb database. First

Example

param is db name, second is the key, and third is the value.

```
var result = web3.db.putHex('testDB', 'key', '0x4f554b443');
console.log(result); // true
```

---

web3.db.getHex

This method should be called, when we want to get a HEX value from a local leveldb database. First

Example

param is db name and second is the key of value.

```
var value = web3.db.getHex('testDB', 'key');
console.log(value); // "0x4f554b443"
```

---

web3.shh

Whisper Overview

Example

```
var shh = web3.shh;
```

---

web3.shh.post

web3.shh.post(object [, callback])

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

This method should be called, when we want to post whisper message. `_message` object may have following fields:

- `from` : identity of sender as hexString
- `to` : identity of receiver as hexString
- `payload` : message payload
- `ttl` : time to live in seconds
- `workToProve` : // or priority TODO
- `topics` : array of strings or hexStrings, with message topics ```js var identity = web3.shh.newIdentity(); var topic = 'example'; var payload = 'hello whisper world!';

var message = { from: identity, topics: [topic], payload: payload, ttl: 100, workToProve: 100 // or priority TODO };

web3.shh.post(message);

```
***

#### web3.shh.newIdentity

    web3.shh.newIdentity([callback])

If you pass an optional callback the HTTP request is made asynchronous. See [this note](#using-callbacks) for details.

Should be called to create new identity.

##### Returns

a new identity hex string.

```js
var identity = web3.shh.newIdentity();
console.log(identity); // "0xc931d93e97ab07fe42d923478ba2465f283f440fd6cabea4dd7a2c807108f651b7135d1d6ca9007d5b68aa497e
```

web3.shh.hasIdentity

```
web3.shh.hasIdentity([callback])
```

If you pass an optional callback the HTTP request is made asynchronous. See this note for details.

Should be called, if we want to check if user has given identity. Accepts one param. Returns true if he has, otherwise false.

Returns

a boolean whether or not the identity exists.

Example

```
var identity = web3.shh.newIdentity();
var result = web3.shh.hasIdentity(identity);
console.log(result); // true

var result2 = web3.shh.hasIdentity(identity + "0");
console.log(result2); // false
```

web3.shh.newGroup

Example

```
// TODO: not implemented yet
```

web3.shh.addToGroup

Example

```
// TODO: not implemented yet
```
```

web3.shh.filter

```
wev3.shh.filter(options)
```

This method should be used, when you want to watch whisper messages. Available filter options are:

- `topics` : array of strings. filter messages with by this topic(s)
- `to` : filter the identity of receiver of the message

Example

```
var topic = 'example';

var options = {
  topics: [topic],
  to: '0x32jkh23kjh4j23h5j34h5j3464'
};

var filter = web3.shh.filter(options);

filter.watch(function(res) {
  console.log(res);
  /* {
  "expiry":1422565026,
  "from":"0x3ec052fc3376f8a218b24b652803a5892038c39419a9a44923a61b113b7785d16ed1572df628859af3504670e4df31dcd8b3ee9a211
  "hash":"0x33eb2da77bf3527e28f8bf493650b1879b08c4f2a362beae4ba2f71bafcd91f9",
  "payload": 'The send message',
  "payloadRaw": "0xjk5h34kj5h34kj6kj346456547trhfghfdgh",
  "sent": 1422564926,
  "to":"0x32jkh23kjh4j23h5j34h5j3464",
  "topics":['myTopic'],
  "ttl":100,
  "workProved":0
  }
  */
});
```

Example

A simple HTML snippet that will display the user's primary account balance of Ether:

```
<html><body>
<div>You have <span id="ether">?</span> Weis</div>
<script>
web3.eth.filter('pending').watch(function() {
    var balance = web3.eth.getBalance(web3.eth.coinbase);
    document.getElementById("ether").innerText = balance.toString(10);
});

...

filter.stopWatching();
</script>
</body></html>
```

To test it, just put it in file and save. Load it in AlethZero and point the URL to file:///WHEREVER_YOU_SAVED_IT

Job done. Now go create.

more examples can be found here

Smart Contracts

Solidity Features

This is a list to explain and demonstrate new Solidity features as soon as they are completed.

Special Type Treatment for Integer Literals

PT Expressions only involving integer literals are now essentially treated as "big integers" (i.e. they do not overflow) until they are actually used with a non-literal. The type of the expression is only determined at that point as the smallest integer type that can contain the resulting value. Example:

```
contract IntegerLiterals {
  function f() {
    // The folloing would have caused a type error in earlier versions (cannot add int8 to uint8),
    // now the value of x is set to 9 and the type to uint8.
    var x = -1 + 10;
    // It is even possible to use literals that do not fit any of the Solidity types as long as
    // the final value is small enough. The value of y will be 1, its type uint8.
    var y = (0x100000000000000000001 * 0x100000000000000000001 * 0x100000000000000000001) & 0xff;
  }
}
```

Contracts Inherit all Members from Address

PT Contract types are implicitly convertible to `address` and explicitly convertible to and from all integer types. Furthermore, a contract type contains all members of the address type with the semantics applying to the contract's address, unless overwritten by the contract.

```
contract Helper {
  function getBalance() returns (uint bal) {
    return this.balance; // balance is "inherited" from the address type
  }
}
contract IsAnAddress {
  Helper helper;
  function setHelper(address a) {
    helper = Helper(a); // Explicit conversion, comparable to a downcast
  }
  function sendAll() {
    // send all funds to helper (both balance and send are address members)
    helper.send(this.balance);
  }
}
```

ABI requires arguments to be padded to 32 bytes

PT The latest version of the ABI specification requires arguments to be padded to multiples of 32 bytes. This is not a language feature that can be demonstrated as code examples. Please see the automated tests `SolidityEndToEndTests::packing_unpacking_types` and `SolidityEndToEndTests::packing_signed_types`.

Specify value and gas for function calls

PT External functions have member functions "gas" and "value" that allow to change the default amount of gas (all) and wei (0) sent to the called contract. "new expressions" also have the value member.

```
contract Helper {
  function getBalance() returns (uint bal) { return this.balance; }
}
contract Main {
  Helper helper;
  function Main() { helper = new Helper.value(20)(); }
  /// @notice Send `val` Wei to `helper` and return its new balance.
  function sendValue(uint val) returns (uint balanceOfHelper) {
    return helper.getBalance.value(val)();
  }
}
```

delete for structs

PT `delete` clears all members of a struct.

```
contract Contract {
  struct Data {
    uint deadline;
    uint amount;
  }
  Data data;
  function set(uint id, uint deadline, uint amount) {
    data.deadline = deadline;
    data.amount = amount;
  }
  function clear(uint id) { delete data; }
}
```

Note that, unfortunately, this only works directly on structs for now, so I would propose to not announce "delete" as a feature yet.

Contract Inheritance

PT1 PT2 Contracts can inherit from each other.

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

// Use "is" to derive from another contract. Derived contracts can access all members
// including private functions and storage variables.
contract mortal is owned {
    function kill() { if (msg.sender == owner) suicide(owner); }
}

// These are only provided to make the interface known to the compiler.
contract Config { function lookup(uint id) returns (address adr) {} }
contract NameReg { function register(string32 name) {} function unregister() {} }

// Multiple inheritance is possible. Note that "owned" is also a base class of
// "mortal", yet there is only a single instance of "owned" (as for virtual
// inheritance in C++).
contract named is owned, mortal {
    function named(string32 name) {
        address ConfigAddress = 0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970;
        NameReg(Config(ConfigAddress).lookup(1)).register(name);
    }

// Functions can be overridden, both local and message-based function calls take
// these overrides into account.
    function kill() {
        if (msg.sender == owner) {
```

```
            address ConfigAddress = 0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970;
            NameReg(Config(ConfigAddress).lookup(1)).unregister();
// It is still possible to call a specific overridden function.
            mortal.kill();
        }
    }
}

// If a constructor takes an argument, it needs to be provided in the header.
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) {
      if (msg.sender == owner) info = newInfo;
   }

   function get() constant returns(uint r) { return info; }

   uint info;
}
```

Function Modifiers

PT Modifiers can be used to easily change the behaviour of functions, for example to automatically check a condition prior to executing the function. They are inheritable properties of contracts and may be overridden by derived contracts.

```
contract owned {
  function owned() { owner = msg.sender; }
  address owner;

  // This contract only defines a modifier but does not use it - it will
  // be used in derived contracts.
  // The function body is inserted where the special symbol "_" in the
  // definition of a modifier appears.
  modifier onlyowner { if (msg.sender == owner) _ }
}
contract mortal is owned {
  // This contract inherits the "onlyowner"-modifier from "owned" and
  // applies it to the "kill"-function, which causes that calls to "kill"
  // only have an effect if they are made by the stored owner.
  function kill() onlyowner {
    suicide(owner);
  }
}
contract priced {
  // Modifiers can receive arguments:
  modifier costs(uint price) { if (msg.value >= price) _ }
}
contract Register is priced, owned {
  mapping (address => bool) registeredAddresses;
  uint price;
  function Register(uint initialPrice) { price = initialPrice; }
  function register() costs(price) {
    registeredAddresses[msg.sender] = true;
  }
  function changePrice(uint _price) onlyowner {
    price = _price;
  }
}
```

Multiple modifiers can be applied to a function by specifying them in a whitespace-separated list and will be evaluated in order. Explicit returns from a modifier or function body immediately leave the whole function, while control flow reaching the end of a function or modifier body continues after the "_" in the previous modifier. Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

Conversion between String and Hash types

PT The explicit conversion between `string` and `hash` types of equal size is now allowed. Example:

```
contract Test {
  function convert(hash160 h, string20 s) returns (string20 res_s, hash160 res_h) {
    res_s = string20(h);
    res_h = hash160(s);
  }
}
```

Access to super

PT In the following contract, the function `kill` is overridden by sibling classes. Due to the fact that the sibling classes do not know of each other, they can only call `mortal.kill()` with the effect that one of the overrides is completely bypassed. A reasonable implementation would call the kill functions in all classes in the inheritance hierarchy.

```
contract mortal { function kill() { suicide(msg.sender); } }
contract named is mortal { function kill() { /*namereg.unregister();*/ mortal.kill(); } }
contract tokenStorage is mortal { function kill() { /*returnAllTokens();*/ mortal.kill(); } }
contract MyContract is named, tokenStorage {}
```

The `super` keyword solves this. Its type is the type of the current contract if it were empty, i.e. it contains all members of the current class' bases. Access to a member of super invokes the usual virtual member lookup, but it ends just above the current class. Using this keyword, the following works as expected:

```
contract mortal { function kill() { suicide(msg.sender); } }
contract named is mortal { function kill() { /*namereg.unregister();*/ super.kill(); } }
contract tokenStorage is mortal { function kill() { /*returnAllTokens();*/ super.kill(); } }
contract MyContract is named, tokenStorage {}
```

State Variable Accessors

PT Public state variables now have accessors created for them. Basically any `public` state variable can be accessed by calling a function with the same name as the variable.

```
contract test {
    function test() {
        data = 42;
    }
    uint256 data;
}
```

For example in the above contract if you tried to call test's `data()` method then you would obtain the result 42.

```
contract test {
    function test() {
        data = 42;
    }
private:
    uint256 data;
}
```

On the other hand on the above contract there is no accessor generated since the state variable is private.

Events

PT Events allow the convenient usage of the EVM logging facilities. Events are inheritable members of contracts. When they are called, they cause the arguments to be stored in the transaction's log. Up to three parameters can receive the attribute `indexed` which will cause the respective arguments to be treated as log topics instead of data. The hash of the signature of the event is always one of the topics. All non-indexed arguments will be stored in the data part of the log. Example:

```
contract ClientReceipt {
  event Deposit(address indexed _from, hash indexed _id, uint _value);
  function deposit(hash _id) {
    Deposit(msg.sender, _id, msg.value);
  }
}
```

Here, the call to `Deposit` will behave identical to `log3(msg.value, 0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20, sha3(msg.sender), _id);`. Note that the large hex number is equal to the sha3-hash of "Deposit(address,hash256,uint256)", the event's signature.

Fallback Functions

PT A contract can have exactly one unnamed function. This function cannot have arguments and is executed on a call to the contract if none of the other functions matches the given function identifier (or if no data was supplied at all).

```
contract Test {
  function() { x = 1; }
  uint x;
}

contract Caller {
  function callTest(address testAddress) {
    Test(testAddress).send(0);
    // results in Test(testAddress).x becoming == 1.
  }
}
```

Events in Exported Interfaces

PT Events are exported to the JSON and Solidity interfaces generated by the compiler. The contract

```
contract c {
    event ev(uint indexed a);
}
```

generates the JSON interface

```
[
   {
      "inputs" : [
         {
            "indexed" : true,
            "name" : "a",
            "type" : "uint256"
         },
         {
```

```
            "indexed" : false,
            "name" : "b",
            "type" : "uint256"
         }
      ],
      "name" : "ev",
      "type" : "event"
   }
]
```

and the Solidity interface `contract c{event ev(uint256 indexed a,uint256 b);}`.

Visibility Specifiers

PT Functions and storage variables can be specified as being `public`, `protected` or `private`, where the default for functions is `public` `protected` for storage variables. Public functions are part of the external interface and can be called externally, while for storage variables, an automatic accessor function is generated. Non-public functions are only visible inside a contract and its derived contracts (there is no distinction between `protected` and `private` for now).

```
contract c {
  function f(uint a) private returns (uint b) { return a + 1; }
  uint public data;
}
```

External functions can call `c.data()` to retrieve the value of `data` in storage, but are not able to call `f`.

Numeric Literals with Ether Subdenominations

PT Numeric literals can also be followed by the common ether subdenominations and the value of the assigned to variable will be multiplied by the proper amount.

```
contract c {
  function c()
  {
      val1 = 1 wei;    // 1
      val2 = 1 szabo;  // 1 * 10 ** 12
      val3 = 1 finney; // 1 * 10 ** 15
      val4 = 1 ether;  // 1 * 10 ** 18
 }
  uint256 val1;
  uint256 val2;
  uint256 val3;
  uint256 val4;
}
```

SHA3 with arbitrary arguments

PT. `sha3()` can now take an arbitrary number and type of arguments.

```
contract c {
  function c()
  {
      val2 = 123;
      val1 = sha3("foo"); // sha3(0x666f6f)
      val3 = sha3(val2, "bar", 1031); //sha3(0x7b626172407)
  }
  uint256 val1;
```

```
  uint16 val2;
  uint256 val3;
}
```

Optional Parameter Names

PT. The names for function parameters and return parameters are now optional.

```
contract test {
  function func(uint k, uint) returns(uint){
    return k;
  }
}
```

Generic call Method

PT Address types (and contracts by inheritance) have a method `call` that can receive an arbitrary number of arguments of arbitrary types (which can be serialized in memory) and will invoke a message call on that address while the arguments are ABI-serialized. If the first type has a memory-length of exactly four bytes, it is not padded to 32 bytes, so it is possible to specify a function signature.

```
contract test {
  function f(address addr, uint a) {
    addr.call(string4(string32(sha3("fun(uint256)"))), a);
  }
}
```

Byte arrays

PT Basic support for variable-length byte arrays. This includes

- `bytes` type for storage variables
- `msg.data` is of `bytes` type and contains the calldata
- functions taking arbitrary parameters (`call` , `sha3` , ...) can be called with `bytes` arguments.
- copying between `msg.data` and `bytes` storage variables

What is not possible yet:

- function parameters of `bytes` type
- local variables of `bytes` type
- index or slice access

```
contract c {
  bytes data;
  function() { data = msg.data; }
  function forward(address addr) { addr.call(data); }
  function getLength() returns (uint) { return addr.length; }
  function clear() { delete data; }
}
```

Enums

PT Solidity now supports enums. Enums are explicitly convertible to all integer types but implicit conversion is not allowed.

```
contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill };
    function test()
    {
        choices = ActionChoices.GoStraight;
    }
    function getChoice() returns (uint d)
    {
        d = uint256(choices);
    }
    ActionChoices choices;
    }
```

Visibility Specifiers

PT The visibility of a function can be specified by giving at most one of the specifiers `external`, `public`, `inheritable` or `private`, where `public` is the default. "External" functions can only be called via message-calls, i.e. from other contracts or from the same contract using `this.function()` (note that this also prevents calls to overwritten functions in base classes). Furthermore, parameters of "external" functions are immutable. "Public" functions can be called from other contracts and from the same contract using stack-based calls. "Inheritable" and "private" functions can only be called via stack-based calls, while "inheritable" functions are only visible in the contract itself and its derived contracts and "private" functions are not even visible in derived contracts.

```
contract Base {
  function exte() external { }
  function publ() public /* can be omitted */ { }
  function inhe() inheritable { priv(); }
  function priv() private { }
}
contract Derived is Base {
  function g() {
    this.exte();
    // impossible: exte();
    this.publ();
    publ();
    // impossible: this.inhe();
    inhe();
    // impossible: this.priv();
    // impossible: priv();
  }
}
```

Import Statement

PT We can now import other contracts and/or standard library contracts using the `import` keyword.

```
import "mortal";

contract Test is mortal {
    // since we import the standard library "mortal" contract and we inherit from it
    // we can call the kill() function that it provides
    function killMe() { kill();}
}
```

Inline members initialization

PT Inline members can be initialized at declaration time.

```
contract test {
  function test(){
    m_b = 6;
  }
  uint m_a = 5;
  uint m_b;
}
```

The arguments of the constructor of base contract

PT It is possible to pass arguments to the base contracts constructor. The arguments for the base constructor in the header will be optional later.

```
contract Base {
    function Base(uint i)
    {
        m_i = i;
    }
    uint public m_i;
}
contract Derived is Base(0) {
    function Derived(uint i) Base(i) {}
}
```

Detect failed CALLs

PT If a CALL fails, do not just silently continue. Currently, this issues a STOP but it will throw an exception once we have exceptions.

```
contract C {
  function willFail() returns (uint) {
    address(709).call();
    return 1;
  }
}
```

`willFail` will always return an empty byte array (unless someone finds the correct private key...).

Basic features for arrays

PT Byte arrays and generic arrays of fixed and dynamic size are supported in calldata and storage with the following features: Index access, copying (from calldata to storage, inside storage, both including implicit type conversion), enlarging and shrinking and deleting. Not supported are memory-based arrays (i.e. usage in non-external functions or local variables), array accessors and features like slicing. Access to an array beyond its length will cause the execution to STOP (exceptions are planned for the future).

```
contract ArrayExample {
  uint[7][] data;
  bytes byteData;
  function assign(uint[4][] input, bytes byteInput) external {
    data = input; // will assign uint[4] to uint[7] correctly, would produce type error if reversed
    byteData = byteInput; // bytes are stored in a compact way
  }
```

```
    function indexAccess() {
      data.length += 20;
      data[3][5] = data[3][2];
      byteData[2] = byteData[7]; // this will access sigle bytes
    }
    function clear() {
      delete data[2]; // will clear all seven elements
      data.length = 2; // clears everything after the second element
      delete data; // clears the whole array
    }
  }
```

Now Variable

PT The global scope contains an immutable variable called `now` which is an alias to `block.timestamp`, i.e. it contains the timestamp of the current block.

```
contract TimedContract {
  uint timeout = now + 4 weeks;
}
```

HashXX and StringXX to bytesXX

[Link to PT] (https://www.pivotaltracker.com/story/show/88146508)

- We replace `hash(XX*8)` and `stringXX` by `bytesXX`.
- `bytesXX` behaves as `hash(XX*8)` in terms of convertability and operators and as `stringXX` in terms of layout in memory (alignment, etc).
- `byte` is an alias for `bytes1`.
- `string` is reserved for future use.

`msg.sig` returns the function's signature hash

[Link to PT] (https://www.pivotaltracker.com/story/show/86896308) New magic type `msg.sig` that will provide the hash of the current function signature as `bytes4` type.

```
contract test {
    function foo(uint256 a) returns (bytes4 value) {
        return msg.sig;
    }
}
```

Calling that function will return `2FBEBD38` which is the hash of the signature of `foo(uint256)`.

Constant variables

PT Added `constant` specifier for uint, mapping and bytesXX types. Variables declared with `constant` specifier should be initialized at declaration time and can not be changed later. For now local variables can not be constant. Constant variables are not stored in Storage.

```
contract Foo {
    function getX() returns (uint r) { return x; }
    uint constant x = 56;
}
```

Anonymous Events

PT Added `anonymous` specifier for Event. For the event declared as anonymous the hash of the signature of the event will not be added as a first topic. The format is

```
event <name>([index list]) anonymous;
```

Anonymous property is also visible for ABI document.

Tightly packed storage

PT Items in storage are packed tighly as far as possible according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only that many bytes that are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

Examples:

```
contract C {
  uint248 x; // 31 bytes: slot 0, offset 0
  uint16 y; // 2 bytes: slot 1, offset 0 (does not fit in slot 0)
  uint240 z; // 30 bytes: slot 1, offset 2 bytes
  uint8 a; // 1 byte: slot 2, offset 0 bytes
  struct S {
    uint8 a; // 1 byte, slot +0, offset 0 bytes
    uint256 b; // 32 bytes, slot +1, offset 0 bytes (does not fit)
  }
  S structData; // 2 slots, slot 3, offset 0 bytes (does not really apply)
  uint8 alpha; // 1 byte, slot 4 (start new slot after struct)
  uint16[3] beta; // 3*16 bytes, slots 5+6 (start new slot for array)
  uint8 gamma; // 1 byte, slot 7 (start new slot after array)
}
```

Common Subexpression Elimination Excluding Memory and Storage

PT The optimizer splits code into blocks (at all operations that have non-local side effects like JUMP, CALL, CREATE and for also all instructions that access or modify memory or storage), analyses these blocks by creating an expression graph and establishes equivalences in a bottom-up way, simplifying expressions that e.g. involve constants. In the following code-generation phase, it re-creates the set of instructions that transform a given initial stack configuration into a given target stack configuration utilizing the simplest representatives of these equivalence classes. In conjunction with the already present jump-optimization, the two code snippets given below should be compiled into the same sequence of instructions:

```
contract test {
```

```
  function f(uint x, uint y) returns (uint z) {
    var c = x + 3;
    var b = 7 + (c * (8 - 7)) - x;
    return -(-b | 0);
  }
}
```

```
contract test {
  function f(uint x, uint y) returns (uint z) {
    return 10;
  }
}
```

Common Subexpression Elimination for Memory and Storage

PT This adds support for memory and storage operations to the common subexpression eliminator. This makes it possible to e.g. stretch the equality inference engine across SSTORE, MSTORE and even SHA3 computations (which go via memory). Without optimizer (because of packed storage), there are 4 SLOAD, 3 SSTORE and 4 SHA3 operations. The optimizer reduces those to a single SLOAD, SHA3 and SSTORE each.

```
contract test {
  struct s { uint8 a; uint8 b; uint8 c; }
  mapping(uint => s) data;
  function f(uint x, uint8 _a, uint8 _b, uint8 _c) {
    data[x].a = _a;
    data[x].b = _b;
    data[x].c = data[x].a;
  }
}
```

External Types

PT All functions with visibility more than internal should have external types (ABI types) otherwise raise an error. For Contract type external type is address type.

```
contract Foo {}
contract Test {
    function func() {
        Foo arg;
        this.Poo(arg);
        Poo(arg);
    }
    function Poo(Foo c) external {}
}
```

the ABI interface for Poo is Poo(address) when the Solidity interface is still Poo(Foo).

Accessor for Arrays

PT For Arrays the accessor is generated which accepts the index as parameter and returns an array element

```
contract test {
    uint[3] public data;
    function test() {
        data[0] = 0;
```

```
        data[1] = 1;
        data[2] = 2;
    }
}
```

In the above contract if you tried to call the data(1) method of the test you would obtain the result 1.

Serpent Features

Serpent is one of the high-level programming languages used to write Ethereum contracts. The language, as suggested by its name, is designed to be very similar to Python; it is intended to be maximally clean and simple, combining many of the efficiency benefits of a low-level language with ease-of-use in programming style, and at the same time adding special domain-specific features for contract programming. The latest version of the Serpent compiler, available on github, is written in C++, allowing it to be easily included in any client.

This tutorial assumes basic knowledge of how Ethereum works, including the concept of blocks, transactions, contracts and messages and the fact that contracts take a byte array as input and provide a byte array as output. If you do not, then go here for a basic tutorial.

Differences Between Serpent and Python

The important differences between Serpent and Python are:

- Python numbers have potentially unlimited size, Serpent numbers wrap around 2^{256}. For example, in Serpent the expression `3^(2^254)` suprisingly evaluates to 1, even though in reality the actual integer is too large to be recorded in its entirety within the universe.
- Serpent has no decimals.
- Serpent has no list comprehensions (expressions like `[x**2 for x in my_list]`), dictionaries or most other advanced features
- Serpent has no concept of first-class functions. Contracts do have functions, and can call their own functions, but variables (except storage) do not persist across calls.
- Serpent has a concept of persistent storage variables
- Serpent has an `extern` statement used to call functions from other contracts

Serpent Tutorials

Let's write our first contract in serpent. Paste the following into a file called "mul2.se":

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

This contract is a simple two lines of code, and defines a function. Functions can be called either by transactions or by other contracts, and are the way that Serpent contracts provide an "interface" to other contracts and to transactions; for example, a contract defining a currency might have functions `send(to, value)` and `check_balance(address)`.

Additionally, the Pyethereum testing environment that we will be using simply assumes that data input and output are in this format.

Now, let's try actually compiling the code. Type:

```
> serpent compile mul2.se
602380600b600039602e5660003560001a60001415602257602060016020376002602051026040526020604040f25b5b6000f2
```

And there we go, that's the hexadecimal form of the code that you can put into transactions. Or, if you want to see opcodes:

```
> serpent pretty_compile mul2.se
[PUSH1, 35, DUP1, PUSH1, 11, PUSH1, 0, CODECOPY, PUSH1, 46, JUMP, PUSH1, 0, CALLDATALOAD, PUSH1, 0, BYTE, PUSH1, 0, EQ,
```

Alternatively, you can compile to LLL to get an intermediate representation:

```
> serpent compile_to_lll mul2.se
(seq
  (return 0
    (lll
      (seq
        (def ('double 'x)
          (seq
            (set '_temp7_1 (mul (get 'x) 2))
            (return (ref '_temp7_1) 32)
          )
        )
      )
      0
    )
  )
)
```

This shows you the machinery that is going on inside. As with most contracts, the outermost layer of code exists only to copy the data of the inner code during initialization and return it, since the code returned during initialization is the code that will be executed every time the contract is called; in the EVM you can see this with the `CODECOPY` opcode, and in LLL this corresponds to the `lll` meta-operation. In the innermost layer, we set a variable `'_temp7_1` to equal twice the input, and then supply the memory address of that variable, and the length 32, to the `RETURN` opcode. The `def` mechanism itself is later translated into opcodes that actually unpack the bytes of the transaction data into the variables seen in the LLL.

Now, what if you want to actually run the contract? That is where pyethereum comes in. Open up a Python console in the same directory, and run:

```
> from pyethereum import tester as t
> s = t.state()
> c = s.contract('mul2.se')
> s.send(t.k0, c, 0, funid=0, abi=[42])
[84]
> s.send(t.k0, c, 0, funid=0, abi=[303030])
[606060]
```

The second line initializes a new state (ie. a genesis block). The third line creates a new contract, and sets `c` to its address. The fourth line sends a transaction from private key zero (the `tester` submodule includes ten pre-made key/address pairs at `k0`, `a0`, `k1`, `a1`, etc) to the address of the contract with 0 wei, calling function 0 (ie. the first function in the contract) with data 42, and we see 84 predictably come out.

Note that if you want to send a transaction to such a contract in the testnet or livenet, you will need to package up the transaction data for "call function 0 with data 42". The command line instruction for this is:

```
> serpent encode_abi 0 42
0000000000000000000000000000000000000000000000000000000000000002a
```

The first byte (ie. two hex characters) always represents the function ID; for example, if we were calling the function with ID 9 in some contract instead, we would do:

```
> serpent encode_abi 9 42
0900000000000000000000000000000000000000000000000000000000000002a
```

Then, after that, each argument is packed into 32 bytes. So if we wanted to encode two variables we would do:

```
> serpent encode_abi 9 42 20125
0900000000000000000000000000000000000000000000000000000000000002a0000000000000000000000000000000000000000000000000000000000004e5d
```

Note that there are advanced features in Serpent for making each argument take up less than 32 bytes, but we will ignore these for now.

Example: Name Registry

Having a multiply-by-two function on the blockchain is kind of boring. So let's do something marginally more interesting: a name registry. The main function will be a `register(key, value)` operation which checks if a given key was already taken, and if is unoccupied then register it with the desired value and return 1; if the key is already occupied return 0. We will also add a second function to check the value associated with a particular key. Let's try it out:

```
def register(key, value):
    # Key not yet claimed
    if not self.storage[key]:
        self.storage[key] = value
        return(1)
    else:
        return(0)  # Key already claimed

def ask(key):
    return(self.storage[key])
```

Here, we see a few parts in action. First, we have the `key` and `value` variables that the function takes as arguments. The third line is a comment; it does not get compiled and only serves to remind you what the code does. Then, we have a

standard if/else clause, which checks if `self.storage[key]` is zero (ie. unclaimed), and if it is then it sets `self.storage[key] = value` and returns 1. Otherwise, it returns zero (which we expressed as a literal array just to exhibit this other style of returning; it's more cumbersome but lets you do return multiple values). `self.storage` is also a pseudo-array, acting like an array but without any particular memory location.

Now, paste the code into "namecoin.se", if you wish try compiling it to LLL, opcodes or EVM, and let's try it out in the pyethereum tester environment:

```
> from pyethereum import tester as t
> s = t.state()
> c = s.contract('namecoin.se')
> s.send(t.k0, c, 0, funid=0, abi=[0x67656f726765, 45])
[1]
> s.send(t.k1, c, 0, funid=0, abi=[0x67656f726765, 20])
[0]
> s.send(t.k2, c, 0, funid=0, abi=[0x6861727279, 65])
[1]
> s.send(t.k3, c, 0, funid=1, abi=[0x6861727279])
[65]
```

`funid` is the index of the function, in the order in which it appears in the contract; here, 0 is for `register` and 1 is for `ask`. If we wanted to encode the transaction data for that first call, we would do:

```
> serpent encode_abi 0 0x67656f726765 45
0000000000000000000000000000000000000000000000000000067656f726765000000000000000000000000000000000000000000000000000000000000002d
```

Including files, and calling other contracts

Once your projects become larger, you will not want to put everything into the same file; things become particularly inconvenient when one piece of code needs to create a contract. Fortunately, the process for splitting code into multiple files is quite simple. Make the following two files:

mul2.se:

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

returnten.se:

```
extern mul2: [double]

MUL2 = create('mul2.se')
return(MUL2.double(5, as=mul2))
```

And open Python:

```
> from pyethereum import tester as t
> s = t.state()
> c = s.contract('returnten.se')
> s.send(t.k0, c, 0, [])
[10]
```

Note that here we introduced several new features. Particularly:

- The `create` command to create a contract using code from another file
- The `extern` keyword to declare a class of contract for which we know the names of the functions
- The interface for calling other contracts

`create` is self-explanatory; it creates a contract and returns the address to the contract. The way `extern` works is that you declare a class of contract, in this case `mul2`, and then list in an array the names of the functions, in this case just `double`. From there, given any variable containing an address, you can do `x.double(arg1, as=mul2)` to call the address stored by that variable. The `as` keyword provides the class that we're using (to determine which function ID `double` corresponds to; there may be another class where `double` corresponds to 7). You do have the option of omitting the `as` keyword, but this comes at the peril of ambiguity if you are using two classes that have the same function name. The arguments are the values provided to the function. If you provide too few arguments, the rest are filled to zero, and if you provide too many the extra ones are ignored. Function calling also has some other optional arguments:

- `gas=12414` - call the function with 12414 gas instead of the default (all gas)
- `value=10^19` - send 10^19 wei (10 ether) along with the message
- `data=x`, `datasz=5` - call the function with 5 values from the array `x`; note that this replaces other function arguments. `data` without `datasz` is illegal
- `outsz=7` - by default, Serpent processes the output of a function by taking the first 32 bytes and returning it as a value. However, if `outsz` is used as here, the function will instead return an array containing 7 values; if you type `y = x.fun(arg1, outsz=7)` then you will be able to access the output via `y[0]`, `y[1]`, etc.

Another, similar, operation to `create` is `(inset 'filename')`, which simply puts code into a particular place without adding a separate contract.

Storage data structures

In more complicated contracts, you will often want to store data structures in storage to represent certain objects. For example, you might have a decentralized exchange contract that stores the balances of users in multiple currencies, as well as open bid and ask orders where each order has a price and a quantity. For this, Serpent has a built-in mechanism for defining your own structures. For example, in such a decentralized exchange contract you might see:

```
data user_balances[][]
data orders[](buys[](user, price, quantity), sells[](user, price, quantity))
```

Then, you might do something like:

```
def fill_buy_order(currency, order_id):
    # Available amount buyer is willing to buy
    q = self.orders[currency].buys[order_id].quantity
    # My balance in the currency
    bal = self.user_balances[msg.sender][currency]
    # The buyer
    buyer = self.orders[currency].buys[order_id].user
    if q > 0:
        # The amount we can actually trade
        amount = min(q, bal)
        # Trade the currency against the base currency
        self.user_balances[msg.sender][currency] -= amount
        self.user_balances[buyer][currency] += amount
        self.user_balances[msg.sender][0] += amount * self.orders[currency].buys[order_id].price
        self.user_balances[buyer][0] -= amount * self.orders[currency].buys[order_id].price
        # Reduce the remaining quantity on the order
        self.orders[currency].buys[order_id].quantity -= amount
```

Notice how we define the data structures at the top, and then use them throughout the contract. These data structure gets and sets are converted into storage accesses in the background, so the data structures are persistent.

The language for doing data structures is simple. First, we can do simple variables:

```
data blah

x = self.blah
self.blah = x + 1
```

Then, we can do arrays, both finite and infinite:

```
data blah[1243]
data blaz[]

x = self.blah[505]
y = self.blaz[3**160]
self.blah[125] = x + y
```

Note that finite arrays are always preferred, because it will cost less gas to calculate the storage index associated with a finite array index lookup. And we can do tuples, where each element of the tuple is itself a valid data structure:

```
data body(head(eyes[2], nose, mouth), arms[2], legs[2])

x = self.body.head.nose
y = self.body.arms[1]
```

And we can do arrays of tuples:

```
data bodies[100](head(eyes[2], nose, mouth), arms[2](fingers[5], elbow), legs[2])

x = self.bodies[45].head.eyes[1]
y = self.bodies[x].arms[1].fingers[3]
```

Note that the following is unfortunately not legal:

```
data body(head(eyes[2], nose, mouth), arms[2], legs[2])

x = self.body.head
y = x.eyes[0]
```

Accesses have to descend fully in a single statement. To see how this could be used in a simpler example, let's go back to our name registry, and upgrade it so that when a user registers a key they become the owner of that key, and the owner of a key has the ability to (1) transfer ownership, and (2) change the value. We'll remove the return values here for simplicity.

```
data registry[](owner, value)

def register(key):
    # Key not yet claimed
    if not self.registry[key].owner:
        self.registry[key].owner = msg.sender

def transfer_ownership(key, new_owner):
    if self.registry[key].owner == msg.sender:
        self.registry[key].owner = new_owner

def set_value(key, new_value):
    if self.registry[key].owner == msg.sender:
        self.registry[key].value = new_value
```

```
def ask(key):
    return([self.registry[key].owner, self.registry[key].value], 2)
```

Note that in the last ask command, the function returns an array of 2 values. If you wanted to call the registry, you would have needed to do something like `o = registry.ask(key, outsz=2)` and you could have then used `o[0]` and `o[1]` to recover the owner and value.

Simple arrays in memory

The syntax for arrays in memory are different: they can only be finite and cannot have tuples or more complicated structures.

Example:

```
def bitwise_or(x, y):
    blah = array(1243)
    blah[567] = x
    blah[568] = y
    blah[569] = blah[567] | blah[568]
    return(blah[569])
```

There are also two functions for dealing with arrays:

```
len(x)
```

Returns the length of array x.

```
shrink_array(x, s)
```

Shrink the length of array x to size s (useful just before passing the array as an argument to a function).

Arrays and Functions

Functions can also take arrays as arguments, and return arrays.

```
def compose(inputs:a):
    return(inputs[0] + inputs[1] * 10 + inputs[2] * 100)

def decompose(x):
    return([x % 10, (x % 100) / 10, x / 100]:a)
```

Putting the `:a` after a function argument means it is an array, and putting it inside a return statement returns the value as an array (just doing `return([x,y,z])` would return the integer which is the memory location of the array).

If a contract calls itself, then it will autodetect which arguments should be arrays and parse them accordingly, so this works fine:

```
def compose(inputs:a, radix):
    return(inputs[0] + inputs[1] * radix + inputs[1] * radix ** 2)

def main():
    return self.compose([1,2,3,4,5], 100)
```

However, if you want to call another contract that takes arrays as arguments, then you will need to put a "signature" into the extern declaration:

```
extern composer: [compose:ai, main]
```

Here, `ai` means "an array followed by an integer".

Strings

There are two types of strings in Serpent: short strings, eg. `"george"` , and long strings, eg. `text("afjqwhruqwhurhqkwrhguqwhrkuqwrkqwhwhrugquwrguwegtwetwet")` . Short strings, given simply in quotes as above, are treated as numbers; long strings, surrounded by the `text` keyword as above, are treated as array-like objects; you can do `getch(str, index)` and `setch(str, index)` to manipulate characters in strings (doing `str[0]` will treat the string as an array and try to fetch the first 32 characters as a number).

To use strings as function arguments or outputs, use the `s` tag, much like you would use `a` for arrays. `len(s)` gives you the length of a string, and `shrink` works for strings the same way as for arrays too.

Macros

WARNING: Relatively new/untested feature, here be ~~dragons~~ serpents

Macros allow you to create rewrite rules which provide additional expressivity to the language. For example, suppose that you wanted to create a command that would compute the median of three values. You could simply do:

```
macro median($a, $b, $c):
    min(min(smax($a, $b), max($a, $c)), max($b, $c))
```

Then, if you wanted to use it somewhere in your code, you just do:

```
x = median(5, 9, 7)
```

Or to take the max of an array:

```
macro maxarray($a:$asz):
    $m = 0
    $i = 0
    while i < $asz:
        $m = max($m, $a[i])
        $i += 1
    $m

x = maxarray([1, 9, 5, 6, 2, 4]:6)
```

For a highly contrived example of just how powerful macros can be, see https://github.com/ethereum/serpent/blob/poc7/examples/peano.se

Note that macros are not functions; they are copied into code every time they are used. Hence, if you have a long macro, you may instead with to make the macro call an actual function. Additionally, note that the dollar signs on variables are important; if you omit a dollar sign in the pattern $a then the macro will only match a variable actually called a. You can also create dollar sign variables that are in the substitution pattern, but not the search pattern; this will generate a variable with a random prefix each instance of the macro. You can also create new variables without a dollar sign inside a substitution

pattern, but then the same variable will be shared across all instances of the pattern and with uses of that variable outside the pattern.

Types

WARNING: Relatively new/untested feature, here be ~~dragons~~ serpents

An excellent compliment to macros is Serpent's ghetto type system, which can be combined with macros to produce quite interesting results. Let us simply show this with an example:

```
type float: [a, b, c]

macro float($x) + float($y):
    float($x + $y)

macro float($x) - float($y):
    float($x - $y)

macro float($x) * float($y):
    float($x * $y / 2^32)

macro float($x) / float($y):
    float($x * 2^32 / $y)

macro unfloat($x):
    $x / 2^32

macro floatfy($x):
    float($x * 2^32)

macro float($x) = float($y):
    $x = $y

macro with(float($x), float($y), $z):
    with($x, $y, $z)

a = floatfy(25)
b = a / floatfy(2)
c = b * b
return(unfloat(c))
```

This returns 156, the integer portion of 12.5^2. A purely integer-based version of this code would have simply returned 144. An interesting use case would be rewriting the elliptic curve signature pubkey recovery code using types in order to make the code neater by making all additions and multiplications implicitly modulo P, or using long integer types to do RSA and other large-value-based cryptography in EVM code.

Miscellaneous

Additional Serpent coding examples can be found here: https://github.com/ethereum/serpent/tree/master/examples

The three other useful features in the tester environment are:

- Block access - you can dig around `s.block` to see block data (eg. `s.block.number`, `s.block.get_balance(addr)`, `s.block.get_storage_data(addr, index)`)
- Snapshots - you can do `x = s.snapshot()` and `s.revert(x)`
- Advancing blocks - you can do `s.mine(100)` and 100 blocks magically pass by with a 60-second interval between blocks. `s.mine(100, addr)` mines into a particular address.
- Full block data dump - type `s.block.to_dict()`

Serpent also gives you access to many "special variables"; the full list is:

- `tx.origin` - the sender of the transaction

- `tx.gas` - gas remaining
- `tx.gasprice` - gas price of the transaction
- `msg.sender` - the sender of the message
- `msg.value` - the number of wei (smallest units of ether) sent with the message
- `self` - the contract's own address
- `self.balance` - the contract's balance
- `x.balance` (for any x) - that account's balance
- `block.coinbase` - current block miner's address
- `block.timestamp` - current block timestamp
- `block.prevhash` - previous block hash
- `block.difficulty` - current block difficulty
- `block.number` - current block number
- `block.gaslimit` - current block gaslimit

Serpent recognises the following "special functions":

- `init` - executed upon contract creation
- `shared` - executed before running `init` and user functions
- `any` - executed before any user functions

There are also special commands for a few crypto operations; particularly:

- `addr = ecrecover(h, v, r, s)` - determines the address that produced the elliptic curve signature `v, r, s` of the hash `h`
- `x = sha256(a, 4)` - returns the sha256 hash of the 128 bytes consisting of the 4-item array starting from `a`
- `x = ripemd160(a, 4)` - same as above but for ripemd160
- To hash an arbitrary number of bytes, use chars syntax. Example: `x = sha256([0xf1fc122bc7f5d74df2b9441a42a14695000000000000000000000000000000000], chars=16)` - returns the sha256 of the first 16 bytes. Note: padding with trailing zeroes, otherwise the first 16 bytes will be zeroes, and the sha256 of it will be computed instead of the desired.

Tips

- If a function is not returning the result you expect, double-check that all variables are correct: there is no error/warning when using an undeclared variable.
- `Invalid argument count or LLL function` usually means you just called foo() instead of `self.foo()`.
- Sometimes you may be intending to use unsigned operators. eg div() and lt() instead of '/' and '<'.

Solidity Tutorials

Solidity is a high-level language whose syntax is similar to that of JavaScript and it is designed to compile to code for the Ethereum Virtual Machine. This tutorial provides a basic introduction to Solidity and assumes some knowledge of the Ethereum Virtual Machine and programming in general. For more details, please see the Solidity specficiation (yet to be written). This tutorial does not cover features like the natural language documentation or formal verification and is also not meant as a final specification of the language.

You can start using Solidity in your browser, with no need to download or compile anything. This application only supports compilation - if you want to run the code or inject it into the blockchain, you have to use a client like AlethZero.

Simple Example

```
contract SimpleStorage {
    uint storedData;
    function set(uint x) {
        storedData = x;
    }
    function get() constant returns (uint retVal) {
        return storedData;
    }
}
```

`uint storedData` declares a state variable called `storedData` of type `uint` (unsigned integer of 256 bits) whose position in storage is automatically allocated by the compiler. The functions `set` and `get` can be used to modify or retrieve the value of the variable.

Subcurrency Example

```
contract Coin {
    address minter;
    mapping (address => uint) balances;
    function Coin() {
        minter = msg.sender;
    }
    function mint(address owner, uint amount) {
        if (msg.sender != minter) return;
        balances[owner] += amount;
    }
    function send(address receiver, uint amount) {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
    }
    function queryBalance(address addr) constant returns (uint balance) {
        return balances[addr];
    }
}
```

This contract introduces some new concepts. One of them is the `address` type, which is a 160 bit value that does not allow any arithmetic operations. Furthermore, the state variable `balance` is of a complex datatype that maps addresses to unsigned integers. Mappings can be seen as hashtables which are virtually initialized such that every possible key exists and is mapped to a value whose byte-representation is all zeros. The special function `Coin` is the constructor which is run during creation of the contract and cannot be called afterwards. It permanently stores the address of the person creating the contract: Together with `tx` and `block`, `msg` is a magic global variable that contains some properties which allow

access to the world outside of the contract. The function `queryBalance` is declared `constant` and thus is not allowed to modify the state of the contract (note that this is not yet enforced, though). In Solidity, return "parameters" are named and essentially create a local variable. So to return the balance, we could also just use `balance = balances[addr];` without any return statement.

Comments

Single-line comments (`//`) and multi-line comments (`/*...*/`) are possible, while triple-slash comments (`///`) right in front of function declarations introduce NatSpec comments (which are not covered here).

Types

The currently implemented (elementary) types are booleans (`bool`), integer and fixed-length string/byte array (bytes0 to bytes32) types. The integer types are signed and unsigned integers of various bit widths (`int8` / `uint8` to `int256` / `uint256` in steps of 8 bits, where `uint` / `int` are aliases for `uint256` / `int256`) and addresses (of 160 bits).

Comparisons (`<=` , `!=` , `==` , etc.) always yield booleans which can be combined using `&&` , `||` and `!` . Note that the usual short-circuiting rules apply for `&&` and `||` , which means that for expressions of the form `(0 < 1 || fun())` , the function is actually never called.

If an operator is applied to different types, the compiler tries to implicitly convert one of the operands to the type of the other (the same is true for assignments). In general, an implicit conversion is possible if it makes sense semantically and no information is lost: `uint8` is convertible to `uint16` and `int120` to `int256` , but `int8` is not convertible to `uint256` . Furthermore, unsigned integers can be converted to bytes of the same or larger size, but not vice-versa. Any type that can be converted to `uint160` can also be converted to `address` .

If the compiler does not allow implicit conversion but you know what you are doing, an explicit type conversion is sometimes possible:

```
int8 y = -3;
uint x = uint(y);
```

At the end of this code snippet, `x` will have the value `0xfffff..fd` (64 hex characters), which is -3 in two's complement representation of 256 bits.

For convenience, it is not always necessary to explicitly specify the type of a variable, the compiler automatically infers it from the type of the first expression that is assigned to the variable:

```
uint20 x = 0x123;
var y = x;
```

Here, the type of `y` will be `uint20` . Using `var` is not possible for function parameters or return parameters. State variables of integer and bytesXX types can be declared as constant.

```
uint constant x = 32;
bytes3 constant text = "abc";
```

Integer Literals

The type of integer literals is not determined as long as integer literals are combined with themselves. This is probably best explained with examples:

```
var x = 1 - 2;
```

The value of `1 - 2` is `-1`, which is assigned to `x` and thus `x` receives the type `int8` -- the smallest type that contains `-1`. The following code snippet behaves differently, though:

```
var one = 1;
var two = 2;
var x = one - two;
```

Here, `one` and `two` both have type `uint8` which is also propagated to `x`. The subtraction inside the type `uint8` causes wrapping and thus the value of `x` will be `255`.

It is even possible to temporarily exceed the maximum of 256 bits as long as only integer literals are used for the computation:

```
var x = (0xffffffffffffffffff * 0xffffffffffffffffff) * 0;
```

Here, `x` will have the value `0` and thus the type `uint8`.

Ether and Time Units

A literal number can take a suffix of `wei`, `finney`, `szabo` or `ether` to convert between the subdenominations of ether, where Ether currency numbers without a postfix are assumed to be "wei", e.g. `2 ether == 2000 finney` evaluates to `true`.

Furthermore, suffixes of `seconds`, `minutes`, `hours`, `days`, `weeks` and `years` can be used to convert between units of time where seconds are the base unit and units are converted naively (i.e. a year is always exactly 365 days, etc.).

Control Structures

Most of the control structures from C/JavaScript are available in Solidity except for `switch` (not planned) and `goto` (note that it's called Solidity). So there is: `if`, `else`, `while`, `for`, `break`, `continue`, `return`. Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so `if (1) { ... }` is *not* valid Solidity.

Function Calls

Functions of the current contract can be called directly, also recursively, as seen in this nonsensical example:

```
contract c {
  function g(uint a) returns (uint ret) { return f(); }
  function f() returns (uint ret) { return g(7) + f(); }
}
```

The expression `this.g(8);` is also a valid function call, but this time, the function will be called via a message call and not directly via jumps. When calling functions of other contracts, the amount of Wei sent with the call and the gas can be specified:

```
contract InfoFeed {
  function info() returns (uint ret) { return 42; }
}
contract Consumer {
  InfoFeed feed;
  function setFeed(address addr) { feed = InfoFeed(addr); }
  function callFeed() { feed.info.value(10).gas(800)(); }
}
```

Note that the expression `InfoFeed(addr)` performs an explicit type conversion stating that "we know that the type of the contract at the given address is `InfoFeed`" and this does not execute a constructor. Be careful in that `feed.info.value(10).gas(800)` only (locally) set the value and amount of gas sent with the function call and only the parentheses at the end perform the actual call.

Function call arguments can also be given by name, in any order:

```
contract c {
function f(uint key, uint value) { ... }
function g() {
  f({value: 2, key: 3});
}
}
```

The names for function parameters and return parameters are optional.

```
contract test {
  function func(uint k, uint) returns(uint){
    return k;
  }
}
```

Special Variables and Functions

There are special variables and functions which always exist in the global namespace.

Block and Transaction Properties

- `block.coinbase` (`address`): current block miner's address
- `block.difficulty` (`uint`): current block difficulty
- `block.gaslimit` (`uint`): current block gaslimit
- `block.number` (`uint`): current block number
- `block.blockhash` (`function(uint) returns (bytes32)`): hash of the given block
- `block.timestamp` (`uint`): current block timestamp
- `msg.data` (`bytes`): complete calldata
- `msg.gas` (`uint`): remaining gas
- `msg.sender` (`address`): sender of the message (current call)
- `msg.value` (`uint`): number of wei sent with the message
- `now` (`uint`): current block timestamp (alias for `block.timestamp`)
- `tx.gasprice` (`uint`): gas price of the transaction
- `tx.origin` (`address`): sender of the transaction (full call chain)

Cryptographic Functions

- `sha3(...) returns (bytes32)` : compute the SHA3 hash of the (tightly packed) arguments

- `sha256(...) returns (bytes32)` : compute the SHA256 hash of the (tightly packed) arguments
- `ripemd160(...) returns (bytes20)` : compute RIPEMD of 256 the (tightly packed) arguments
- `ecrecover(bytes32, byte, bytes32, bytes32) returns (address)` : recover public key from elliptic curve signature

In the above, "tightly packed" means that the arguments are concatenated without padding, i.e. `sha3("ab", "c") == sha3("abc") == sha3(0x616263) == sha3(6382179) = sha3(97, 98, 99)` . If padding is needed, explicit type conversions can be used.

Contract Related

- `this` (current contract's type): the current contract, explicitly convertible to `address`
- `suicide(address)` : suicide the current contract, sending its funds to the given address

Furthermore, all functions of the current contract are callable directly including the current function.

Functions on addresses

It is possible to query the balance of an address using the property `balance` and to send Ether (in units of wei) to an address using the `send` function:

```
address x = 0x123;
if (x.balance < 10 && address(this).balance >= 10) x.send(10);
```

Furthermore, to interface with contracts that do not adhere to the ABI (like the classic NameReg contract), the function `call` is provided which takes an arbitrary number of arguments of any type. These arguments are ABI-serialized (i.e. also padded to 32 bytes). One exception is the case where the first argument is encoded to exactly four bytes. In this case, it is not padded to allow the use of function signatures here.

```
address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
nameReg.call("register", "MyName");
nameReg.call(bytes4(sha3("fun(uint256)")), a);
```

Note that contracts inherit all members of address, so it is possible to query the balance of the current contract using `this.balance` .

Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

Arrays

Both variably and fixed size arrays are supported in storage and as parameters of external functions:

```
contract ArrayContract {
  uint[2**20] m_aLotOfIntegers;
  bool[2][] m_pairsOfFlags;
  function setAllFlagPairs(bool[2][] newPairs) {
    // assignment to array replaces the complete array
    m_pairsOfFlags = newPairs;
```

```
  }
  function setFlagPair(uint index, bool flagA, bool flagB) {
    // access to a non-existing index will stop execution
    m_pairsOfFlags[index][0] = flagA;
    m_pairsOfFlags[index][1] = flagB;
  }
  function changeFlagArraySize(uint newSize) {
    // if the new size is smaller, removed array elements will be cleared
    m_pairsOfFlags.length = newSize;
  }
  function clear() {
    // these clear the arrays completely
    delete m_pairsOfFlags;
    delete m_aLotOfIntegers;
    // identical effect here
    m_pairsOfFlags.length = 0;
  }
  bytes m_byteData;
  function byteArrays(bytes data) external {
    // byte arrays ("bytes") are different as they are stored without padding,
    // but can be treated identical to "uint8[]"
    m_byteData = data;
    m_byteData.length += 7;
    m_byteData[3] = 8;
    delete m_byteData[2];
  }
}
```

Structs

Solidity provides a way to define new types in the form of structs, which is shown in the following example:

```
contract CrowdFunding {
  struct Funder {
    address addr;
    uint amount;
  }
  struct Campaign {
    address beneficiary;
    uint fundingGoal;
    uint numFunders;
    uint amount;
    mapping (uint => Funder) funders;
  }
  uint numCampaigns;
  mapping (uint => Campaign) campaigns;
  function newCampaign(address beneficiary, uint goal) returns (uint campaignID) {
    campaignID = numCampaigns++; // campaignID is return variable
    Campaign c = campaigns[campaignID];  // assigns reference
    c.beneficiary = beneficiary;
    c.fundingGoal = goal;
  }
  function contribute(uint campaignID) {
    Campaign c = campaigns[campaignID];
    Funder f = c.funders[c.numFunders++];
    f.addr = msg.sender;
    f.amount = msg.value;
    c.amount += f.amount;
  }
  function checkGoalReached(uint campaignID) returns (bool reached) {
    Campaign c = campaigns[campaignID];
    if (c.amount < c.fundingGoal)
      return false;
    c.beneficiary.send(c.amount);
    c.amount = 0;
    return true;
  }
}
```

The contract does not provide the full functionality of a crowdfunding contract, but it contains the basic concepts necessary

to understand structs. Struct types can be used as value types for mappings and they can itself contain mappings (even the struct itself can be the value type of the mapping, although it is not possible to include a struct as is inside of itself). Note how in all the functions, a struct type is assigned to a local variable. This does not copy the struct but only store a reference so that assignments to members of the local variable actually write to the state.

Enums

Enums are another way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The variable of enum type can be declared as constant.

```
contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choices;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;
    function setGoStraight()
    {
        choices = ActionChoices.GoStraight;
    }
    function getChoice() returns (uint)
    {
        return uint(choices);
    }
    function getDefaultChoice() returns (uint)
    {
        return uint(defaultChoice);
    }
}
```

Interfacing with other Contracts

There are two ways to interface with other contracts: Either call a method of a contract whose address is known or create a new contract. Both uses are shown in the example below. Note that (obviously) the source code of a contract to be created needs to be known, which means that it has to come before the contract that creates it (and cyclic dependencies are not possible since the bytecode of the new contract is actually contained in the bytecode of the creating contract).

```
contract OwnedToken {
  // TokenCreator is a contract type that is defined below. It is fine to reference it
  // as long as it is not used to create a new contract.
  TokenCreator creator;
  address owner;
  bytes32 name;
  function OwnedToken(bytes32 _name) {
    address nameReg = 0x72ba7d8e73fe8eb666ea66babc8116a41bfb10e2;
    nameReg.call("register", _name);
    owner = msg.sender;
    // We do an explicit type conversion from `address` to `TokenCreator` and assume that the type of
    // the calling contract is TokenCreator, there is no real way to check.
    creator = TokenCreator(msg.sender);
    name = _name;
  }
  function changeName(bytes32 newName) {
    // Only the creator can alter the name -- contracts are explicitly convertible to addresses.
    if (msg.sender == address(creator)) name = newName;
  }
  function transfer(address newOwner) {
    // Only the current owner can transfer the token.
    if (msg.sender != owner) return;
    // We also want to ask the creator if the transfer is fine.
    // Note that this calls a function of the contract defined below.
    // If the call fails (e.g. due to out-of-gas), the execution here stops
    // immediately (the ability to catch this will be added later).
    if (creator.isTokenTransferOK(owner, newOwner))
      owner = newOwner;
  }
```

```
}
contract TokenCreator {
  function createToken(bytes32 name) returns (address tokenAddress) {
    // Create a new Token contract and return its address.
    return address(new OwnedToken(name));
  }
  function changeName(address tokenAddress, bytes32 name) {
    // We need an explicit type conversion because contract types are not part of the ABI.
    OwnedToken token = OwnedToken(tokenAddress);
    token.changeName(name);
  }
  function isTokenTransferOK(address currentOwner, address newOwner) returns (bool ok) {
    // Check some arbitrary condition.
    address tokenAddress = msg.sender;
    return (sha3(newOwner) & 0xff) == (bytes20(tokenAddress) & 0xff);
  }
}
```

Constructor Arguments

A Solidity contract expects constructor arguments after the end of the contract data itself. This means that you pass the arguments to a contract by putting them after the compiled bytes as returned by the compiler in the usual ABI format.

Contract Inheritance

Solidity supports multiple inheritance by copying code including polymorphism. Details are given in the following example.

```
contract owned {
    function owned() { owner = msg.sender; }
    address owner;
}

// Use "is" to derive from another contract. Derived contracts can access all members
// including private functions and storage variables.
contract mortal is owned {
    function kill() { if (msg.sender == owner) suicide(owner); }
}

// These are only provided to make the interface known to the compiler.
contract Config { function lookup(uint id) returns (address adr) {} }
contract NameReg { function register(bytes32 name) {} function unregister() {} }

// Multiple inheritance is possible. Note that "owned" is also a base class of
// "mortal", yet there is only a single instance of "owned" (as for virtual
// inheritance in C++).
contract named is owned, mortal {
    function named(bytes32 name) {
        address ConfigAddress = 0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970;
        NameReg(Config(ConfigAddress).lookup(1)).register(name);
    }

// Functions can be overridden, both local and message-based function calls take
// these overrides into account.
    function kill() {
        if (msg.sender == owner) {
            address ConfigAddress = 0xd5f9d8d94886e70b06e474c3fb14fd43e2f23970;
            NameReg(Config(ConfigAddress).lookup(1)).unregister();
// It is still possible to call a specific overridden function.
            mortal.kill();
        }
    }
}

// If a constructor takes an argument, it needs to be provided in the header.
contract PriceFeed is owned, mortal, named("GoldFeed") {
   function updateInfo(uint newInfo) {
      if (msg.sender == owner) info = newInfo;
   }
```

```
    function get() constant returns(uint r) { return info; }

    uint info;
}
```

Note that above, we call `mortal.kill()` to "forward" the destruction request. The way this is done is problematic, as seen in the following example:

```
contract mortal is owned {
    function kill() { if (msg.sender == owner) suicide(owner); }
}
contract Base1 is mortal {
    function kill() { /* do cleanup 1 */ mortal.kill(); }
}
contract Base2 is mortal {
    function kill() { /* do cleanup 2 */ mortal.kill(); }
}
contract Final is Base1, Base2 {
}
```

A call to `Final.kill()` will call `Base2.kill` as the most derived override, but this function will bypass `Base1.kill`, basically because it does not even know about `Base1`. The way around this is to use `super`:

```
contract mortal is owned {
    function kill() { if (msg.sender == owner) suicide(owner); }
}
contract Base1 is mortal {
    function kill() { /* do cleanup 1 */ super.kill(); }
}
contract Base2 is mortal {
    function kill() { /* do cleanup 2 */ super.kill(); }
}
contract Final is Base1, Base2 {
}
```

If `Base1` calls a function of `super`, it does not simply call this function on one of its base contracts, it rather calls this function on the next base contract in the final inheritance graph, so it will call `Base2.kill()`. Note that the actual function that is called when using super is not known in the context of the class where it is used, although its type is known. This is similar for ordinary virtual method lookup.

Multiple Inheritance and Linearization

Languages that allow multiple inheritance have to deal with several problems, one of them being the Diamond Problem. Solidity follows the path of Python and uses "C3 Linearization" to force a specific order in the DAG of base classes. This results in the desirable property of monotonicity but disallows some inheritance graphs. Especially, the order in which the base classes are given in the `is` directive is important. In the following code, Solidity will give the error "Linearization of inheritance graph impossible".

```
contract X {}
contract A is X {}
contract C is A, X {}
```

The reason for this is that `C` requests `X` to override `A` (by specifying `A, X` in this order), but `A` itself requests to override `X`, which is a contradiction that cannot be resolved.

A simple rule to remember is to specify the base classes in the order from "most base-like" to "most derived".

Visibility Specifiers

Functions and storage variables can be specified as being `public`, `internal` or `private`, where the default for functions is `public` and `internal` for storage variables. In addition, functions can also be specified as `external`.

External: External functions are part of the contract interface and they can be called from other contracts and via transactions. An external function `f` cannot be called internally (i.e. `f()` does not work, but `this.f()` works). Furthermore, all function parameters are immutable.

Public: Public functions are part of the contract interface and can be either called internally or via messages. For public storage variables, an automatic accessor function (see below) is generated.

Inherited: Those functions and storage variables can only be accessed internally.

Private: Private functions and storage variables are only visible for the contract they are defined in and not in derived contracts.

```
contract c {
  function f(uint a) private returns (uint b) { return a + 1; }
  function setData(uint a) inherited { data = a; }
  uint public data;
}
```

Other contracts can call `c.data()` to retrieve the value of data in storage, but are not able to call `f`. Contracts derived from `c` can call `setData` to alter the value of `data` (but only in their own storage).

Accessor Functions

The compiler automatically creates accessor functions for all public state variables. The contract given below will have a function called `data` that does not take any arguments and returns a uint, the value of the state variable `data`. The initialization of state variables can be done at declaration.

```
contract test {
     uint public data = 42;
}
```

The next example is a bit more complex:

```
contract complex {
  struct Data { uint a; bytes3 b; mapping(uint => uint) map; }
  mapping(uint => mapping(bool => Data)) public data;
}
```

It will generate a function of the following form:

```
function data(uint arg1, bool arg2) returns (uint a, bytes3 b)
{
  a = data[arg1][arg2].a;
  b = data[arg1][arg2].b;
}
```

Note that the mapping in the struct is omitted because there is no good way to provide the key for the mapping.

Fallback Functions

A contract can have exactly one unnamed function. This function cannot have arguments and is executed on a call to the contract if none of the other functions matches the given function identifier (or if no data was supplied at all).

```
contract Test {
  function() { x = 1; }
  uint x;
}

contract Caller {
  function callTest(address testAddress) {
    Test(testAddress).send(0);
    // results in Test(testAddress).x becoming == 1.
  }
}
```

Function Modifiers

Modifiers can be used to easily change the behaviour of functions, for example to automatically check a condition prior to executing the function. They are inheritable properties of contracts and may be overridden by derived contracts.

```
contract owned {
  function owned() { owner = msg.sender; }
  address owner;

  // This contract only defines a modifier but does not use it - it will
  // be used in derived contracts.
  // The function body is inserted where the special symbol "_" in the
  // definition of a modifier appears.
  modifier onlyowner { if (msg.sender == owner) _ }
}
contract mortal is owned {
  // This contract inherits the "onlyowner"-modifier from "owned" and
  // applies it to the "kill"-function, which causes that calls to "kill"
  // only have an effect if they are made by the stored owner.
  function kill() onlyowner {
    suicide(owner);
  }
}
contract priced {
  // Modifiers can receive arguments:
  modifier costs(uint price) { if (msg.value >= price) _ }
}
contract Register is priced, owned {
  mapping (address => bool) registeredAddresses;
  uint price;
  function Register(uint initialPrice) { price = initialPrice; }
  function register() costs(price) {
    registeredAddresses[msg.sender] = true;
  }
  function changePrice(uint _price) onlyowner {
    price = _price;
  }
}
```

Multiple modifiers can be applied to a function by specifying them in a whitespace-separated list and will be evaluated in order. Explicit returns from a modifier or function body immediately leave the whole function, while control flow reaching the end of a function or modifier body continues after the "_" in the preceding modifier. Arbitrary expressions are allowed for modifier arguments and in this context, all symbols visible from the function are visible in the modifier. Symbols introduced in the modifier are not visible in the function (as they might change by overriding).

Events

Events allow the convenient usage of the EVM logging facilities. Events are inheritable members of contracts. When they are called, they cause the arguments to be stored in the transaction's log. Up to three parameters can receive the attribute `indexed` which will cause the respective arguments to be treated as log topics instead of data. The hash of the signature of the event is one of the topics except you declared the event with `anonymous` specifier. All non-indexed arguments will be stored in the data part of the log. Example:

```
contract ClientReceipt {
  event Deposit(address indexed _from, bytes32 indexed _id, uint _value);
  function deposit(bytes32 _id) {
    Deposit(msg.sender, _id, msg.value);
  }
}
```

Here, the call to `Deposit` will behave identical to `log3(msg.value, 0x50cb9fe53daa9737b786ab3646f04d0150dc50ef4e75f59509d83667ad5adb20, sha3(msg.sender), _id);`. Note that the large hex number is equal to the sha3-hash of "Deposit(address,bytes32,uint256)", the event's signature.

Layout of Storage

Statically-sized variables (everything except mapping and dynamically-sized array types) are laid out contiguously in storage starting from position `0`. Multiple items that need less than 32 bytes are packed into a single storage slot if possible, according to the following rules:

- The first item in a storage slot is stored lower-order aligned.
- Elementary types use only that many bytes that are necessary to store them.
- If an elementary type does not fit the remaining part of a storage slot, it is moved to the next storage slot.
- Structs and array data always start a new slot and occupy whole slots (but items inside a struct or array are packed tightly according to these rules).

The elements of structs and arrays are stored after each other, just as if they were given explicitly.

Due to their unpredictable size, mapping and dynamically-sized array types use a `sha3` computation to find the starting position of the value or the array data. These starting positions are always full stack slots.

The mapping or the dynamic array itself occupies an (unfilled) slot in storage at some position `p` according to the above rule (or by recursively applying this rule for mappings to mappings or arrays of arrays). For a dynamic array, this slot stores the number of elements in the array. For a mapping, the slot is unused (but it is needed so that two equal mappings after each other will use a different hash distribution). Array data is located at `sha3(p)` and the value corresponding to a mapping key `k` is located at `sha3(k . p)` where `.` is concatenation. If the value is again a non-elementary type, the positions are found by adding an offset of `sha3(k . p)`.

So for the following contract snippet:

```
contract c {
  struct S { uint a; uint b; }
  uint x;
  mapping(uint => mapping(uint => S)) data;
}
```

The position of `data[4][9].b` is at `sha3(uint256(9) . sha3(uint256(4) . uint(256(1))) + 1`.

Esoteric Features

There are some types in Solidity's type system that have no counterpart in the syntax. One of these types are the types of functions. But still, using `var` it is possible to have local variables of these types:

```
contract FunctionSelector {
  function select(bool useB, uint x) returns (uint z) {
    var f = a;
    if (useB) f = b;
    return f(x);
  }
  function a(uint x) returns (uint z) {
    return x * x;
  }
  function b(uint x) returns (uint z) {
    return 2 * x;
  }
}
```

Calling `select(false, x)` will compute `x * x` and `select(true, x)` will compute `2 * x`.

Internals - the Optimizer

The Solidity optimizer operates on assembly, so it can be and also is used by other languages. It splits the sequence of instructions into basic blocks at points that are hard to move. These are basically all instructions that modify change the control flow (jumps, calls, etc), instructions that have side effects apart from `MSTORE` and `SSTORE` (like `LOGi`, `EXTCODECOPY`, but also `CALLDATALOAD` and others). Inside of such a block, the instructions are analysed and every modification to the stack, to memory or storage is recorded as an expression which consists of an instruction and a list of arguments which are essentially pointers to other expressions. The main idea is now to find expressions that are always equal (or every input) and combine them into an expression class. The optimizer first tries to find each new expression in a list of already known expressions. If this does not work, the expression is simplified according to rules like `constant` + `constant` = `sum_of_constants` or `X` * 1 = `X`. Since this is done recursively, we can also apply the latter rule if the second factor is a more complex expression where we know that it will always evaluate to one. Modifications to storage and memory locations have to erase knowledge about storage and memory locations which are not known to be different: If we first write to location x and then to location y and both are input variables, the second could overwrite the first, so we actually do not know what is stored at x after we wrote to y. On the other hand, if a simplification of the expression x - y evaluates to a non-zero constant, we know that we can keep our knowledge about what is stored at x.

At the end of this process, we know which expressions have to be on the stack in the end and have list of modifications to memory and storage. From these expressions which are actually needed, a dependency graph is created and every operation that is not part of this graph is essentially dropped. Now new code is generated that applies the modifications to memory and storage in the order they were made in the original code (dropping modifications which were found not to be needed) and finally, generates all values that are required to be on the stack in the correct place.

These steps are applied to each basic block and the newly generated code is used as replacement if it is smaller. If a basic block is split at a JUMPI and during the analysis, the condition evaluates to a constant, the JUMPI is replaced depending on the value of the constant, and thus code like

```
var x = 7;
data[7] = 9;
if (data[x] != x + 2)
  return 2;
else
  return 1;
```

is simplified to code which can also be compiled from

```
data[7] = 9;
return 1;
```

even though the instructions contained a jump in the beginning.

Ethereum Whitepaper

A Next-Generation Smart Contract and Decentralized Application Platform

Satoshi Nakamoto's development of Bitcoin in 2009 has often been hailed as a radical development in money and currency, being the first example of a digital asset which simultaneously has no backing or "intrinsic value" and no centralized issuer or controller. However, another, arguably more important, part of the Bitcoin experiment is the underlying blockchain technology as a tool of distributed consensus, and attention is rapidly starting to shift to this other aspect of Bitcoin. Commonly cited alternative applications of blockchain technology include using on-blockchain digital assets to represent custom currencies and financial instruments ("colored coins"), the ownership of an underlying physical device ("smart property"), non-fungible assets such as domain names ("Namecoin"), as well as more complex applications involving having digital assets being directly controlled by a piece of code implementing arbitrary rules ("smart contracts") or even blockchain-based "decentralized autonomous organizations" (DAOs). What Ethereum intends to provide is a blockchain with a built-in fully fledged Turing-complete programming language that can be used to create "contracts" that can be used to encode arbitrary state transition functions, allowing users to create any of the systems described above, as well as many others that we have not yet imagined, simply by writing up the logic in a few lines of code.

Table of Contents

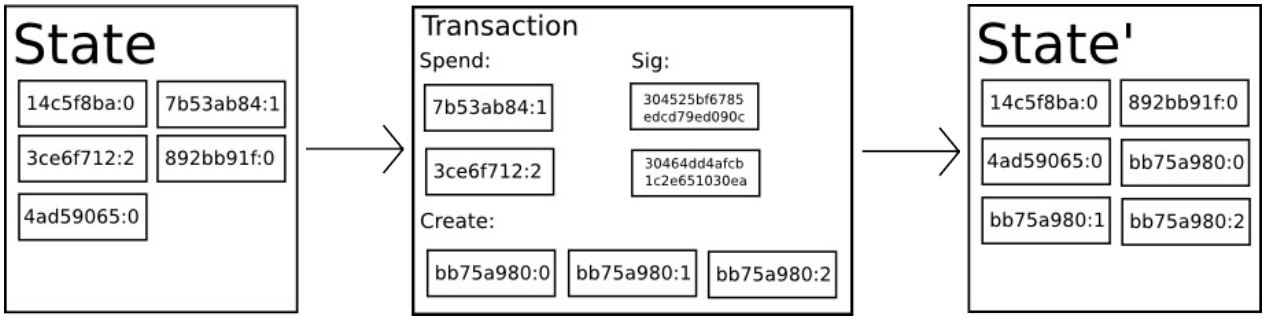
Introduction to Bitcoin and Existing Concepts

History

The concept of decentralized digital currency, as well as alternative applications like property registries, has been around for decades. The anonymous e-cash protocols of the 1980s and the 1990s, mostly reliant on a cryptographic primitive known as Chaumian blinding, provided a currency with a high degree of privacy, but the protocols largely failed to gain traction because of their reliance on a centralized intermediary. In 1998, Wei Dai's b-money became the first proposal to introduce the idea of creating money through solving computational puzzles as well as decentralized consensus, but the proposal was scant on details as to how decentralized consensus could actually be implemented. In 2005, Hal Finney introduced a concept of "reusable proofs of work", a system which uses ideas from b-money together with Adam Back's computationally difficult Hashcash puzzles to create a concept for a cryptocurrency, but once again fell short of the ideal by relying on trusted computing as a backend. In 2009, a decentralized currency was for the first time implemented in practice by Satoshi Nakamoto, combining established primitives for managing ownership through public key cryptography with a consensus algorithm for keeping track of who owns coins, known as "proof of work".

The mechanism behind proof of work was a breakthrough in the space because it simultaneously solved two problems. First, it provided a simple and moderately effective consensus algorithm, allowing nodes in the network to collectively agree on a set of canonical updates to the state of the Bitcoin ledger. Second, it provided a mechanism for allowing free entry into the consensus process, solving the political problem of deciding who gets to influence the consensus, while simultaneously preventing sybil attacks. It does this by substituting a formal barrier to participation, such as the requirement to be registered as a unique entity on a particular list, with an economic barrier - the weight of a single node in the consensus voting process is directly proportional to the computing power that the node brings. Since then, an alternative approach has been proposed called *proof of stake*, calculating the weight of a node as being proportional to its currency holdings and not computational resources; the discussion of the relative merits of the two approaches is beyond the scope of this paper but it should be noted that both approaches can be used to serve as the backbone of a cryptocurrency.

Bitcoin As A State Transition System



From a technical standpoint, the ledger of a cryptocurrency such as Bitcoin can be thought of as a state transition system, where there is a "state" consisting of the ownership status of all existing bitcoins and a "state transition function" that takes a state and a transaction and outputs a new state which is the result. In a standard banking system, for example, the state is a balance sheet, a transaction is a request to move $X from A to B, and the state transition function reduces the value in A's account by $X and increases the value in B's account by $X. If A's account has less than $X in the first place, the state transition function returns an error. Hence, one can formally define:

```
APPLY(S,TX) -> S' or ERROR
```

In the banking system defined above:

```
APPLY({ Alice: $50, Bob: $50 },"send $20 from Alice to Bob") = { Alice: $30, Bob: $70 }
```

But:

```
APPLY({ Alice: $50, Bob: $50 },"send $70 from Alice to Bob") = ERROR
```

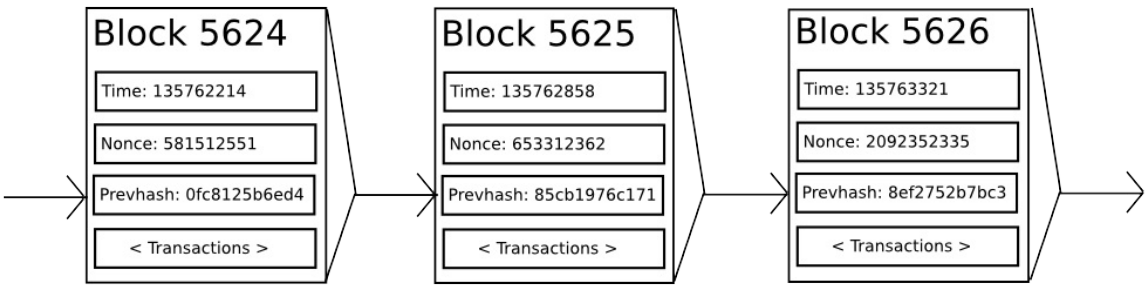
The "state" in Bitcoin is the collection of all coins (technically, "unspent transaction outputs" or UTXO) that have been minted and not yet spent, with each UTXO having a denomination and an owner (defined by a 20-byte address which is essentially a cryptographic public key[1]). A transaction contains one or more inputs, with each input containing a reference to an existing UTXO and a cryptographic signature produced by the private key associated with the owner's address, and one or more outputs, with each output containing a new UTXO to be added to the state.

The state transition function `APPLY(S,TX) -> S'` can be defined roughly as follows:

1. For each input in `TX`:
   - If the referenced UTXO is not in `S`, return an error.
   - If the provided signature does not match the owner of the UTXO, return an error.
2. If the sum of the denominations of all input UTXO is less than the sum of the denominations of all output UTXO, return an error.
3. Return `S` with all input UTXO removed and all output UTXO added.

The first half of the first step prevents transaction senders from spending coins that do not exist, the second half of the first step prevents transaction senders from spending other people's coins, and the second step enforces conservation of value. In order to use this for payment, the protocol is as follows. Suppose Alice wants to send 11.7 BTC to Bob. First, Alice will look for a set of available UTXO that she owns that totals up to at least 11.7 BTC. Realistically, Alice will not be able to get exactly 11.7 BTC; say that the smallest she can get is 6+4+2=12. She then creates a transaction with those three inputs and two outputs. The first output will be 11.7 BTC with Bob's address as its owner, and the second output will be the remaining 0.3 BTC "change", with the owner being Alice herself.

Mining



If we had access to a trustworthy centralized service, this system would be trivial to implement; it could simply be coded exactly as described, using a centralized server's hard drive to keep track of the state. However, with Bitcoin we are trying to build a decentralized currency system, so we will need to combine the state transaction system with a consensus system in order to ensure that everyone agrees on the order of transactions. Bitcoin's decentralized consensus process requires nodes in the network to continuously attempt to produce packages of transactions called "blocks". The network is intended to produce roughly one block every ten minutes, with each block containing a timestamp, a nonce, a reference to (ie. hash of) the previous block and a list of all of the transactions that have taken place since the previous block. Over time, this creates a persistent, ever-growing, "blockchain" that constantly updates to represent the latest state of the Bitcoin ledger.

The algorithm for checking if a block is valid, expressed in this paradigm, is as follows:

1. Check if the previous block referenced by the block exists and is valid.
2. Check that the timestamp of the block is greater than that of the previous block[2] and less than 2 hours into the future
3. Check that the proof of work on the block is valid.

4. Let `S[0]` be the state at the end of the previous block.
5. Suppose `TX` is the block's transaction list with `n` transactions. For all `i` in `0...n-1`, set `S[i+1] = APPLY(S[i],TX[i])` If any application returns an error, exit and return false.
6. Return true, and register `S[n]` as the state at the end of this block.

Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block. Additionally, note that the order in which the miner includes transactions into the block matters; if there are two transactions A and B in a block such that B spends a UTXO created by A, then the block will be valid if A comes before B but not otherwise.

The one validity condition present in the above list that is not found in other systems is the requirement for "proof of work". The precise condition is that the double-SHA256 hash of every block, treated as a 256-bit number, must be less than a dynamically adjusted target, which as of the time of this writing is approximately 2^{187}. The purpose of this is to make block creation computationally "hard", thereby preventing sybil attackers from remaking the entire blockchain in their favor. Because SHA256 is designed to be a completely unpredictable pseudorandom function, the only way to create a valid block is simply trial and error, repeatedly incrementing the nonce and seeing if the new hash matches.

At the current target of ~2^{187}, the network must make an average of ~2^{69} tries before a valid block is found; in general, the target is recalibrated by the network every 2016 blocks so that on average a new block is produced by some node in the network every ten minutes. In order to compensate miners for this computational work, the miner of every block is entitled to include a transaction giving themselves 25 BTC out of nowhere. Additionally, if any transaction has a higher total denomination in its inputs than in its outputs, the difference also goes to the miner as a "transaction fee". Incidentally, this is also the only mechanism by which BTC are issued; the genesis state contained no coins at all.

In order to better understand the purpose of mining, let us examine what happens in the event of a malicious attacker. Since Bitcoin's underlying cryptography is known to be secure, the attacker will target the one part of the Bitcoin system that is not protected by cryptography directly: the order of transactions. The attacker's strategy is simple:

1. Send 100 BTC to a merchant in exchange for some product (preferably a rapid-delivery digital good)
2. Wait for the delivery of the product
3. Produce another transaction sending the same 100 BTC to himself
4. Try to convince the network that his transaction to himself was the one that came first.

Once step (1) has taken place, after a few minutes some miner will include the transaction in a block, say block number 270000. After about one hour, five more blocks will have been added to the chain after that block, with each of those blocks indirectly pointing to the transaction and thus "confirming" it. At this point, the merchant will accept the payment as finalized and deliver the product; since we are assuming this is a digital good, delivery is instant. Now, the attacker creates another transaction sending the 100 BTC to himself. If the attacker simply releases it into the wild, the transaction will not be processed; miners will attempt to run `APPLY(S,TX)` and notice that `TX` consumes a UTXO which is no longer in the state. So instead, the attacker creates a "fork" of the blockchain, starting by mining another version of block 270000 pointing to the same block 269999 as a parent but with the new transaction in place of the old one. Because the block data is different, this requires redoing the proof of work. Furthermore, the attacker's new version of block 270000 has a different hash, so the original blocks 270001 to 270005 do not "point" to it; thus, the original chain and the attacker's new chain are completely separate. The rule is that in a fork the longest blockchain is taken to be the truth, and so legitimate miners will work on the 270005 chain while the attacker alone is working on the 270000 chain. In order for the attacker to make his blockchain the longest, he would need to have more computational power than the rest of the network combined in order to catch up (hence, "51% attack").

Merkle Trees

[SPV in bitcoin]

Left: it suffices to present only a small number of nodes in a Merkle tree to give a proof of the validity of a branch.

Right: any attempt to change any part of the Merkle tree will eventually lead to an inconsistency somewhere up the chain.

An important scalability feature of Bitcoin is that the block is stored in a multi-level data structure. The "hash" of a block is actually only the hash of the block header, a roughly 200-byte piece of data that contains the timestamp, nonce, previous block hash and the root hash of a data structure called the Merkle tree storing all transactions in the block. A Merkle tree is a type of binary tree, composed of a set of nodes with a large number of leaf nodes at the bottom of the tree containing the underlying data, a set of intermediate nodes where each node is the hash of its two children, and finally a single root node, also formed from the hash of its two children, representing the "top" of the tree. The purpose of the Merkle tree is to allow the data in a block to be delivered piecemeal: a node can download only the header of a block from one source, the small part of the tree relevant to them from another source, and still be assured that all of the data is correct. The reason why this works is that hashes propagate upward: if a malicious user attempts to swap in a fake transaction into the bottom of a Merkle tree, this change will cause a change in the node above, and then a change in the node above that, finally changing the root of the tree and therefore the hash of the block, causing the protocol to register it as a completely different block (almost certainly with an invalid proof of work).

The Merkle tree protocol is arguably essential to long-term sustainability. A "full node" in the Bitcoin network, one that stores and processes the entirety of every block, takes up about 15 GB of disk space in the Bitcoin network as of April 2014, and is growing by over a gigabyte per month. Currently, this is viable for some desktop computers and not phones, and later on in the future only businesses and hobbyists will be able to participate. A protocol known as "simplified payment verification" (SPV) allows for another class of nodes to exist, called "light nodes", which download the block headers, verify the proof of work on the block headers, and then download only the "branches" associated with transactions that are relevant to them. This allows light nodes to determine with a strong guarantee of security what the status of any Bitcoin transaction, and their current balance, is while downloading only a very small portion of the entire blockchain.

Alternative Blockchain Applications

The idea of taking the underlying blockchain idea and applying it to other concepts also has a long history. In 2005, Nick Szabo came out with the concept of "secure property titles with owner authority", a document describing how "new advances in replicated database technology" will allow for a blockchain-based system for storing a registry of who owns what land, creating an elaborate framework including concepts such as homesteading, adverse possession and Georgian land tax. However, there was unfortunately no effective replicated database system available at the time, and so the protocol was never implemented in practice. After 2009, however, once Bitcoin's decentralized consensus was developed a number of alternative applications rapidly began to emerge.

- **Namecoin** - created in 2010, Namecoin is best described as a decentralized name registration database. In decentralized protocols like Tor, Bitcoin and BitMessage, there needs to be some way of identifying accounts so that other people can interact with them, but in all existing solutions the only kind of identifier available is a pseudorandom hash like `1LW79wp5ZBqaHW1jL5TCiBCrhQYtHagUWy`. Ideally, one would like to be able to have an account with a name like "george". However, the problem is that if one person can create an account named "george" then someone else can use the same process to register "george" for themselves as well and impersonate them. The only solution is a first-to-file paradigm, where the first registerer succeeds and the second fails - a problem perfectly suited for the Bitcoin consensus protocol. Namecoin is the oldest, and most successful, implementation of a name registration system using such an idea.
- **Colored coins** - the purpose of colored coins is to serve as a protocol to allow people to create their own digital currencies - or, in the important trivial case of a currency with one unit, digital tokens, on the Bitcoin blockchain. In the colored coins protocol, one "issues" a new currency by publicly assigning a color to a specific Bitcoin UTXO, and the protocol recursively defines the color of other UTXO to be the same as the color of the inputs that the transaction creating them spent (some special rules apply in the case of mixed-color inputs). This allows users to maintain wallets containing only UTXO of a specific color and send them around much like regular bitcoins, backtracking through the blockchain to determine the color of any UTXO that they receive.
- **Metacoins** - the idea behind a metacoin is to have a protocol that lives on top of Bitcoin, using Bitcoin transactions to store metacoin transactions but having a different state transition function, `APPLY'`. Because the metacoin protocol

cannot prevent invalid metacoin transactions from appearing in the Bitcoin blockchain, a rule is added that if `APPLY'(S,TX)` returns an error, the protocol defaults to `APPLY'(S,TX) = S`. This provides an easy mechanism for creating an arbitrary cryptocurrency protocol, potentially with advanced features that cannot be implemented inside of Bitcoin itself, but with a very low development cost since the complexities of mining and networking are already handled by the Bitcoin protocol. Metacoins have been used to implement some classes of financial contracts, name registration and decentralized exchange.

Thus, in general, there are two approaches toward building a consensus protocol: building an independent network, and building a protocol on top of Bitcoin. The former approach, while reasonably successful in the case of applications like Namecoin, is difficult to implement; each individual implementation needs to bootstrap an independent blockchain, as well as building and testing all of the necessary state transition and networking code. Additionally, we predict that the set of applications for decentralized consensus technology will follow a power law distribution where the vast majority of applications would be too small to warrant their own blockchain, and we note that there exist large classes of decentralized applications, particularly decentralized autonomous organizations, that need to interact with each other.

The Bitcoin-based approach, on the other hand, has the flaw that it does not inherit the simplified payment verification features of Bitcoin. SPV works for Bitcoin because it can use blockchain depth as a proxy for validity; at some point, once the ancestors of a transaction go far enough back, it is safe to say that they were legitimately part of the state. Blockchain-based meta-protocols, on the other hand, cannot force the blockchain not to include transactions that are not valid within the context of their own protocols. Hence, a fully secure SPV meta-protocol implementation would need to backward scan all the way to the beginning of the Bitcoin blockchain to determine whether or not certain transactions are valid. Currently, all "light" implementations of Bitcoin-based meta-protocols rely on a trusted server to provide the data, arguably a highly suboptimal result especially when one of the primary purposes of a cryptocurrency is to eliminate the need for trust.

Scripting

Even without any extensions, the Bitcoin protocol actually does facilitate a weak version of a concept of "smart contracts". UTXO in Bitcoin can be owned not just by a public key, but also by a more complicated script expressed in a simple stack-based programming language. In this paradigm, a transaction spending that UTXO must provide data that satisfies the script. Indeed, even the basic public key ownership mechanism is implemented via a script: the script takes an elliptic curve signature as input, verifies it against the transaction and the address that owns the UTXO, and returns 1 if the verification is successful and 0 otherwise. Other, more complicated, scripts exist for various additional use cases. For example, one can construct a script that requires signatures from two out of a given three private keys to validate ("multisig"), a setup useful for corporate accounts, secure savings accounts and some merchant escrow situations. Scripts can also be used to pay bounties for solutions to computational problems, and one can even construct a script that says something like "this Bitcoin UTXO is yours if you can provide an SPV proof that you sent a Dogecoin transaction of this denomination to me", essentially allowing decentralized cross-cryptocurrency exchange.

However, the scripting language as implemented in Bitcoin has several important limitations:

- **Lack of Turing-completeness** - that is to say, while there is a large subset of computation that the Bitcoin scripting language supports, it does not nearly support everything. The main category that is missing is loops. This is done to avoid infinite loops during transaction verification; theoretically it is a surmountable obstacle for script programmers, since any loop can be simulated by simply repeating the underlying code many times with an if statement, but it does lead to scripts that are very space-inefficient. For example, implementing an alternative elliptic curve signature algorithm would likely require 256 repeated multiplication rounds all individually included in the code.
- **Value-blindness** - there is no way for a UTXO script to provide fine-grained control over the amount that can be withdrawn. For example, one powerful use case of an oracle contract would be a hedging contract, where A and B put in $1000 worth of BTC and after 30 days the script sends $1000 worth of BTC to A and the rest to B. This would require an oracle to determine the value of 1 BTC in USD, but even then it is a massive improvement in terms of trust and infrastructure requirement over the fully centralized solutions that are available now. However, because UTXO are all-or-nothing, the only way to achieve this is through the very inefficient hack of having many UTXO of varying denominations (eg. one UTXO of 2^k for every k up to 30) and having O pick which UTXO to send to A and which to B.
- **Lack of state** - UTXO can either be spent or unspent; there is no opportunity for multi-stage contracts or scripts which

keep any other internal state beyond that. This makes it hard to make multi-stage options contracts, decentralized exchange offers or two-stage cryptographic commitment protocols (necessary for secure computational bounties). It also means that UTXO can only be used to build simple, one-off contracts and not more complex "stateful" contracts such as decentralized organizations, and makes meta-protocols difficult to implement. Binary state combined with value-blindness also mean that another important application, withdrawal limits, is impossible.

- **Blockchain-blindness** - UTXO are blind to blockchain data such as the nonce, the timestamp and previous block hash. This severely limits applications in gambling, and several other categories, by depriving the scripting language of a potentially valuable source of randomness.

Thus, we see three approaches to building advanced applications on top of cryptocurrency: building a new blockchain, using scripting on top of Bitcoin, and building a meta-protocol on top of Bitcoin. Building a new blockchain allows for unlimited freedom in building a feature set, but at the cost of development time, bootstrapping effort and security. Using scripting is easy to implement and standardize, but is very limited in its capabilities, and meta-protocols, while easy, suffer from faults in scalability. With Ethereum, we intend to build an alternative framework that provides even larger gains in ease of development as well as even stronger light client properties, while at the same time allowing applications to share an economic environment and blockchain security.

Ethereum

The intent of Ethereum is to create an alternative protocol for building decentralized applications, providing a different set of tradeoffs that we believe will be very useful for a large class of decentralized applications, with particular emphasis on situations where rapid development time, security for small and rarely used applications, and the ability of different applications to very efficiently interact, are important. Ethereum does this by building what is essentially the ultimate abstract foundational layer: a blockchain with a built-in Turing-complete programming language, allowing anyone to write smart contracts and decentralized applications where they can create their own arbitrary rules for ownership, transaction formats and state transition functions. A bare-bones version of Namecoin can be written in two lines of code, and other protocols like currencies and reputation systems can be built in under twenty. Smart contracts, cryptographic "boxes" that contain value and only unlock it if certain conditions are met, can also be built on top of the platform, with vastly more power than that offered by Bitcoin scripting because of the added powers of Turing-completeness, value-awareness, blockchain-awareness and state.

Ethereum Accounts

In Ethereum, the state is made up of objects called "accounts", with each account having a 20-byte address and state transitions being direct transfers of value and information between accounts. An Ethereum account contains four fields:

- The **nonce**, a counter used to make sure each transaction can only be processed once
- The account's current **ether balance**
- The account's **contract code**, if present
- The account's **storage** (empty by default)

"Ether" is the main internal crypto-fuel of Ethereum, and is used to pay transaction fees. In general, there are two types of accounts: **externally owned accounts**, controlled by private keys, and **contract accounts**, controlled by their contract code. An externally owned account has no code, and one can send messages from an externally owned account by creating and signing a transaction; in a contract account, every time the contract account receives a message its code activates, allowing it to read and write to internal storage and send other messages or create contracts in turn.

Note that "contracts" in Ethereum should not be seen as something that should be "fulfilled" or "complied with"; rather, they are more like "autonomous agents" that live inside of the Ethereum execution environment, always executing a specific piece of code when "poked" by a message or transaction, and having direct control over their own ether balance and their own key/value store to keep track of persistent variables.

Messages and Transactions

The term "transaction" is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account. Transactions contain:

- The recipient of the message
- A signature identifying the sender
- The amount of ether to transfer from the sender to the recipient
- An optional data field
- A `STARTGAS` value, representing the maximum number of computational steps the transaction execution is allowed to take
- A `GASPRICE` value, representing the fee the sender pays per computational step

The first three are standard fields expected in any cryptocurrency. The data field has no function by default, but the virtual machine has an opcode using which a contract can access the data; as an example use case, if a contract is functioning as an on-blockchain domain registration service, then it may wish to interpret the data being passed to it as containing two "fields", the first field being a domain to register and the second field being the IP address to register it two. The contract would read these values from the message data and appropriately place them in storage.

The `STARTGAS` and `GASPRICE` fields are crucial for Ethereum's anti-denial of service model. In order to prevent accidental or hostile infinite loops or other computational wastage in code, each transaction is required to set a limit to how many computational steps of code execution it can use. The fundamental unit of computation is "gas"; usually, a computational step costs 1 gas, but some operations cost higher amounts of gas because they are more computationally expensive, or increase the amount of data that must be stored as part of the state. There is also a fee of 5 gas for every byte in the transaction data. The intent of the fee system is to require an attacker to pay proportionately for every resource that they consume, including computation, bandwidth and storage; hence, any transaction that leads to the network consuming a greater amount of any of these resources must have a gas fee roughly proportional to the increment.

Messages

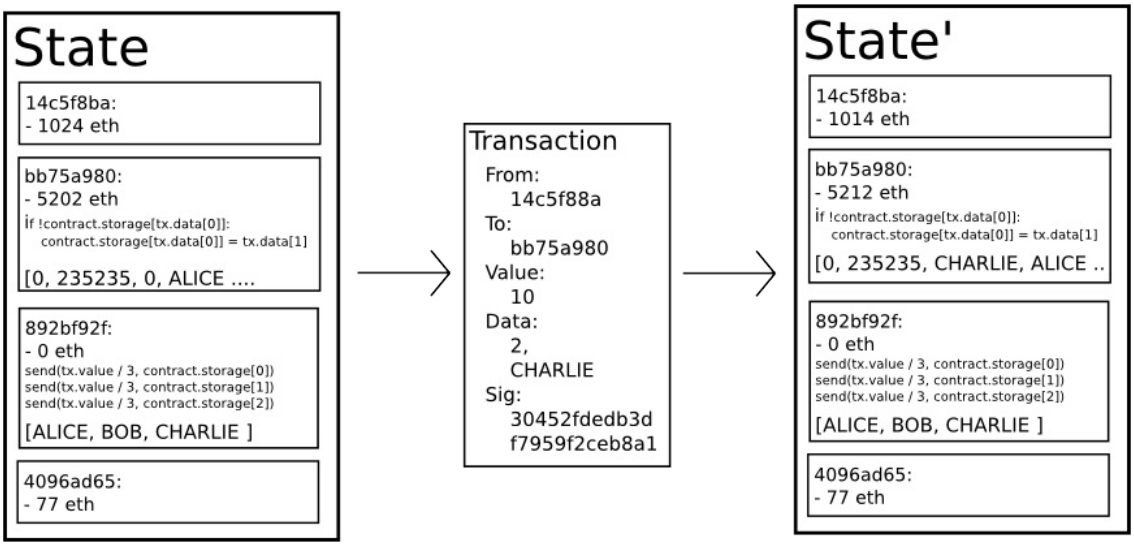
Contracts have the ability to send "messages" to other contracts. Messages are virtual objects that are never serialized and exist only in the Ethereum execution environment. A message contains:

- The sender of the message (implicit)
- The recipient of the message
- The amount of ether to transfer alongside the message
- An optional data field
- A `STARTGAS` value

Essentially, a message is like a transaction, except it is produced by a contract and not an external actor. A message is produced when a contract currently executing code executes the `CALL` opcode, which produces and executes a message. Like a transaction, a message leads to the recipient account running its code. Thus, contracts can have relationships with other contracts in exactly the same way that external actors can.

Note that the gas allowance assigned by a transaction or contract applies to the total gas consumed by that transaction and all sub-executions. For example, if an external actor A sends a transaction to B with 1000 gas, and B consumes 600 gas before sending a message to C, and the internal execution of C consumes 300 gas before returning, then B can spend another 100 gas before running out of gas.

Ethereum State Transition Function



The Ethereum state transition function, `APPLY(S,TX) -> S'` can be defined as follows:

1. Check if the transaction is well-formed (ie. has the right number of values), the signature is valid, and the nonce matches the nonce in the sender's account. If not, return an error.
2. Calculate the transaction fee as `STARTGAS * GASPRICE`, and determine the sending address from the signature. Subtract the fee from the sender's account balance and increment the sender's nonce. If there is not enough balance to spend, return an error.
3. Initialize `GAS = STARTGAS`, and take off a certain quantity of gas per byte to pay for the bytes in the transaction.
4. Transfer the transaction value from the sender's account to the receiving account. If the receiving account does not yet exist, create it. If the receiving account is a contract, run the contract's code either to completion or until the execution runs out of gas.
5. If the value transfer failed because the sender did not have enough money, or the code execution ran out of gas, revert all state changes except the payment of the fees, and add the fees to the miner's account.
6. Otherwise, refund the fees for all remaining gas to the sender, and send the fees paid for gas consumed to the miner.

For example, suppose that the contract's code is:

```
if !self.storage[calldataload(0)]:
    self.storage[calldataload(0)] = calldataload(32)
```

Note that in reality the contract code is written in the low-level EVM code; this example is written in Serpent, one of our high-level languages, for clarity, and can be compiled down to EVM code. Suppose that the contract's storage starts off empty, and a transaction is sent with 10 ether value, 2000 gas, 0.001 ether gasprice, and 64 bytes of data, with bytes 0-31 representing the number `2` and bytes 32-63 representing the string `CHARLIE`. The process for the state transition function in this case is as follows:

1. Check that the transaction is valid and well formed.
2. Check that the transaction sender has at least 2000 * 0.001 = 2 ether. If it is, then subtract 2 ether from the sender's account.
3. Initialize gas = 2000; assuming the transaction is 170 bytes long and the byte-fee is 5, subtract 850 so that there is 1150 gas left.
4. Subtract 10 more ether from the sender's account, and add it to the contract's account.
5. Run the code. In this case, this is simple: it checks if the contract's storage at index `2` is used, notices that it is not, and so it sets the storage at index `2` to the value `CHARLIE`. Suppose this takes 187 gas, so the remaining amount of

gas is 1150 - 187 = 963
6. Add 963 * 0.001 = 0.963 ether back to the sender's account, and return the resulting state.

If there was no contract at the receiving end of the transaction, then the total transaction fee would simply be equal to the provided `GASPRICE` multiplied by the length of the transaction in bytes, and the data sent alongside the transaction would be irrelevant.

Note that messages work equivalently to transactions in terms of reverts: if a message execution runs out of gas, then that message's execution, and all other executions triggered by that execution, revert, but parent executions do not need to revert. This means that it is "safe" for a contract to call another contract, as if A calls B with G gas then A's execution is guaranteed to lose at most G gas. Finally, note that there is an opcode, `CREATE`, that creates a contract; its execution mechanics are generally similar to `CALL`, with the exception that the output of the execution determines the code of a newly created contract.

Code Execution

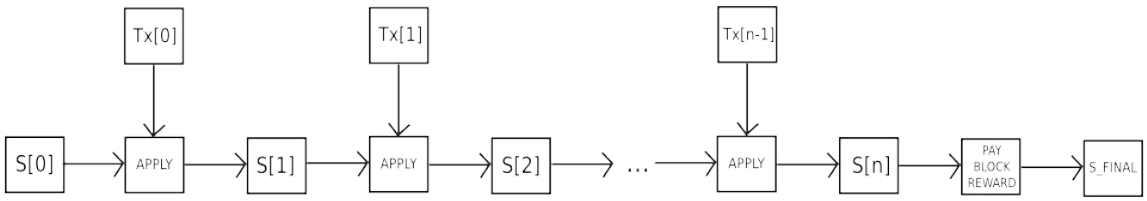
The code in Ethereum contracts is written in a low-level, stack-based bytecode language, referred to as "Ethereum virtual machine code" or "EVM code". The code consists of a series of bytes, where each byte represents an operation. In general, code execution is an infinite loop that consists of repeatedly carrying out the operation at the current program counter (which begins at zero) and then incrementing the program counter by one, until the end of the code is reached or an error or `STOP` or `RETURN` instruction is detected. The operations have access to three types of space in which to store data:

- The **stack**, a last-in-first-out container to which values can be pushed and popped
- **Memory**, an infinitely expandable byte array
- The contract's long-term **storage**, a key/value store. Unlike stack and memory, which reset after computation ends, storage persists for the long term.

The code can also access the value, sender and data of the incoming message, as well as block header data, and the code can also return a byte array of data as an output.

The formal execution model of EVM code is surprisingly simple. While the Ethereum virtual machine is running, its full computational state can be defined by the tuple `(block_state, transaction, message, code, memory, stack, pc, gas)`, where `block_state` is the global state containing all accounts and includes balances and storage. At the start of every round of execution, the current instruction is found by taking the `pc` th byte of `code` (or 0 if `pc >= len(code)`), and each instruction has its own definition in terms of how it affects the tuple. For example, `ADD` pops two items off the stack and pushes their sum, reduces `gas` by 1 and increments `pc` by 1, and `SSTORE` pushes the top two items off the stack and inserts the second item into the contract's storage at the index specified by the first item. Although there are many ways to optimize Ethereum virtual machine execution via just-in-time compilation, a basic implementation of Ethereum can be done in a few hundred lines of code.

Blockchain and Mining



The Ethereum blockchain is in many ways similar to the Bitcoin blockchain, although it does have some differences. The

main difference between Ethereum and Bitcoin with regard to the blockchain architecture is that, unlike Bitcoin, Ethereum blocks contain a copy of both the transaction list and the most recent state. Aside from that, two other values, the block number and the difficulty, are also stored in the block. The basic block validation algorithm in Ethereum is as follows:

1. Check if the previous block referenced exists and is valid.
2. Check that the timestamp of the block is greater than that of the referenced previous block and less than 15 minutes into the future
3. Check that the block number, difficulty, transaction root, uncle root and gas limit (various low-level Ethereum-specific concepts) are valid.
4. Check that the proof of work on the block is valid.
5. Let `S[0]` be the state at the end of the previous block.
6. Let `TX` be the block's transaction list, with `n` transactions. For all in in `0...n-1`, set `S[i+1] = APPLY(S[i],TX[i])`. If any applications returns an error, or if the total gas consumed in the block up until this point exceeds the `GASLIMIT`, return an error.
7. Let `S_FINAL` be `S[n]`, but adding the block reward paid to the miner.
8. Check if the Merkle tree root of the state `S_FINAL` is equal to the final state root provided in the block header. If it is, the block is valid; otherwise, it is not valid.

The approach may seem highly inefficient at first glance, because it needs to store the entire state with each block, but in reality efficiency should be comparable to that of Bitcoin. The reason is that the state is stored in the tree structure, and after every block only a small part of the tree needs to be changed. Thus, in general, between two adjacent blocks the vast majority of the tree should be the same, and therefore the data can be stored once and referenced twice using pointers (ie. hashes of subtrees). A special kind of tree known as a "Patricia tree" is used to accomplish this, including a modification to the Merkle tree concept that allows for nodes to be inserted and deleted, and not just changed, efficiently. Additionally, because all of the state information is part of the last block, there is no need to store the entire blockchain history - a strategy which, if it could be applied to Bitcoin, can be calculated to provide 5-20x savings in space.

A commonly asked question is "where" contract code is executed, in terms of physical hardware. This has a simple answer: the process of executing contract code is part of the definition of the state transition function, which is part of the block validation algorithm, so if a transaction is added into block `B` the code execution spawned by that transaction will be executed by all nodes, now and in the future, that download and validate block `B`.

Applications

In general, there are three types of applications on top of Ethereum. The first category is financial applications, providing users with more powerful ways of managing and entering into contracts using their money. This includes sub-currencies, financial derivatives, hedging contracts, savings wallets, wills, and ultimately even some classes of full-scale employment contracts. The second category is semi-financial applications, where money is involved but there is also a heavy non-monetary side to what is being done; a perfect example is self-enforcing bounties for solutions to computational problems. Finally, there are applications such as online voting and decentralized governance that are not financial at all.

Token Systems

On-blockchain token systems have many applications ranging from sub-currencies representing assets such as USD or gold to company stocks, individual tokens representing smart property, secure unforgeable coupons, and even token systems with no ties to conventional value at all, used as point systems for incentivization. Token systems are surprisingly easy to implement in Ethereum. The key point to understand is that all a currency, or token system, fundamentally is is a database with one operation: subtract X units from A and give X units to B, with the proviso that (i) A had at least X units before the transaction and (2) the transaction is approved by A. All that it takes to implement a token system is to implement this logic into a contract.

The basic code for implementing a token system in Serpent looks as follows:

```
def send(to, value):
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

This is essentially a literal implementation of the "banking system" state transition function described further above in this document. A few extra lines of code need to be added to provide for the initial step of distributing the currency units in the first place and a few other edge cases, and ideally a function would be added to let other contracts query for the balance of an address. But that's all there is to it. Theoretically, Ethereum-based token systems acting as sub-currencies can potentially include another important feature that on-chain Bitcoin-based meta-currencies lack: the ability to pay transaction fees directly in that currency. The way this would be implemented is that the contract would maintain an ether balance with which it would refund ether used to pay fees to the sender, and it would refill this balance by collecting the internal currency units that it takes in fees and reselling them in a constant running auction. Users would thus need to "activate" their accounts with ether, but once the ether is there it would be reusable because the contract would refund it each time.

Financial derivatives and Stable-Value Currencies

Financial derivatives are the most common application of a "smart contract", and one of the simplest to implement in code. The main challenge in implementing financial contracts is that the majority of them require reference to an external price ticker; for example, a very desirable application is a smart contract that hedges against the volatility of ether (or another cryptocurrency) with respect to the US dollar, but doing this requires the contract to know what the value of ETH/USD is. The simplest way to do this is through a "data feed" contract maintained by a specific party (eg. NASDAQ) designed so that that party has the ability to update the contract as needed, and providing an interface that allows other contracts to send a message to that contract and get back a response that provides the price.

Given that critical ingredient, the hedging contract would look as follows:

1. Wait for party A to input 1000 ether.
2. Wait for party B to input 1000 ether.
3. Record the USD value of 1000 ether, calculated by querying the data feed contract, in storage, say this is $x.
4. After 30 days, allow A or B to "reactivate" the contract in order to send $x worth of ether (calculated by querying the data feed contract again to get the new price) to A and the rest to B.

Such a contract would have significant potential in crypto-commerce. One of the main problems cited about cryptocurrency is the fact that it's volatile; although many users and merchants may want the security and convenience of dealing with cryptographic assets, they many not wish to face that prospect of losing 23% of the value of their funds in a single day. Up until now, the most commonly proposed solution has been issuer-backed assets; the idea is that an issuer creates a sub-currency in which they have the right to issue and revoke units, and provide one unit of the currency to anyone who provides them (offline) with one unit of a specified underlying asset (eg. gold, USD). The issuer then promises to provide one unit of the underlying asset to anyone who sends back one unit of the crypto-asset. This mechanism allows any non-cryptographic asset to be "uplifted" into a cryptographic asset, provided that the issuer can be trusted.

In practice, however, issuers are not always trustworthy, and in some cases the banking infrastructure is too weak, or too hostile, for such services to exist. Financial derivatives provide an alternative. Here, instead of a single issuer providing the funds to back up an asset, a decentralized market of speculators, betting that the price of a cryptographic reference asset (eg. ETH) will go up, plays that role. Unike issuers, speculators have no option to default on their side of the bargain because the hedging contract holds their funds in escrow. Note that this approach is not fully decentralized, because a trusted source is still needed to provide the price ticker, although arguably even still this is a massive improvement in terms of reducing infrastructure requirements (unlike being an issuer, issuing a price feed requires no licenses and can likely be categorized as free speech) and reducing the potential for fraud.

Identity and Reputation Systems

The earliest alternative cryptocurrency of all, Namecoin, attempted to use a Bitcoin-like blockchain to provide a name

registration system, where users can register their names in a public database alongside other data. The major cited use case is for a DNS system, mapping domain names like "bitcoin.org" (or, in Namecoin's case, "bitcoin.bit") to an IP address. Other use cases include email authentication and potentially more advanced reputation systems. Here is the basic contract to provide a Namecoin-like name registration system on Ethereum:

```
def register(name, value):
    if !self.storage[name]:
        self.storage[name] = value
```

The contract is very simple; all it is is a database inside the Ethereum network that can be added to, but not modified or removed from. Anyone can register a name with some value, and that registration then sticks forever. A more sophisticated name registration contract will also have a "function clause" allowing other contracts to query it, as well as a mechanism for the "owner" (ie. the first registerer) of a name to change the data or transfer ownership. One can even add reputation and web-of-trust functionality on top.

Decentralized File Storage

Over the past few years, there have emerged a number of popular online file storage startups, the most prominent being Dropbox, seeking to allow users to upload a backup of their hard drive and have the service store the backup and allow the user to access it in exchange for a monthly fee. However, at this point the file storage market is at times relatively inefficient; a cursory look at various existing solutions shows that, particularly at the "uncanny valley" 20-200 GB level at which neither free quotas nor enterprise-level discounts kick in, monthly prices for mainstream file storage costs are such that you are paying for more than the cost of the entire hard drive in a single month. Ethereum contracts can allow for the development of a decentralized file storage ecosystem, where individual users can earn small quantities of money by renting out their own hard drives and unused space can be used to further drive down the costs of file storage.

The key underpinning piece of such a device would be what we have termed the "decentralized Dropbox contract". This contract works as follows. First, one splits the desired data up into blocks, encrypting each block for privacy, and builds a Merkle tree out of it. One then makes a contract with the rule that, every N blocks, the contract would pick a random index in the Merkle tree (using the previous block hash, accessible from contract code, as a source of randomness), and give X ether to the first entity to supply a transaction with a simplified payment verification-like proof of ownership of the block at that particular index in the tree. When a user wants to re-download their file, they can use a micropayment channel protocol (eg. pay 1 szabo per 32 kilobytes) to recover the file; the most fee-efficient approach is for the payer not to publish the transaction until the end, instead replacing the transaction with a slightly more lucrative one with the same nonce after every 32 kilobytes.

An important feature of the protocol is that, although it may seem like one is trusting many random nodes not to decide to forget the file, one can reduce that risk down to near-zero by splitting the file into many pieces via secret sharing, and watching the contracts to see each piece is still in some node's possession. If a contract is still paying out money, that provides a cryptographic proof that someone out there is still storing the file.

Decentralized Autonomous Organizations

The general concept of a "decentralized autonomous organization" is that of a virtual entity that has a certain set of members or shareholders which, perhaps with a 67% majority, have the right to spend the entity's funds and modify its code. The members would collectively decide on how the organization should allocate its funds. Methods for allocating a DAO's funds could range from bounties, salaries to even more exotic mechanisms such as an internal currency to reward work. This essentially replicates the legal trappings of a traditional company or nonprofit but using only cryptographic blockchain technology for enforcement. So far much of the talk around DAOs has been around the "capitalist" model of a "decentralized autonomous corporation" (DAC) with dividend-receiving shareholders and tradable shares; an alternative, perhaps described as a "decentralized autonomous community", would have all members have an equal share in the decision making and require 67% of existing members to agree to add or remove a member. The requirement that one person can only have one membership would then need to be enforced collectively by the group.

A general outline for how to code a DAO is as follows. The simplest design is simply a piece of self-modifying code that changes if two thirds of members agree on a change. Although code is theoretically immutable, one can easily get around this and have de-facto mutability by having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage. In a simple implementation of such a DAO contract, there would be three transaction types, distinquished by the data provided in the transaction:

- `[0,i,K,V]` to register a proposal with index `i` to change the address at storage index `K` to value `V`
- `[0,i]` to register a vote in favor of proposal `i`
- `[2,i]` to finalize proposal `i` if enough votes have been made

The contract would then have clauses for each of these. It would maintain a record of all open storage changes, along with a list of who voted for them. It would also have a list of all members. When any storage change gets to two thirds of members voting for it, a finalizing transaction could execute the change. A more sophisticated skeleton would also have built-in voting ability for features like sending a transaction, adding members and removing members, and may even provide for Liquid Democracy-style vote delegation (ie. anyone can assign someone to vote for them, and assignment is transitive so if A assigns B and B assigns C then C determines A's vote). This design would allow the DAO to grow organically as a decentralized community, allowing people to eventually delegate the task of filtering out who is a member to specialists, although unlike in the "current system" specialists can easily pop in and out of existence over time as individual community members change their alignments.

An alternative model is for a decentralized corporation, where any account can have zero or more shares, and two thirds of the shares are required to make a decision. A complete skeleton would involve asset management functionality, the ability to make an offer to buy or sell shares, and the ability to accept offers (preferably with an order-matching mechanism inside the contract). Delegation would also exist Liquid Democracy-style, generalizing the concept of a "board of directors".

Further Applications

1. Savings wallets. Suppose that Alice wants to keep her funds safe, but is worried that she will lose or someone will hack her private key. She puts ether into a contract with Bob, a bank, as follows:

- Alice alone can withdraw a maximum of 1% of the funds per day.
- Bob alone can withdraw a maximum of 1% of the funds per day, but Alice has the ability to make a transaction with her key shutting off this ability.
- Alice and Bob together can withdraw anything.

Normally, 1% per day is enough for Alice, and if Alice wants to withdraw more she can contact Bob for help. If Alice's key gets hacked, she runs to Bob to move the funds to a new contract. If she loses her key, Bob will get the funds out eventually. If Bob turns out to be malicious, then she can turn off his ability to withdraw.

2. Crop insurance. One can easily make a financial derivatives contract but using a data feed of the weather instead of any price index. If a farmer in Iowa purchases a derivative that pays out inversely based on the precipitation in Iowa, then if there is a drought, the farmer will automatically receive money and if there is enough rain the farmer will be happy because their crops would do well. This can be expanded to natural disaster insurance generally.

3. A decentralized data feed. For financial contracts for difference, it may actually be possible to decentralize the data feed via a protocol called "SchellingCoin". SchellingCoin basically works as follows: N parties all put into the system the value of a given datum (eg. the ETH/USD price), the values are sorted, and everyone between the 25th and 75th percentile gets one token as a reward. Everyone has the incentive to provide the answer that everyone else will provide, and the only value that a large number of players can realistically agree on is the obvious default: the truth. This creates a decentralized protocol that can theoretically provide any number of values, including the ETH/USD price, the temperature in Berlin or even the result of a particular hard computation.

4. Smart multisignature escrow. Bitcoin allows multisignature transaction contracts where, for example, three out of a given five keys can spend the funds. Ethereum allows for more granularity; for example, four out of five can spend

everything, three out of five can spend up to 10% per day, and two out of five can spend up to 0.5% per day. Additionally, Ethereum multisig is asynchronous - two parties can register their signatures on the blockchain at different times and the last signature will automatically send the transaction.

5. Cloud computing. The EVM technology can also be used to create a verifiable computing environment, allowing users to ask others to carry out computations and then optionally ask for proofs that computations at certain randomly selected checkpoints were done correctly. This allows for the creation of a cloud computing market where any user can participate with their desktop, laptop or specialized server, and spot-checking together with security deposits can be used to ensure that the system is trustworthy (ie. nodes cannot profitably cheat). Although such a system may not be suitable for all tasks; tasks that require a high level of inter-process communication, for example, cannot easily be done on a large cloud of nodes. Other tasks, however, are much easier to parallelize; projects like SETI@home, folding@home and genetic algorithms can easily be implemented on top of such a platform.

6. Peer-to-peer gambling. Any number of peer-to-peer gambling protocols, such as Frank Stajano and Richard Clayton's Cyberdice, can be implemented on the Ethereum blockchain. The simplest gambling protocol is actually simply a contract for difference on the next block hash, and more advanced protocols can be built up from there, creating gambling services with near-zero fees that have no ability to cheat.

7. Prediction markets. Provided an oracle or SchellingCoin, prediction markets are also easy to implement, and prediction markets together with SchellingCoin may prove to be the first mainstream application of futarchy as a governance protocol for decentralized organizations.

8. On-chain decentralized marketplaces, using the identity and reputation system as a base.

Miscellanea And Concerns

Modified GHOST Implementation

The "Greedy Heaviest Observed Subtree" (GHOST) protocol is an innovation first introduced by Yonatan Sompolinsky and Aviv Zohar in December 2013. The motivation behind GHOST is that blockchains with fast confirmation times currently suffer from reduced security due to a high stale rate - because blocks take a certain time to propagate through the network, if miner A mines a block and then miner B happens to mine another block before miner A's block propagates to B, miner B's block will end up wasted and will not contribute to network security. Furthermore, there is a centralization issue: if miner A is a mining pool with 30% hashpower and B has 10% hashpower, A will have a risk of producing a stale block 70% of the time (since the other 30% of the time A produced the last block and so will get mining data immediately) whereas B will have a risk of producing a stale block 90% of the time. Thus, if the block interval is short enough for the stale rate to be high, A will be substantially more efficient simply by virtue of its size. With these two effects combined, blockchains which produce blocks quickly are very likely to lead to one mining pool having a large enough percentage of the network hashpower to have de facto control over the mining process.

As described by Sompolinsky and Zohar, GHOST solves the first issue of network security loss by including stale blocks in the calculation of which chain is the "longest"; that is to say, not just the parent and further ancestors of a block, but also the stale descendants of the block's ancestor (in Ethereum jargon, "uncles") are added to the calculation of which block has the largest total proof of work backing it. To solve the second issue of centralization bias, we go beyond the protocol described by Sompolinsky and Zohar, and also provide block rewards to stales: a stale block receives 87.5% of its base reward, and the nephew that includes the stale block receives the remaining 12.5%. Transaction fees, however, are not awarded to uncles.

Ethereum implements a simplified version of GHOST which only goes down seven levels. Specifically, it is defined as follows:

- A block must specify a parent, and it must specify 0 or more uncles
- An uncle included in block B must have the following properties:

    - It must be a direct child of the kth generation ancestor of B, where 2 <= k <= 7.
    - It cannot be an ancestor of B
    - An uncle must be a valid block header, but does not need to be a previously verified or even valid block
    - An uncle must be different from all uncles included in previous blocks and all other uncles included in the same block (non-double-inclusion)
- For every uncle U in block B, the miner of B gets an additional 3.125% added to its coinbase reward and the miner of U gets 93.75% of a standard coinbase reward.

This limited version of GHOST, with uncles includable only up to 7 generations, was used for two reasons. First, unlimited GHOST would include too many complications into the calculation of which uncles for a given block are valid. Second, unlimited GHOST with compensation as used in Ethereum removes the incentive for a miner to mine on the main chain and not the chain of a public attacker.

Fees

Because every transaction published into the blockchain imposes on the network the cost of needing to download and verify it, there is a need for some regulatory mechanism, typically involving transaction fees, to prevent abuse. The default approach, used in Bitcoin, is to have purely voluntary fees, relying on miners to act as the gatekeepers and set dynamic minimums. This approach has been received very favorably in the Bitcoin community particularly because it is "market-based", allowing supply and demand between miners and transaction senders determine the price. The problem with this line of reasoning is, however, that transaction processing is not a market; although it is intuitively attractive to construe transaction processing as a service that the miner is offering to the sender, in reality every transaction that a miner includes will need to be processed by every node in the network, so the vast majority of the cost of transaction processing is borne by third parties and not the miner that is making the decision of whether or not to include it. Hence, tragedy-of-the-commons problems are very likely to occur.

However, as it turns out this flaw in the market-based mechanism, when given a particular inaccurate simplifying assumption, magically cancels itself out. The argument is as follows. Suppose that:

1. A transaction leads to `k` operations, offering the reward `kR` to any miner that includes it where `R` is set by the sender and `k` and `R` are (roughly) visible to the miner beforehand.
2. An operation has a processing cost of `C` to any node (ie. all nodes have equal efficiency)
3. There are `N` mining nodes, each with exactly equal processing power (ie. `1/N` of total)
4. No non-mining full nodes exist.

A miner would be willing to process a transaction if the expected reward is greater than the cost. Thus, the expected reward is `kR/N` since the miner has a `1/N` chance of processing the next block, and the processing cost for the miner is simply `kC`. Hence, miners will include transactions where `kR/N > kC`, or `R > NC`. Note that `R` is the per-operation fee provided by the sender, and is thus a lower bound on the benefit that the sender derives from the transaction, and `NC` is the cost to the entire network together of processing an operation. Hence, miners have the incentive to include only those transactions for which the total utilitarian benefit exceeds the cost.

However, there are several important deviations from those assumptions in reality:

1. The miner does pay a higher cost to process the transaction than the other verifying nodes, since the extra verification time delays block propagation and thus increases the chance the block will become a stale.
2. There do exist nonmining full nodes.
3. The mining power distribution may end up radically inegalitarian in practice.
4. Speculators, political enemies and crazies whose utility function includes causing harm to the network do exist, and they can cleverly set up contracts where their cost is much lower than the cost paid by other verifying nodes.

(1) provides a tendency for the miner to include fewer transactions, and (2) increases `NC`; hence, these two effects at least partially cancel each other out. (3) and (4) are the major issue; to solve them we simply institute a floating cap: no block can have more operations than `BLK_LIMIT_FACTOR` times the long-term exponential moving average. Specifically:

```
blk.oplimit = floor((blk.parent.oplimit * (EMAFACTOR - 1) + floor(parent.opcount * BLK_LIMIT_FACTOR)) / EMA_FACTOR)
```

`BLK_LIMIT_FACTOR` and `EMA_FACTOR` are constants that will be set to 65536 and 1.5 for the time being, but will likely be changed after further analysis.

There is another factor disincentivizing large block sizes in Bitcoin: blocks that are large will take longer to propagate, and thus have a higher probability of becoming stales. In Ethereum, highly gas-consuming blocks can also take longer to propagate both because they are physically larger and because they take longer to process the transaction state transitions to validate. This delay disincentive is a significant consideration in Bitcoin, but less so in Ethereum because of the GHOST protocol; hence, relying on regulated block limits provides a more stable baseline.

Computation And Turing-Completeness

An important note is that the Ethereum virtual machine is Turing-complete; this means that EVM code can encode any computation that can be conceivably carried out, including infinite loops. EVM code allows looping in two ways. First, there is a `JUMP` instruction that allows the program to jump back to a previous spot in the code, and a `JUMPI` instruction to do conditional jumping, allowing for statements like `while x < 27: x = x * 2`. Second, contracts can call other contracts, potentially allowing for looping through recursion. This naturally leads to a problem: can malicious users essentially shut miners and full nodes down by forcing them to enter into an infinite loop? The issue arises because of a problem in computer science known as the halting problem: there is no way to tell, in the general case, whether or not a given program will ever halt.

As described in the state transition section, our solution works by requiring a transaction to set a maximum number of computational steps that it is allowed to take, and if execution takes longer computation is reverted but fees are still paid. Messages work in the same way. To show the motivation behind our solution, consider the following examples:

- An attacker creates a contract which runs an infinite loop, and then sends a transaction activating that loop to the miner. The miner will process the transaction, running the infinite loop, and wait for it to run out of gas. Even though the execution runs out of gas and stops halfway through, the transaction is still valid and the miner still claims the fee from the attacker for each computational step.
- An attacker creates a very long infinite loop with the intent of forcing the miner to keep computing for such a long time that by the time computation finishes a few more blocks will have come out and it will not be possible for the miner to include the transaction to claim the fee. However, the attacker will be required to submit a value for `STARTGAS` limiting the number of computational steps that execution can take, so the miner will know ahead of time that the computation will take an excessively large number of steps.
- An attacker sees a contract with code of some form like `send(A,contract.storage[A]); contract.storage[A] = 0`, and sends a transaction with just enough gas to run the first step but not the second (ie. making a withdrawal but not letting the balance go down). The contract author does not need to worry about protecting against such attacks, because if execution stops halfway through the changes get reverted.
- A financial contract works by taking the median of nine proprietary data feeds in order to minimize risk. An attacker takes over one of the data feeds, which is designed to be modifiable via the variable-address-call mechanism described in the section on DAOs, and converts it to run an infinite loop, thereby attempting to force any attempts to claim funds from the financial contract to run out of gas. However, the financial contract can set a gas limit on the message to prevent this problem.

The alternative to Turing-completeness is Turing-incompleteness, where `JUMP` and `JUMPI` do not exist and only one copy of each contract is allowed to exist in the call stack at any given time. With this system, the fee system described and the uncertainties around the effectiveness of our solution might not be necessary, as the cost of executing a contract would be bounded above by its size. Additionally, Turing-incompleteness is not even that big a limitation; out of all the contract examples we have conceived internally, so far only one required a loop, and even that loop could be removed by making 26 repetitions of a one-line piece of code. Given the serious implications of Turing-completeness, and the limited benefit, why not simply have a Turing-incomplete language? In reality, however, Turing-incompleteness is far from a neat solution to the

problem. To see why, consider the following contracts:

```
C0: call(C1); call(C1);
C1: call(C2); call(C2);
C2: call(C3); call(C3);
...
C49: call(C50); call(C50);
C50: (run one step of a program and record the change in storage)
```

Now, send a transaction to A. Thus, in 51 transactions, we have a contract that takes up 2^{50} computational steps. Miners could try to detect such logic bombs ahead of time by maintaining a value alongside each contract specifying the maximum number of computational steps that it can take, and calculating this for contracts calling other contracts recursively, but that would require miners to forbid contracts that create other contracts (since the creation and execution of all 26 contracts above could easily be rolled into a single contract). Another problematic point is that the address field of a message is a variable, so in general it may not even be possible to tell which other contracts a given contract will call ahead of time. Hence, all in all, we have a surprising conclusion: Turing-completeness is surprisingly easy to manage, and the lack of Turing-completeness is equally surprisingly difficult to manage unless the exact same controls are in place - but in that case why not just let the protocol be Turing-complete?

Currency And Issuance

The Ethereum network includes its own built-in currency, ether, which serves the dual purpose of providing a primary liquidity layer to allow for efficient exchange between various types of digital assets and, more importantly, of providing a mechanism for paying transaction fees. For convenience and to avoid future argument (see the current mBTC/uBTC/satoshi debate in Bitcoin), the denominations will be pre-labelled:

- 1: wei
- 10^{12}: szabo
- 10^{15}: finney
- 10^{18}: ether

This should be taken as an expanded version of the concept of "dollars" and "cents" or "BTC" and "satoshi". In the near future, we expect "ether" to be used for ordinary transactions, "finney" for microtransactions and "szabo" and "wei" for technical discussions around fees and protocol implementation; the remaining denominations may become useful later and should not be included in clients at this point.

The issuance model will be as follows:

- Ether will be released in a currency sale at the price of 1000-2000 ether per BTC, a mechanism intended to fund the Ethereum organization and pay for development that has been used with success by other platforms such as Mastercoin and NXT. Earlier buyers will benefit from larger discounts. The BTC received from the sale will be used entirely to pay salaries and bounties to developers and invested into various for-profit and non-profit projects in the Ethereum and cryptocurrency ecosystem.
- 0.099x the total amount sold (60102216 ETH) will be allocated to the organization to compensate early contributors and pay ETH-denominated expenses before the genesis block.
- 0.099x the total amount sold will be maintained as a long-term reserve.
- 0.26x the total amount sold will be allocated to miners per year forever after that point.

Group	At launch	After 1 year	After 5 years
Currency units	1.198X	1.458X	2.498X
Purchasers	83.5%	68.6%	40.0%
Reserve spent pre-sale	8.26%	6.79%	3.96%

Reserve used post-sale	8.26%	6.79%	3.96%
Miners	0%	17.8%	52.0%

Long-Term Supply Growth Rate (percent)

[SPV in bitcoin]

Despite the linear currency issuance, just like with Bitcoin over time the supply growth rate nevertheless tends to zero

The two main choices in the above model are (1) the existence and size of an endowment pool, and (2) the existence of a permanently growing linear supply, as opposed to a capped supply as in Bitcoin. The justification of the endowment pool is as follows. If the endowment pool did not exist, and the linear issuance reduced to 0.217x to provide the same inflation rate, then the total quantity of ether would be 16.5% less and so each unit would be 19.8% more valuable. Hence, in the equilibrium 19.8% more ether would be purchased in the sale, so each unit would once again be exactly as valuable as before. The organization would also then have 1.198x as much BTC, which can be considered to be split into two slices: the original BTC, and the additional 0.198x. Hence, this situation is *exactly equivalent* to the endowment, but with one important difference: the organization holds purely BTC, and so is not incentivized to support the value of the ether unit.

The permanent linear supply growth model reduces the risk of what some see as excessive wealth concentration in Bitcoin, and gives individuals living in present and future eras a fair chance to acquire currency units, while at the same time retaining a strong incentive to obtain and hold ether because the "supply growth rate" as a percentage still tends to zero over time. We also theorize that because coins are always lost over time due to carelessness, death, etc, and coin loss can be modeled as a percentage of the total supply per year, that the total currency supply in circulation will in fact eventually stabilize at a value equal to the annual issuance divided by the loss rate (eg. at a loss rate of 1%, once the supply reaches 26X then 0.26X will be mined and 0.26X lost every year, creating an equilibrium).

Note that in the future, it is likely that Ethereum will switch to a proof-of-stake model for security, reducing the issuance requirement to somewhere between zero and 0.05X per year. In the event that the Ethereum organization loses funding or for any other reason disappears, we leave open a "social contract": anyone has the right to create a future candidate version of Ethereum, with the only condition being that the quantity of ether must be at most equal to `60102216 * (1.198 + 0.26 * n)` where `n` is the number of years after the genesis block. Creators are free to crowd-sell or otherwise assign some or all of the difference between the PoS-driven supply expansion and the maximum allowable supply expansion to pay for development. Candidate upgrades that do not comply with the social contract may justifiably be forked into compliant versions.

Mining Centralization

The Bitcoin mining algorithm works by having miners compute SHA256 on slightly modified versions of the block header millions of times over and over again, until eventually one node comes up with a version whose hash is less than the target (currently around 2^{192}). However, this mining algorithm is vulnerable to two forms of centralization. First, the mining ecosystem has come to be dominated by ASICs (application-specific integrated circuits), computer chips designed for, and therefore thousands of times more efficient at, the specific task of Bitcoin mining. This means that Bitcoin mining is no longer a highly decentralized and egalitarian pursuit, requiring millions of dollars of capital to effectively participate in. Second, most Bitcoin miners do not actually perform block validation locally; instead, they rely on a centralized mining pool to provide the block headers. This problem is arguably worse: as of the time of this writing, the top three mining pools indirectly control roughly 50% of processing power in the Bitcoin network, although this is mitigated by the fact that miners can switch to other mining pools if a pool or coalition attempts a 51% attack.

The current intent at Ethereum is to use a mining algorithm where miners are required to fetch random data from the state, compute some randomly selected transactions from the last N blocks in the blockchain, and return the hash of the result. This has two important benefits. First, Ethereum contracts can include any kind of computation, so an Ethereum ASIC would essentially be an ASIC for general computation - ie. a better CPU. Second, mining requires access to the entire blockchain, forcing miners to store the entire blockchain and at least be capable of verifying every transaction. This

removes the need for centralized mining pools; although mining pools can still serve the legitimate role of evening out the randomness of reward distribution, this function can be served equally well by peer-to-peer pools with no central control.

This model is untested, and there may be difficulties along the way in avoiding certain clever optimizations when using contract execution as a mining algorithm. However, one notably interesting feature of this algorithm is that it allows anyone to "poison the well", by introducing a large number of contracts into the blockchain specifically designed to stymie certain ASICs. The economic incentives exist for ASIC manufacturers to use such a trick to attack each other. Thus, the solution that we are developing is ultimately an adaptive economic human solution rather than purely a technical one.

Scalability

One common concern about Ethereum is the issue of scalability. Like Bitcoin, Ethereum suffers from the flaw that every transaction needs to be processed by every node in the network. With Bitcoin, the size of the current blockchain rests at about 15 GB, growing by about 1 MB per hour. If the Bitcoin network were to process Visa's 2000 transactions per second, it would grow by 1 MB per three seconds (1 GB per hour, 8 TB per year). Ethereum is likely to suffer a similar growth pattern, worsened by the fact that there will be many applications on top of the Ethereum blockchain instead of just a currency as is the case with Bitcoin, but ameliorated by the fact that Ethereum full nodes need to store just the state instead of the entire blockchain history.

The problem with such a large blockchain size is centralization risk. If the blockchain size increases to, say, 100 TB, then the likely scenario would be that only a very small number of large businesses would run full nodes, with all regular users using light SPV nodes. In such a situation, there arises the potential concern that the full nodes could band together and all agree to cheat in some profitable fashion (eg. change the block reward, give themselves BTC). Light nodes would have no way of detecting this immediately. Of course, at least one honest full node would likely exist, and after a few hours information about the fraud would trickle out through channels like Reddit, but at that point it would be too late: it would be up to the ordinary users to organize an effort to blacklist the given blocks, a massive and likely infeasible coordination problem on a similar scale as that of pulling off a successful 51% attack. In the case of Bitcoin, this is currently a problem, but there exists a blockchain modification suggested by Peter Todd which will alleviate this issue.

In the near term, Ethereum will use two additional strategies to cope with this problem. First, because of the blockchain-based mining algorithms, at least every miner will be forced to be a full node, creating a lower bound on the number of full nodes. Second and more importantly, however, we will include an intermediate state tree root in the blockchain after processing each transaction. Even if block validation is centralized, as long as one honest verifying node exists, the centralization problem can be circumvented via a verification protocol. If a miner publishes an invalid block, that block must either be badly formatted, or the state `S[n]` is incorrect. Since `S[0]` is known to be correct, there must be some first state `S[i]` that is incorrect where `S[i-1]` is correct. The verifying node would provide the index `i`, along with a "proof of invalidity" consisting of the subset of Patricia tree nodes needing to process `APPLY(S[i-1],TX[i]) -> S[i]`. Nodes would be able to use those nodes to run that part of the computation, and see that the `S[i]` generated does not match the `S[i]` provided.

Another, more sophisticated, attack would involve the malicious miners publishing incomplete blocks, so the full information does not even exist to determine whether or not blocks are valid. The solution to this is a challenge-response protocol: verification nodes issue "challenges" in the form of target transaction indices, and upon receiving a node a light node treats the block as untrusted until another node, whether the miner or another verifier, provides a subset of Patricia nodes as a proof of validity.

Conclusion

The Ethereum protocol was originally conceived as an upgraded version of a cryptocurrency, providing advanced features such as on-blockchain escrow, withdrawal limits, financial contracts, gambling markets and the like via a highly generalized programming language. The Ethereum protocol would not "support" any of the applications directly, but the existence of a Turing-complete programming language means that arbitrary contracts can theoretically be created for any transaction type or application. What is more interesting about Ethereum, however, is that the Ethereum protocol moves far beyond just

currency. Protocols around decentralized file storage, decentralized computation and decentralized prediction markets, among dozens of other such concepts, have the potential to substantially increase the efficiency of the computational industry, and provide a massive boost to other peer-to-peer protocols by adding for the first time an economic layer. Finally, there is also a substantial array of applications that have nothing to do with money at all.

The concept of an arbitrary state transition function as implemented by the Ethereum protocol provides for a platform with unique potential; rather than being a closed-ended, single-purpose protocol intended for a specific array of applications in data storage, gambling or finance, Ethereum is open-ended by design, and we believe that it is extremely well-suited to serving as a foundational layer for a very large number of both financial and non-financial protocols in the years to come.

Notes and Further Reading

Notes

1. A sophisticated reader may notice that in fact a Bitcoin address is the hash of the elliptic curve public key, and not the public key itself. However, it is in fact perfectly legitimate cryptographic terminology to refer to the pubkey hash as a public key itself. This is because Bitcoin's cryptography can be considered to be a custom digital signature algorithm, where the public key consists of the hash of the ECC pubkey, the signature consists of the ECC pubkey concatenated with the ECC signature, and the verification algorithm involves checking the ECC pubkey in the signature against the ECC pubkey hash provided as a public key and then verifying the ECC signature against the ECC pubkey.
2. Technically, the median of the 11 previous blocks.
3. Internally, 2 and "CHARLIE" are both numbers, with the latter being in big-endian base 256 representation. Numbers can be at least 0 and at most 2^{256}-1.

Further Reading

1. Intrinsic value: http://bitcoinmagazine.com/8640/an-exploration-of-intrinsic-value-what-it-is-why-bitcoin-doesnt-have-it-and-why-bitcoin-does-have-it/
2. Smart property: https://en.bitcoin.it/wiki/Smart_Property
3. Smart contracts: https://en.bitcoin.it/wiki/Contracts
4. B-money: http://www.weidai.com/bmoney.txt
5. Reusable proofs of work: http://www.finney.org/~hal/rpow/
6. Secure property titles with owner authority: http://szabo.best.vwh.net/securetitle.html
7. Bitcoin whitepaper: http://bitcoin.org/bitcoin.pdf
8. Namecoin: https://namecoin.org/
9. Zooko's triangle: http://en.wikipedia.org/wiki/Zooko's_triangle
10. Colored coins whitepaper: https://docs.google.com/a/buterin.com/document/d/1AnkP_cVZTCMLIzw4DvsW6M8Q2JC0lIzrTLuoWu2z1BE/edit
11. Mastercoin whitepaper: https://github.com/mastercoin-MSC/spec
12. Decentralized autonomous corporations, Bitcoin Magazine: http://bitcoinmagazine.com/7050/bootstrapping-a-decentralized-autonomous-corporation-part-i/
13. Simplified payment verification: https://en.bitcoin.it/wiki/Scalability#Simplifiedpaymentverification
14. Merkle trees: http://en.wikipedia.org/wiki/Merkle_tree
15. Patricia trees: http://en.wikipedia.org/wiki/Patricia_tree
16. GHOST: http://www.cs.huji.ac.il/~avivz/pubs/13/btc_scalability_full.pdf
17. StorJ and Autonomous Agents, Jeff Garzik: http://garzikrants.blogspot.ca/2013/01/storj-and-bitcoin-autonomous-agents.html
18. Mike Hearn on Smart Property at Turing Festival: http://www.youtube.com/watch?v=Pu4PAMFPo5Y
19. Ethereum RLP: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-RLP
20. Ethereum Merkle Patricia trees: https://github.com/ethereum/wiki/wiki/%5BEnglish%5D-Patricia-Tree
21. Peter Todd on Merkle sum trees: http://sourceforge.net/p/bitcoin/mailman/message/31709140/

Glossary

One of the things that cryptocurrency, and really any new genre of technology, is notorious for is the sheer quantity of vocabulary that gets generated to describe all of the new concepts. Anyone dealing with peer-to-peer internet software on anything more than a casual basis needs to deal with concepts of cryptography, including hashes, signatures and public and private keys, symmetric and asymmetric encryption, denial of service protection, as well as arcane constructions such as distributed hash tables and webs of trust. New Bitcoin users are forced to contend with learning not just the common basics of cryptography, but also additional internal jargon such as "blocks", "confirmations", "mining", "SPV clients" and "51% attacks", as well as economic concepts like incentive-compatibility and the fine nuances of centralization and decentralization. Ethereum, being a decentralized application development platform based on a generalization of a cryptocurrency, necessarily incorporates both of these sets of concepts, as well as adding many of its own. To help anyone new to Ethereum, whether they are in it as cryptocurrency enthusiasts, business owners, social or political visionaries, web developers or are simply ordinary people looking to see how the technology can improve their lives, the following list is intended to provide a basic summary of the vocabulary that Ethereum users often tend to use:

Cryptography

See also: http://en.wikipedia.org/wiki/Public-key_cryptography

- **Computational infeasibility**: a process is computationally infeasible if it would take an impracticably long time (eg. billions of years) to do it for anyone who might conceivably have an interest in carrying it out. Generally, 2^{80} computational steps is considered the lower bound for computational infeasibility.
- **Hash**: a hash function (or hash algorithm) is a process by which a document (ie. a piece of data or file) is processed into a small piece of data (usually 32 bytes) which looks completely random, and from which no meaningful data can be recovered about the document, but which has the important property that the result of hashing one particular document is always the same. Additionally, it is crucially important that it is computationally infeasible to find two documents that have the same hash. Generally, changing even one letter in a document will completely randomize the hash; for example, the SHA3 hash of "Saturday" is `c38bbc8e93c09f6ed3fe39b5135da91ad1a99d397ef16948606cdcbd14929f9d`, whereas the SHA3 hash of Caturday is `b4013c0eed56d5a0b448b02ec1d10dd18c1b3832068fbbdc65b98fa9b14b6dbf`. Hashes are usually used as a way of creating a globally agreed-upon identifier for a particular document that cannot be forged.
- **Encryption**: encryption is a process by which a document (**plaintext**) is combined with a shorter string of data, called a **key** (eg. `c85ef7d79691fe79573b1a7064c19c1a9819ebdbd1faaab1a8ec92344438aaf4`), to produce an output (**ciphertext**) which can be "decrypted" back into the original plaintext by someone else who has the key, but which is incomprehensible and computationally infeasible to decrypt for anyone who does not have the key.
- **Public key encryption**: a special kind of encryption where there is a process for generating two keys at the same time (typically called a **private key** and a **public key**), such that documents encrypted using one key can be decrypted with the other. Generally, as suggested by the name, individuals publish their public keys and keep their private keys to themselves.
- **Digital signature**: a digital signing algorithm is a process by which a user can produce a short string of data called a "signature" of a document using a private key such that anyone with the corresponding public key, the signature and the document can verify that (1) the document was "signed" by the owner of that particular private key, and (2) the document was not changed after it was signed. Note that this differs from traditional signatures where you can scribble extra text onto a document after you sign it and there's no way to tell the difference; in a digital signature any change to the document will render the signature invalid.

Blockchains

See also: https://bitcoin.org/en/vocabulary

- **Address**: an address is essentially the representation of a public key belonging to a particular user; for example, the address associated with the private key given above is `cd2a3d9f938e13cd947ec05abc7fe734df8dd826`. Note that in

practice, the address is technically the hash of a public key, but for simplicity it's better to ignore this distinction.

- **Transaction**: a transaction is a document authorizing some particular action associated with the blockchain. In a currency, the dominant transaction type is sending currency units or tokens to someone else; in other systems actions like registering domain names, making and fulfilling trade offers and entering into contracts are also valid transaction types.
- **Block**: a block is a package of data that contains zero or more transactions, the hash of the previous block ("parent"), and optionally other data. The total set of blocks, with every block except for the initial "genesis block" containing the hash of its parent, is called the **blockchain** and contains the entire transaction history of a network. Note that some blockchain-based cryptocurrencies instead use the word "ledger" for a blockchain; the two are roughly equivalent, although in systems that use the term "ledger" each block generally contains a full copy of the current state (eg. currency balances, partially fulfilled contracts, registrations) of every account allowing users to discard outdated historical data.
- **Account**: an account is the entry in a ledger, indexed by its address, that contains the complete data about the state of that account. In a currency system, this involves currency balances and perhaps unfulfilled trade orders; in other cases more complex relationships may be stored inside of accounts.
- **Proof of work**: one important property of a block in Bitcoin, Ethereum and many other crypto-ledgers is that the hash of the block must be smaller than some target value. The reason this is necessary is that in a decentralized system anyone can produce blocks, so in order to prevent the network from being flooded with blocks, and to provide a way of measuring how much consensus there is behind a particular version of the blockchain, it must in some way be hard to produce a block. Because hashes are pseudorandom, finding a block whose hash is less than `0000000100000000000000000000000000000000000000000000000000000000` takes an average of 4.3 billion attempts. In all such systems, the target value self-adjusts so that on average one node in the network finds a block every N minutes (eg. N = 10 for Bitcoin and 1 for Ethereum).
- **Nonce**: a meaningless value in a block which can be adjusted in order to try to satisfy the proof of work condition
- **Mining**: mining is the process of repeatedly aggregating transactions, constructing a block and trying difference nonces until a nonce is found that satisfies the proof of work condition. If a miner gets lucky and produces a valid block, they are granted a certain number of coins as a reward as well as all of the transaction fees in the block, and all miners start trying to create a new block containing the hash of the newly generated block as their parent.
- **Stale**: a stale is a block that is created when there is already another block with the same parent out there; stales typically get discarded and are wasted effort.
- **Fork**: a situation where two blocks are generated pointing to the same block as their parent, and some portion of miners see one block first and some see the other. This may lead to two blockchains growing at the same time. Generally, it is mathematically near-certain that a fork will resolve itself within four blocks as miners on one chain will eventually get lucky and that chain will grow longer and all miners switch to it; however, forks may last longer if miners disagree on whether or not a particular block is valid.
- **Double spend**: a deliberate fork, where a user with a large amount of mining power sends a transaction to purchase some produce, then after receiving the product creates another transaction sending the same coins to themselves. The attacker then creates a block, at the same level as the block containing the original transaction but containing the second transaction instead, and starts mining on the fork. If the attacker has more than 50% of all mining power, the double spend is guaranteed to succeed eventually at any block depth. Below 50%, there is some probability of success, but it is usually only substantial at a depth up to about 2-5; for this reason, most cryptocurrency exchanges, gambling sites and financial services wait until six blocks have been produced ("six confirmations") before accepting a payment.
- **SPV client** (or **light client**) - a client that downloads only a small part of the blockchain, allowing users of low-power or low-storage hardware like smartphones and laptops to maintain almost the same guarantee of security by sometimes selectively downloading small parts of the state without needing to spend megabytes of bandwidth and gigabytes of storage on full blockchain validation and maintennance.

Ethereum Blockchain

See also: http://ethereum.org/ethereum.html

- **Serialization**: the process of converting a data structure into a sequence of bytes. Ethereum internally uses an

encoding format called recursive-length prefix encoding (RLP), described here

- **Patricia tree** (or **trie**): a data structure which stores the state of every account. The trie is built by starting from each individual node, then splitting the nodes into groups of up to 16 and hashing each group, then making hashes of hashes and so forth until there is one final "root hash" for the entire trie. The trie has the important properties that (1) there is exactly one possible trie and therefore one possible root hash for each set of data, (2) it is very easy to update, add or remove nodes in the trie and generate the new root hash, (3) there is no way to modify any part of the tree without changing the root hash, so if the root hash is included in a signed document or a valid block the signature or proof of work secures the entire tree, and (4) one can provide just the "branch" of a tree going down to a particular node as cryptographic proof that that node is indeed in the tree with that exact content. Patricia trees are also used to store the internal storage of accounts as well as transactions and uncles. See here for a more detailed description.
- **GHOST**: GHOST is a protocol by which blocks can contain a hash of not just their parent, but also hashes for stales that are other children of the parent's parent (called **uncles**). This ensures that stales still contribute to blockchain security, and mitigates a problem with fast blockchains that large miners have an advantage because they hear about their own blocks immediately and so are less likely to generate stales.
- **Uncle**: a child of a parent of a parent of a block that is not the parent, or more generally a child of an ancestor that is not an ancestor. If A is an uncle of B, B is a **nephew** of A.
- **Account nonce**: a transaction counter in each account. This prevents replay attacks where a transaction sending eg. 20 coins from A to B can be replayed by B over and over to continually drain A's balance.
- **EVM code**: Ethereum virtual machine code, the programming language in which accounts on the Ethereum blockchain can contain code. The EVM code associated with an account is executed every time a message is sent to that account, and has the ability to read/write storage and itself send messages.
- **Message**: a sort of "virtual transaction" sent by EVM code from one account to another. Note that "transactions" and "messages" in Ethereum are different; a "transaction" in Ethereum parlance specifically refers to a physical digitally signed piece of data that goes in the blockchain, and every transaction triggers an associated message, but messages can also be sent by EVM code, in which case they are never represented in data anywhere.
- **Storage**: a key/value database contained in each account, where keys and values are both 32-byte strings but can otherwise contain anything.
- **Externally owned account**: an account controlled by a private key. Externally owned accounts cannot contain EVM code.
- **Contract**: an account which contains, and is controlled by, EVM code. Contracts cannot be controlled by private keys directly; unless built into the EVM code, a contract has no owner once released.
- **Ether**: the primary internal cryptographic token of the Ethereum network. Ether is used to pay transaction and computation fees for Ethereum transactions.
- **Gas**: a measurement roughly equivalent to computational steps. Every transaction is required to include a gas limit and a fee that it is willing to pay per gas; miners have the choice of including the transaction and collecting the fee or not. If the total number of gas used by the computation spawned by the transaction, including the original message and any sub-messages that may be triggered, is less than or equal to the gas limit, then the transaction processes. If the total gas exceeds the gas limit, then all changes are reverted, except that the transaction is still valid and the fee can still be collected by the miner. Every operation has a gas expenditure; for most operations it is 1, although some expensive operations fave expenditures up to 100 and a transaction itself has an expenditure of 500.

Non-blockchain

- **EtherBrowser**: the upcoming primary client for Ethereum, which will exist in the form of a web browser that can be used to access both normal websites and applications built on top of the Ethereum platform
- **Whisper**: an upcoming P2P messaging protocol that will be integrated into the EtherBrowser
- **Swarm**: an upcoming P2P data storage protocol optimized for static web hosting that will be integrated into the EtherBrowser
- **Solidity**, **LLL**, **Serpent** and **Mutan**: prorgamming languages for writing contract code which can be compiled into EVM code. Serpent can also be compiled into LLL.
- **PoC**: proof-of-concept, another name for a pre-launch release

Surrounding concepts: applications and governance

- **Decentralized application**: a program which is run by many people which either uses or creates a decentralized network for some specific purpose (eg. connecting buyers and sellers in some marketplace, sharing files, online file storage, maintaining a currency). Ethereum-based decentralized applications (also called Đapps, where the Đ is the Norse letter "eth") typically consist of an HTML/Javascript webpage, and if viewed inside the EtherBrowser the browser recognizes special Javascript APIs for sending transactions to the blockchain, reading data from the blockchain and interacting with Whisper and Swarm. A Đapp typically also has a specific associated contract on the blockchain, though Đapps that facilitate the creation of many contracts are quite possible.
- **Decentralized organization**: an organization that has no centralized leadership, instead using a combination of formal democratic voting processes and stigmergic self-organization as their primary operating principles. A less impressive but sometimes confused concept is a "geographically distributed organization", an organization where people work far apart from each other and which may even have no office at all; GDOs can still have formal centralized leadership.
- **Theseus criterion**: a test for determining how decentralized an organization is. The test is as follows: suppose the organization has N people, and aliens try to pick K people in the organization at a time (eg. once per week) and zap them out of existence, replacing each group with K new people who know nothing about the organization. For how high K can the organization function? Dictatorships fail at K = 1 once the dictator is zapped away; the US government does slightly better but would still be in serious trouble if all 638 members of the Senate and Congress were to disappear, where as something like Bitcoin or BitTorrent is resilient for extremely high K because new agents can simply follow their economic incentives to fill in the missing roles. A stricter test, the **Byzantine Theseus criterion**, involves randomly substituting K users at a time with malicious actors for some time before replacing them with new users.
- **Decentralized autonomous organization**: decentralized organizations where the method of governance is in some fashion "autonomous", ie. it's not controlled by some form of discussion process or committee.
- **No common language criterion**: a test for determining how autonomous an organization is. The test is as follows: suppose the N people in the organization speak N different languages, with none in common. Can the organization still function?
- **Delegative democracy** (or **liquid democracy**): a governance mechanism for DOs and DAOs where everyone votes on everything by default, but where individuals can select specific individuals to vote for them on specific issues. The idea is to generalize the tradeoff between full direct democracy with every individual having equal power and the expert opinion and quick decision making ability provided by specific individuals, allowing people to defer to friends, politicians, subject matter experts or anyone else to exactly the extent that they want to.
- **Futarchy**: a governance mechanism originally proposed by Robin Hanson for managing political organizations, but which is actually extremely applicable for DOs and DAOs: rule by prediction market. Essentially, some easily measurable success metrics are chosen, and tokens are released which will pay out some time in the future (eg. after 1 year) depending on the values of those success metrics, with one such token for each possible action to take. Each of these tokens are traded against a corresponding dollar token, which pays out exactly $1 if the corresponding measure is taken (if the corresponding measure is not taken, both types of tokens pay $0, so the probability of the action actually being taken does not affect the price). The action that the market predicts will have the best results, as judged by the token's high price on its market, is the action that will be taken. This provides another, autonomous, mechanism for selecting for and simultaneously rewarding expert opinion.

Economics

- **Token system**: essentially, a fungible virtual good that can be traded. More formally, a token system is a database mapping addresses to numbers with the property that the primary allowed operation is a transfer of N tokens from A to B, with the conditions that N is non-negative, N is not greater than A's current balance, and a document authorizing the transfer is digitally signed by A. Secondary "issuance" and "consumption" operations may also exist, transaction fees may also be collected, and simultaneous multi-transfers with many parties may be possible. Typical use cases include currencies, cryptographic tokens inside of networks, company shares and digital gift cards.
- **Namespace**: a database mapping names to values. In the simplest example anyone can register an entry if the name has not already been taken (perhaps after paying some fee). If a name has been taken then it can only be changed (if at all) by the account that made the original registration (in many systems, ownership can also be transferred). Namespaces can be used to store usernames, public keys, internet domain names, token systems or other namespaces, and many other applications.
- **Identity**: a set of cryptographically verifiable interactions that have the property that they were all created by the same

person

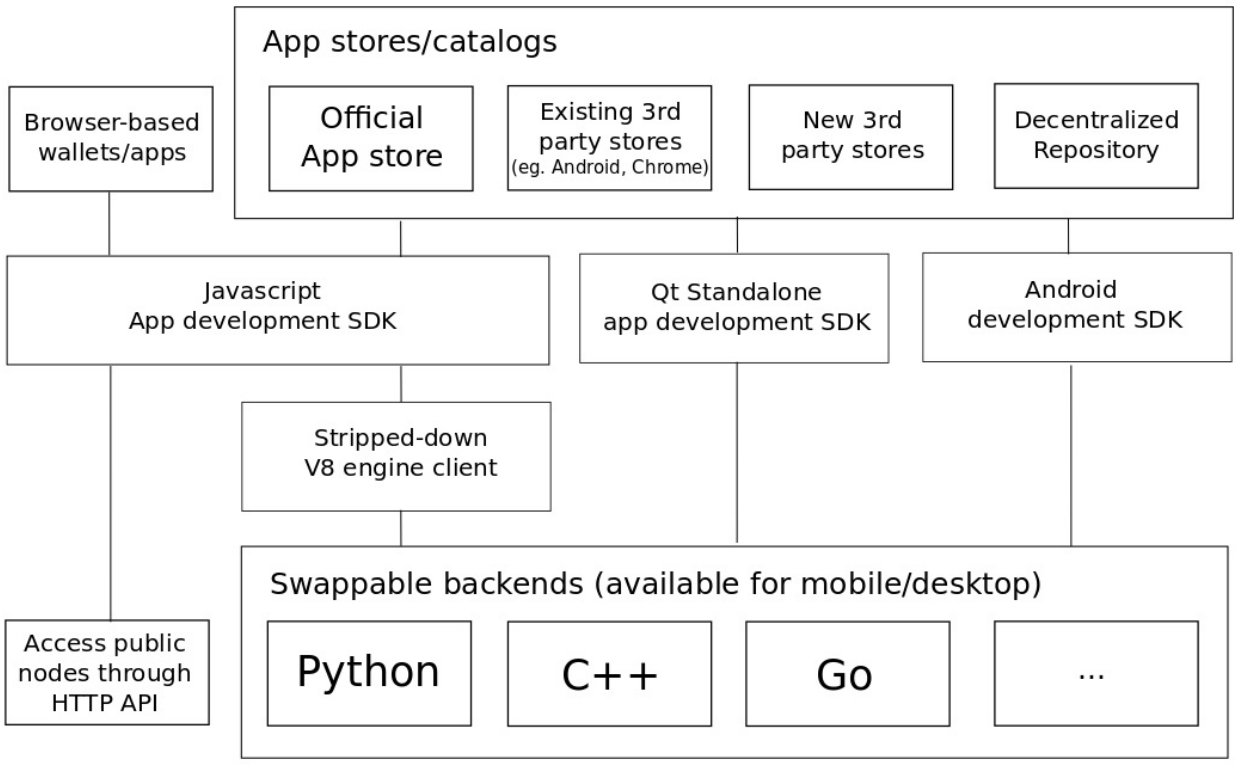
- **Unique identity**: a set of cryptographically verifiable interactions that have the property that they were all created by the same person, with the added constraint that one person cannot have multiple unique identities
- **Incentive compatibility**: a protocol is incentive-compatible if everyone is better off "following the rules" than attempting to cheat, at least unless a very large number of people agree to cheat together at the same time.
- **Basic income**: the idea of issuing some quantity of tokens to every unique identity every period of time (eg. months), with the ultimate intent being that people who do not wish to work or cannot work can survive on this allowance alone. These tokens can simply be crafted out of thin air, or they can come from some revenue stream (eg. profit from some revenue-generating entity, or a government); in order for the BI to be sufficient for a person to live on it alone, a combination of revenue streams will likely have to be used.
- **Public good**: a service which provides a very small benefit to a very large number of people, such that no individual has a large effect on whether or not the good is produced and so no one has the incentive to help pay for it.
- **Reputation**: the property of an identity that other entities believe that identity to be either (1) competent at some specific task, or (2) trustworthy in some context, ie. not likely to betray others even if short-term profitable.
- **Web of trust**: the idea that if A highly rates B, and B highly rates C, then A is likely to trust C. Complicated and powerful mechanisms for determining the reliability of specific individuals in specific concepts can theoretically be gathered from this principle.
- **Escrow**: if two low-reputation entities are engaged in commerce, the payer may wish to leave the funds with a high-reputation third party and instruct that party to send the funds to the payee only when the product is delivered. This reduces the risk of the payer or payee committing fraud.
- **Deposit**: digital property placed into a contract involving another party such that if certain conditions are not satisfied that property is automatically forfeited to the counterparty
- **Hostage**: digital property placed into a contract such that if certain conditions are not satisfied that property is automatically either destroyed or donated to charity or basic income funds, perhaps with widely distributed benefit but necessarily with no significant benefit to any specific individual.

This chapter contains **all the diagrams Vitalik created, adapted or used** until 26 Feb 2015 to explain concepts in various blog posts, presentations and forum posts **since ethereum's inception**.

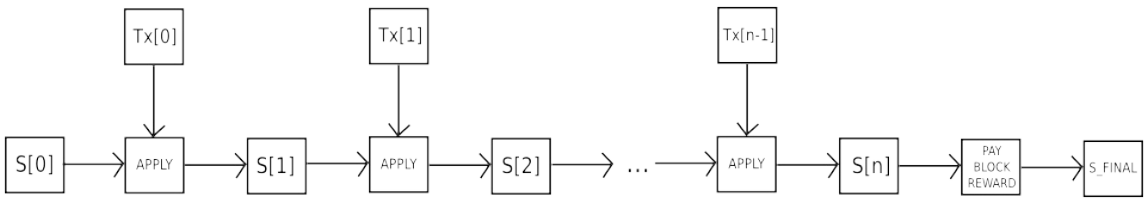
In a true open source fashion, if you want to reuse, edit or update some of the diagrams you can find the editable .svg files here. The only request would be to share back with the community :)

Enjoy!

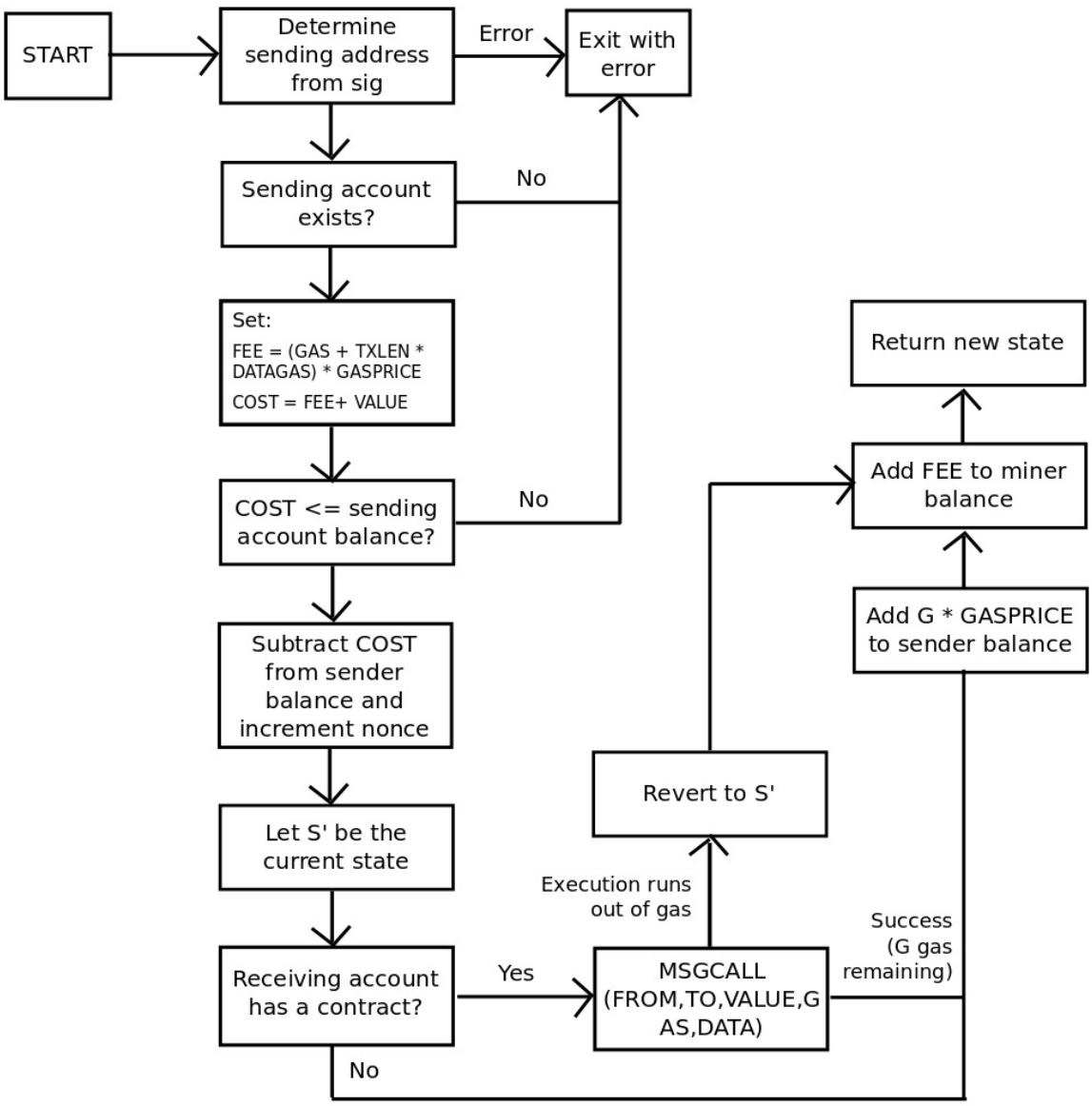
App ecosystem diagram



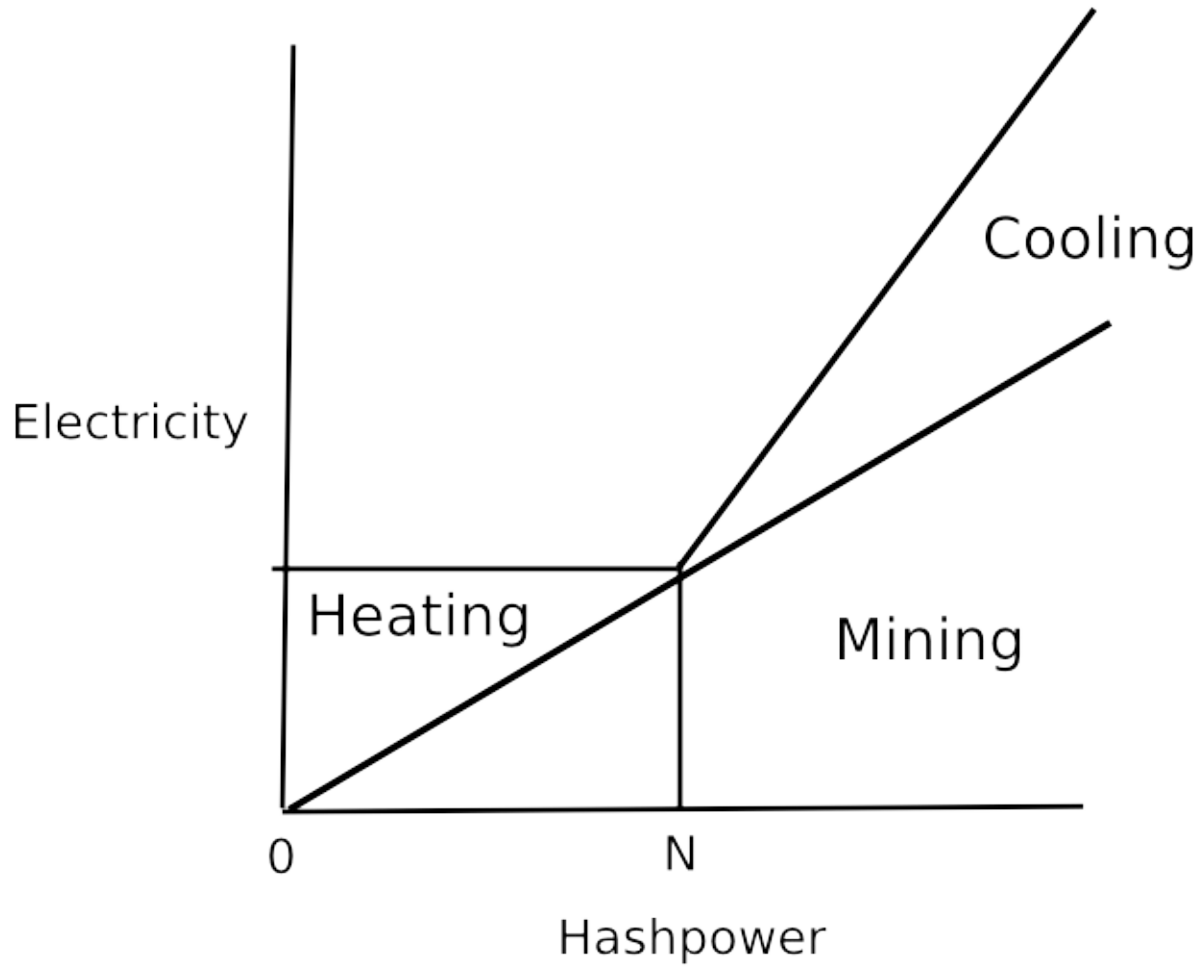
Apply block diagram



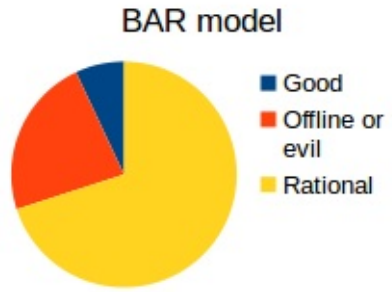
Apply TX diagram


START
Determine sending address from sig
Error
Exit with error
Sending account exists?
No
Set:
FEE = (GAS + TXLEN * DATAGAS) * GASPRICE
COST = FEE+ VALUE
COST <= sending account balance?
No
Subtract COST from sender balance and increment nonce
Let S' be the current state
Receiving account has a contract?
Yes
MSGCALL (FROM,TO,VALUE,GAS,DATA)
Execution runs out of gas
Revert to S'
Success (G gas remaining)
Add G * GASPRICE to sender balance
Add FEE to miner balance
Return new state
No

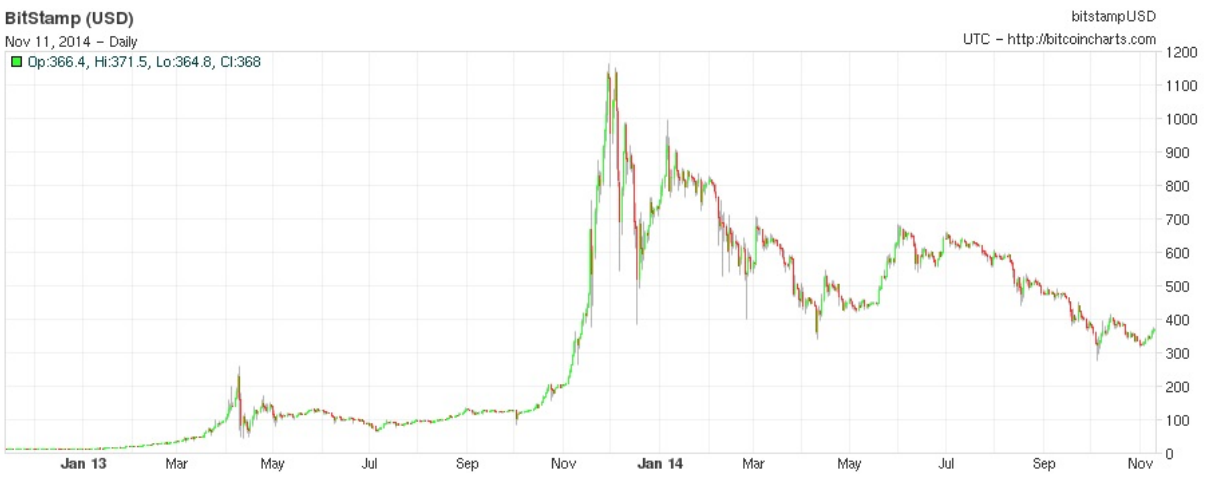
Asic home



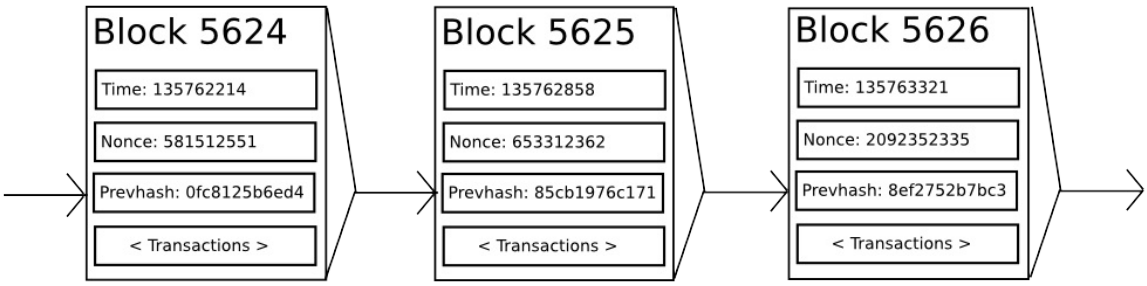
BAR model



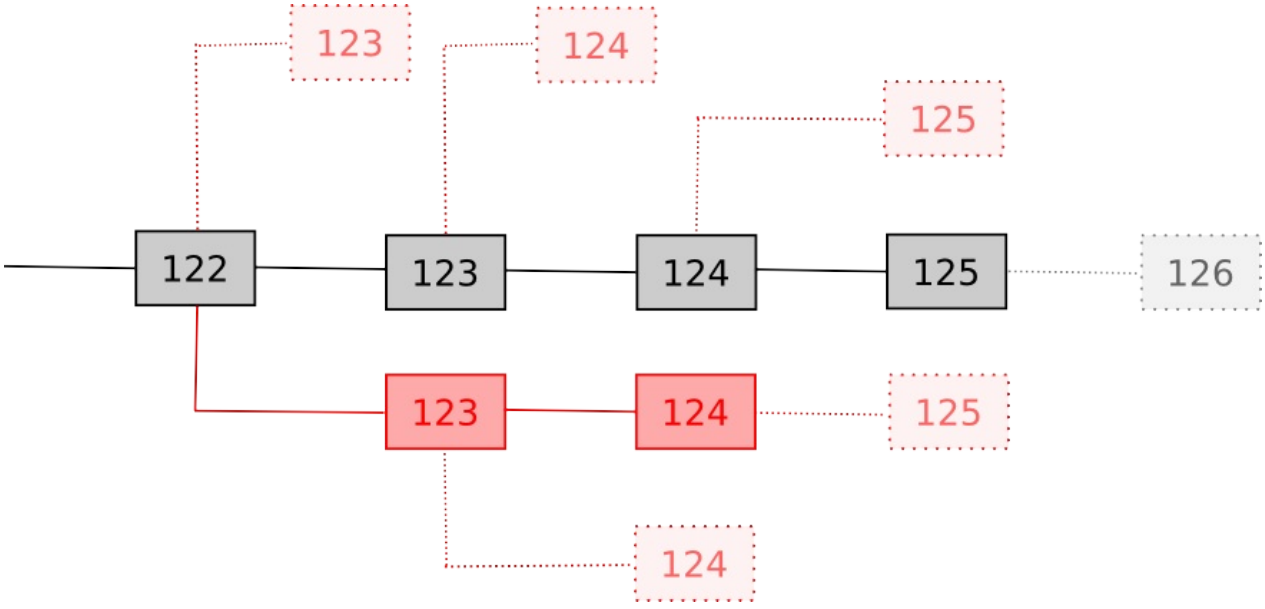
Bitcoin price chart



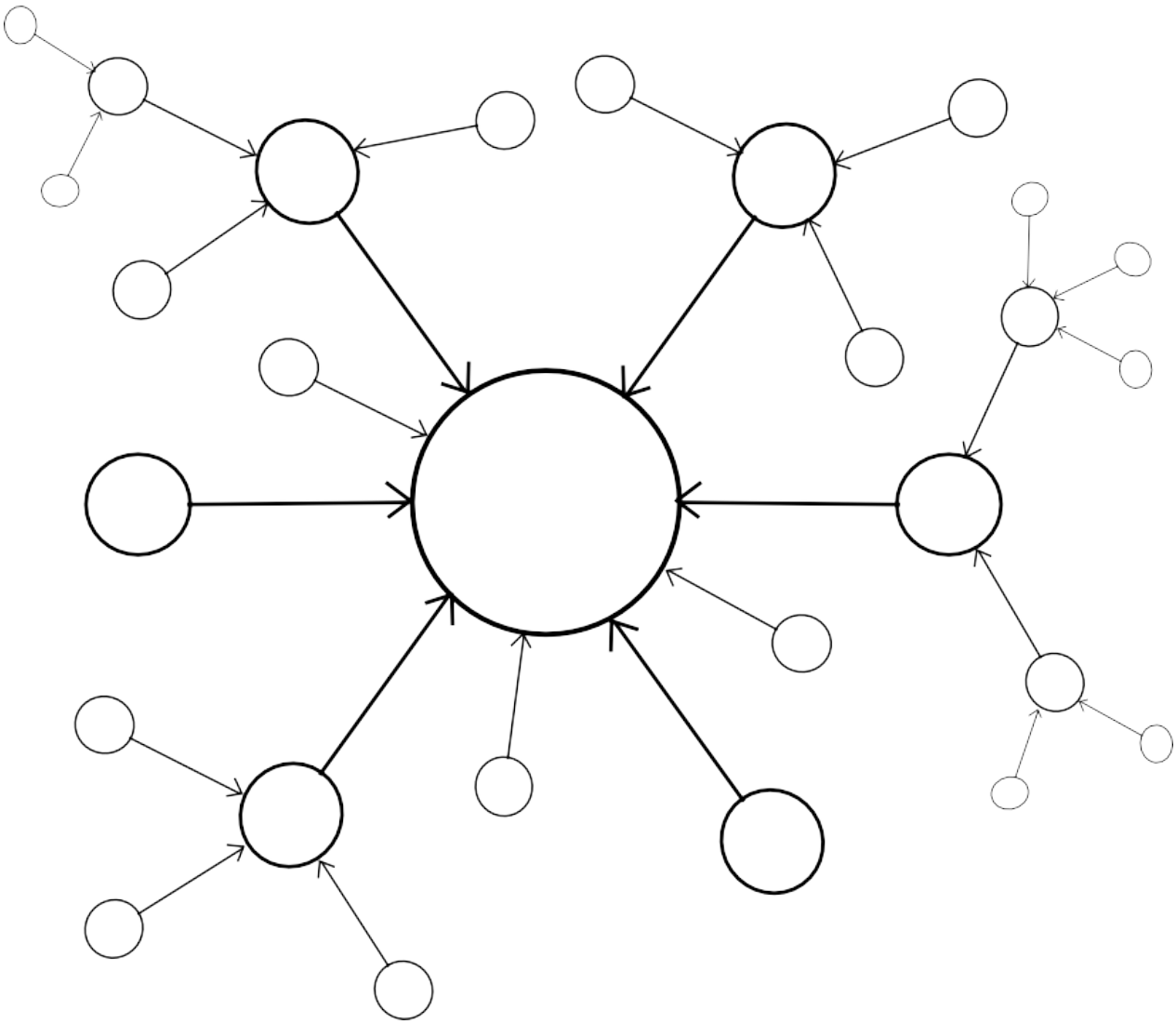
Block



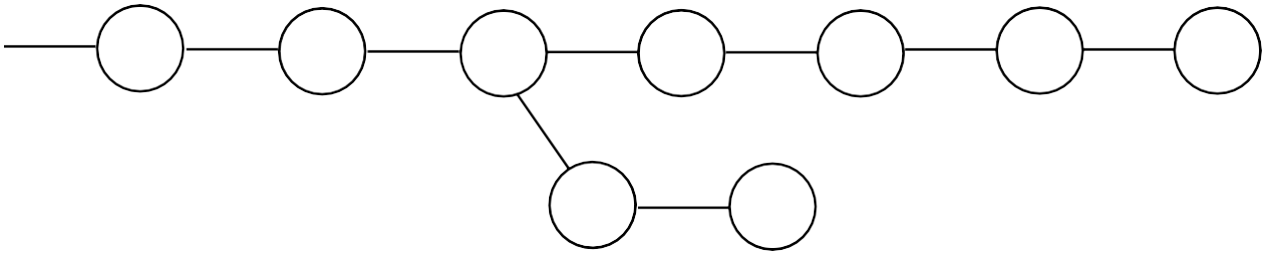
Blockchain



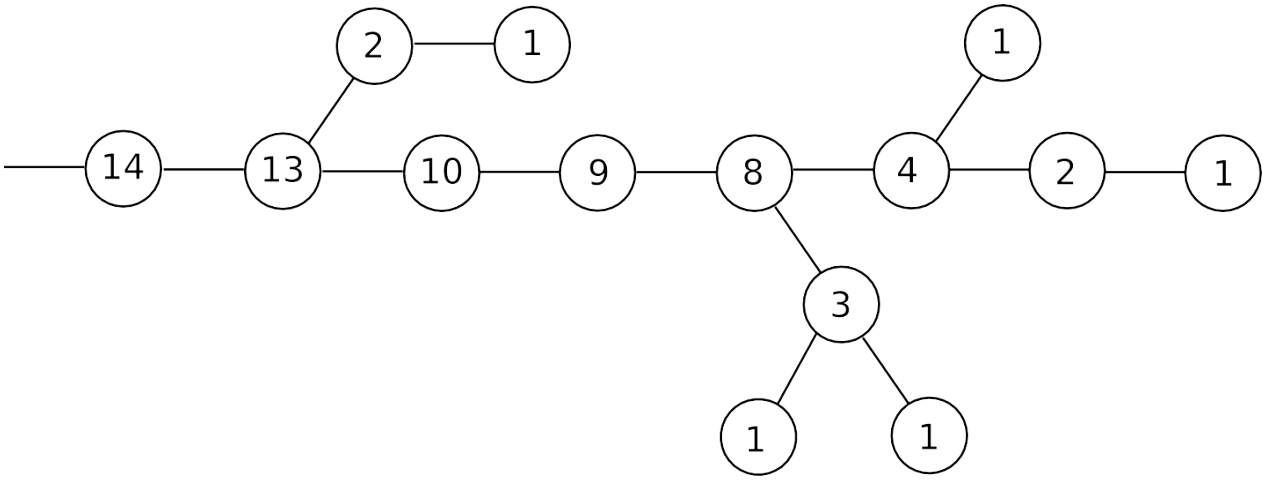
Centralized multichain



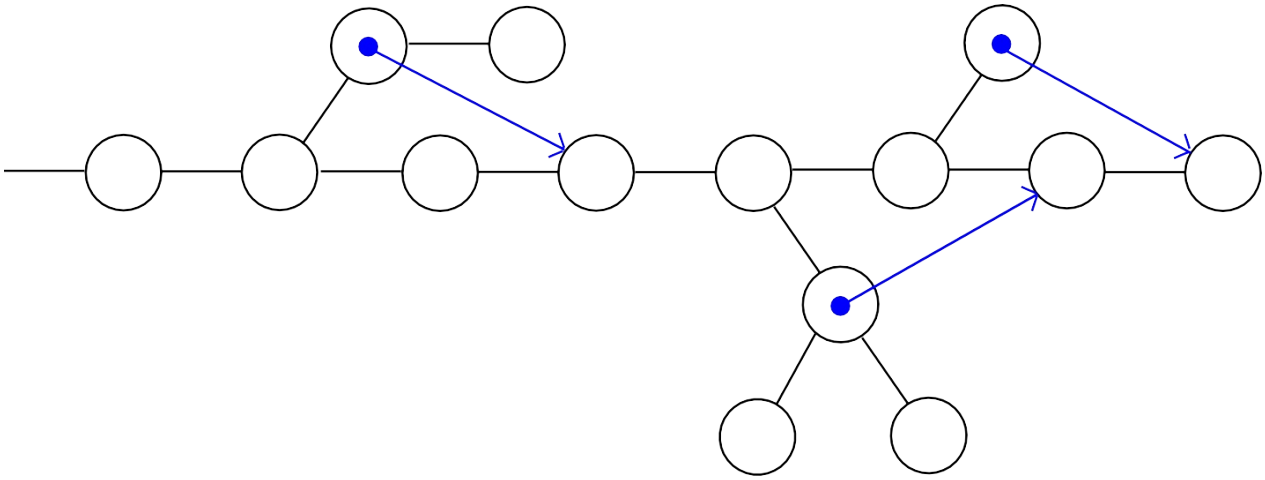
Chain forks



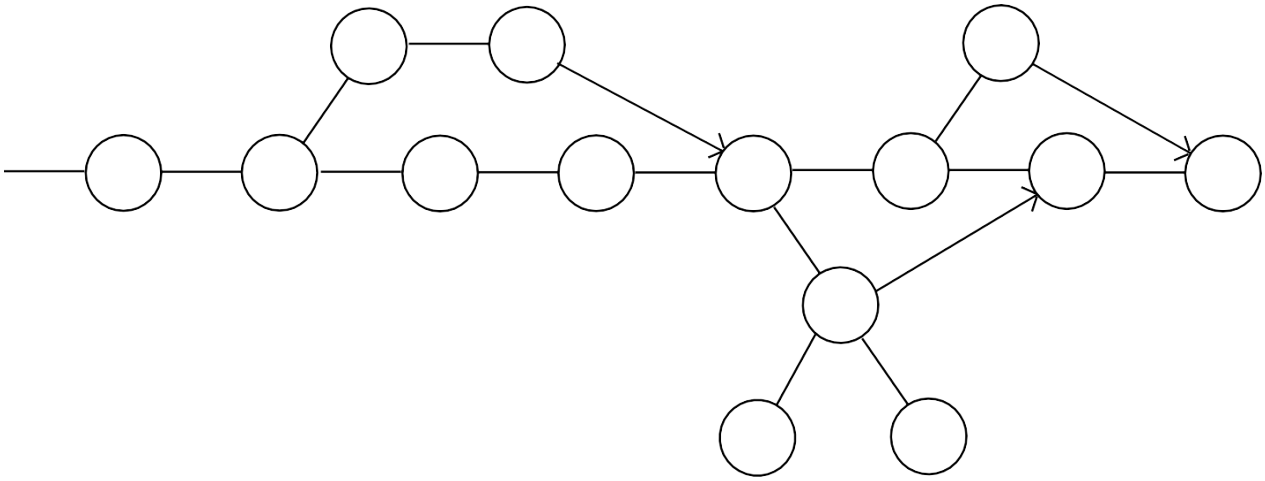
Chain selection



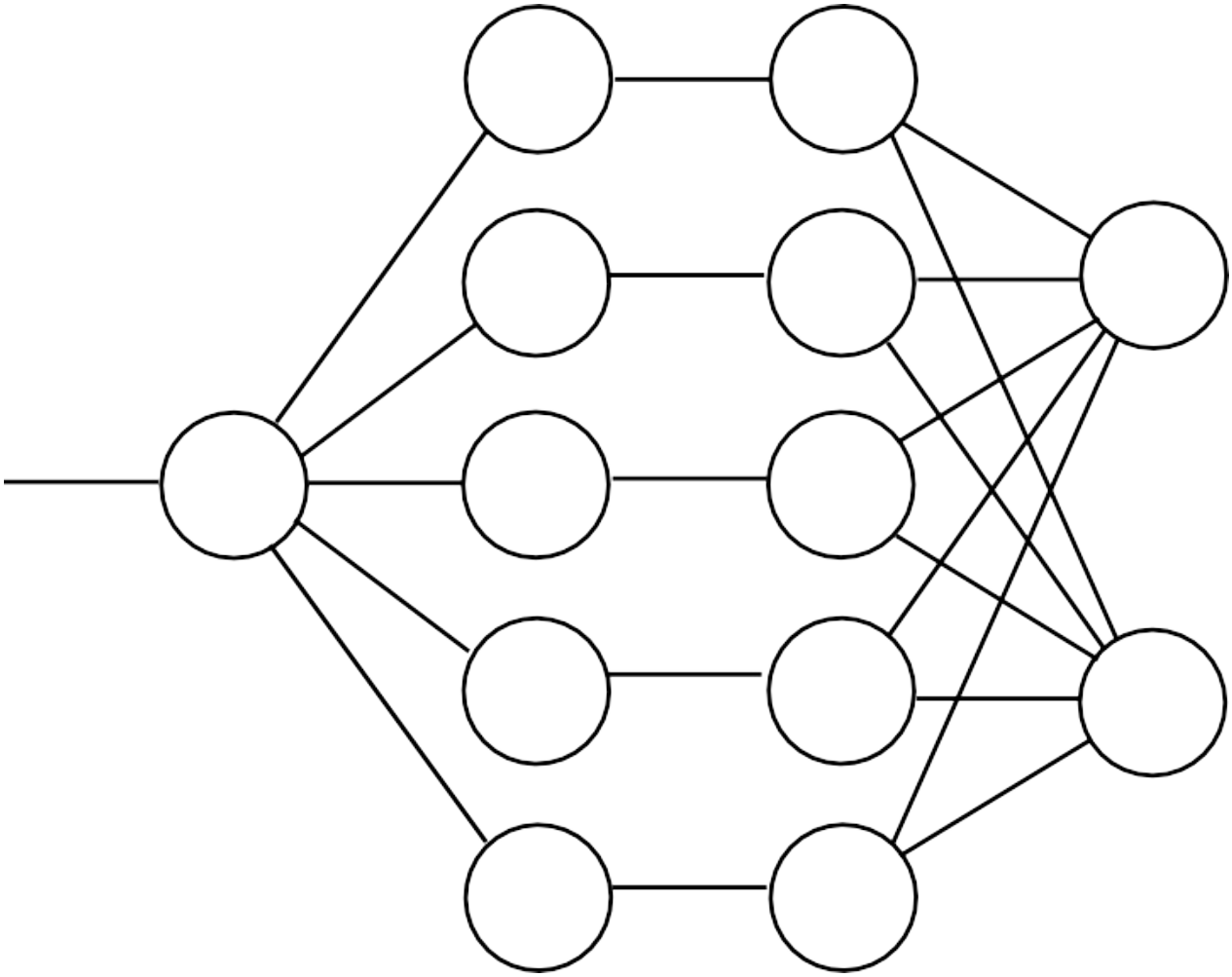
Chain selection 2



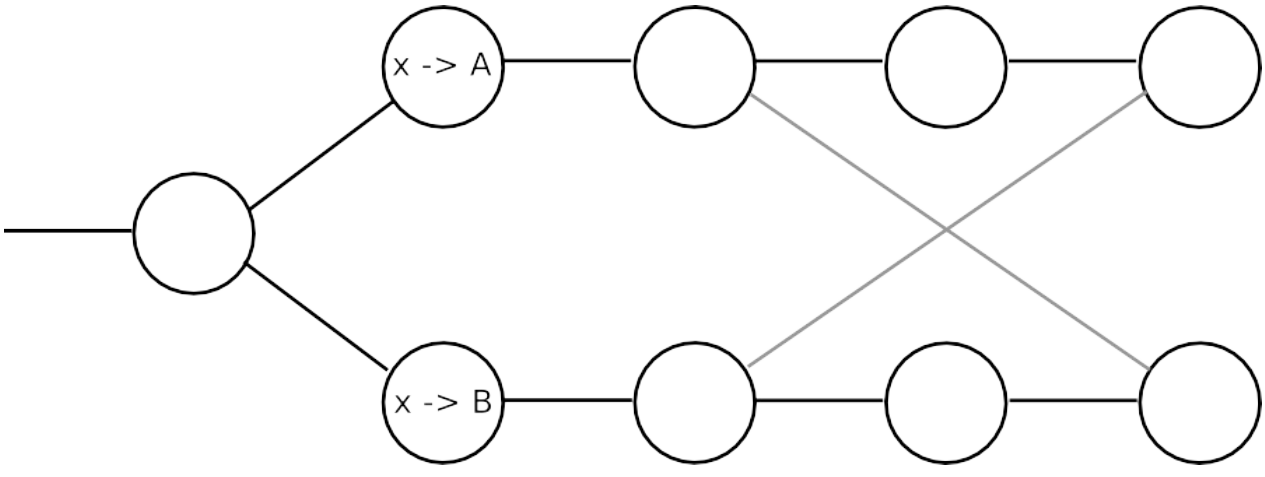
Chain selection 3



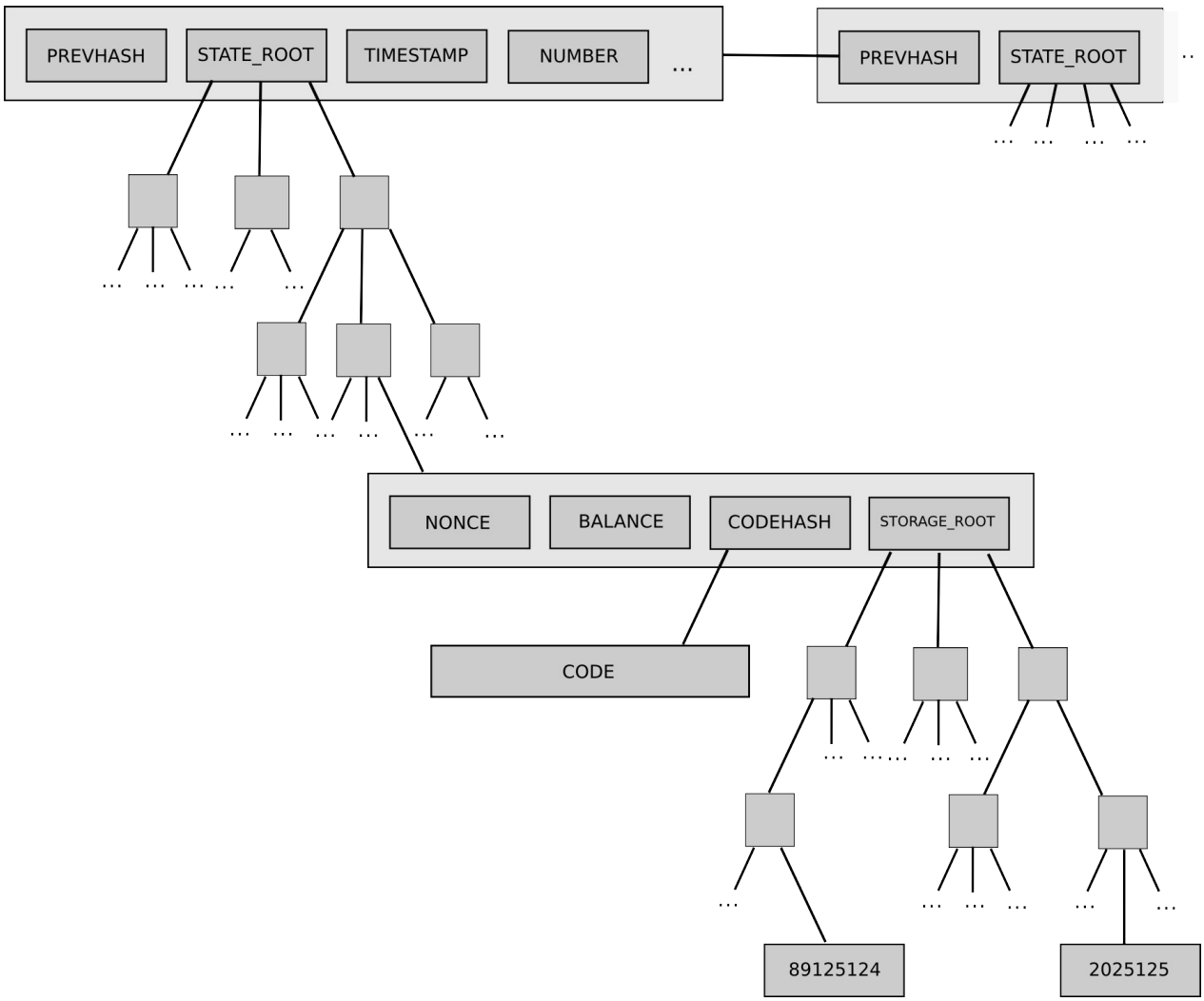
Chain selection 4



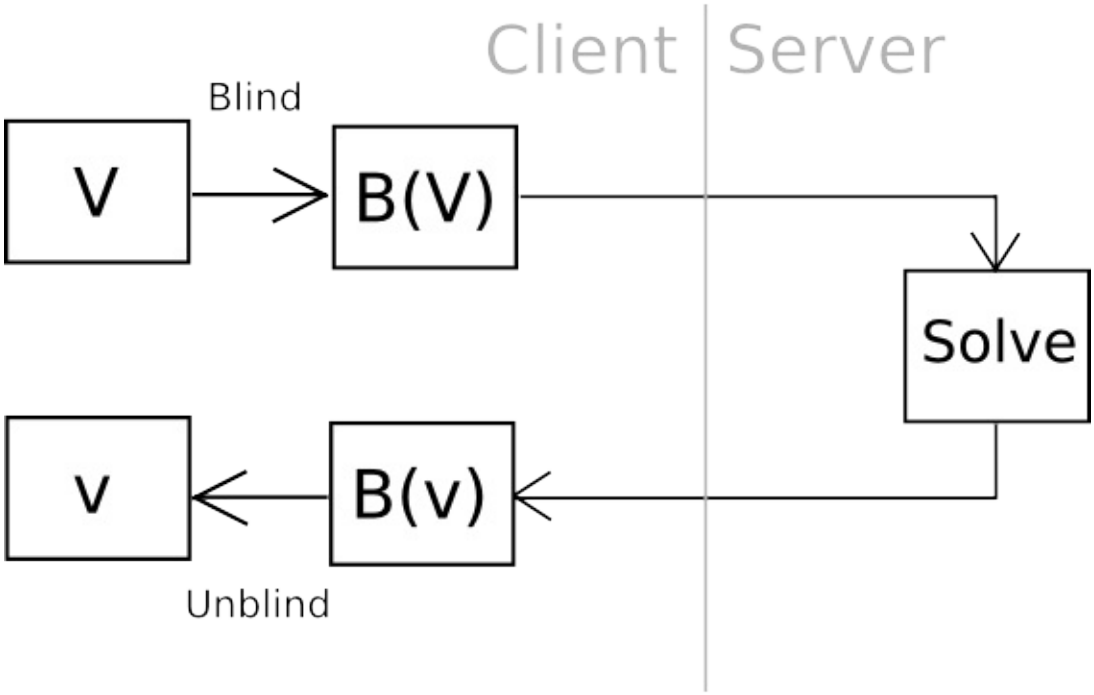
Chain selection 5



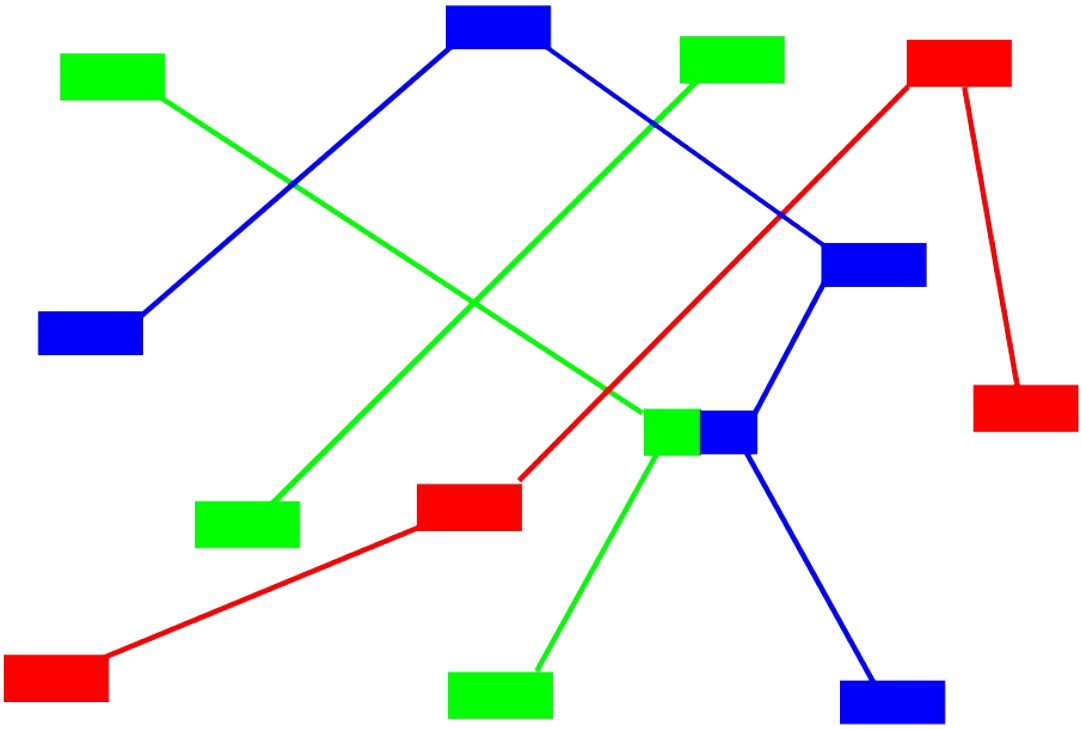
Chain diagram



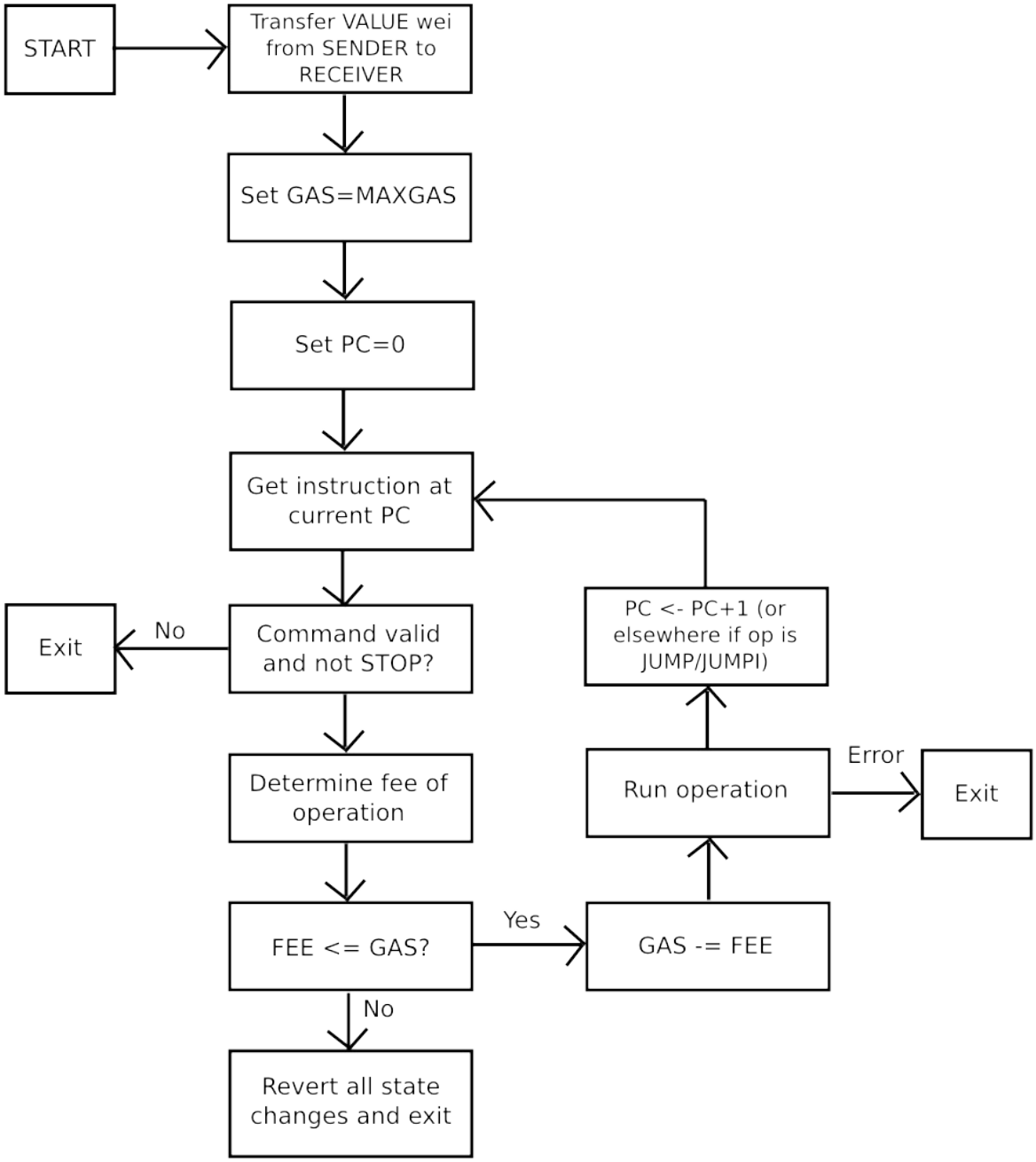
Client server



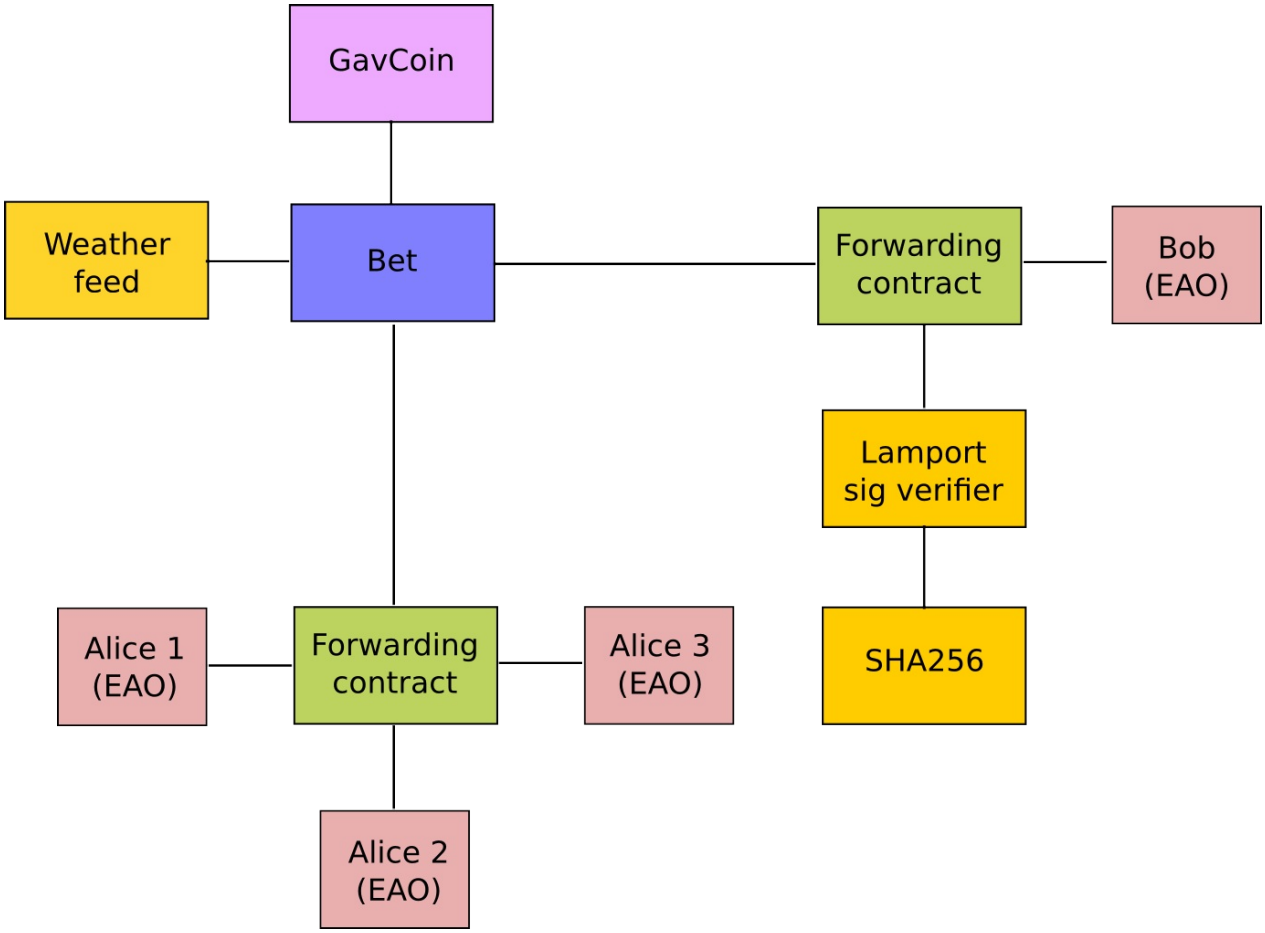
Colored coins UTXO tree



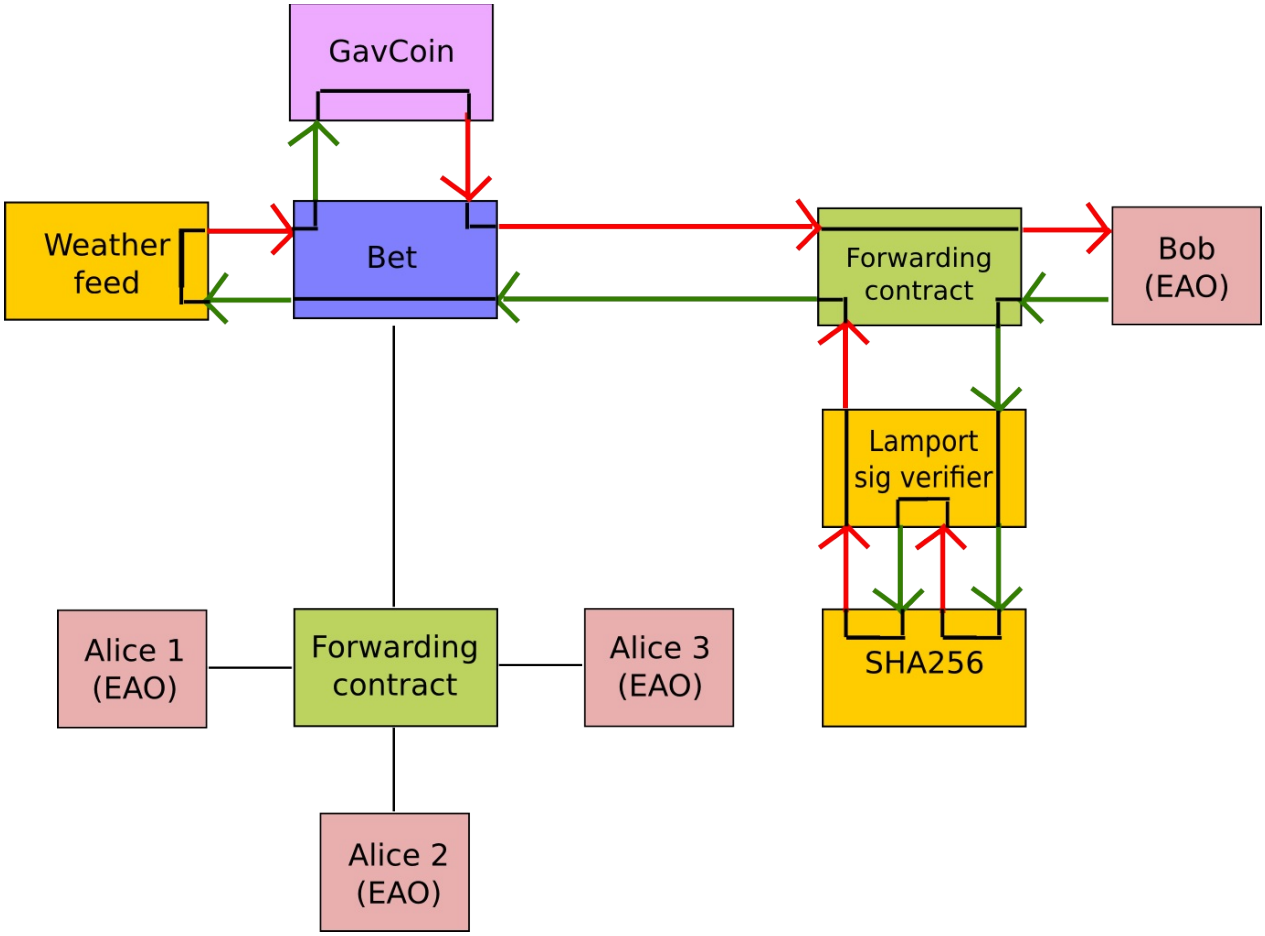
Contract execution flowchart


START
Transfer VALUE wei from SENDER to RECEIVER
Set GAS=MAXGAS
Set PC=0
Get instruction at current PC
Command valid and not STOP?
No
Exit
Determine fee of operation
FEE <= GAS?
Yes
No
Revert all state changes and exit
GAS -= FEE
Run operation
Error
Exit
PC <- PC+1 (or elsewhere if op is JUMP/JUMPI)

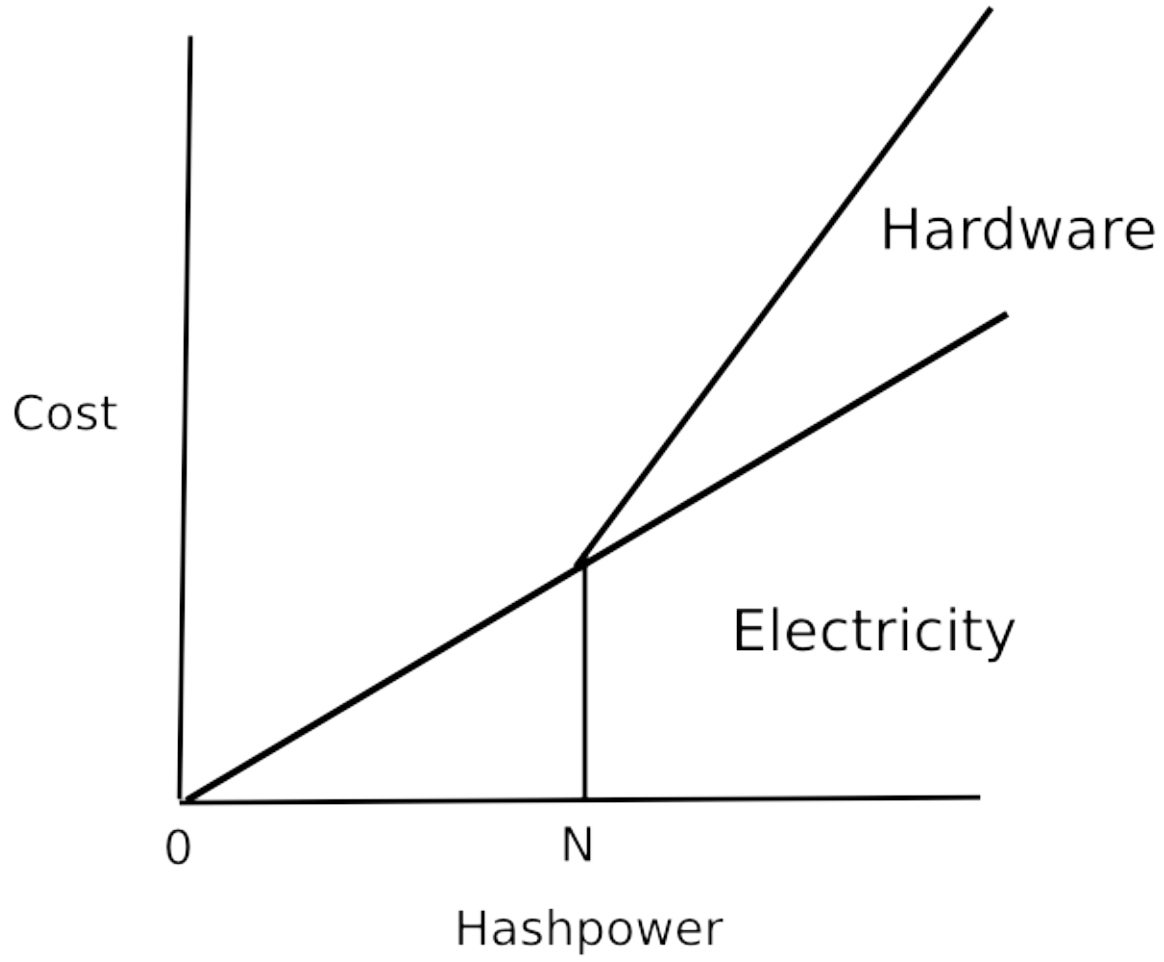
Contract relationship



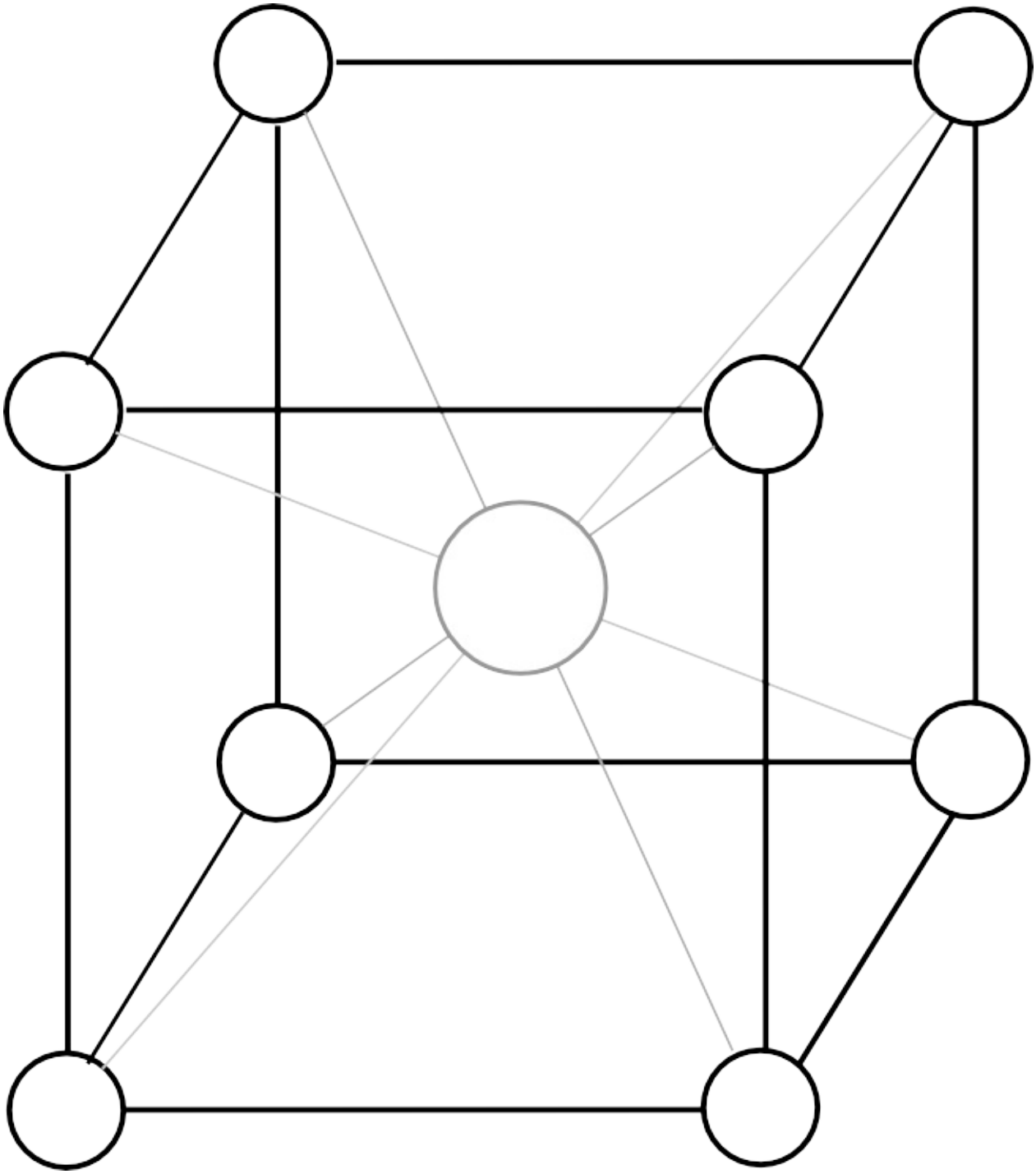
Contract relationship 2


GavCoin
Weather feed
Bet
Forwarding contract
Bob (EAO)
Lamport sig verifier
SHA256
Alice 1 (EAO)
Forwarding contract
Alice 3 (EAO)
Alice 2 (EAO)

CPU home



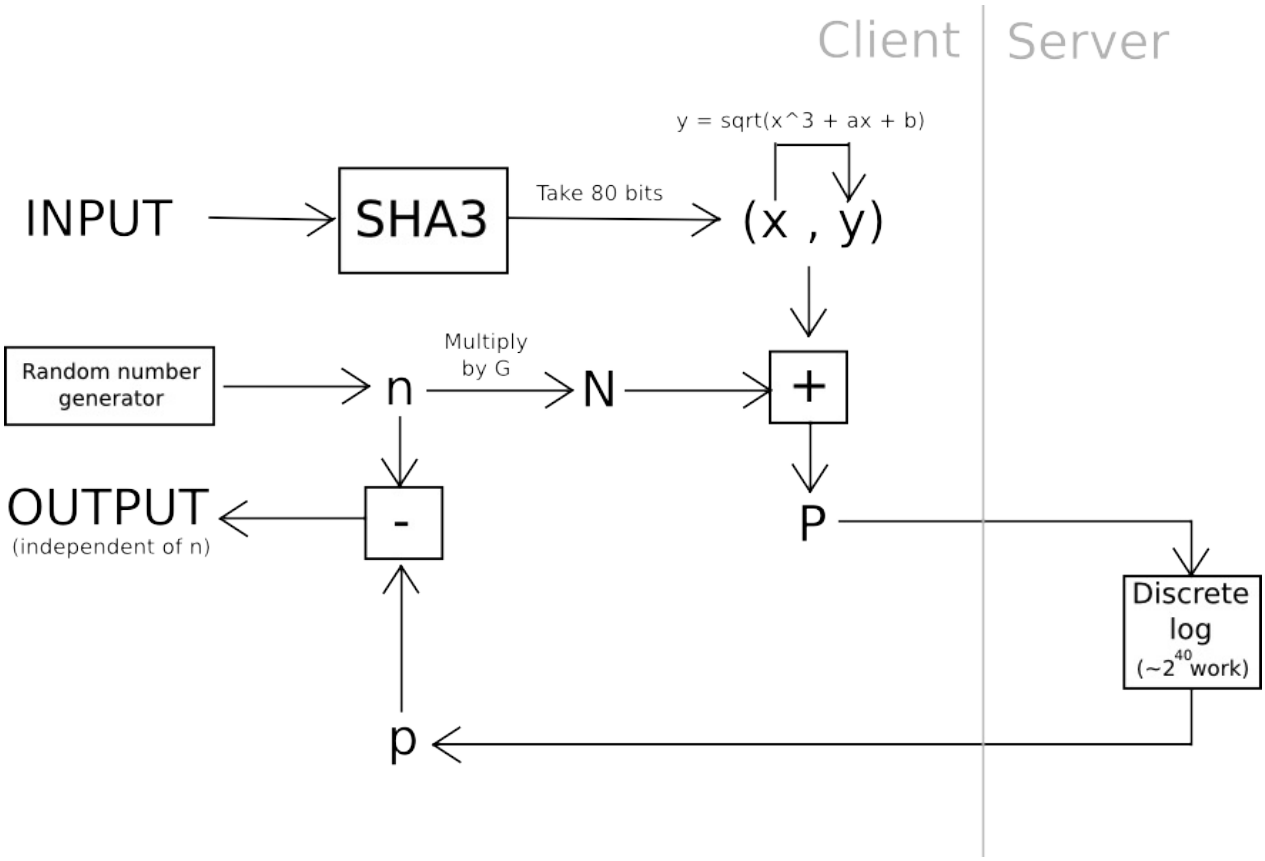
Hypercube



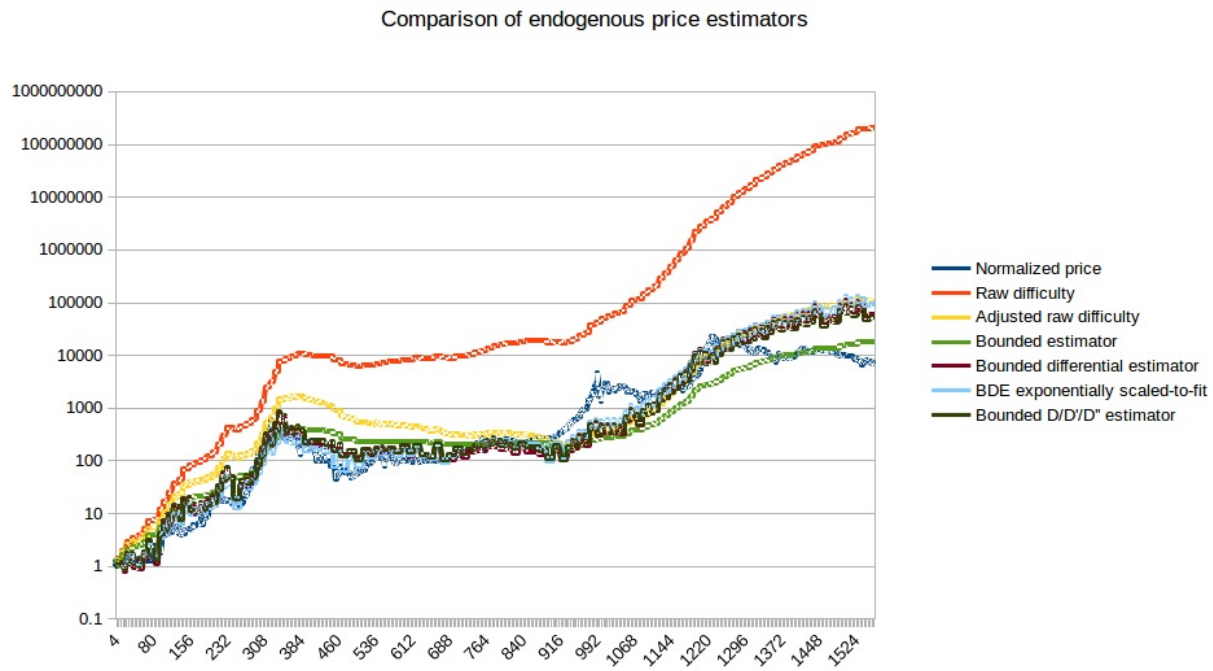
Currency

```
init:
    contract.storage[msg.sender] = 10^6
code:
    from = msg.sender
    to = msg.data[0]
    value = msg.data[1]
    if contract.storage[from] >= value:
        contract.storage[from] -= value
        contract.storage[to] += value
```

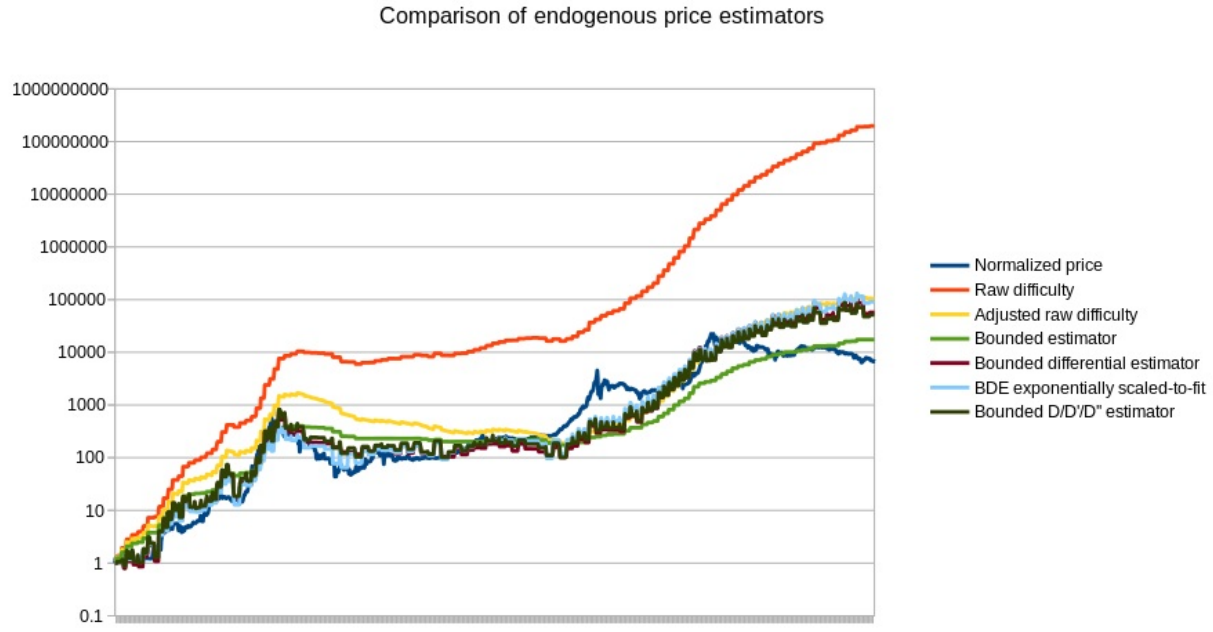
Ecthingy



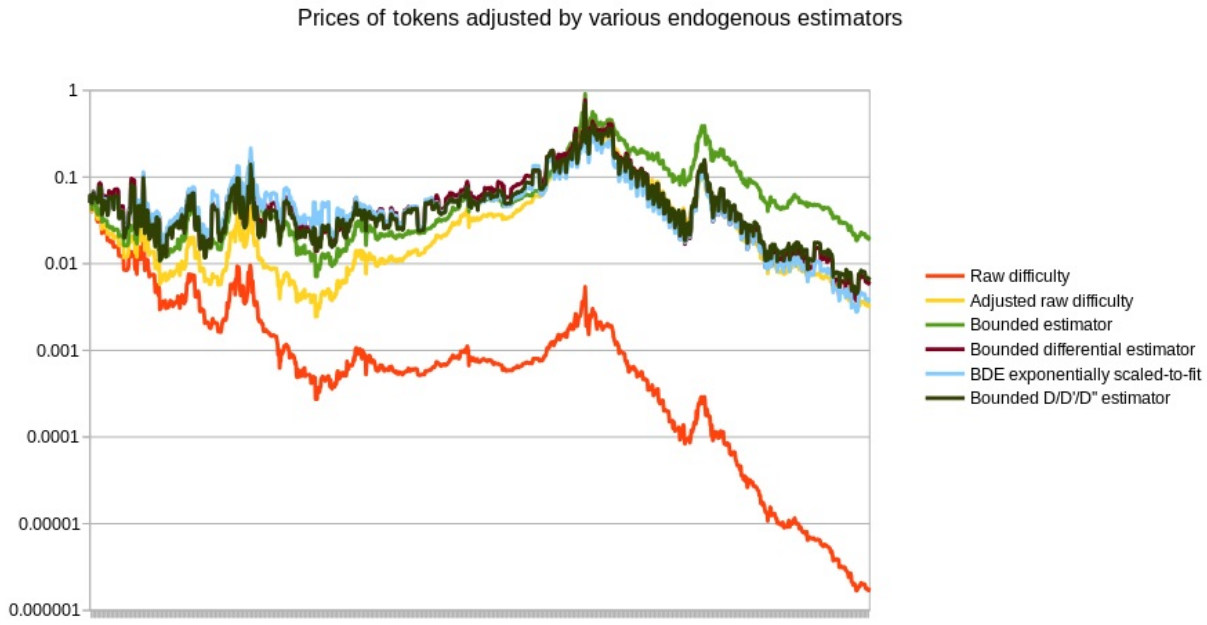
Endo 1



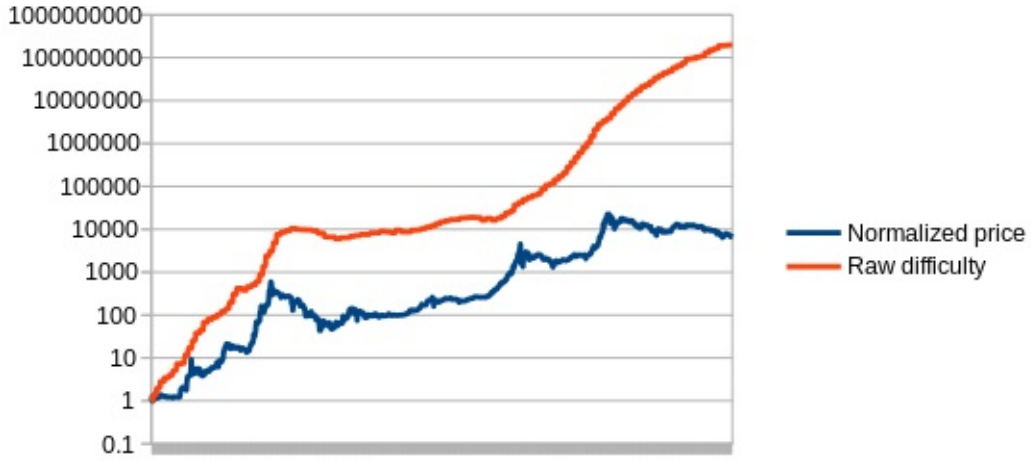
Endo 1a



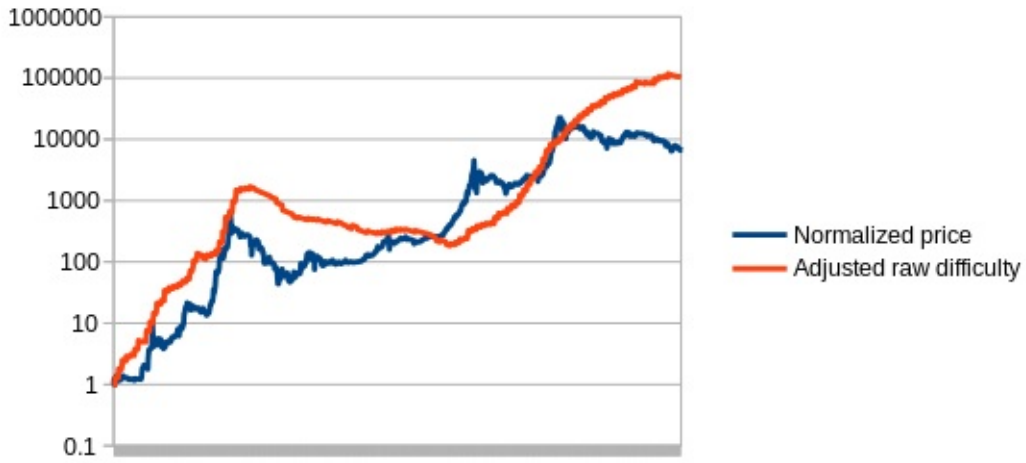
Endo 2


Prices of tokens adjusted by various endogenous estimators
1
0.1
0.01
0.001
0.0001
0.00001
0.000001
Raw difficulty
Adjusted raw difficulty
Bounded estimator
Bounded differential estimator
BDE exponentially scaled-to-fit
Bounded D/D'/D" estimator

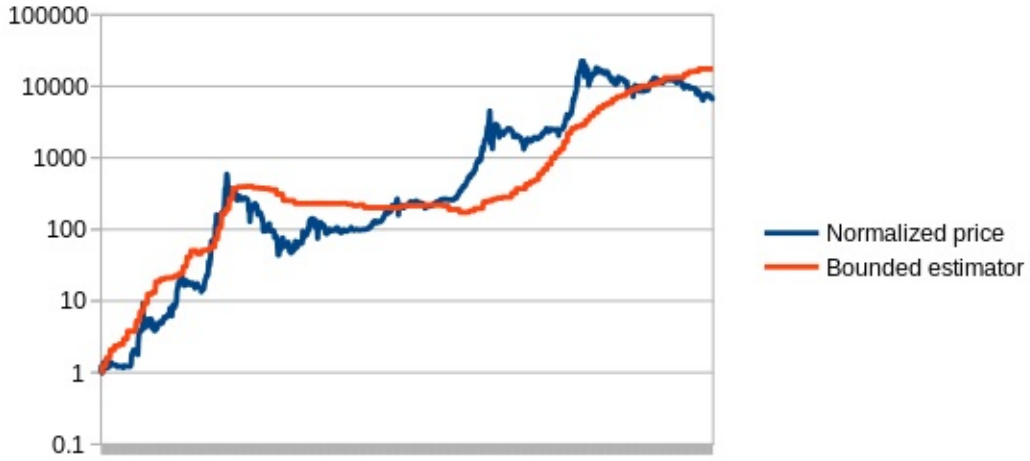
Endo 3


1000000000
100000000
10000000
1000000
100000
10000
1000
100
10
1
0.1
Normalized price
Raw difficulty

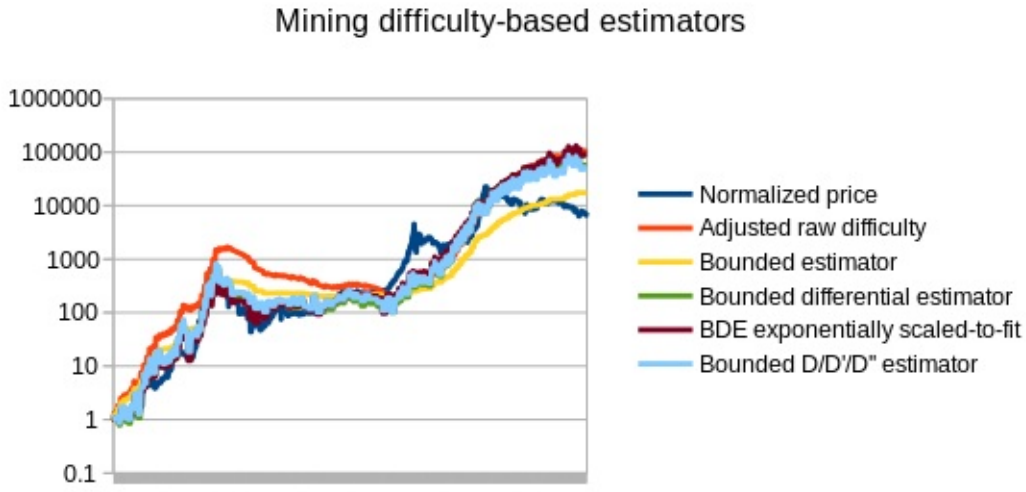
Endo 4



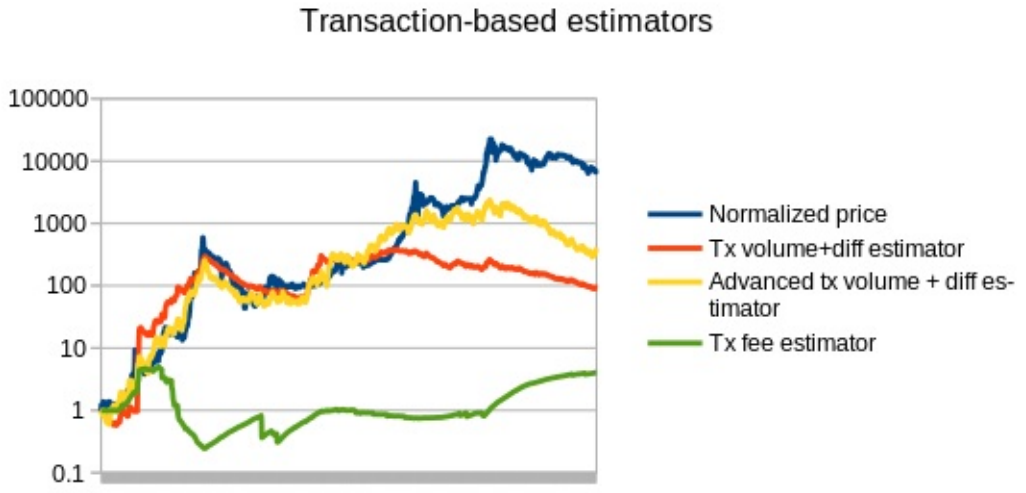
Endo 5



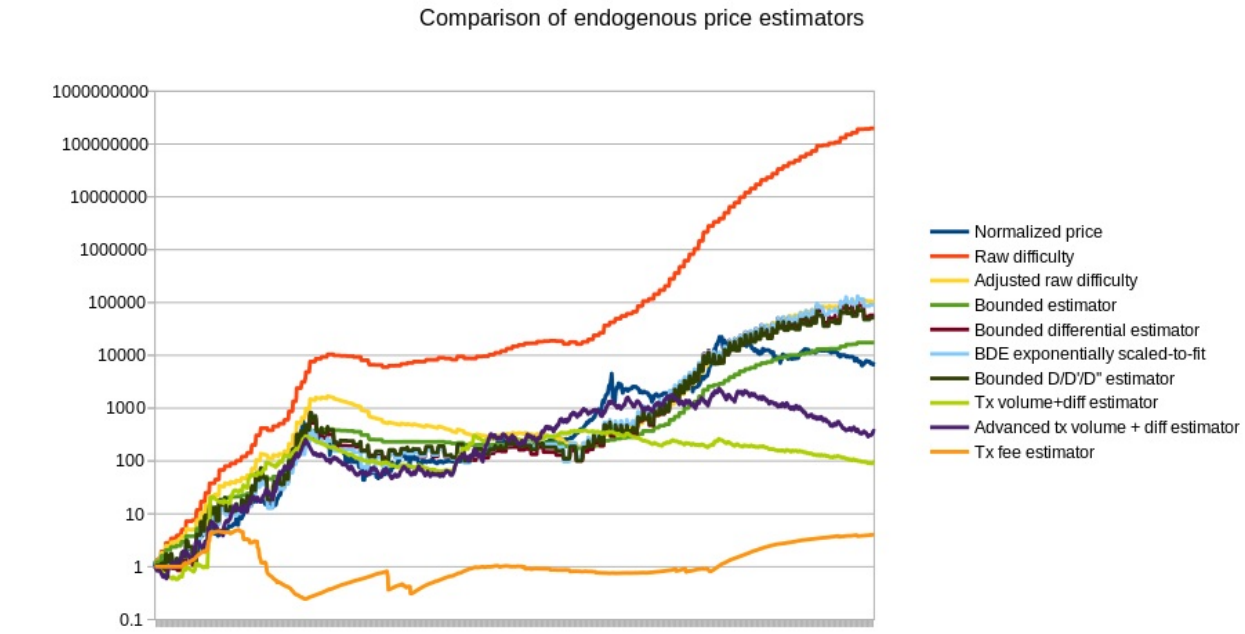
Endo 10



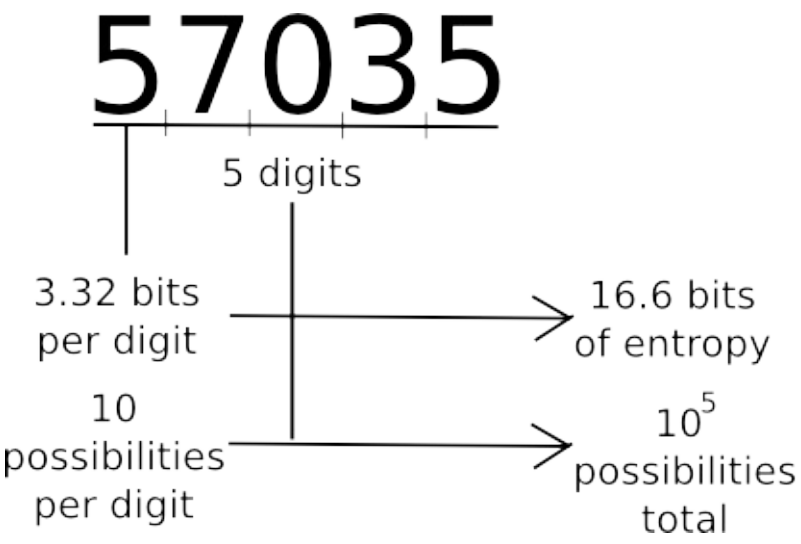
Endo 20



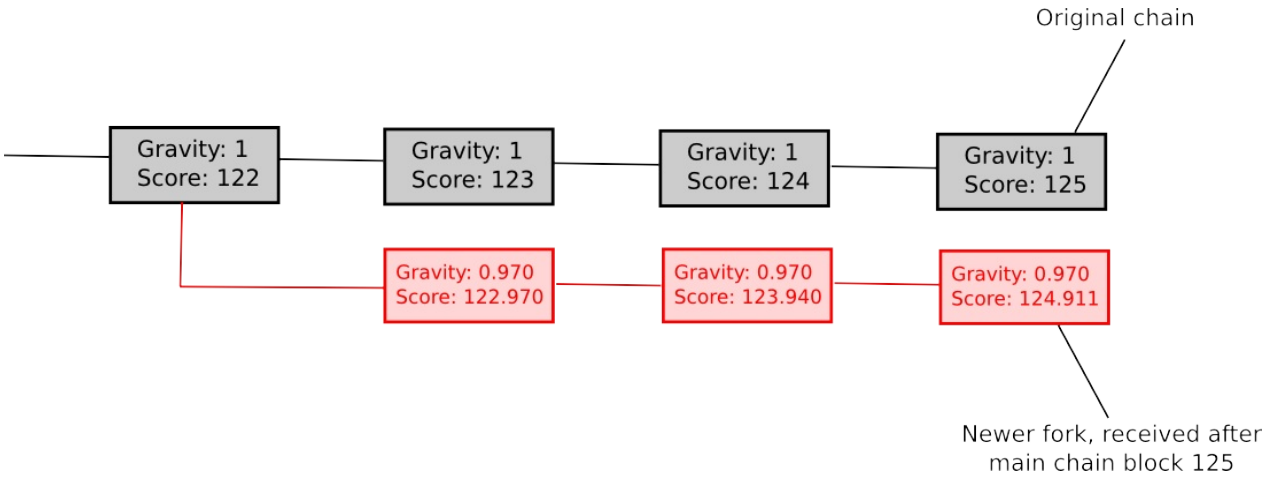
Endo 30



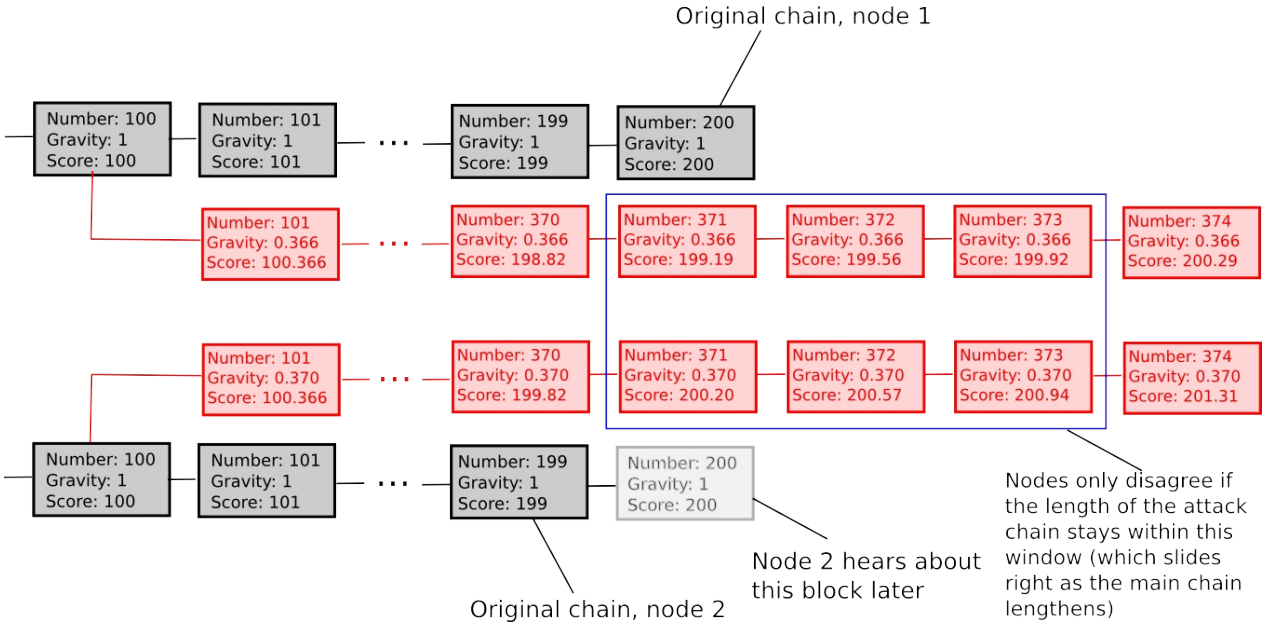
Entropy



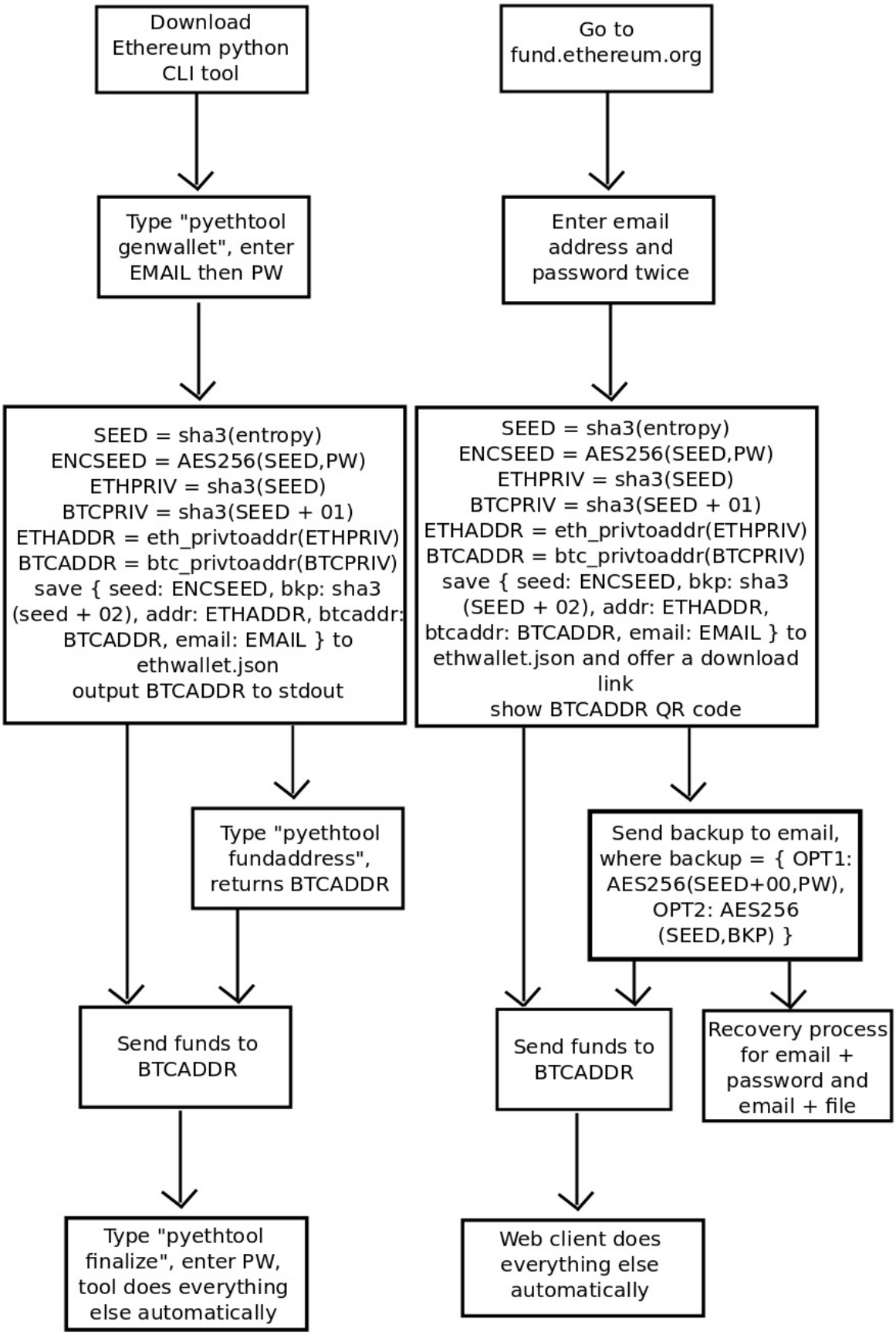
Ess 1



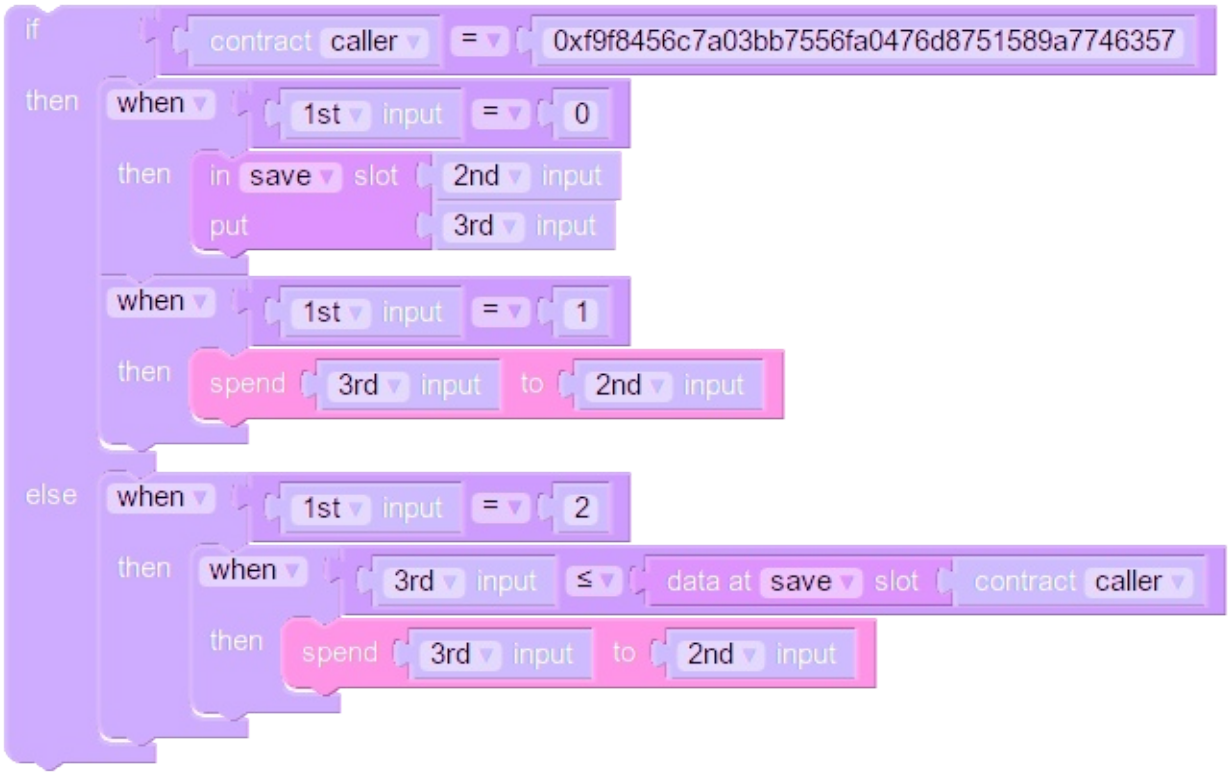
Ess 2



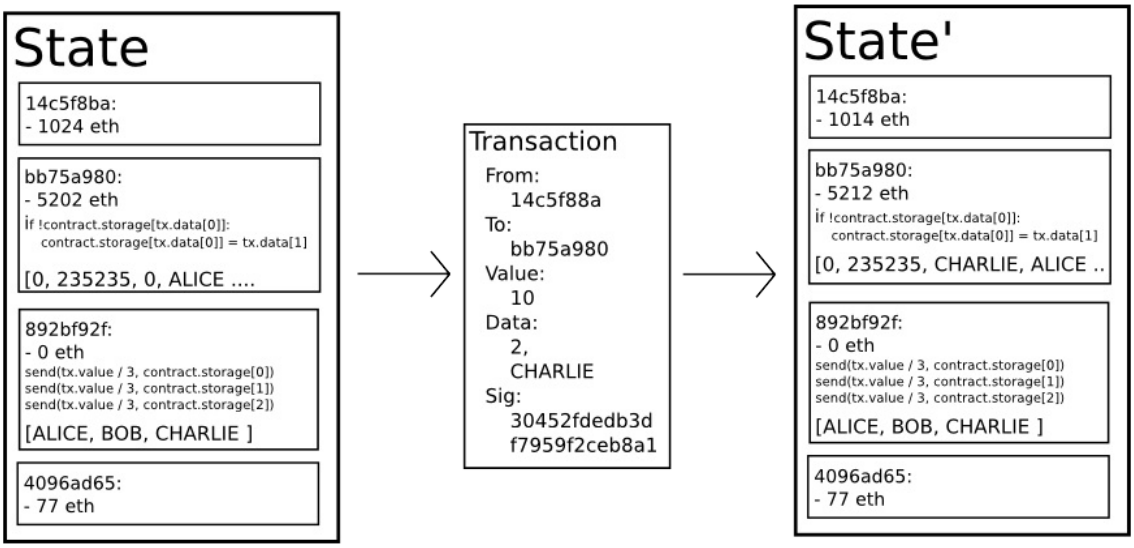
Ether sale process


Download Ethereum python CLI tool
Type "pyethtool genwallet", enter EMAIL then PW
SEED = sha3(entropy)
ENCSEED = AES256(SEED,PW)
ETHPRIV = sha3(SEED)
BTCPRIV = sha3(SEED + 01)
ETHADDR = eth_privtoaddr(ETHPRIV)
BTCADDR = btc_privtoaddr(BTCPRIV)
save { seed: ENCSEED, bkp: sha3 (seed + 02), addr: ETHADDR, btcaddr: BTCADDR, email: EMAIL } to ethwallet.json
output BTCADDR to stdout
Type "pyethtool fundaddress", returns BTCADDR
Send funds to BTCADDR
Type "pyethtool finalize", enter PW, tool does everything else automatically
Go to fund.ethereum.org
Enter email address and password twice
SEED = sha3(entropy)
ENCSEED = AES256(SEED,PW)
ETHPRIV = sha3(SEED)
BTCPRIV = sha3(SEED + 01)
ETHADDR = eth_privtoaddr(ETHPRIV)
BTCADDR = btc_privtoaddr(BTCPRIV)
save { seed: ENCSEED, bkp: sha3 (SEED + 02), addr: ETHADDR, btcaddr: BTCADDR, email: EMAIL } to ethwallet.json and offer a download link
show BTCADDR QR code
Send backup to email, where backup = { OPT1: AES256(SEED+00,PW), OPT2: AES256 (SEED,BKP) }
Send funds to BTCADDR
Recovery process for email + password and email + file
Web client does everything else automatically

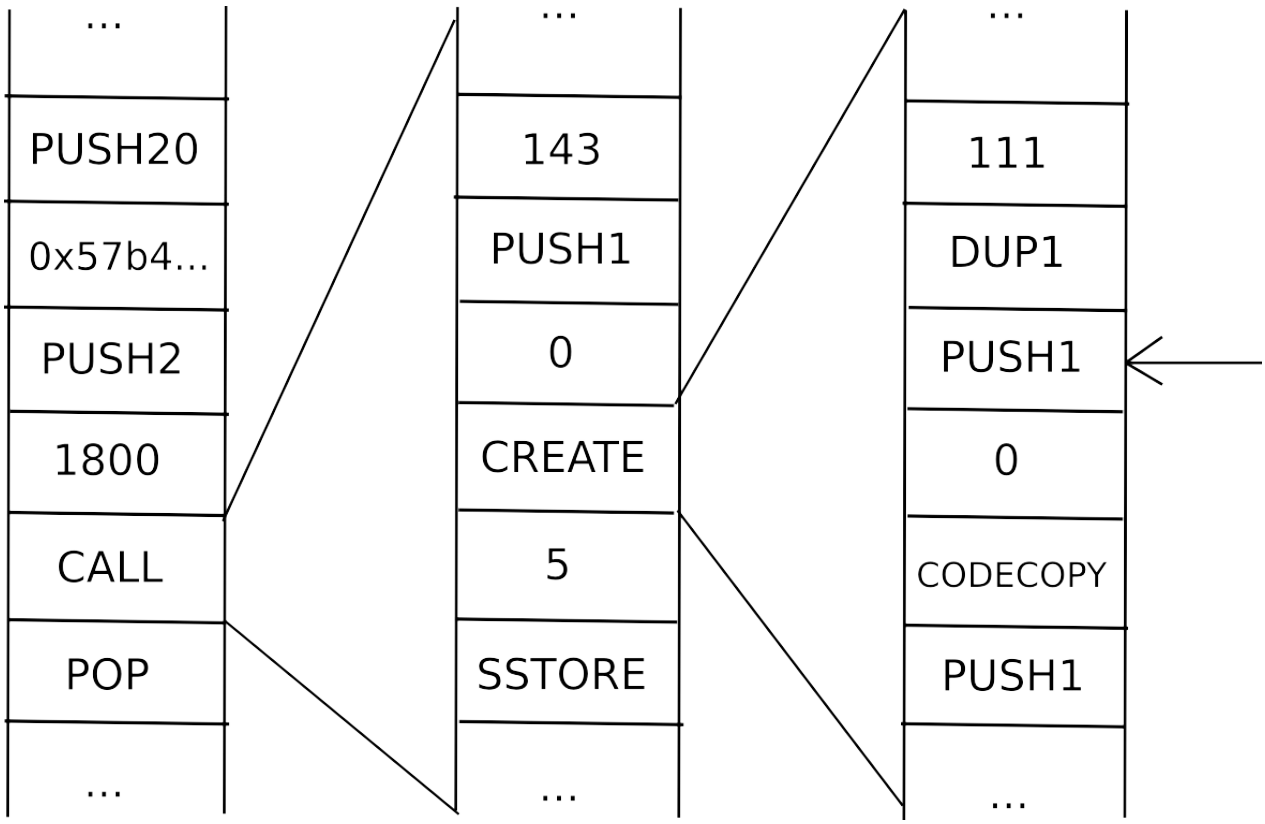
Etherscripter



Ether transition



Execution path



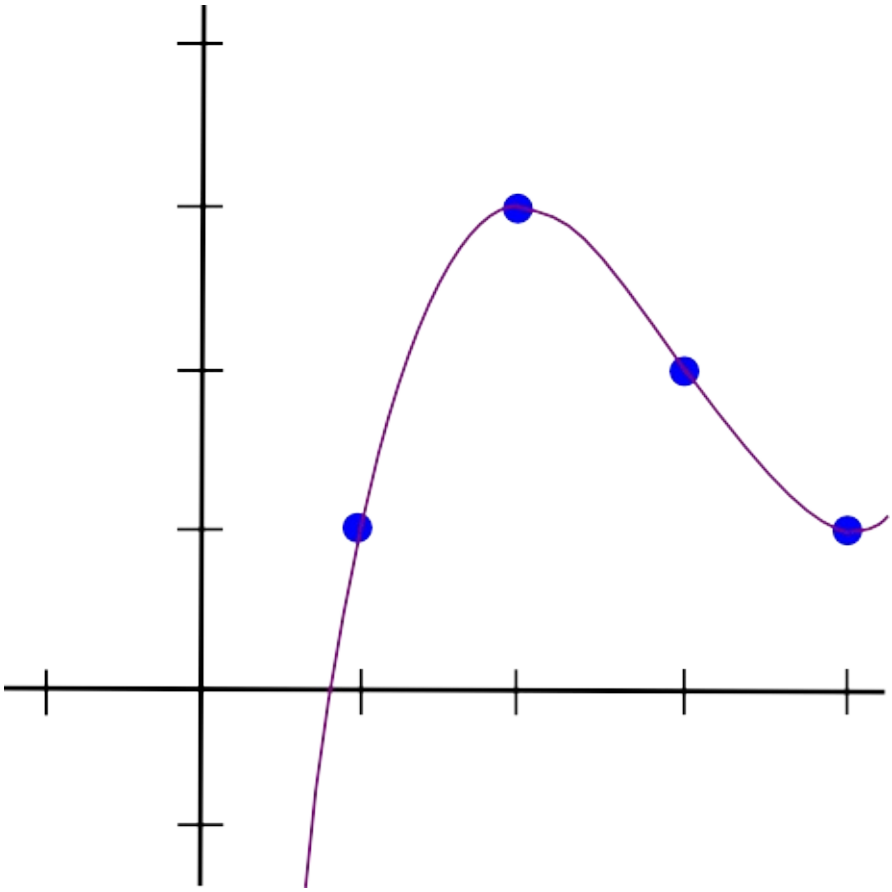
Failure rate

$$FR(m, n) \sim = \binom{m}{n} * FR(base)^{m}$$

Fault tolerance 1

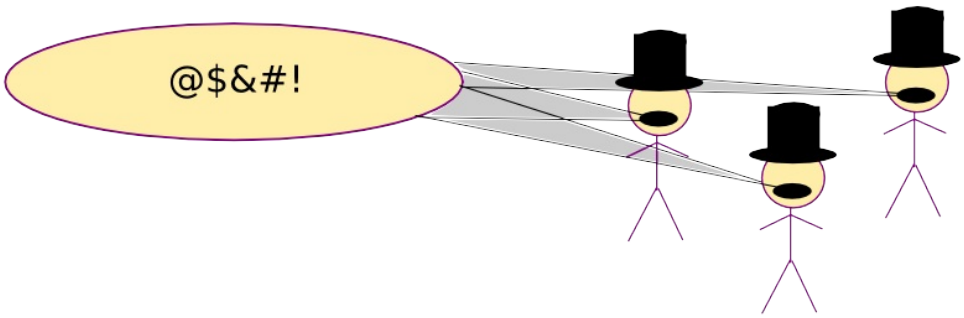
Fault tolerance 2

Four points

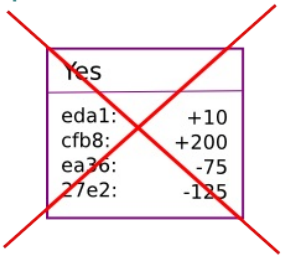


Futarchy bottom

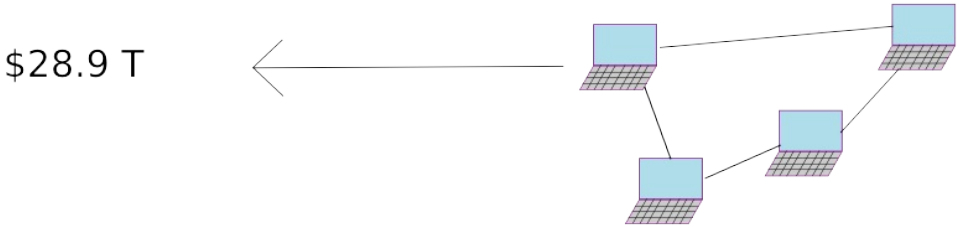
Step 3: close both markets, implement the policy with the higher price



Step 4: revert all trades on losing market



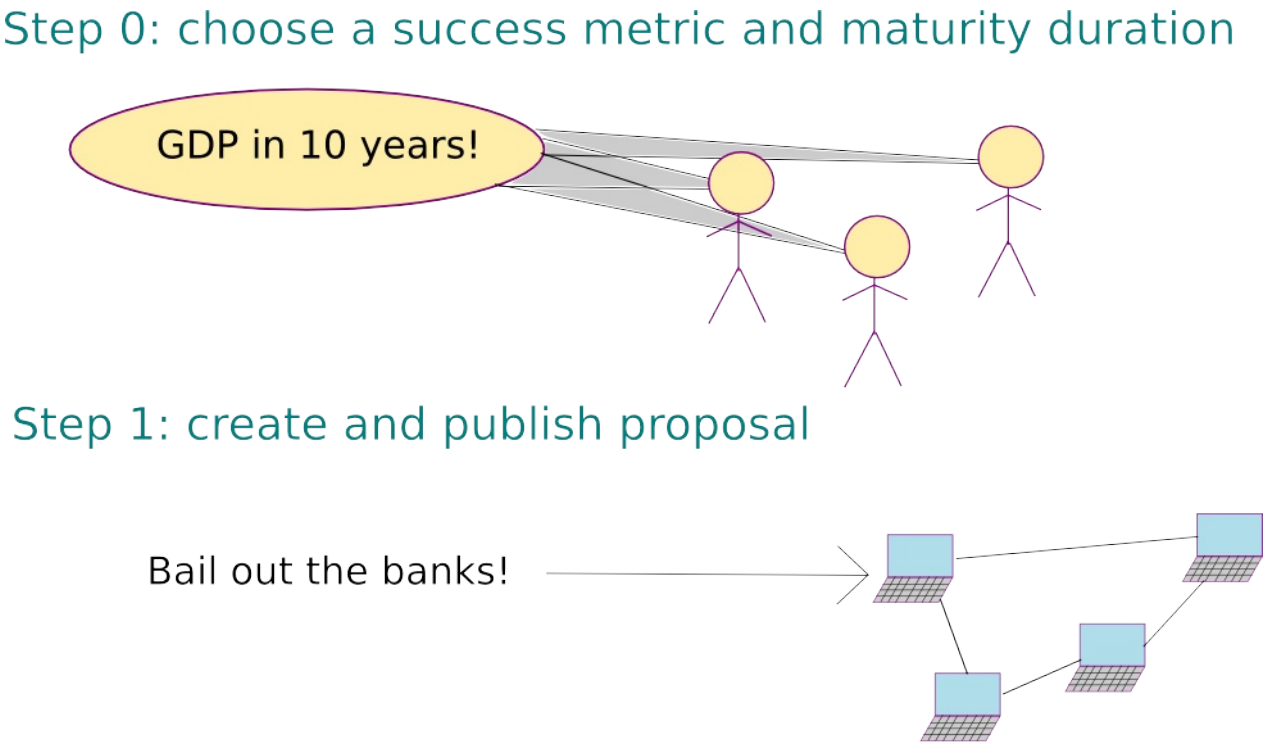
Step 5: wait for maturity, and measure success metric



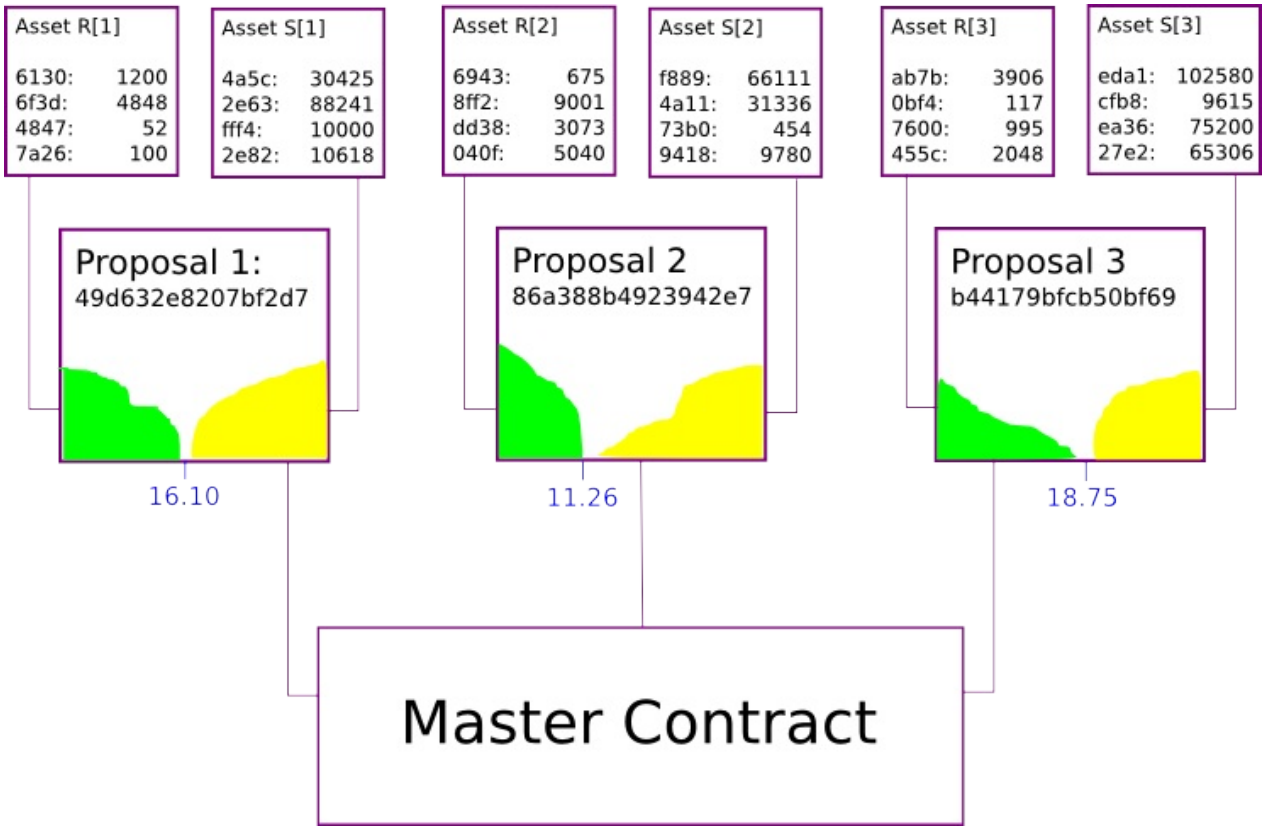
Step 6: reward everyone on the winning market

No		
f889:	+50	+ $1450
4a11:	-500	- $14500
73b0:	+200	+ $5780
9418:	+250	- $7250

Futarchy top

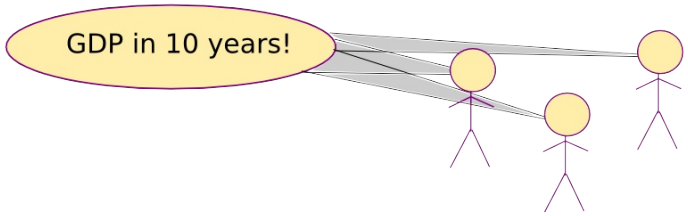

Step 0: choose a success metric and maturity duration
GDP in 10 years!
Step 1: create and publish proposal
Bail out the banks!
Step 2: set up prediction markets for "yes" and "no"
Yes
Bids
Asks
24.94
No
Bids
Asks
26.20
Note the average price of both over some period

Futarchy


Asset R[1]
6130: 1200
6f3d: 4848
4847: 52
7a26: 100
Asset S[1]
4a5c: 30425
2e63: 88241
fff4: 10000
2e82: 10618
Asset R[2]
6943: 675
8ff2: 9001
dd38: 3073
040f: 5040
Asset S[2]
f889: 66111
4a11: 31336
73b0: 454
9418: 9780
Asset R[3]
ab7b: 3906
0bf4: 117
7600: 995
455c: 2048
Asset S[3]
eda1: 102580
cfb8: 9615
ea36: 75200
27e2: 65306
Proposal 1:
49d632e8207bf2d7
Proposal 2
86a388b4923942e7
Proposal 3
b44179bfcb50bf69
16.10
11.26
18.75
Master Contract

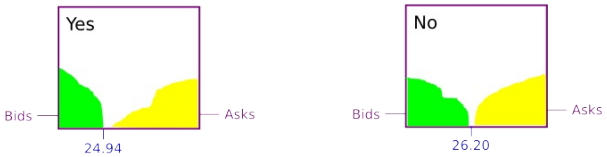
Futarchy 0

Step 0: choose a success metric and maturity duration



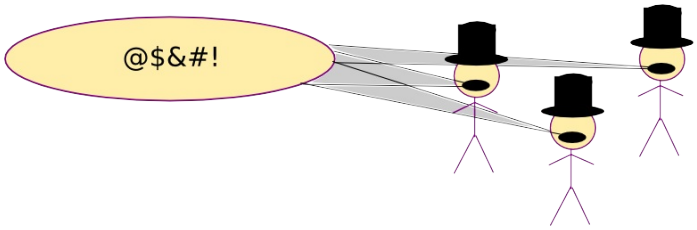
Step 1: create and publish proposal

Step 2: set up prediction markets for "yes" and "no"

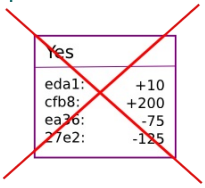


Note the average price of both over some period

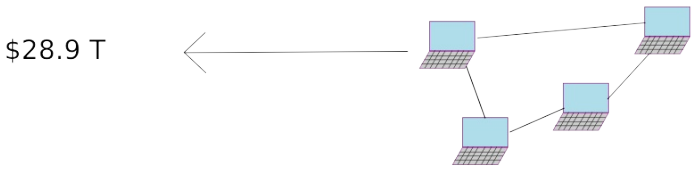
Step 3: close both markets, implement the policy with the higher price



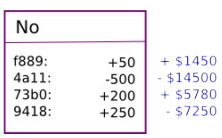
Step 4: revert all trades on losing market



Step 5: wait for maturity, and measure success metric



Step 6: reward everyone on the winning market



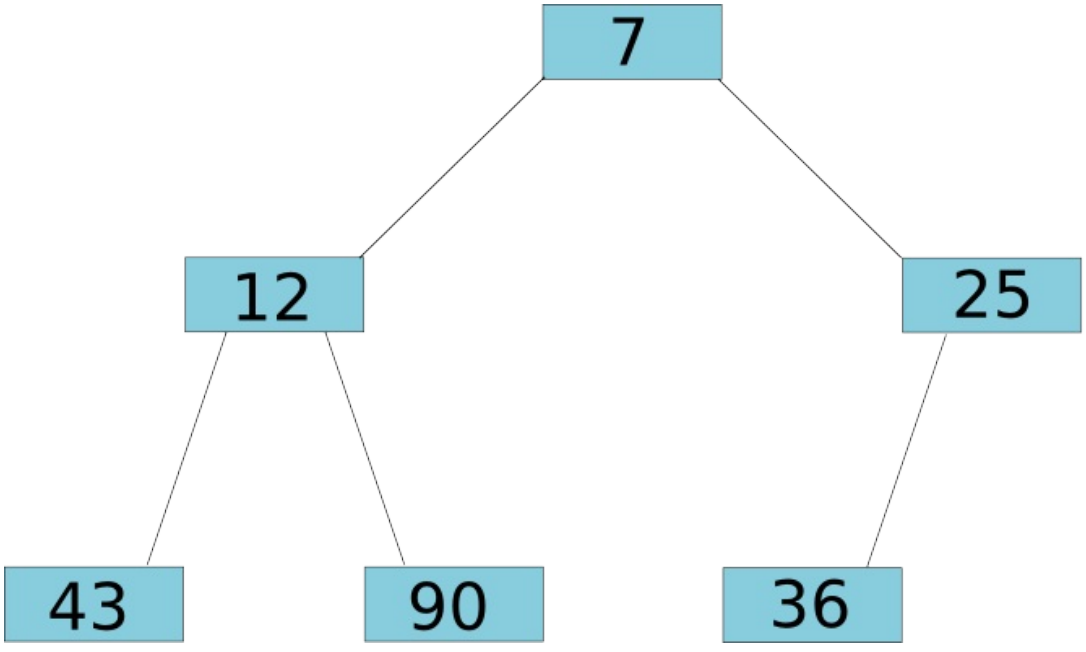
Grand equation

$$\text{Cost of maintenance in-house} + \text{Cost of electricity in-house} + \text{Cost of storage in-house} \stackrel{?}{>} \text{Cost of storage externally} + \text{Cost of electricity externally} + \text{Cost of maintenance externally} + \text{Shipping cost} + \text{Mining revenue} \times \left(1 - \frac{\text{Shipping time}}{\text{Hashpower doubling time}}\right)^{\frac{1}{2}} - \text{Hobbyist/ ideology premium}$$

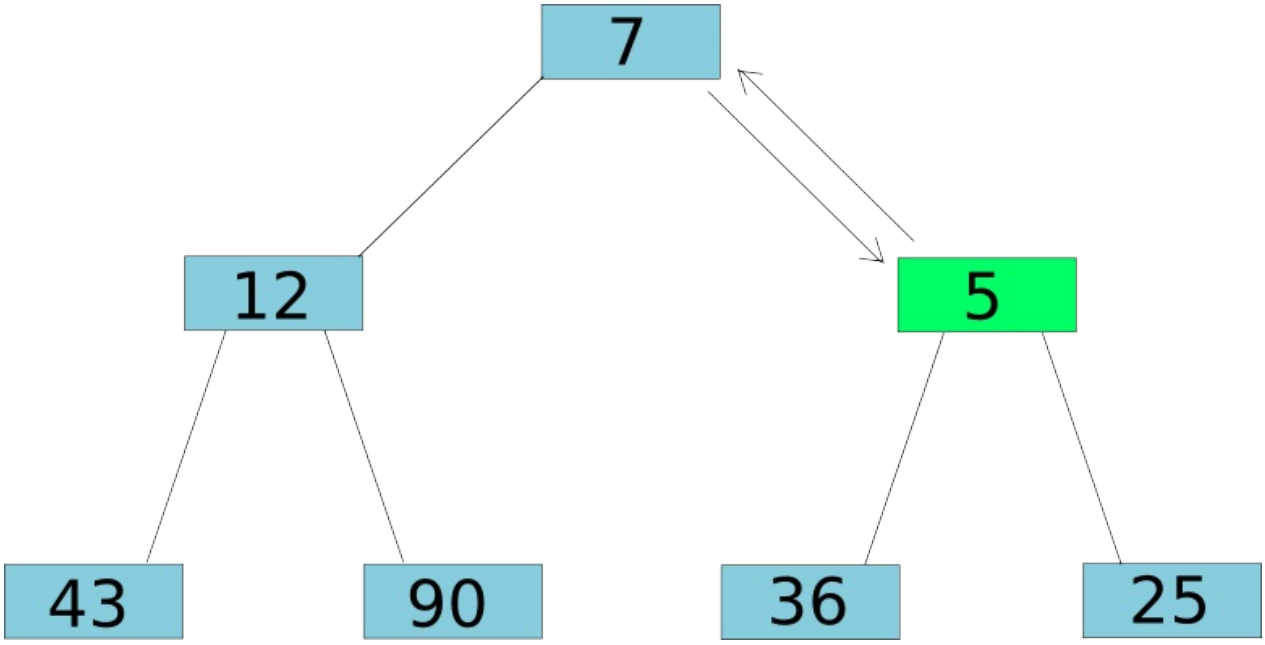
Grand equation 2

$$\frac{E + H}{E} \stackrel{?}{>} A$$

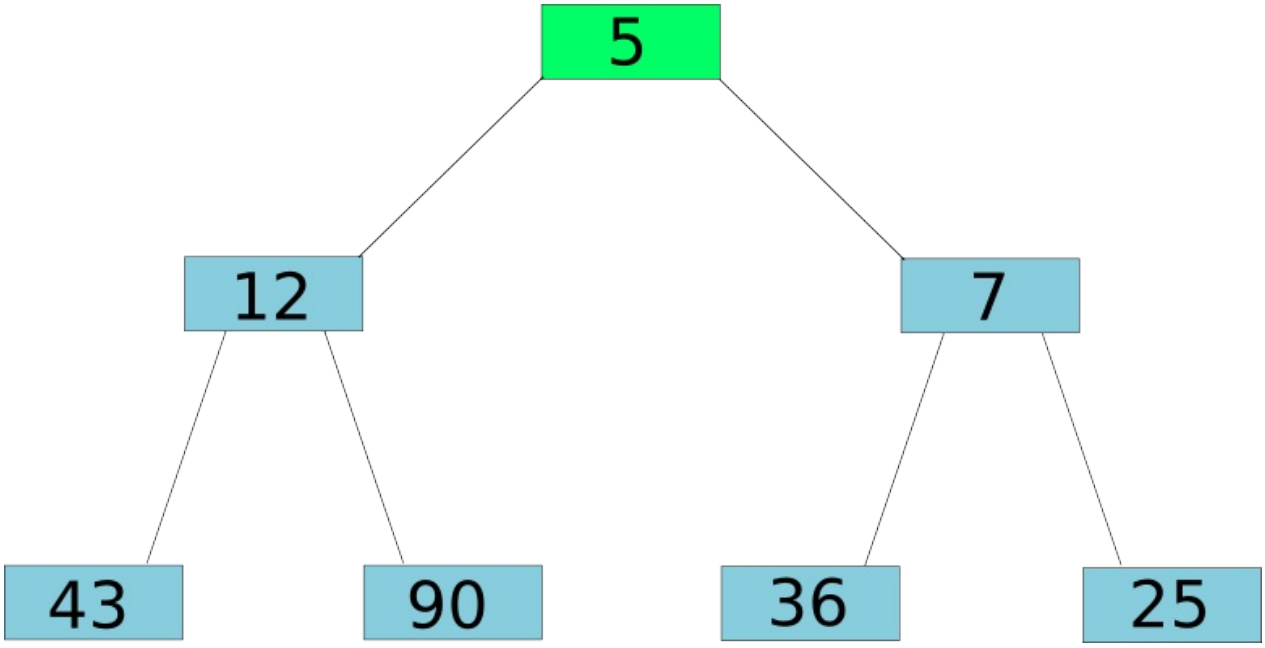
Heap 1



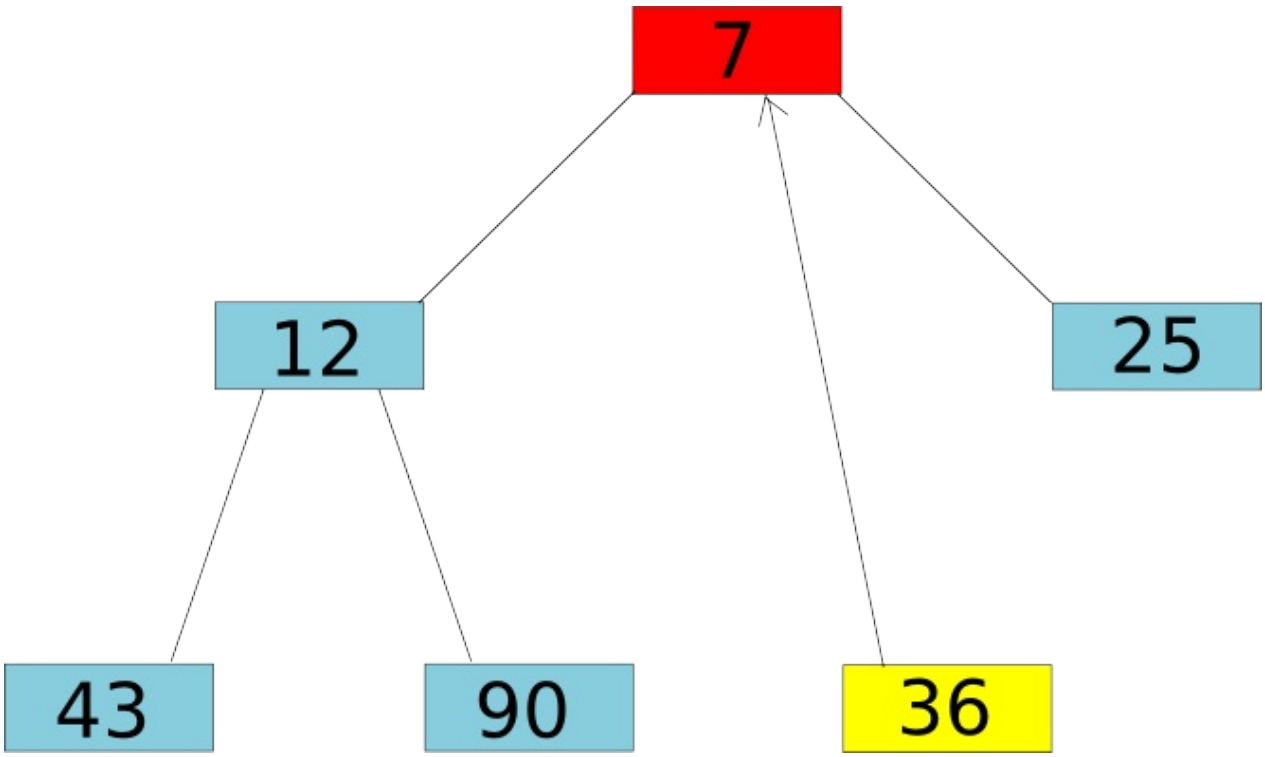
Heap 2



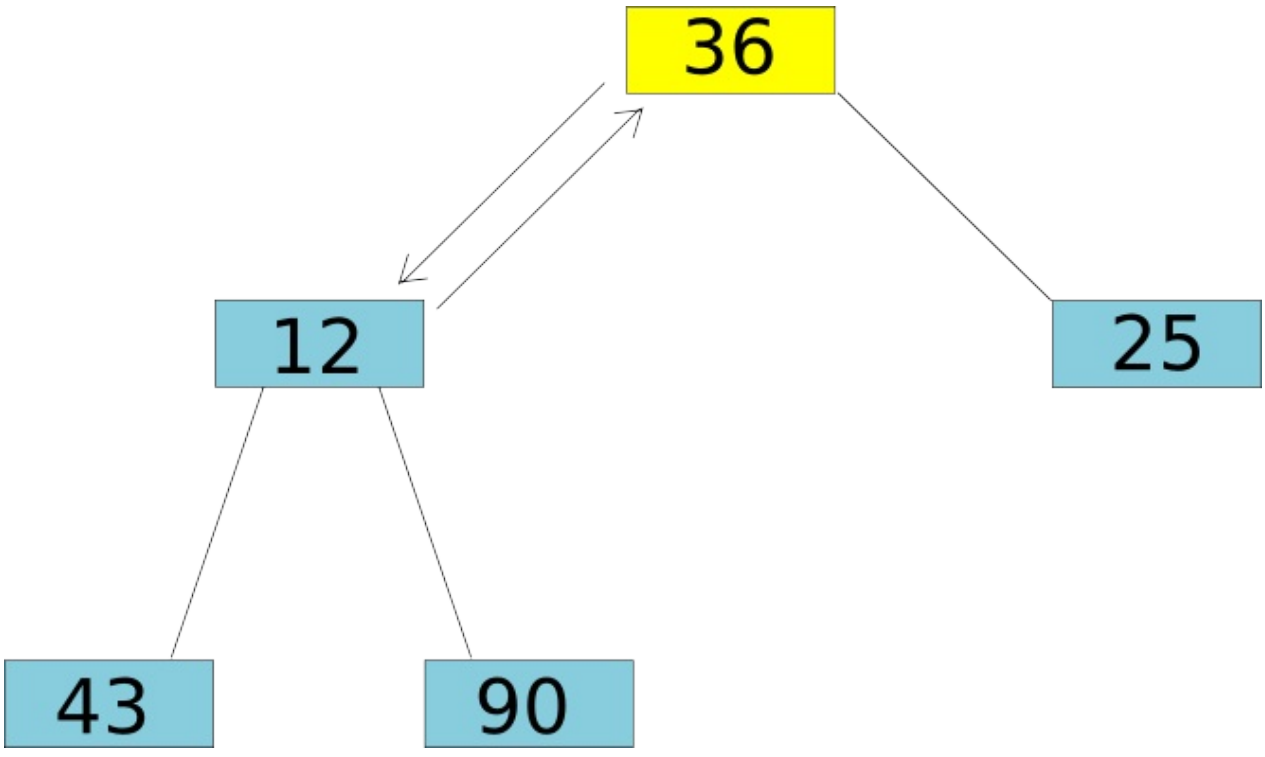
Heap 3



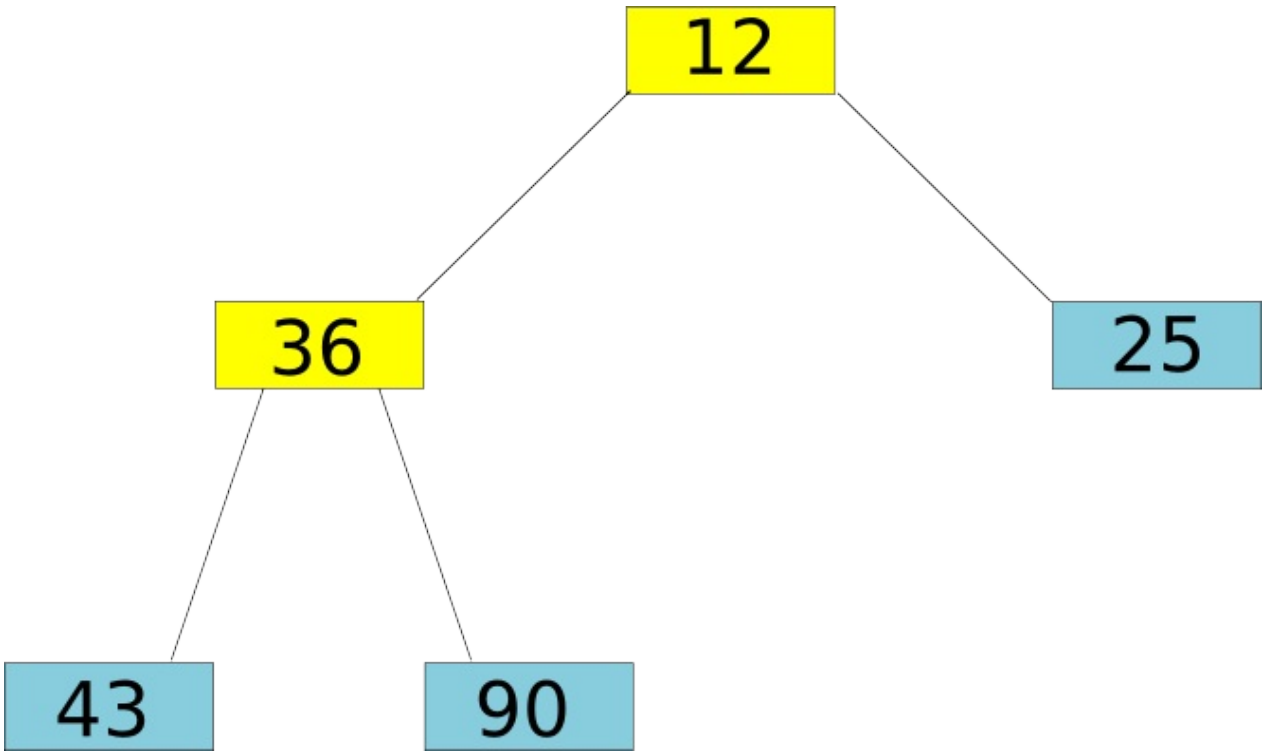
Heap 4



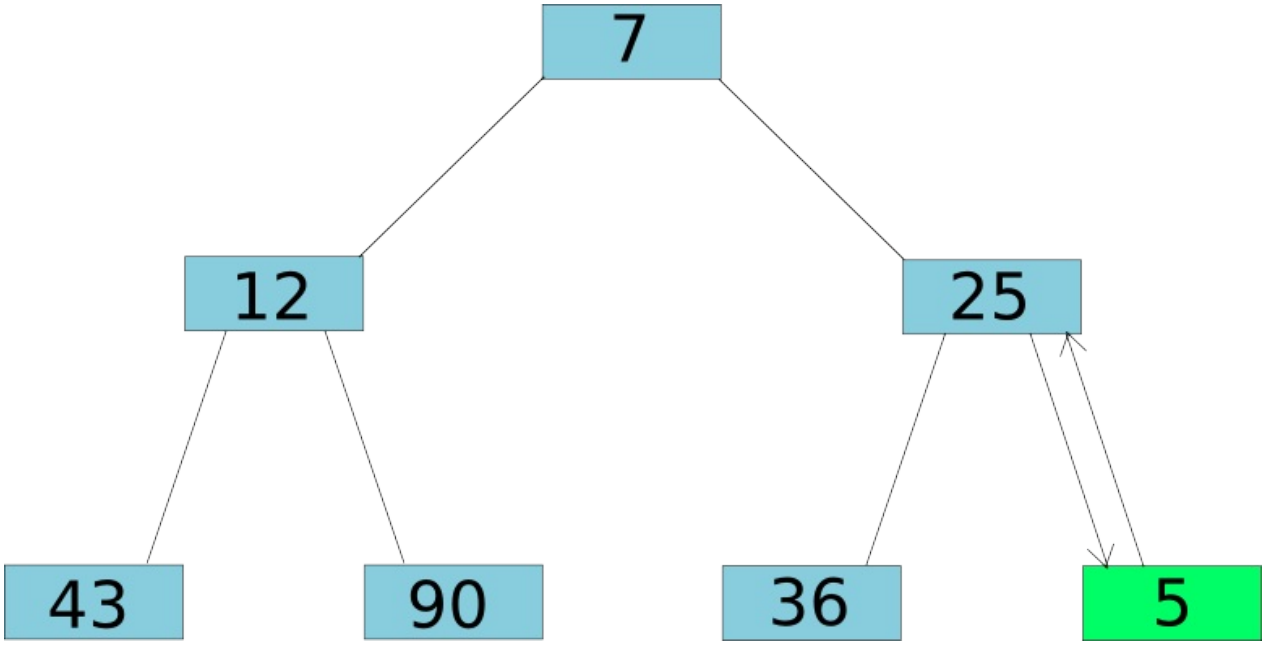
Heap 5



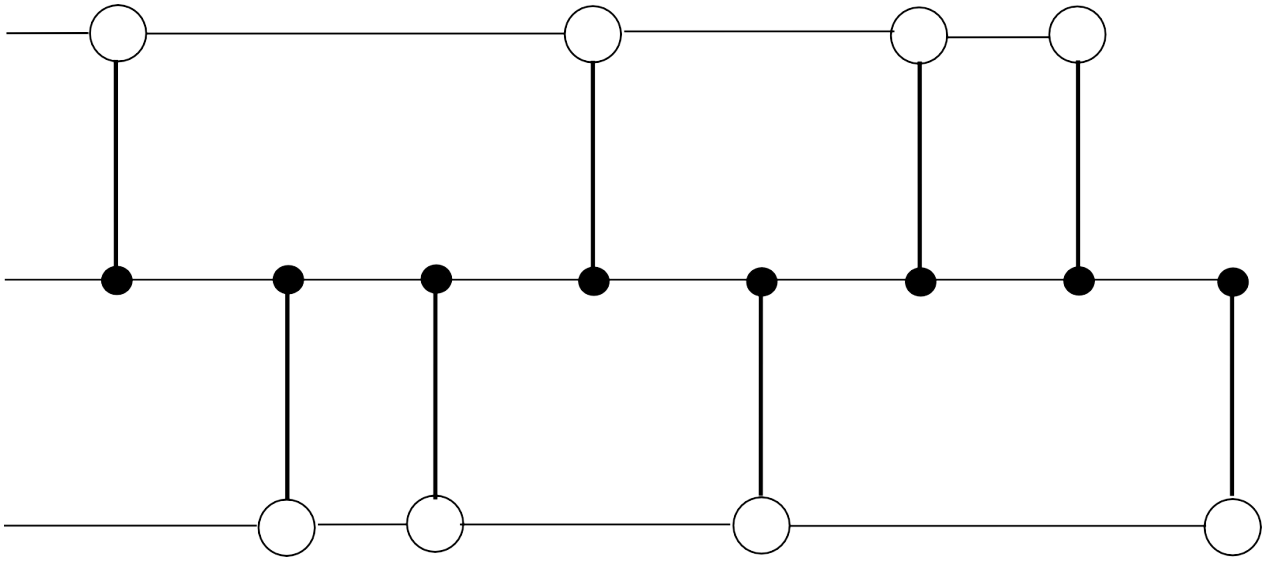
Heap 6



Heap 20



Hubchain



Hubspoke

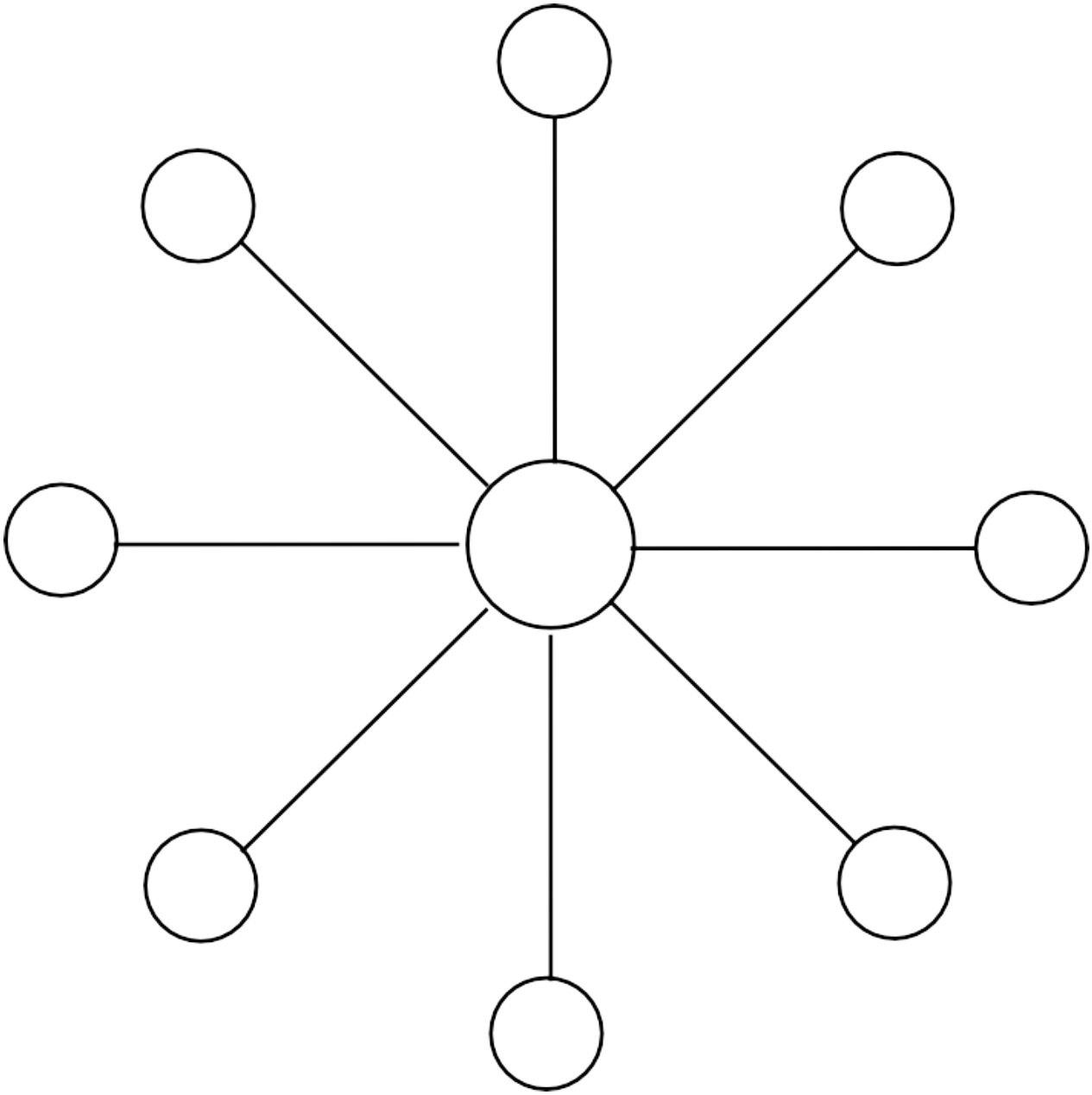
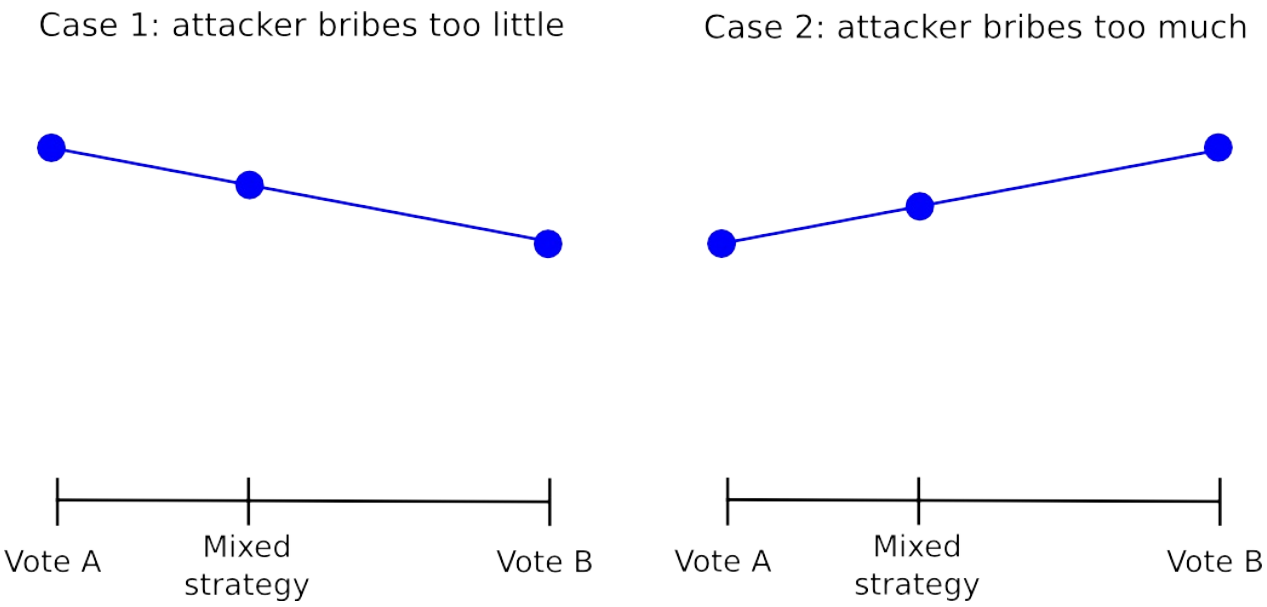
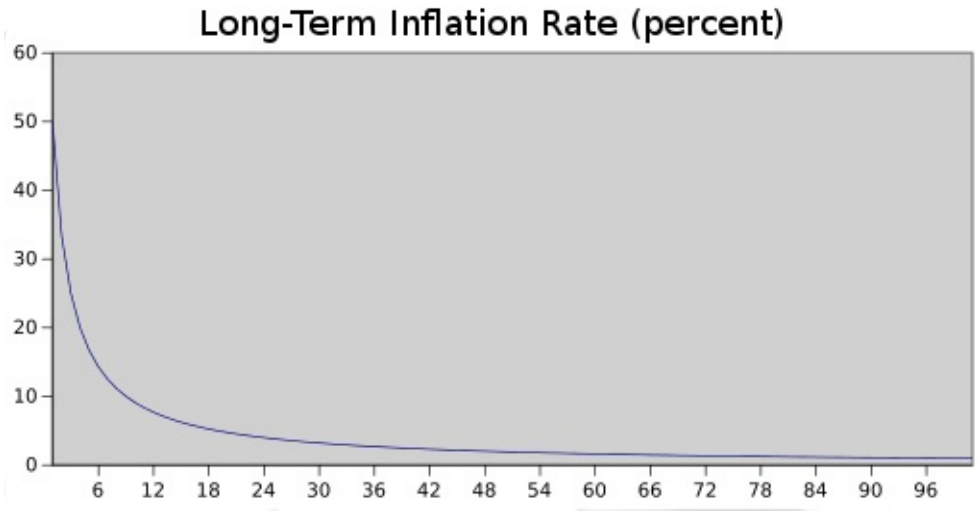


Image 2

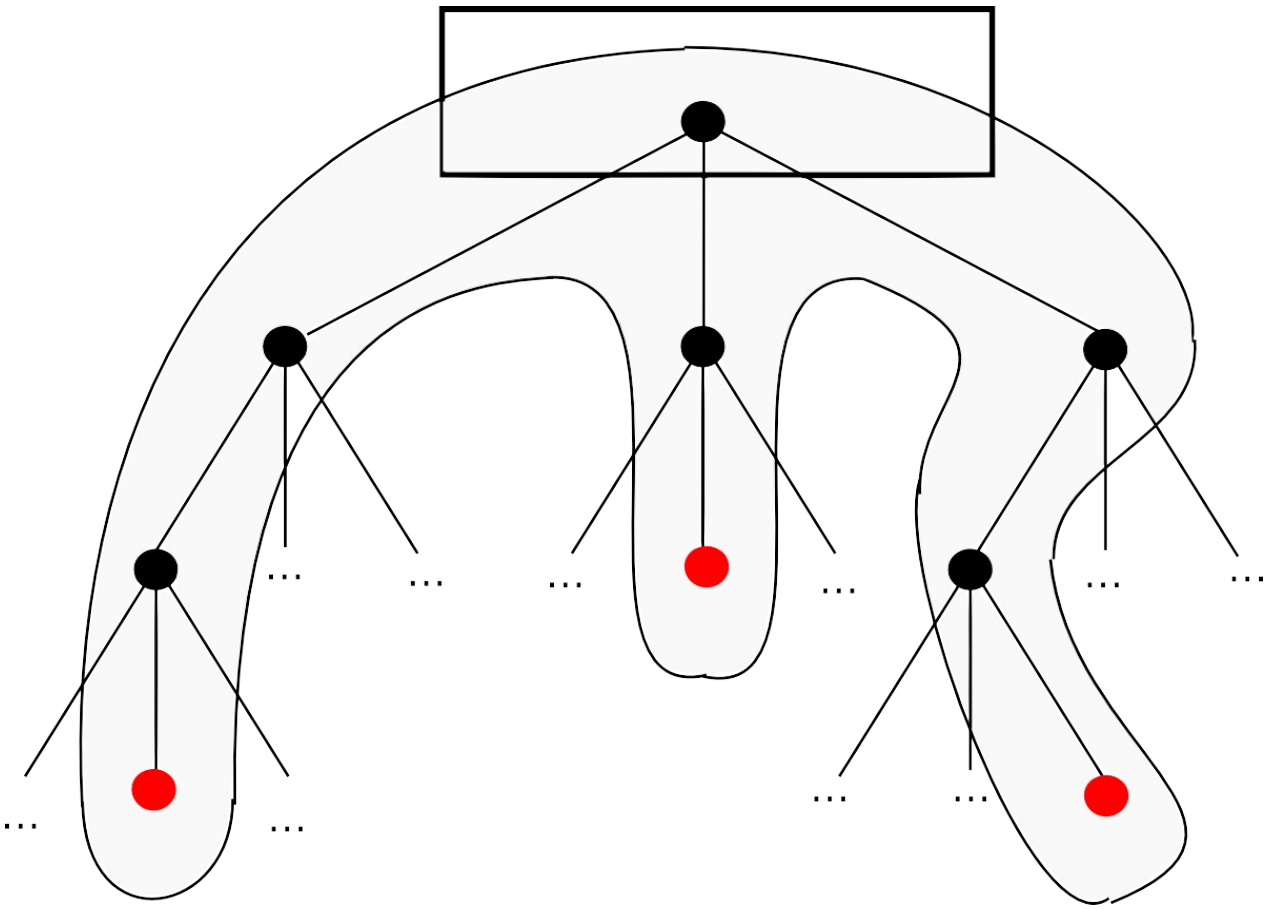
Incentives



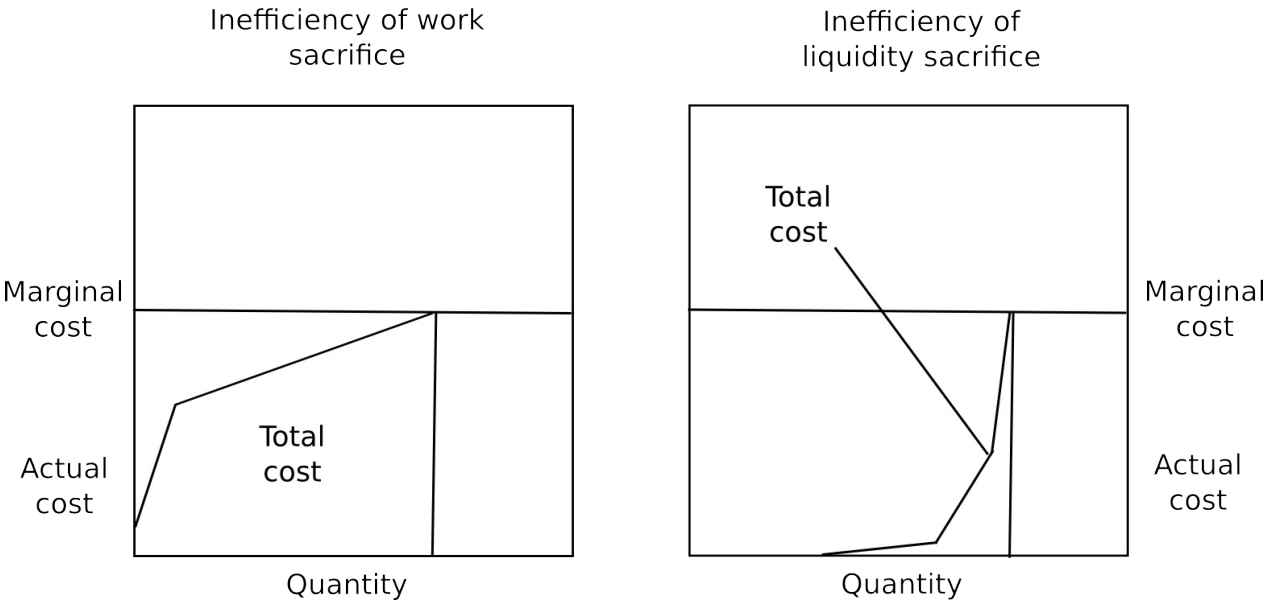
Inflation



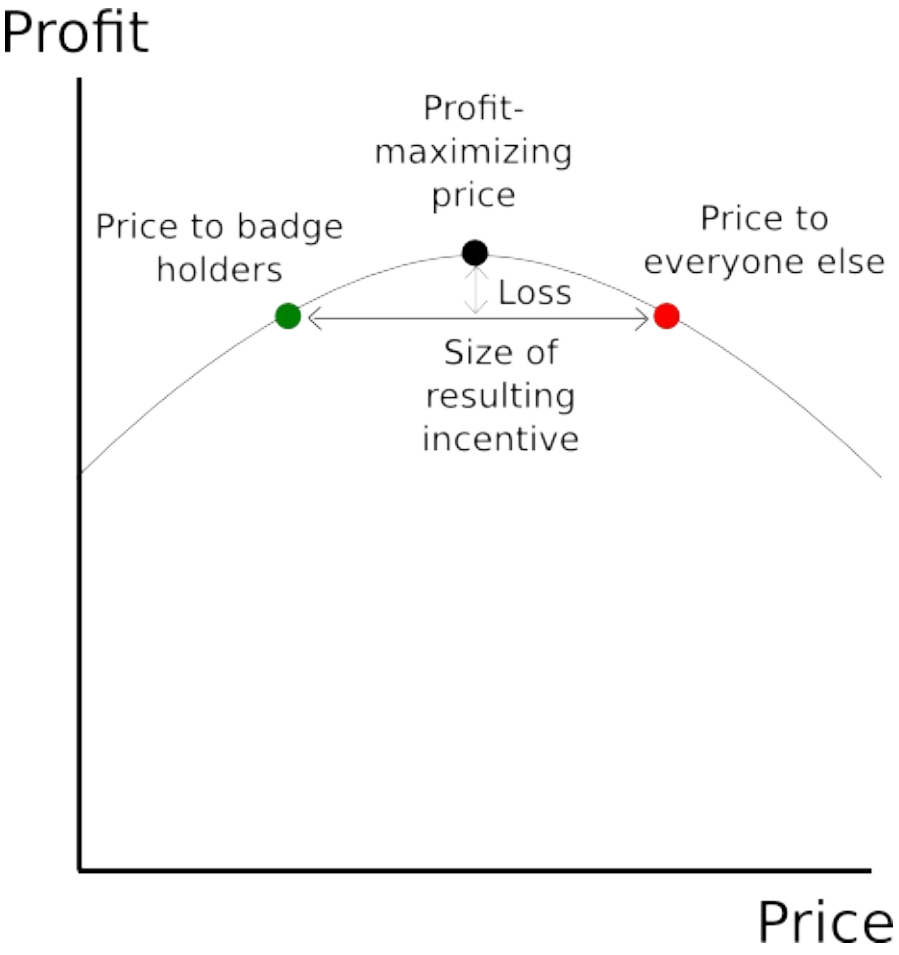
Lightproof



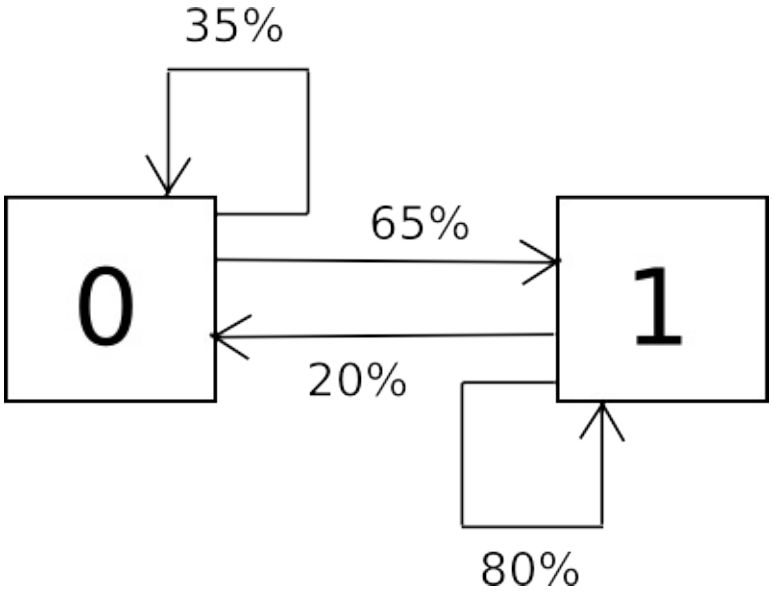
Liquidity



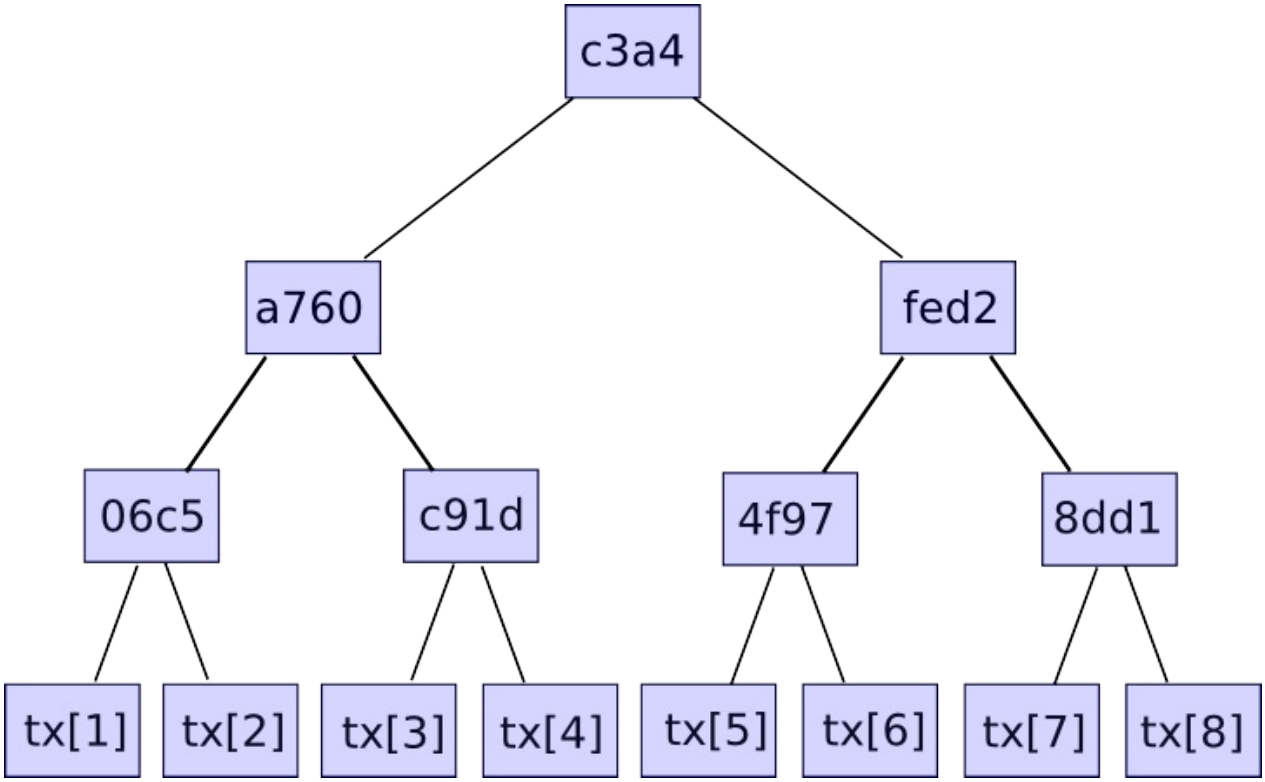
Marginal sanctions



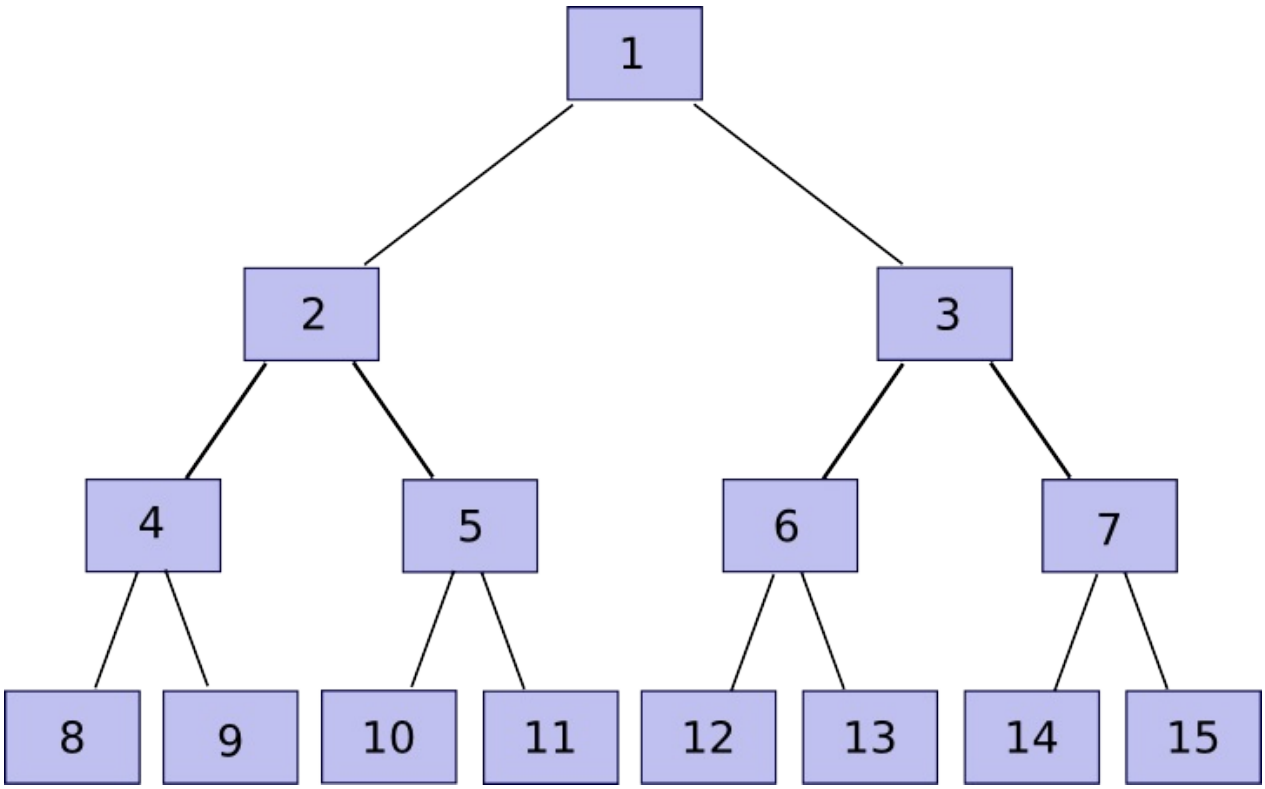
Markov



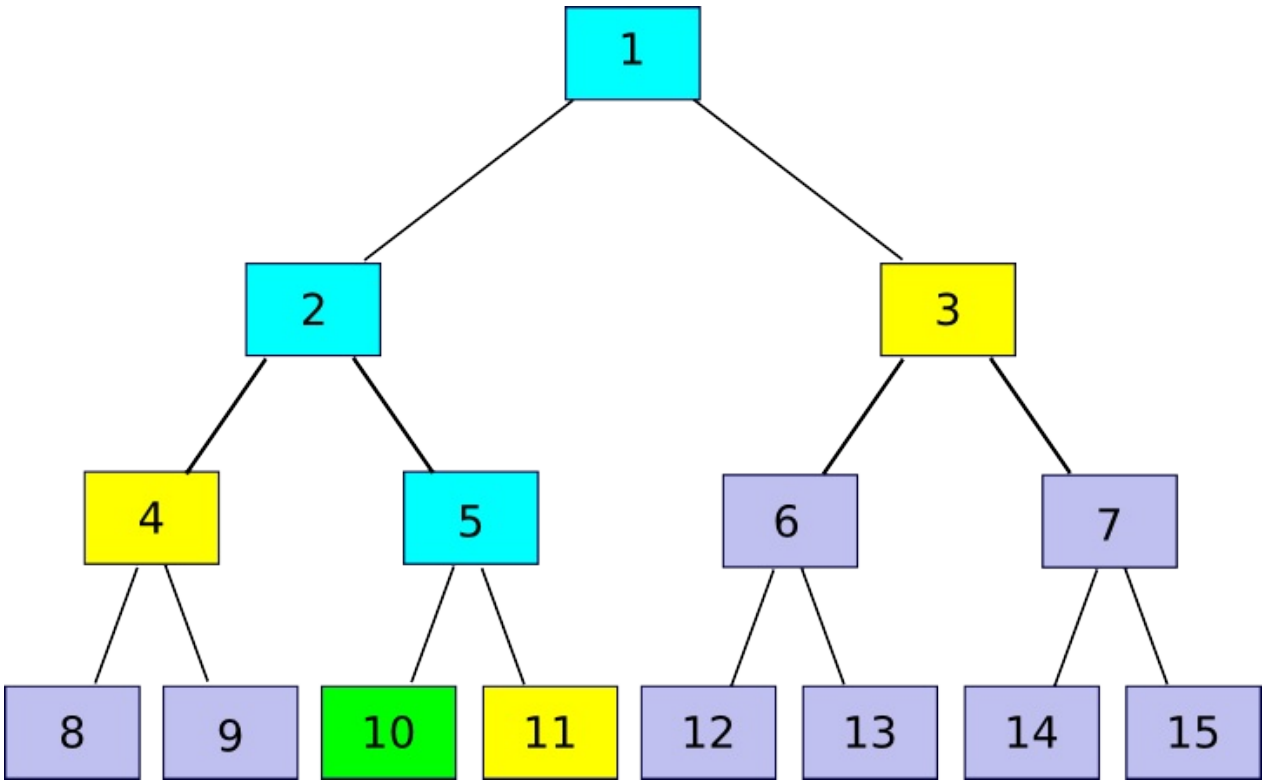
Merkle basic



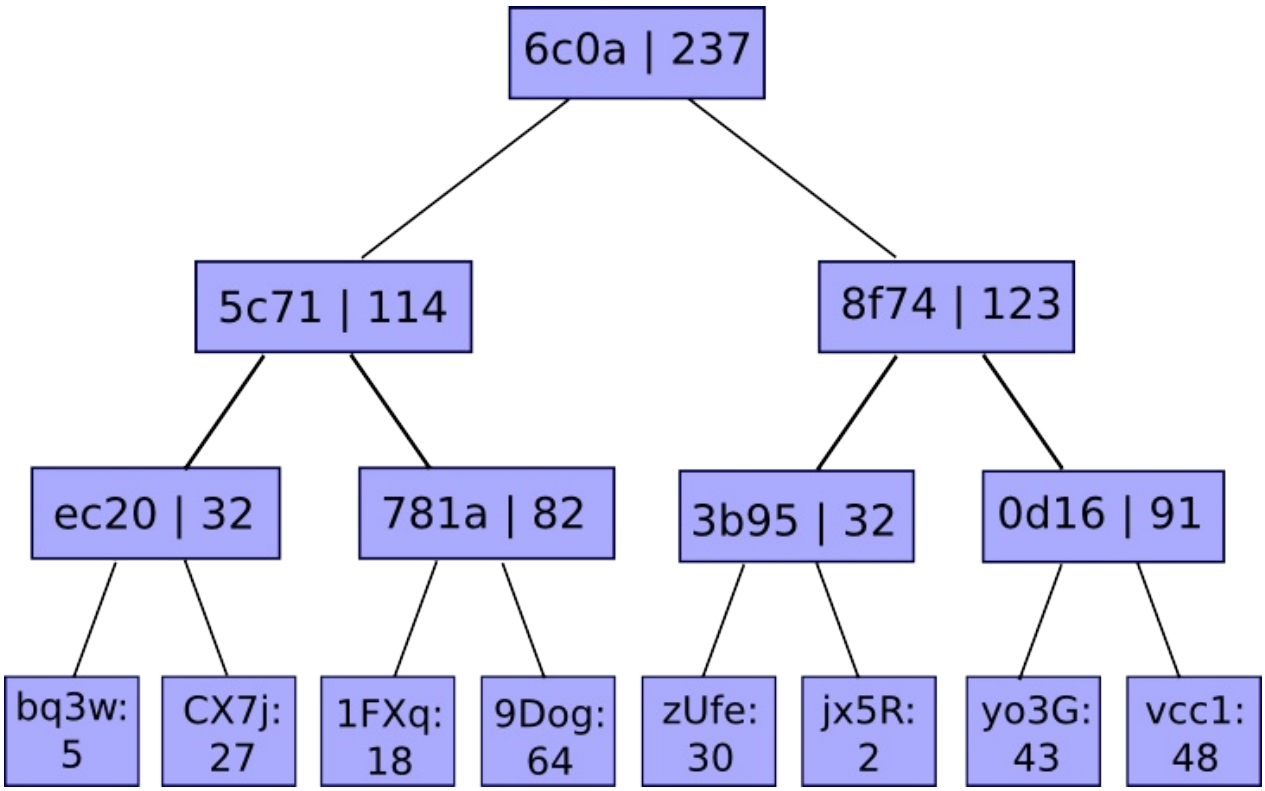
Merkle basic 2



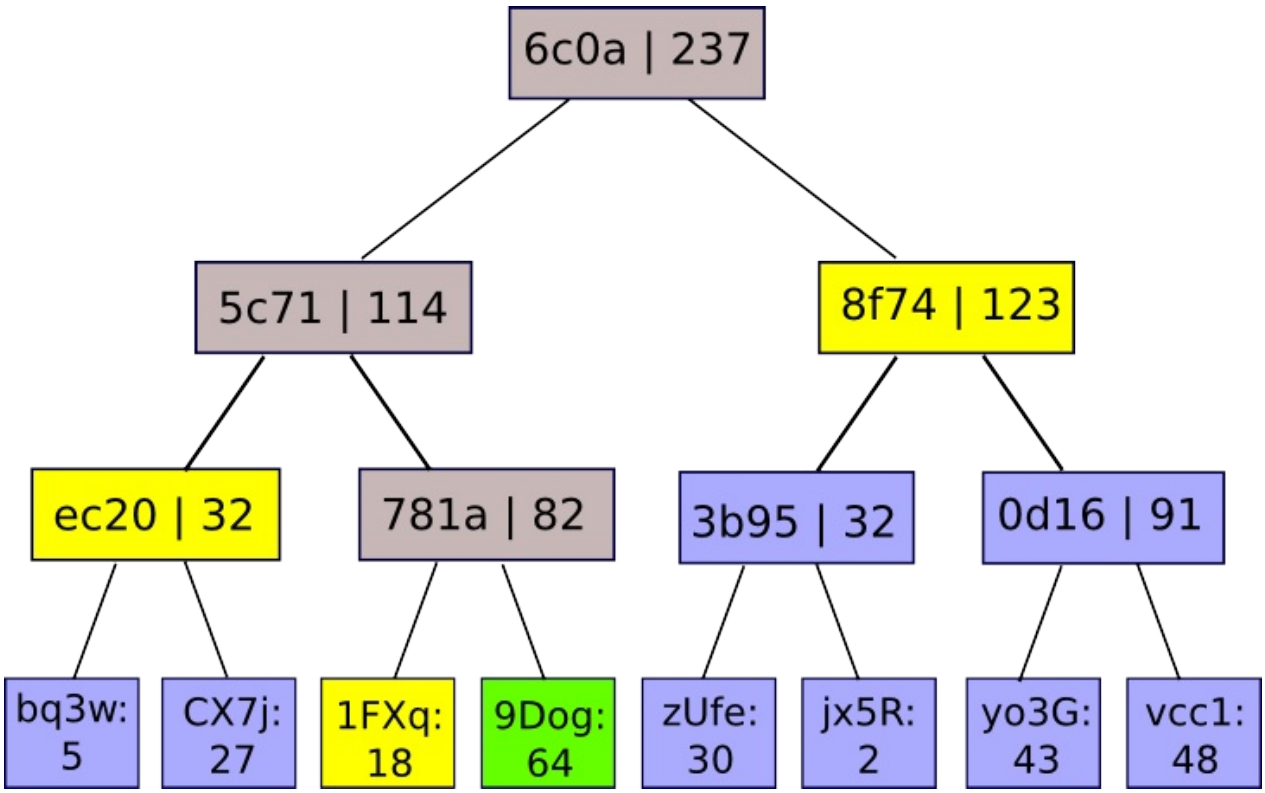
Merkle



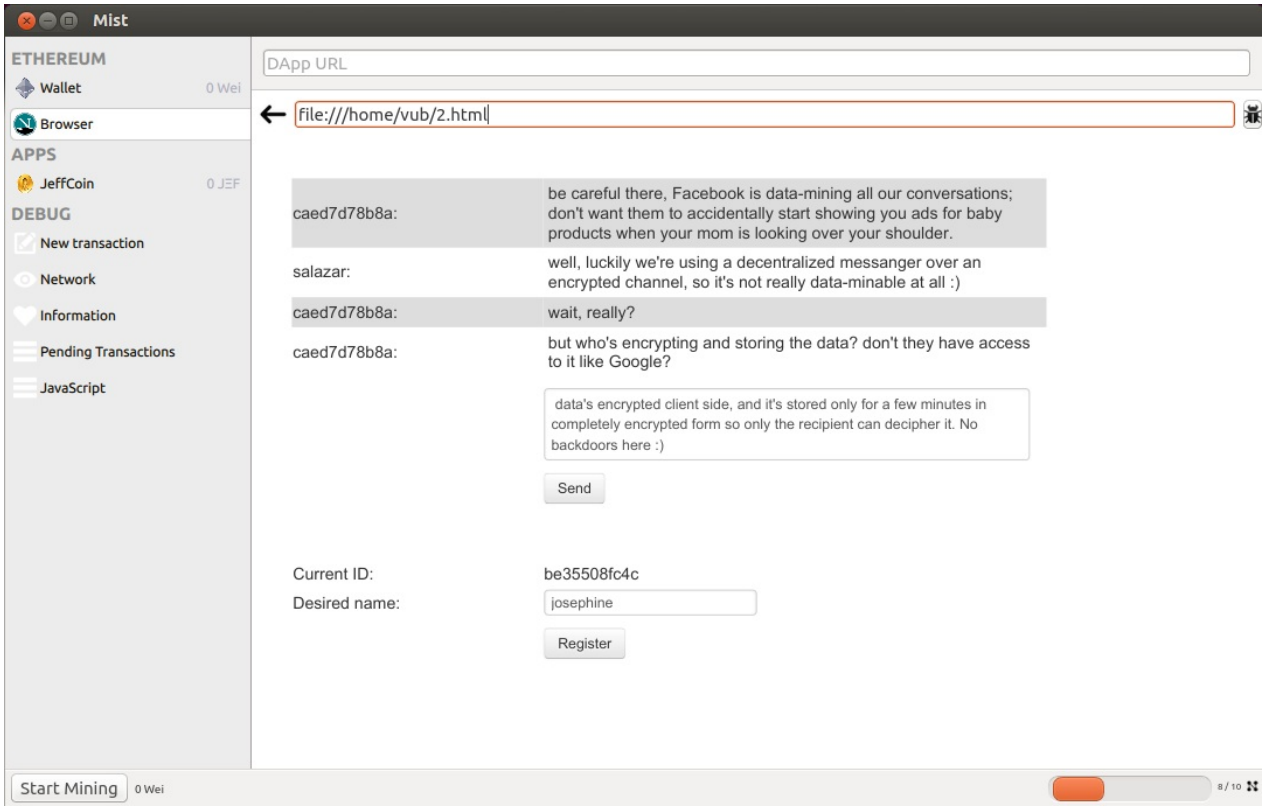
Merkle sum



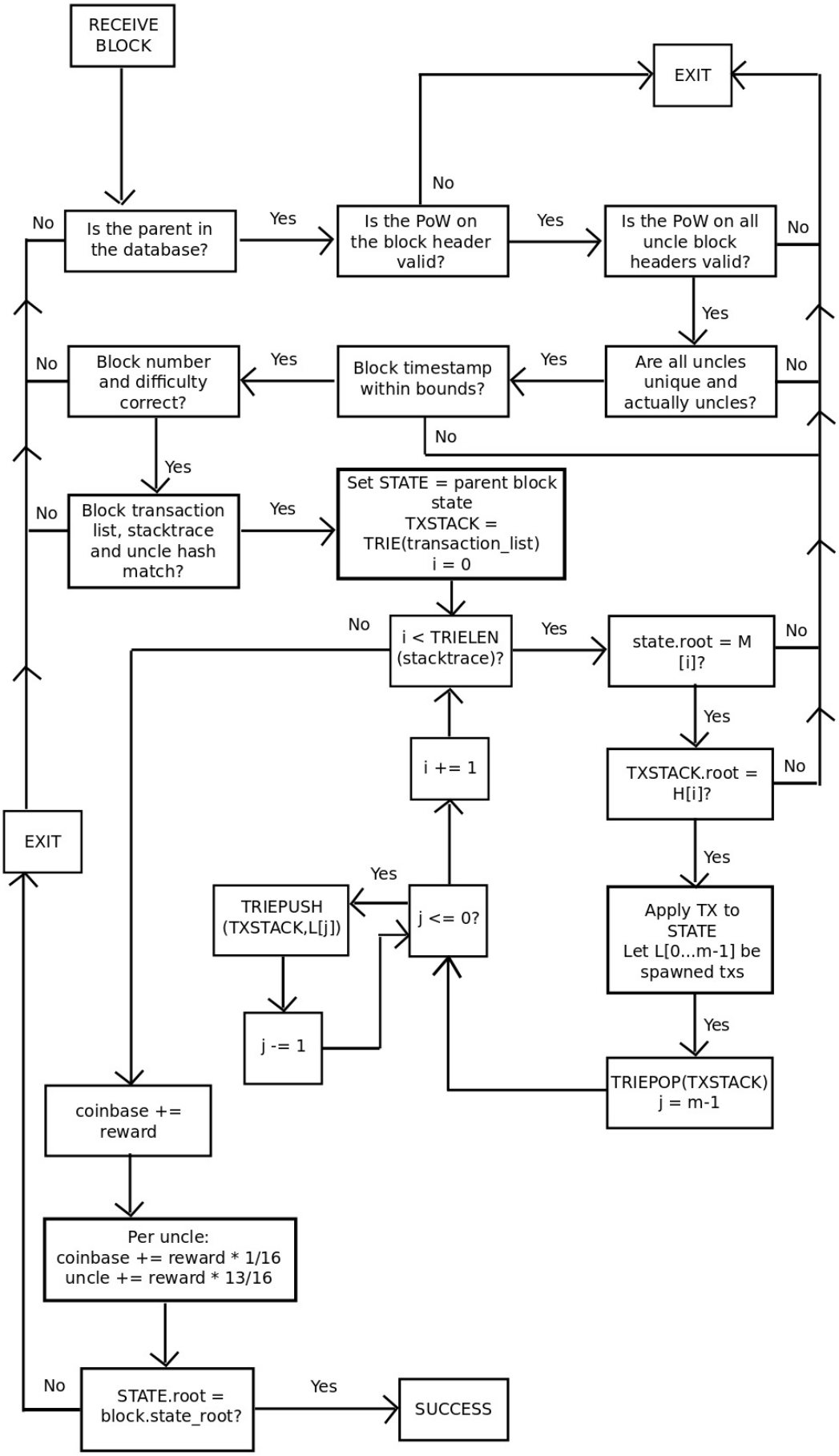
Merkle sum 2



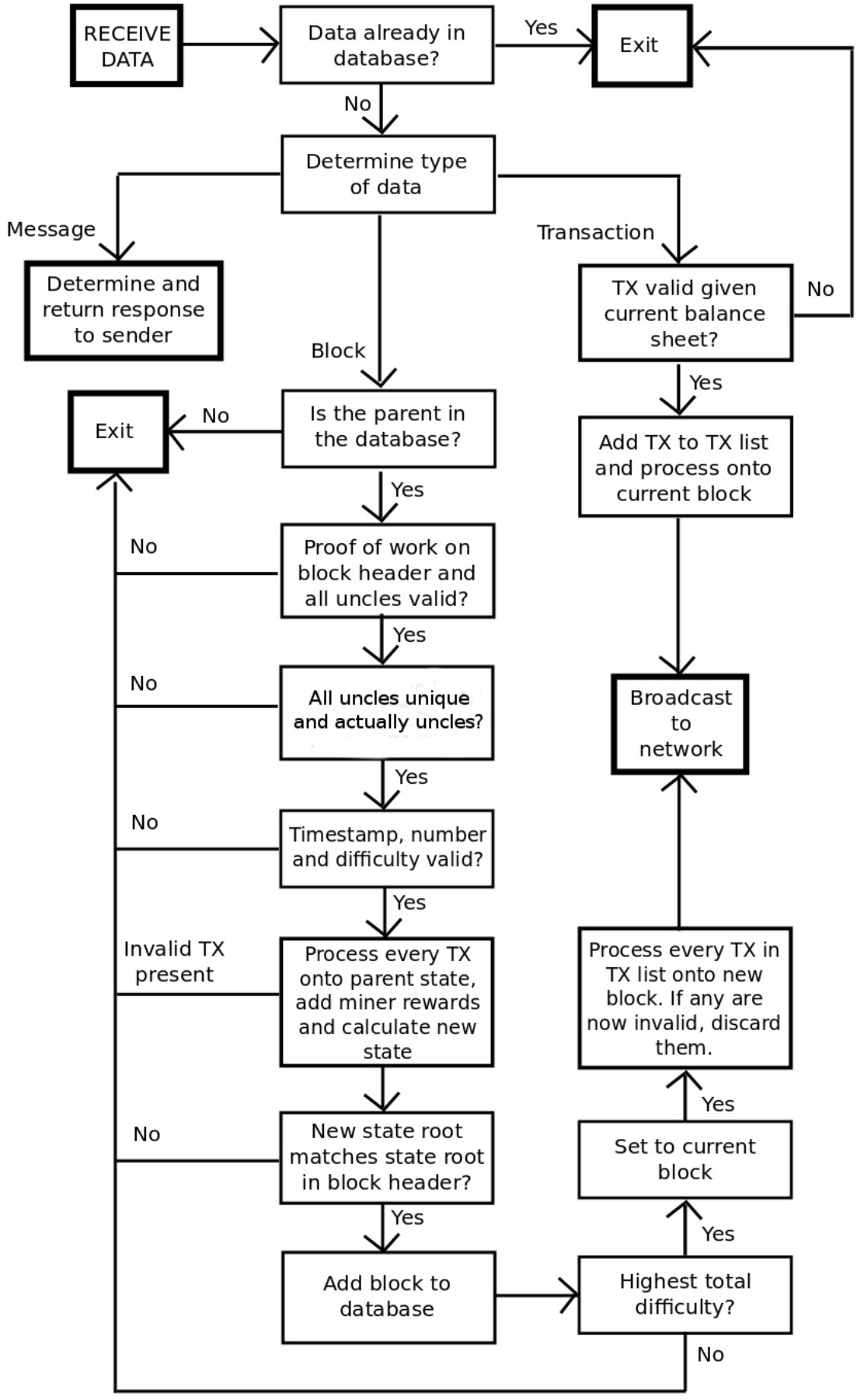
Messenger



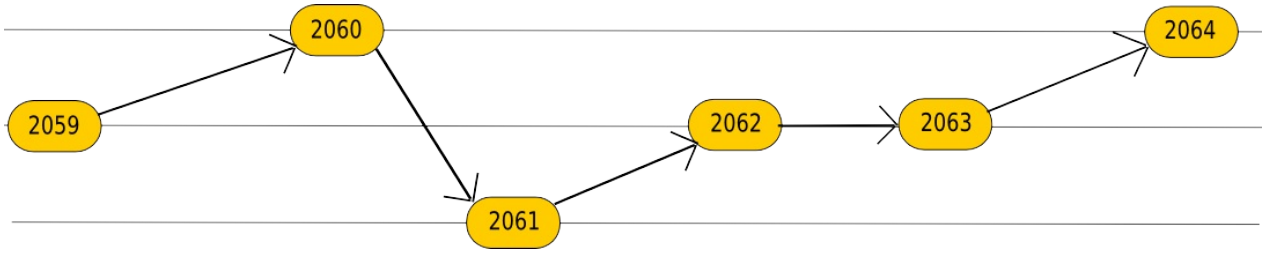
Miner receives block flowchart


RECEIVE BLOCK
EXIT
No
Is the parent in the database?
Yes
Is the PoW on the block header valid?
Yes
Is the PoW on all uncle block headers valid?
No
Yes
Block number and difficulty correct?
Yes
Block timestamp within bounds?
Yes
Are all uncles unique and actually uncles?
No
No
Yes
Block transaction list, stacktrace and uncle hash match?
Yes
Set STATE = parent block state
TXSTACK = TRIE(transaction_list)
i = 0
No
i < TRIELEN (stacktrace)?
Yes
state.root = M[i]?
No
Yes
TXSTACK.root = H[i]?
No
Yes
i += 1
EXIT
Yes
TRIEPUSH (TXSTACK,L[j])
j <= 0?
Apply TX to STATE
Let L[0...m-1] be spawned txs
Yes
j -= 1
Yes
TRIEPOP(TXSTACK)
j = m-1
coinbase += reward
Per uncle:
coinbase += reward * 1/16
uncle += reward * 13/16
No
STATE.root = block.state_root?
Yes
SUCCESS

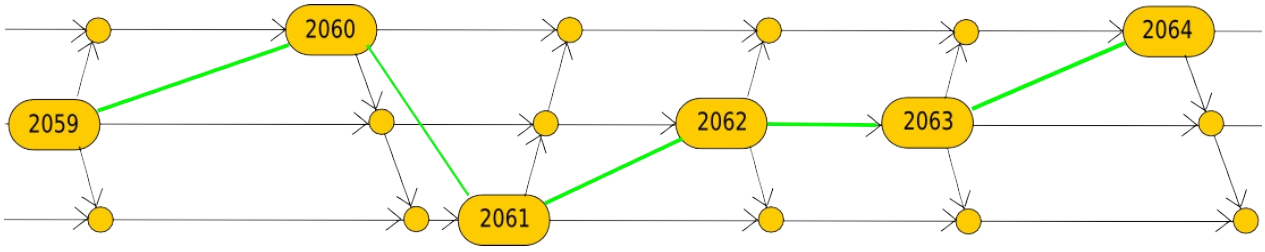
Miner receives data flowchart


RECEIVE DATA
Data already in database?
Yes
Exit
No
Determine type of data
Message
Determine and return response to sender
Transaction
TX valid given current balance sheet?
No
Yes
Block
Exit
No
Is the parent in the database?
Add TX to TX list and process onto current block
Yes
No
Proof of work on block header and all uncles valid?
Yes
No
All uncles unique and actually uncles?
Broadcast to network
Yes
No
Timestamp, number and difficulty valid?
Yes
Invalid TX present
Process every TX onto parent state, add miner rewards and calculate new state
Process every TX in TX list onto new block. If any are now invalid, discard them.
Yes
No
New state root matches state root in block header?
Set to current block
Yes
Yes
Add block to database
Highest total difficulty?
No

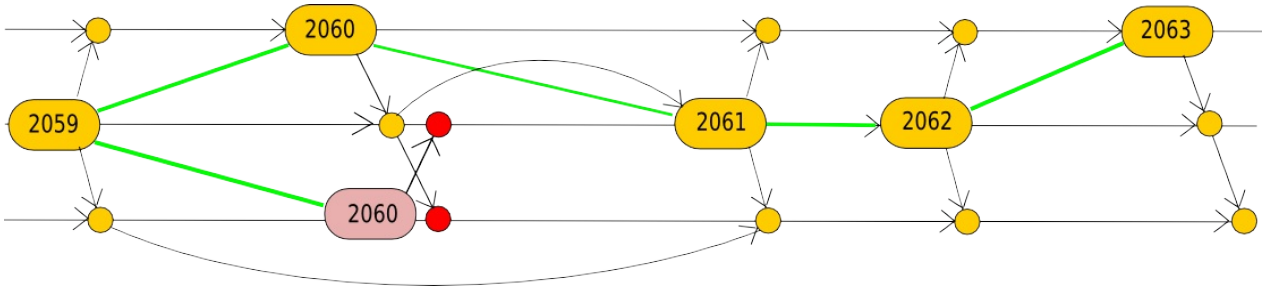
Miner net 1



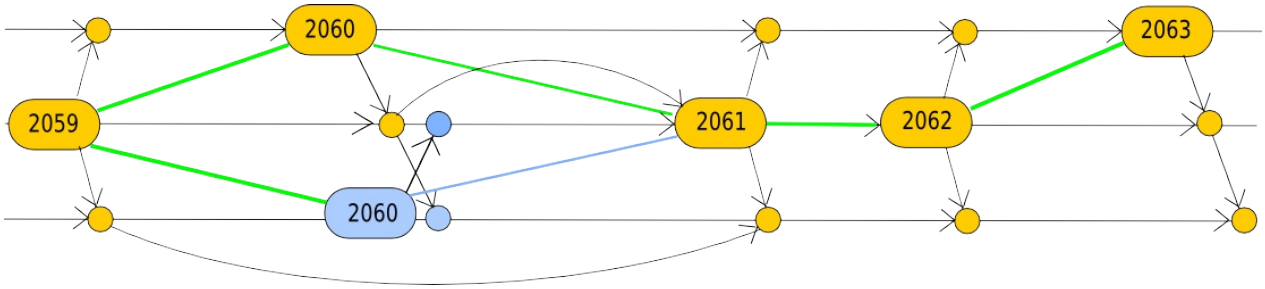
Miner net 2



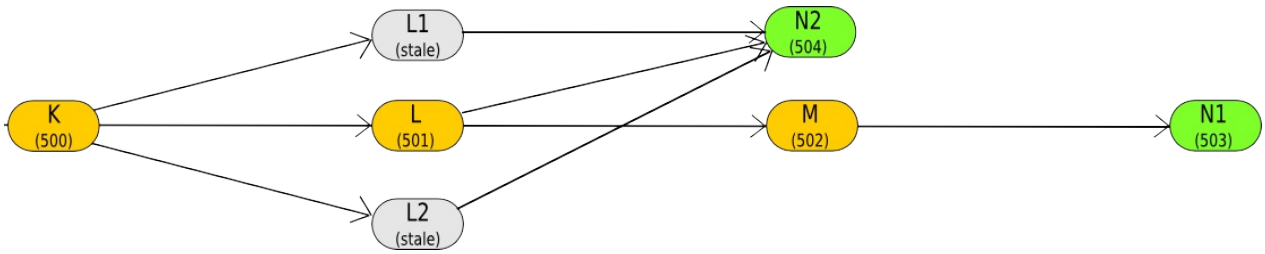
Miner net 3



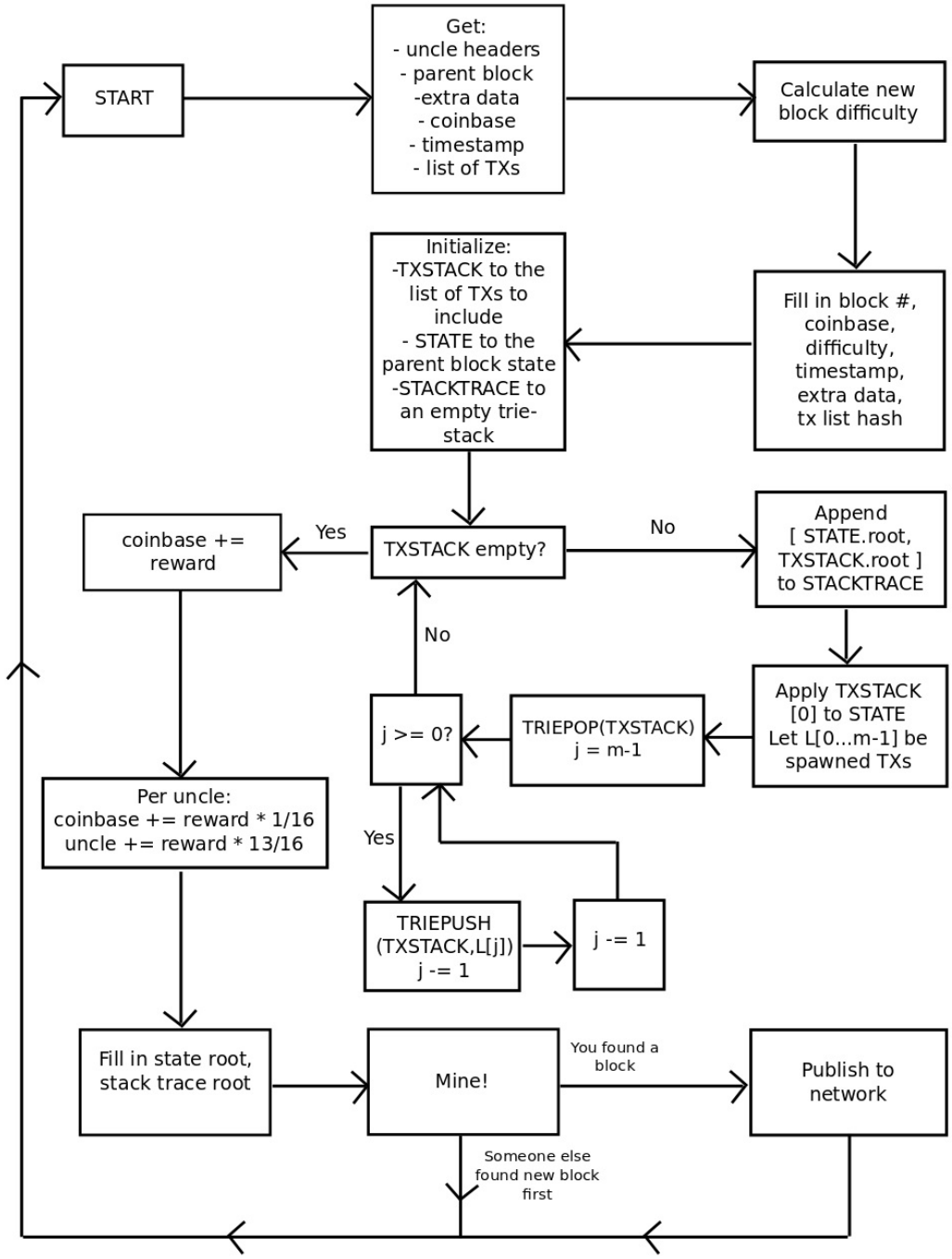
Miner net 4



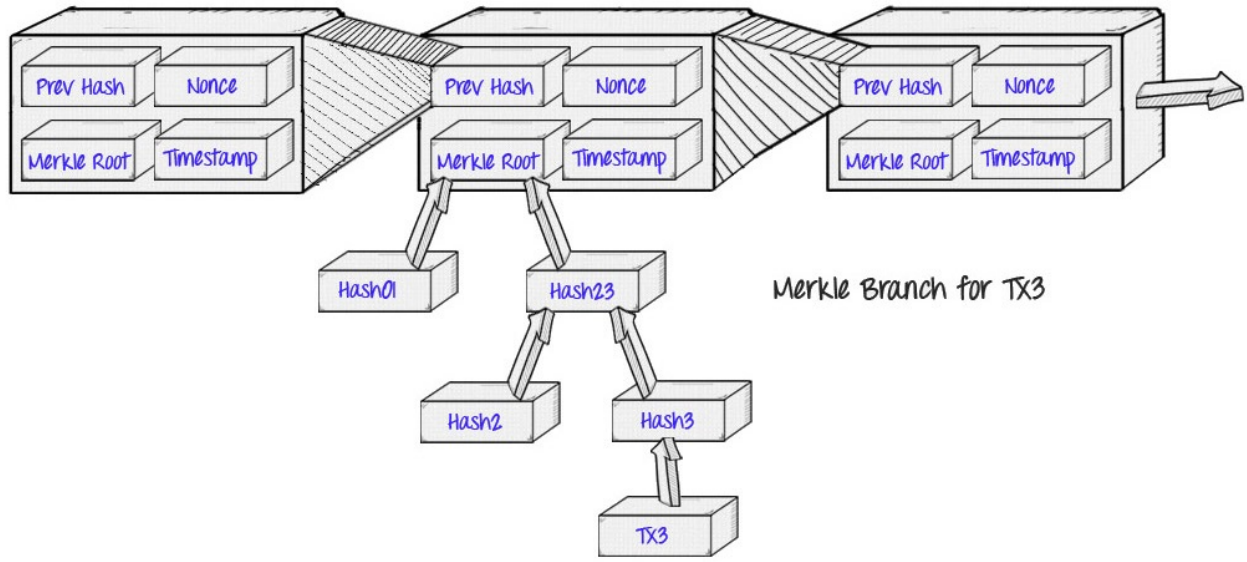
Miner net 5



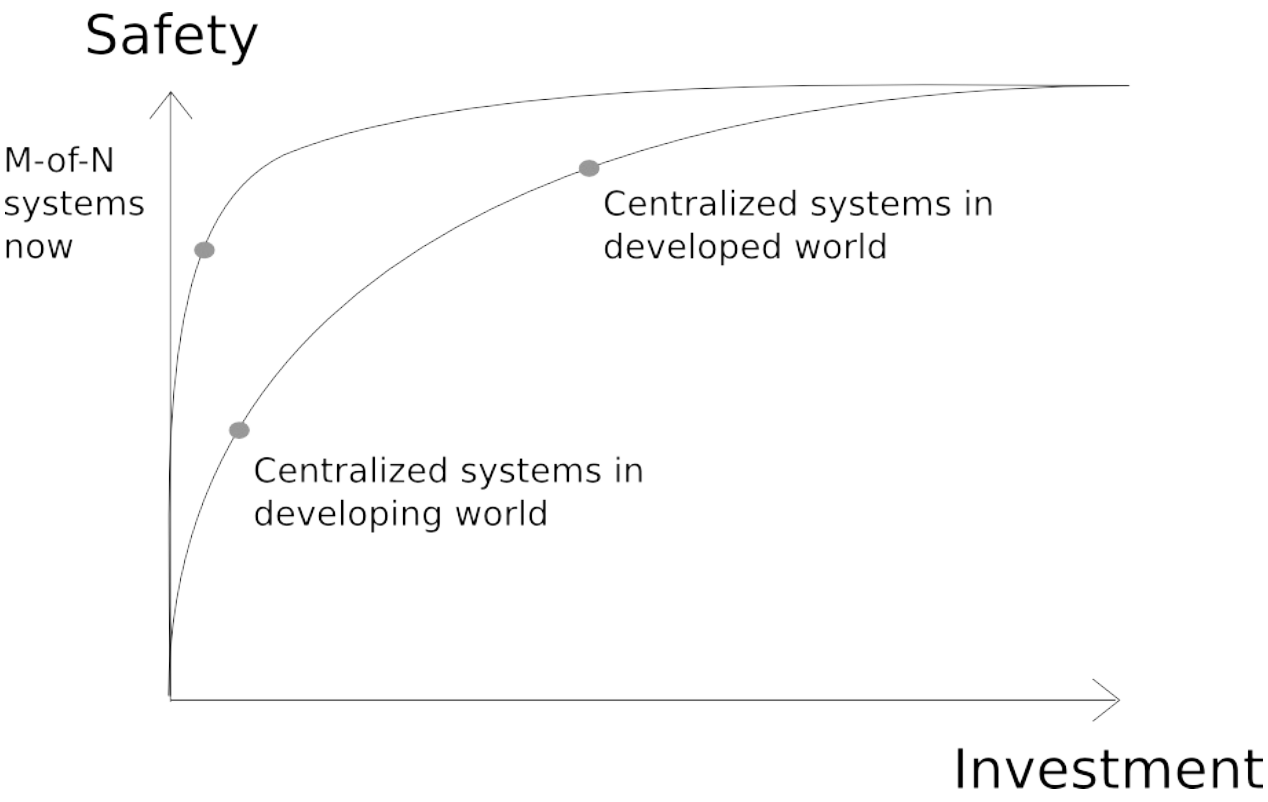
Mining process flowchart


START
Get:
- uncle headers
- parent block
-extra data
- coinbase
- timestamp
- list of TXs
Calculate new block difficulty
Fill in block #, coinbase, difficulty, timestamp, extra data, tx list hash
Initialize:
-TXSTACK to the list of TXs to include
- STATE to the parent block state
-STACKTRACE to an empty trie-stack
TXSTACK empty?
Yes
coinbase += reward
No
Append [STATE.root, TXSTACK.root] to STACKTRACE
Apply TXSTACK [0] to STATE
Let L[0...m-1] be spawned TXs
TRIEPOP(TXSTACK)
j = m-1
j >= 0?
No
Yes
Per uncle:
coinbase += reward * 1/16
uncle += reward * 13/16
TRIEPUSH (TXSTACK,L[j])
j -= 1
j -= 1
Fill in state root, stack trace root
Mine!
You found a block
Publish to network
Someone else found new block first

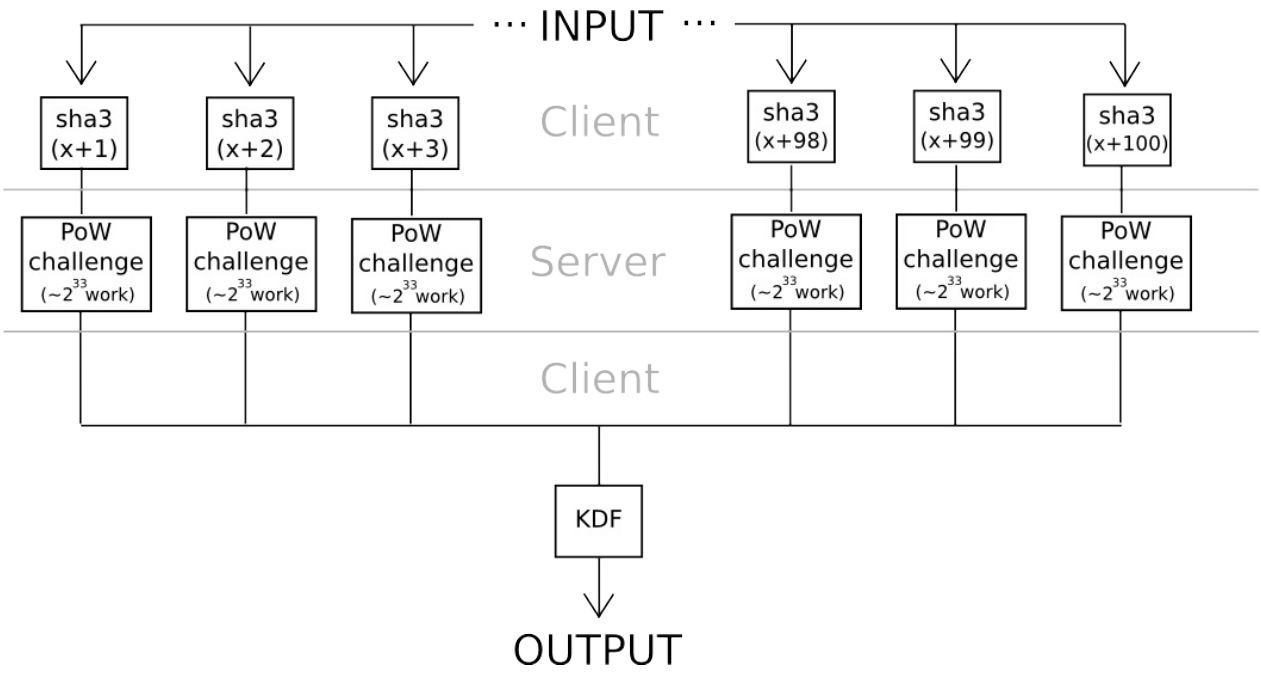
Mining



M of N safety



Multikdf

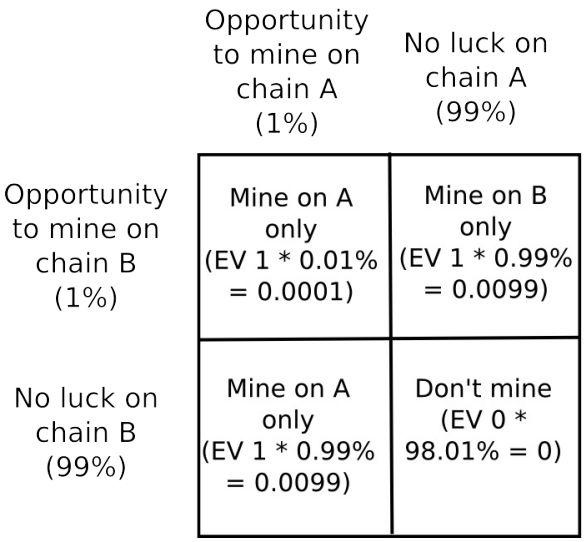


Namereg

```
key = msg.data[0]
value = msg.data[1]
if !contract.storage[key]:
    contract.storage[key] = value
```

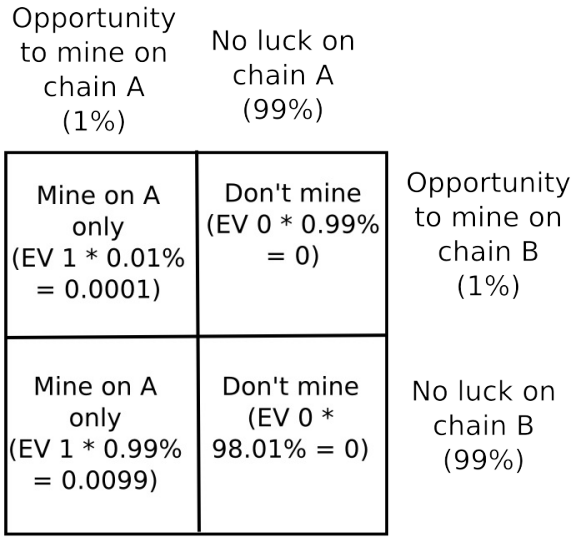
Need for preselection

Dishonest mining (try both chains)



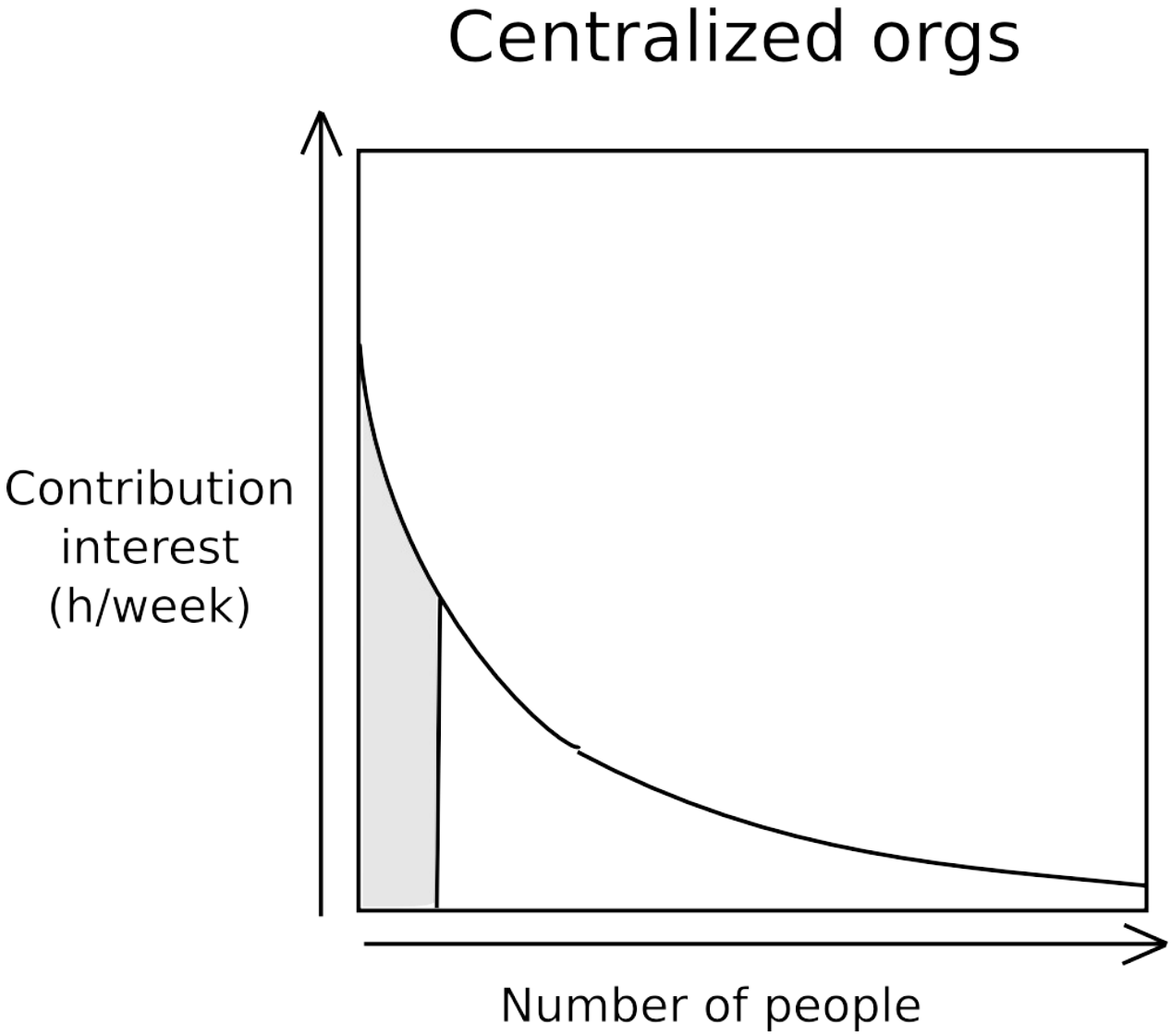
Total EV: 0.0199 per block

Honest mining (mine on A)

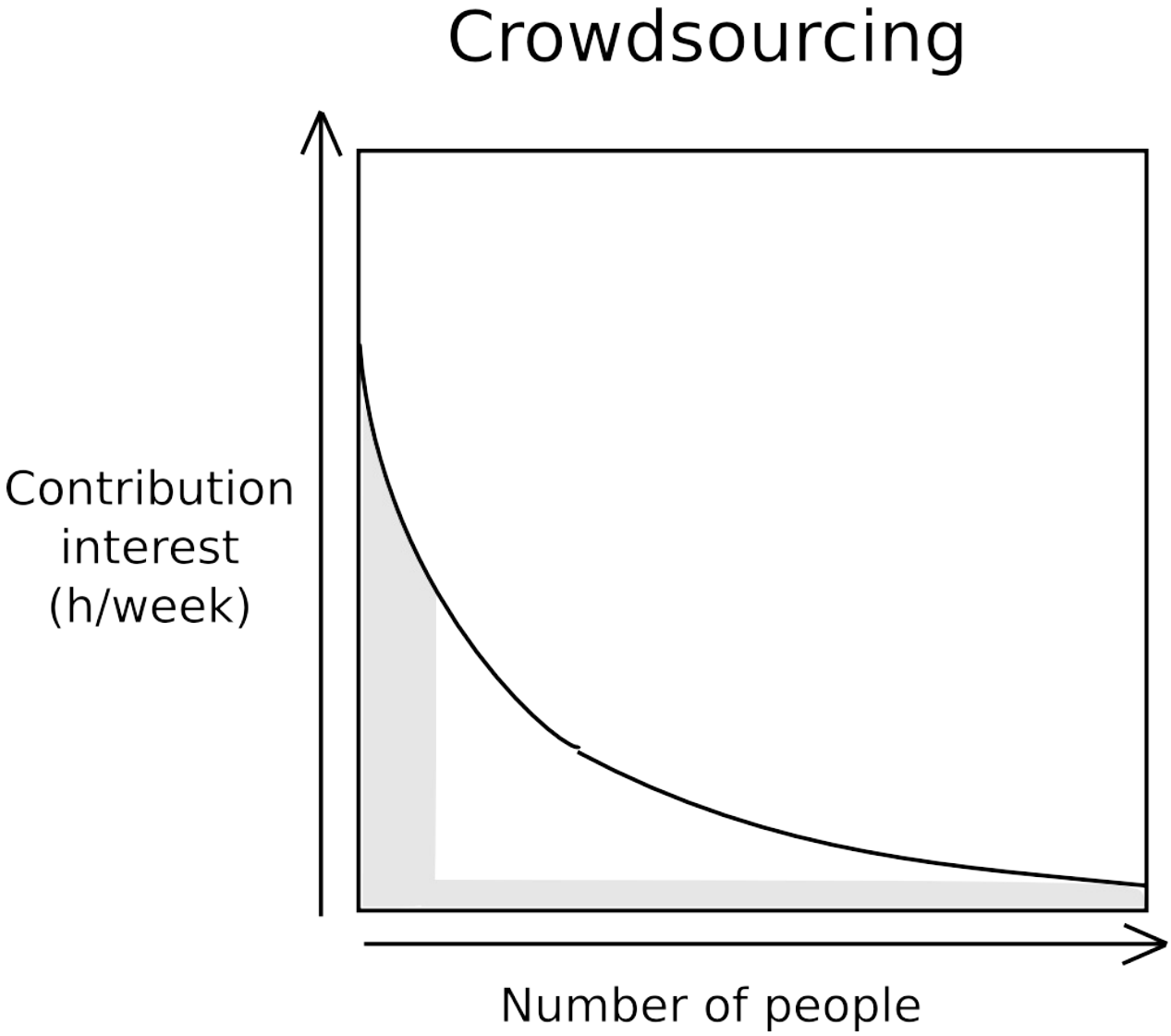


Total EV: 0.01 per block

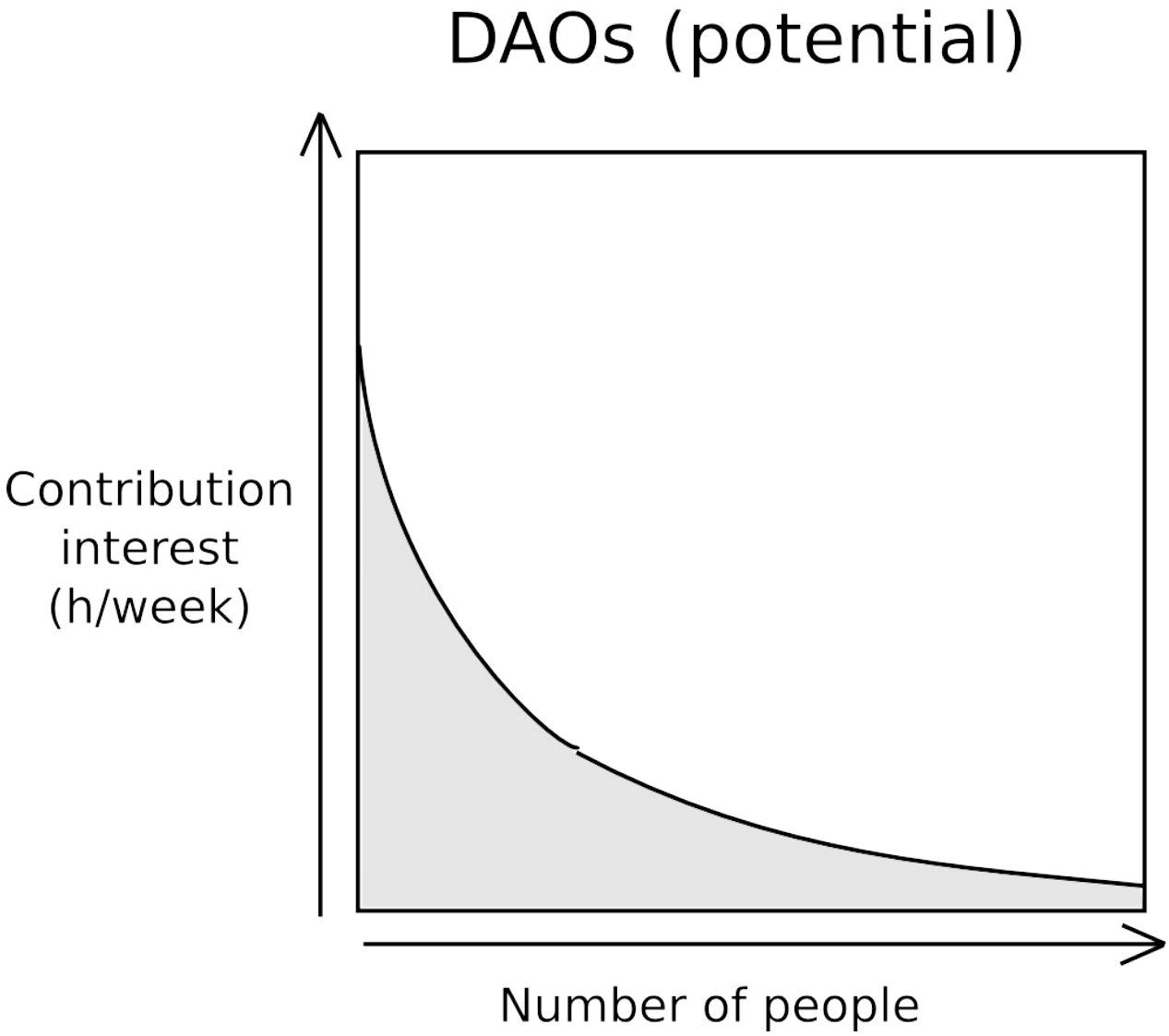
Orgs 1


Centralized orgs
Contribution interest (h/week)
Number of people

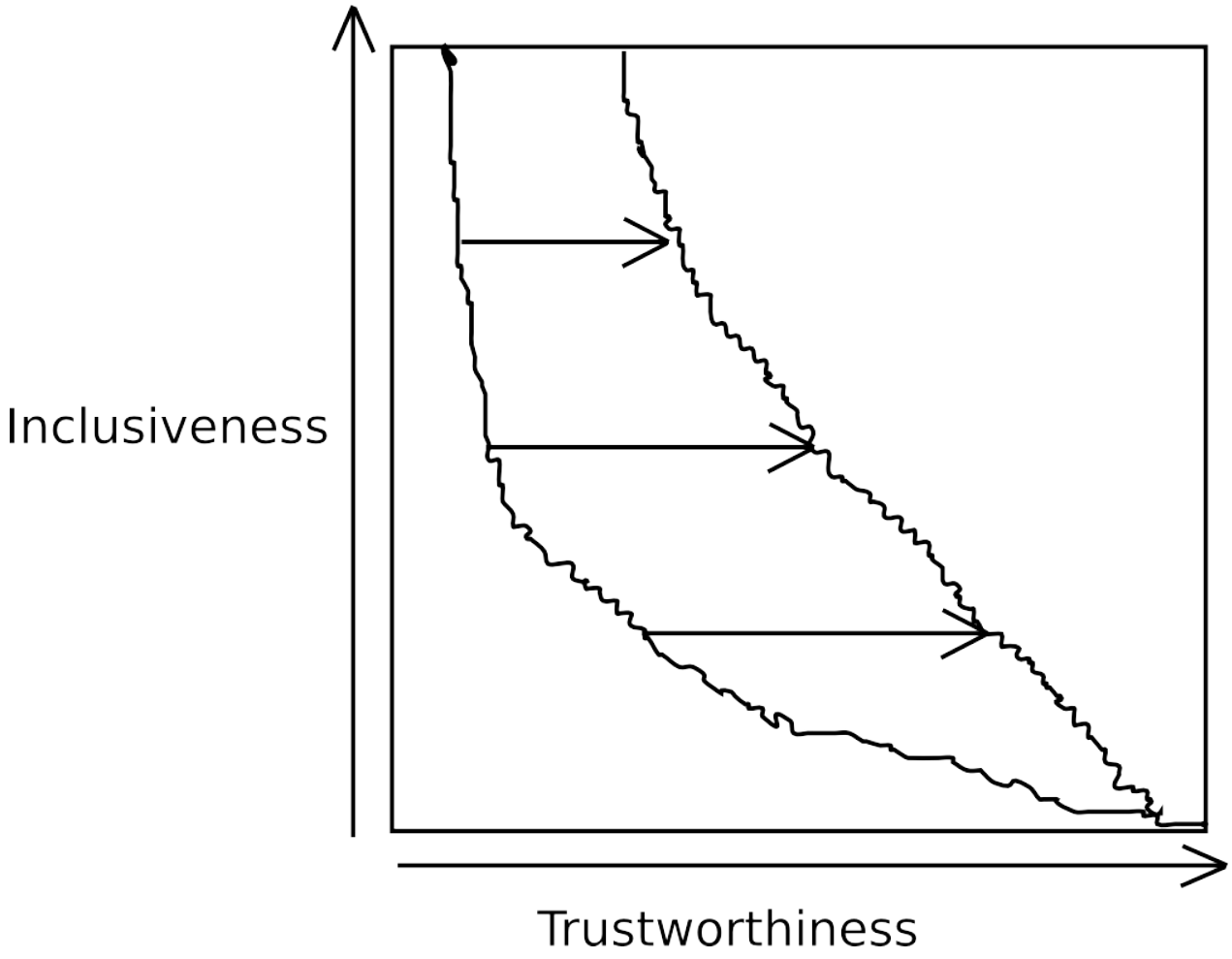
Orgs 2


Crowdsourcing
Contribution interest (h/week)
Number of people

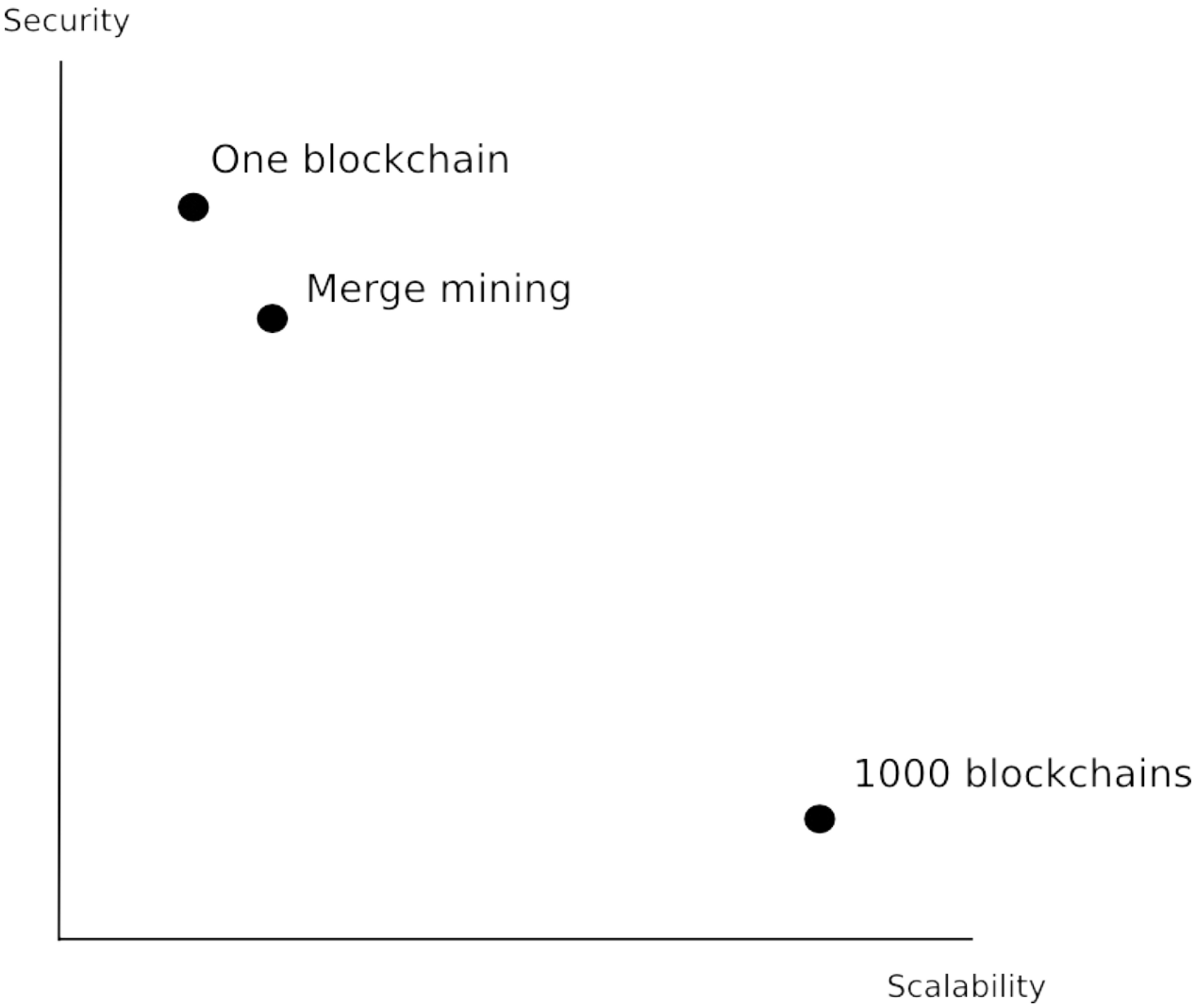
Orgs 3



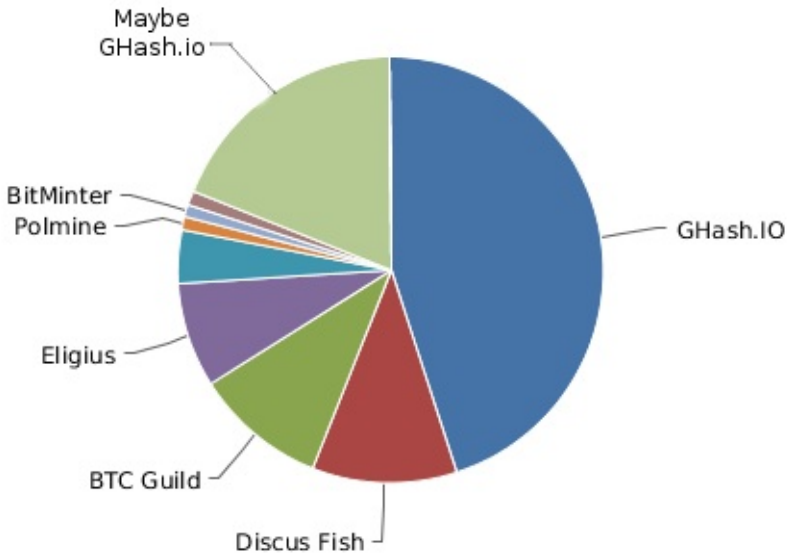
Pareto chart



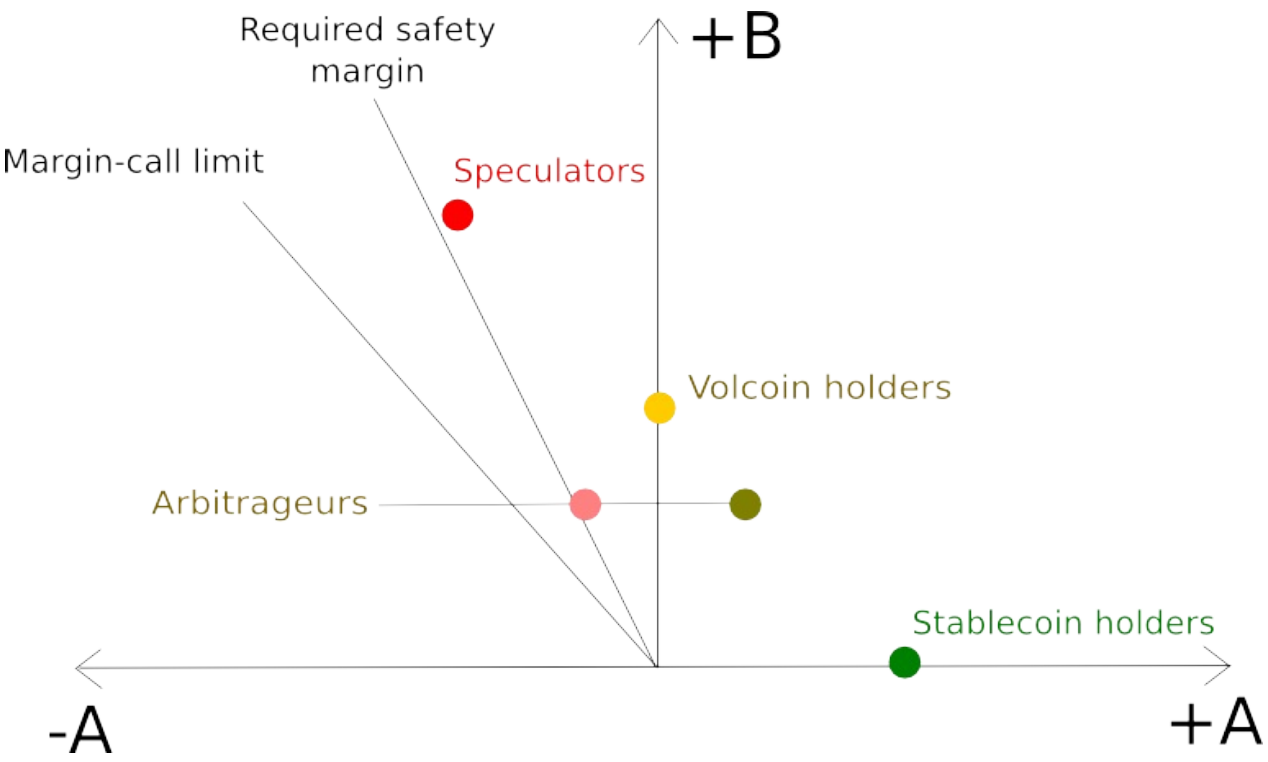
Pareto scaling


Security
One blockchain
Merge mining
1000 blockchains
Scalability

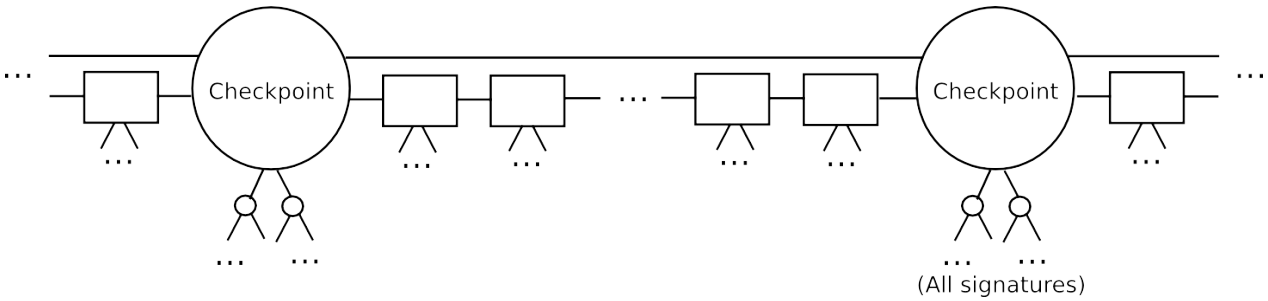
Pools


Maybe
GHash.io
BitMinter
Polmine
Eligius
BTC Guild
Discus Fish
GHash.IO

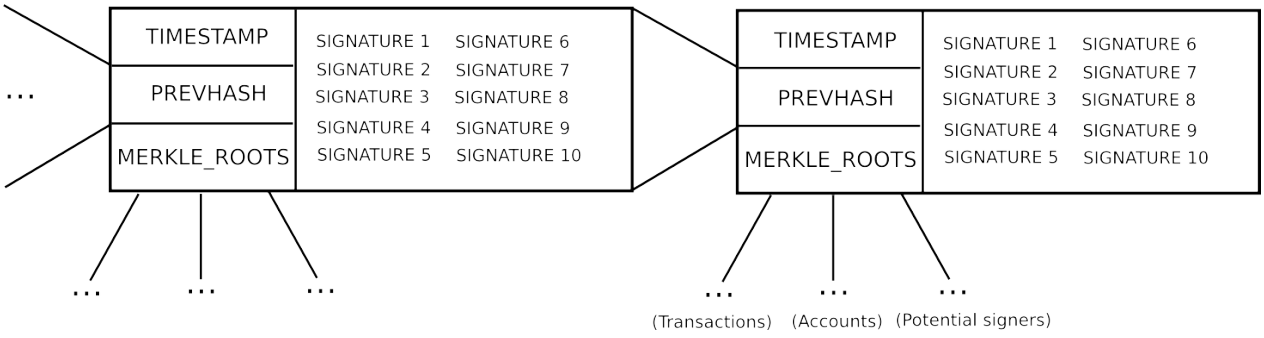
Portofolio replication



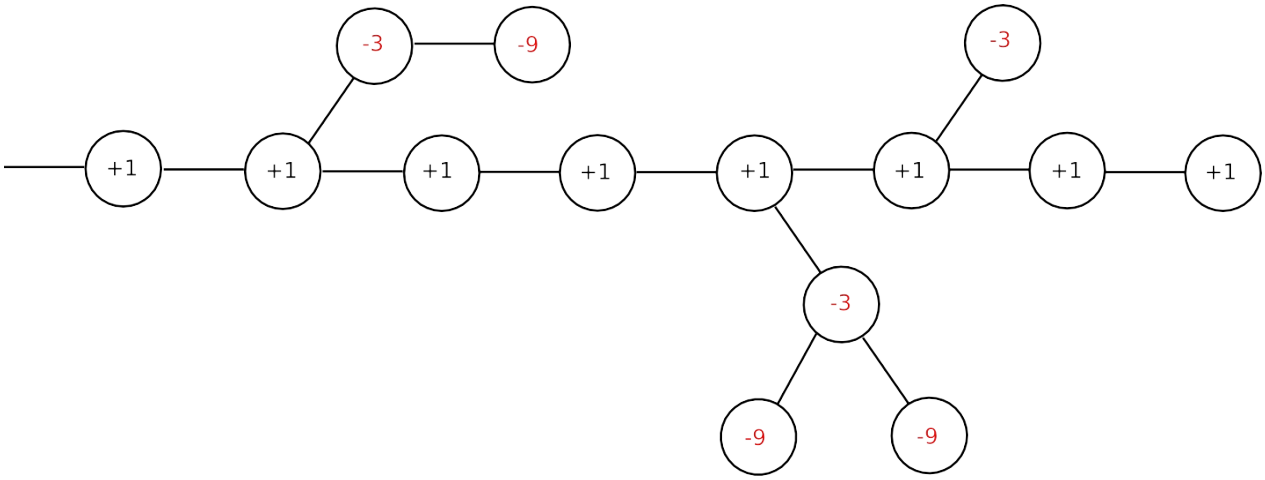
POS checkpoints



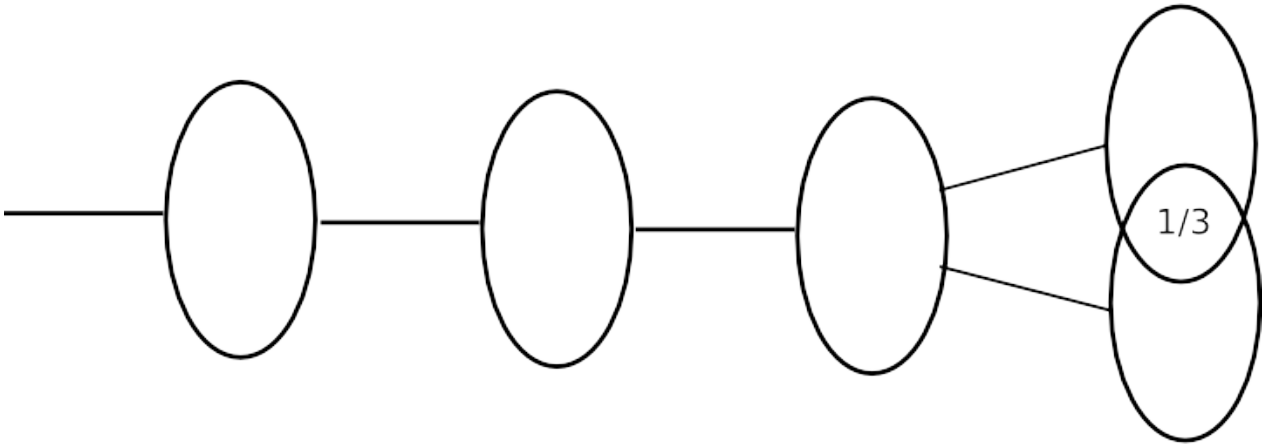
POS header



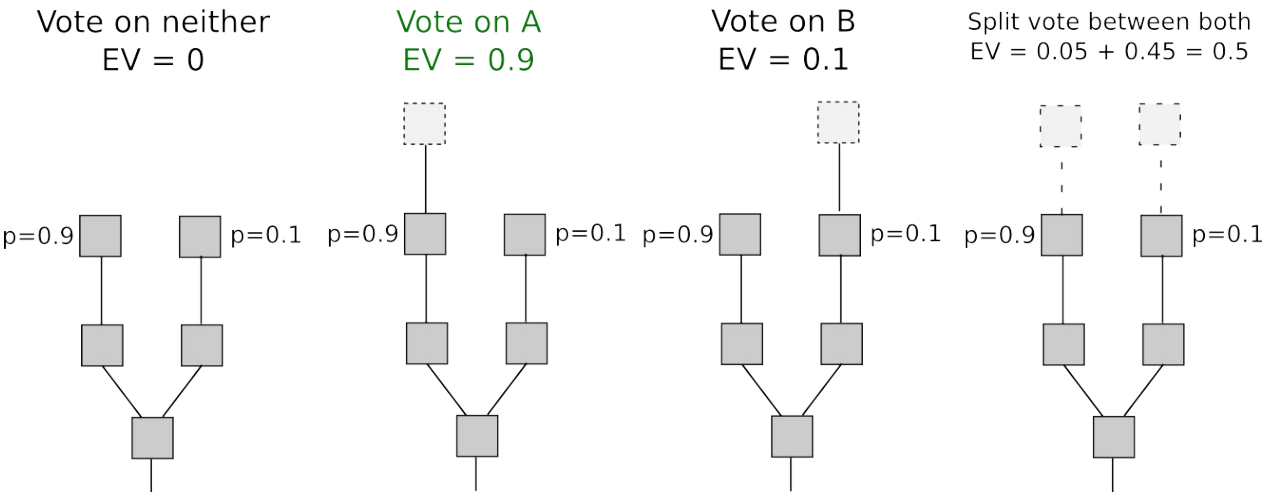
POS selection 2



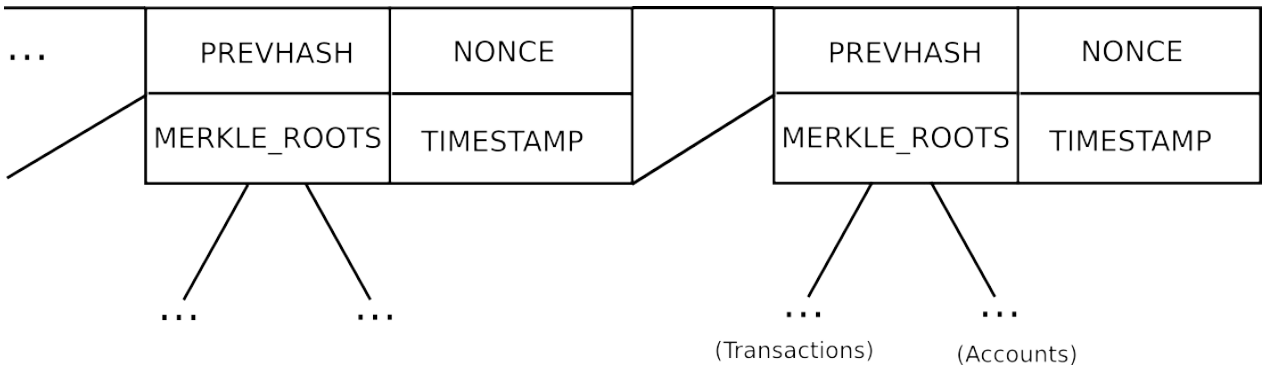
POS split



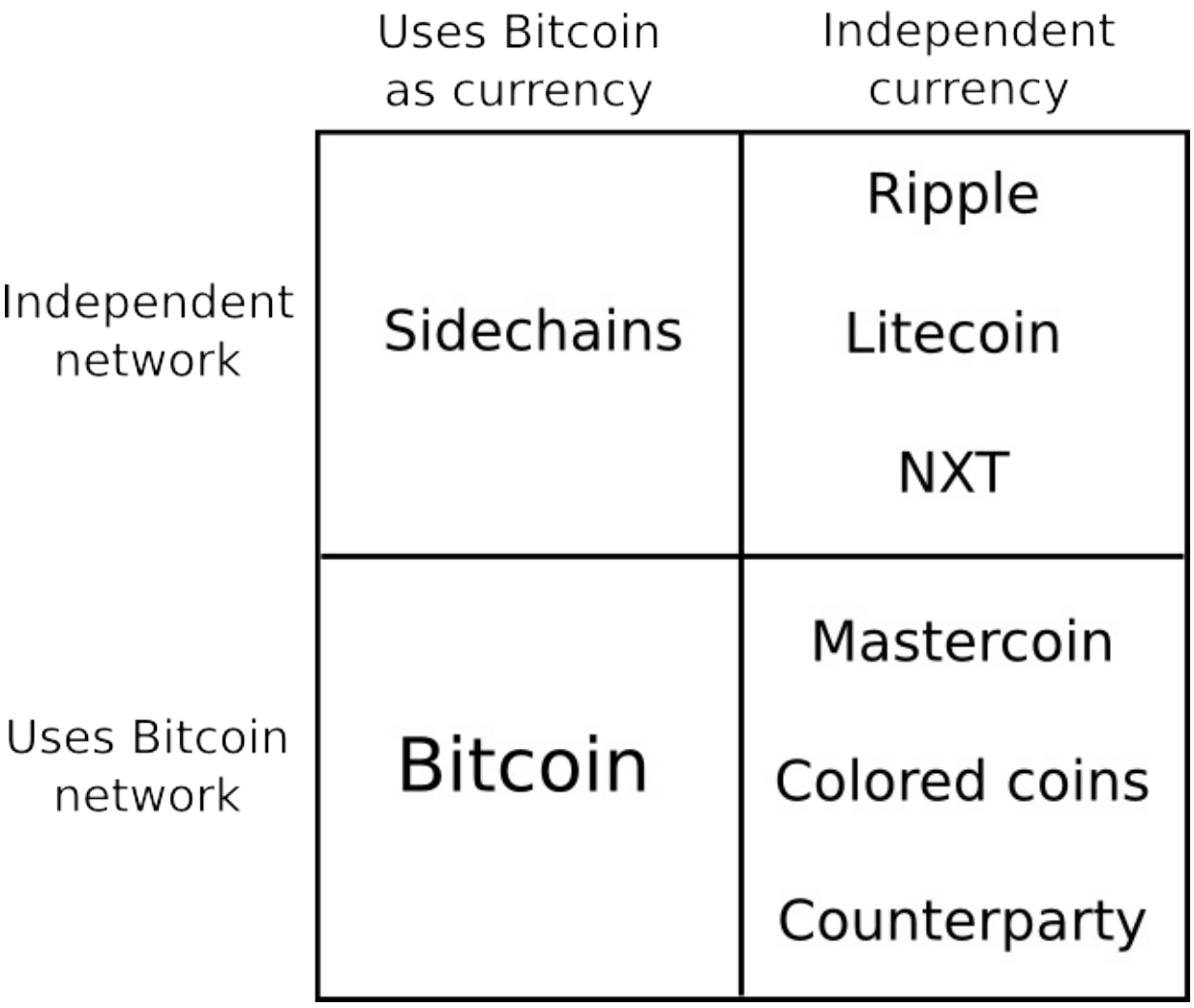
POW sec



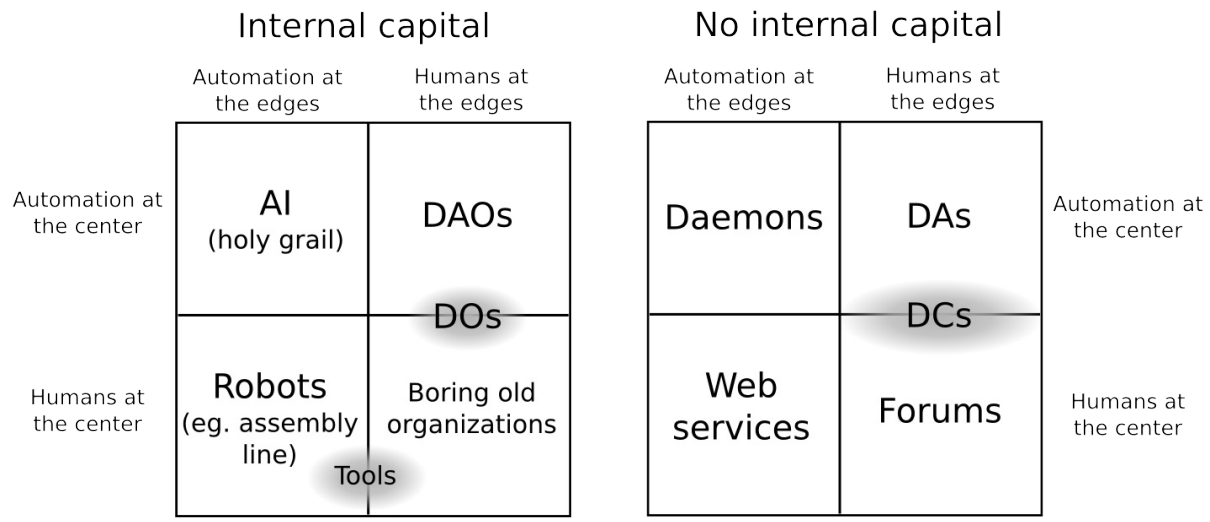
POW header



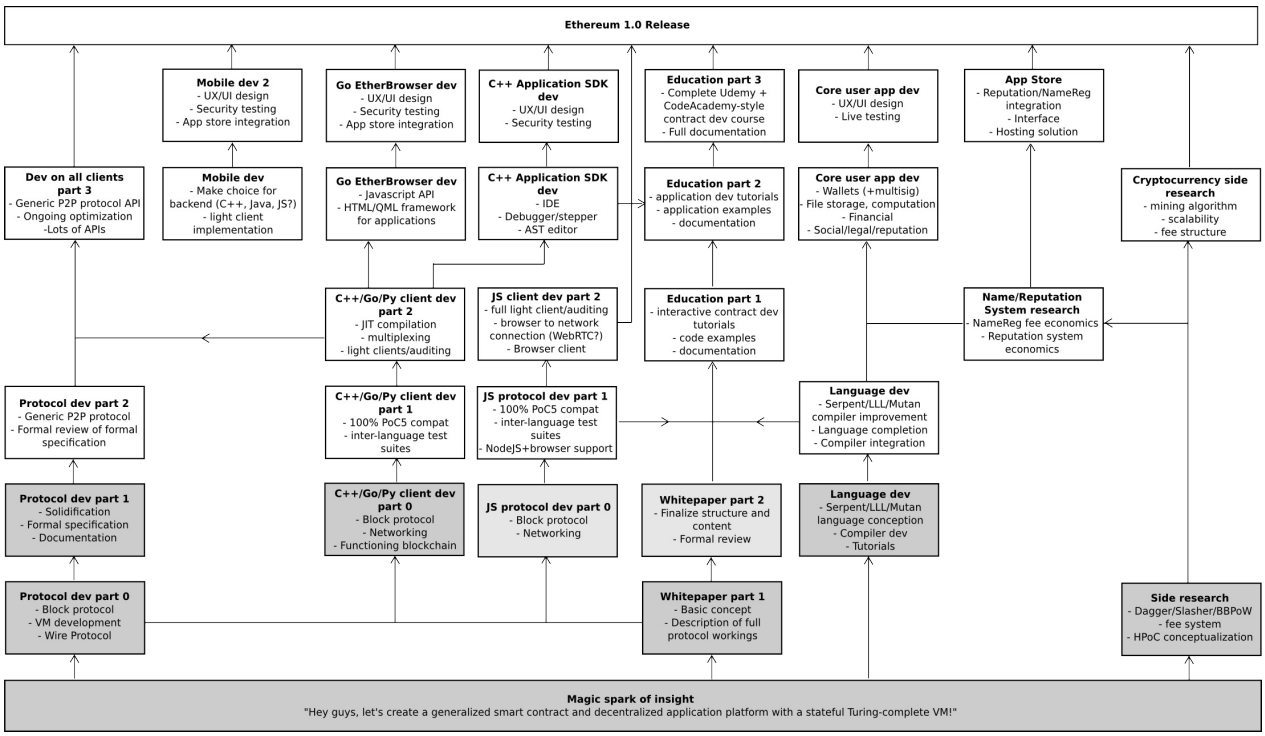
Quadrant chart



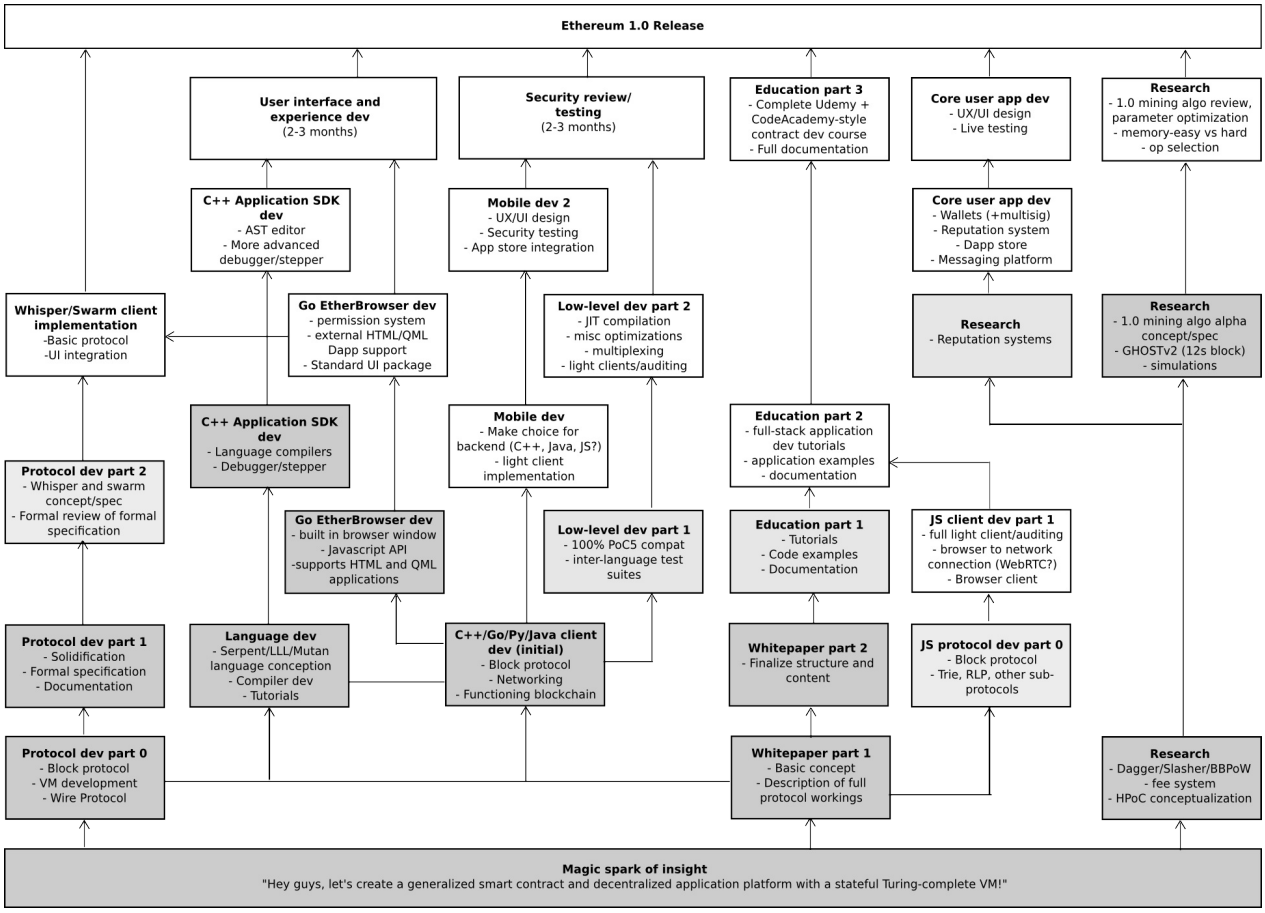
Quadrant chart 2



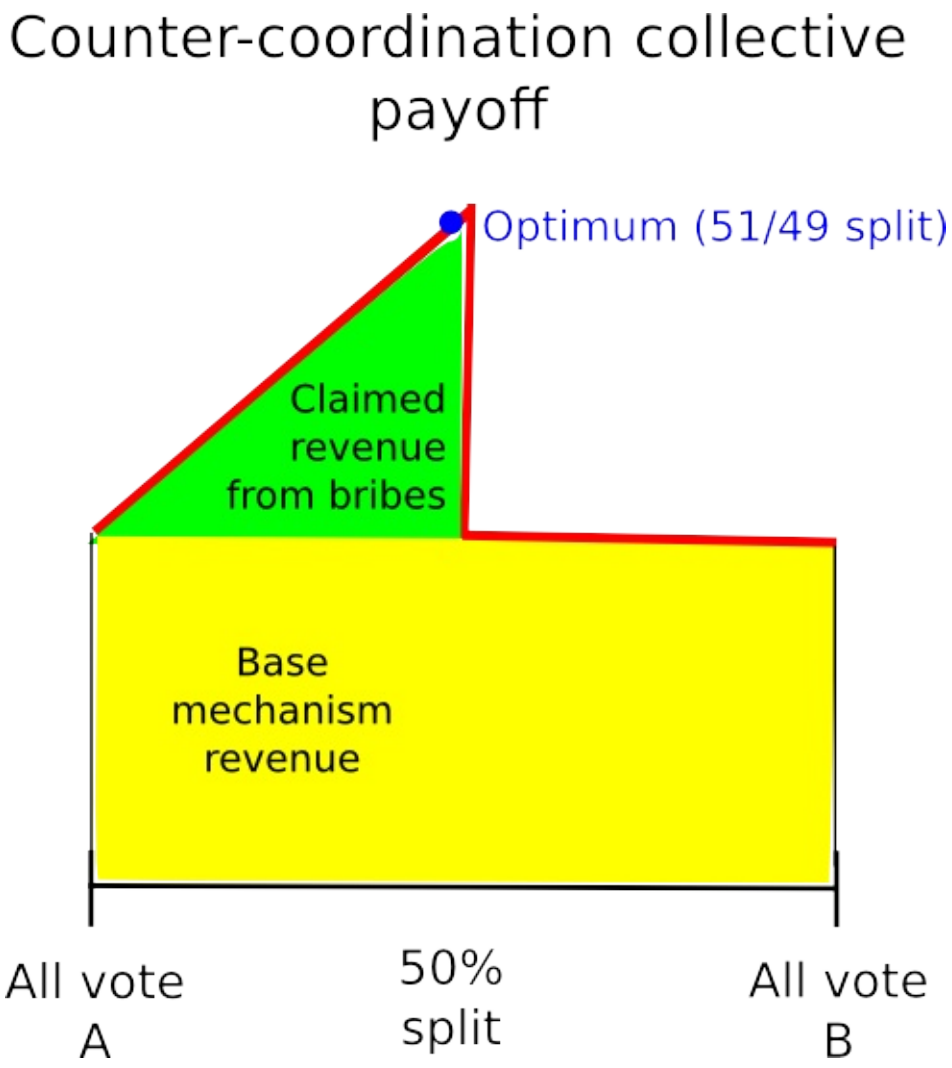
Roadmap



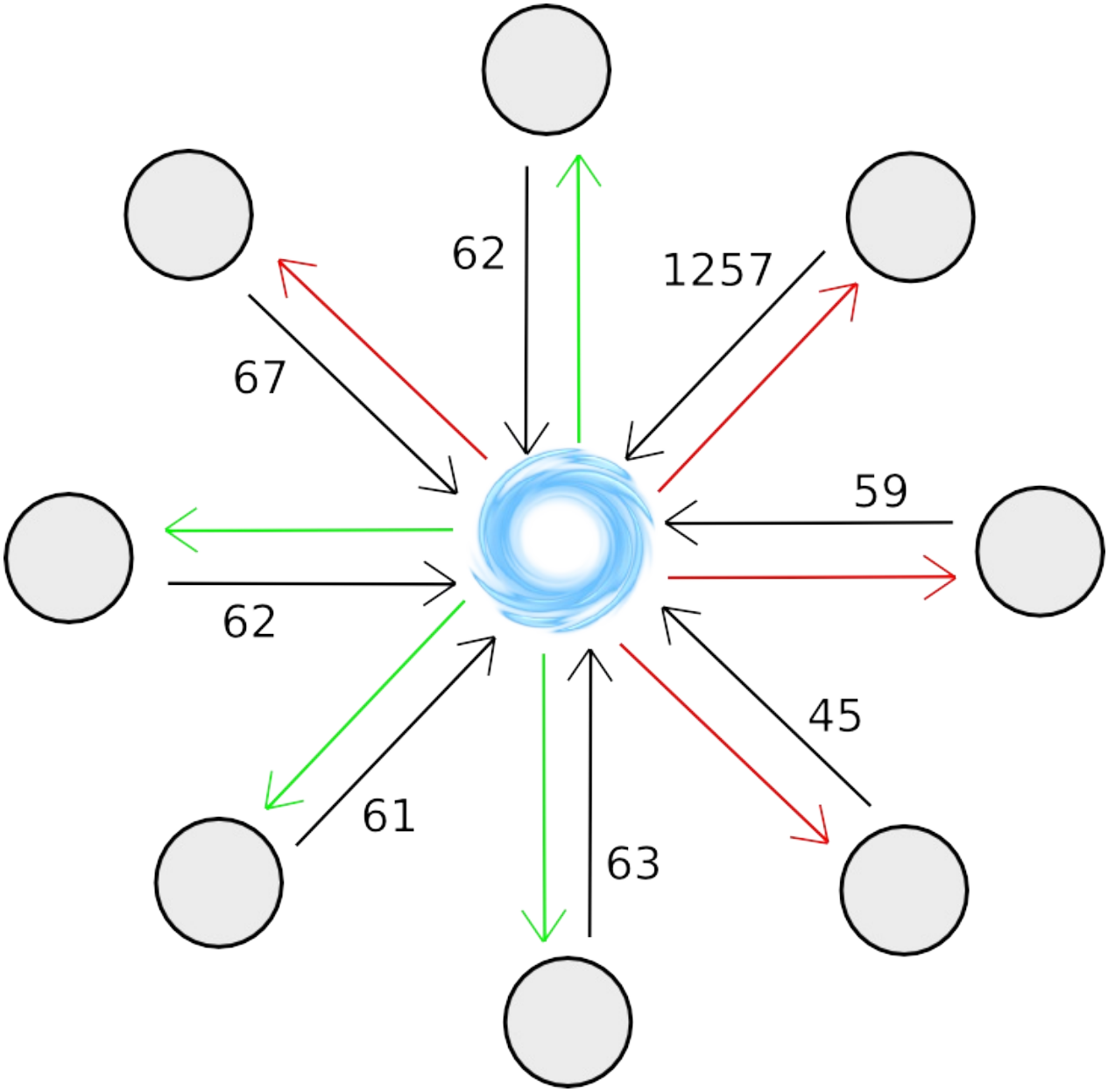
Roadmap 3



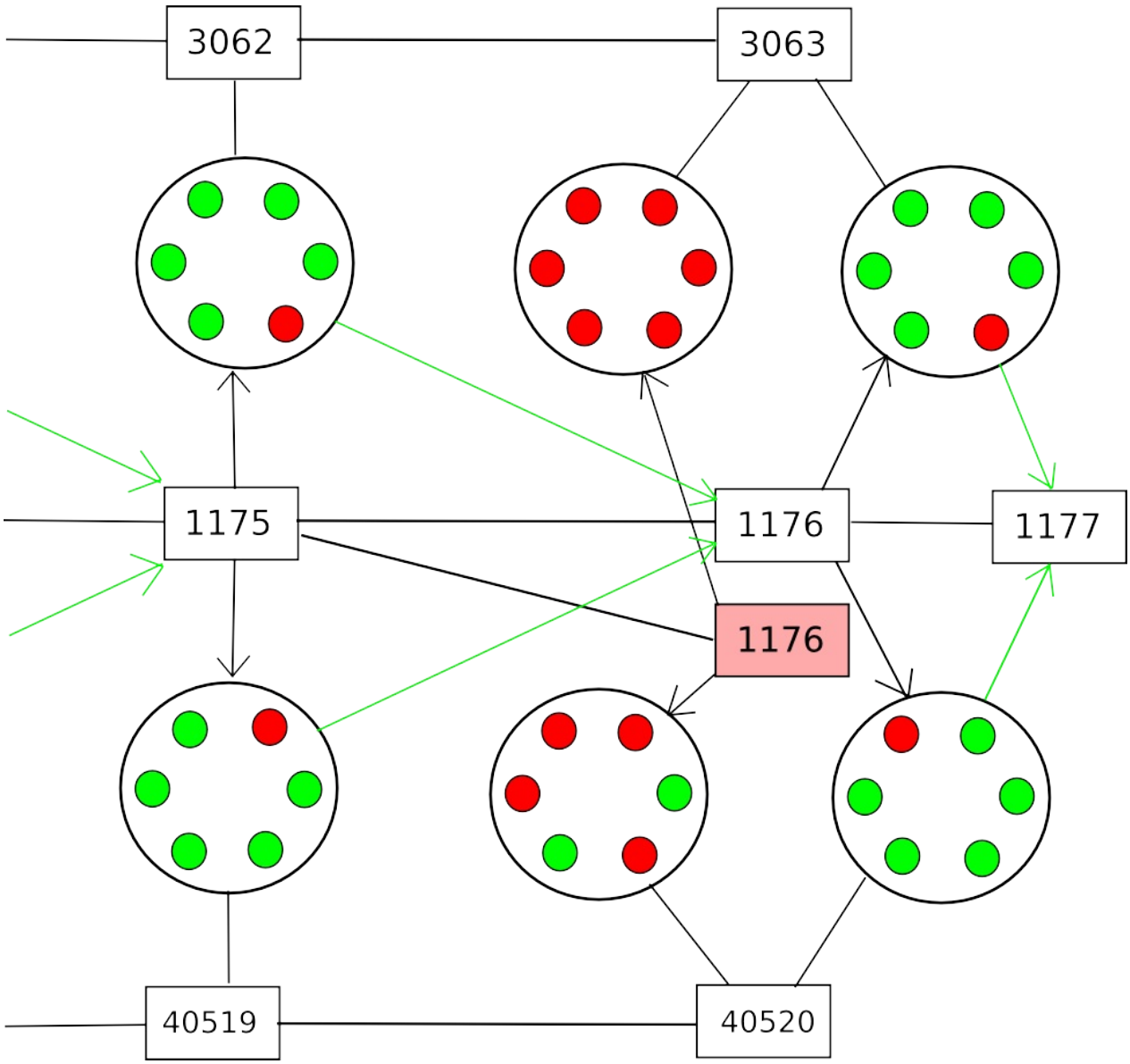
Schelling coin collective payoff



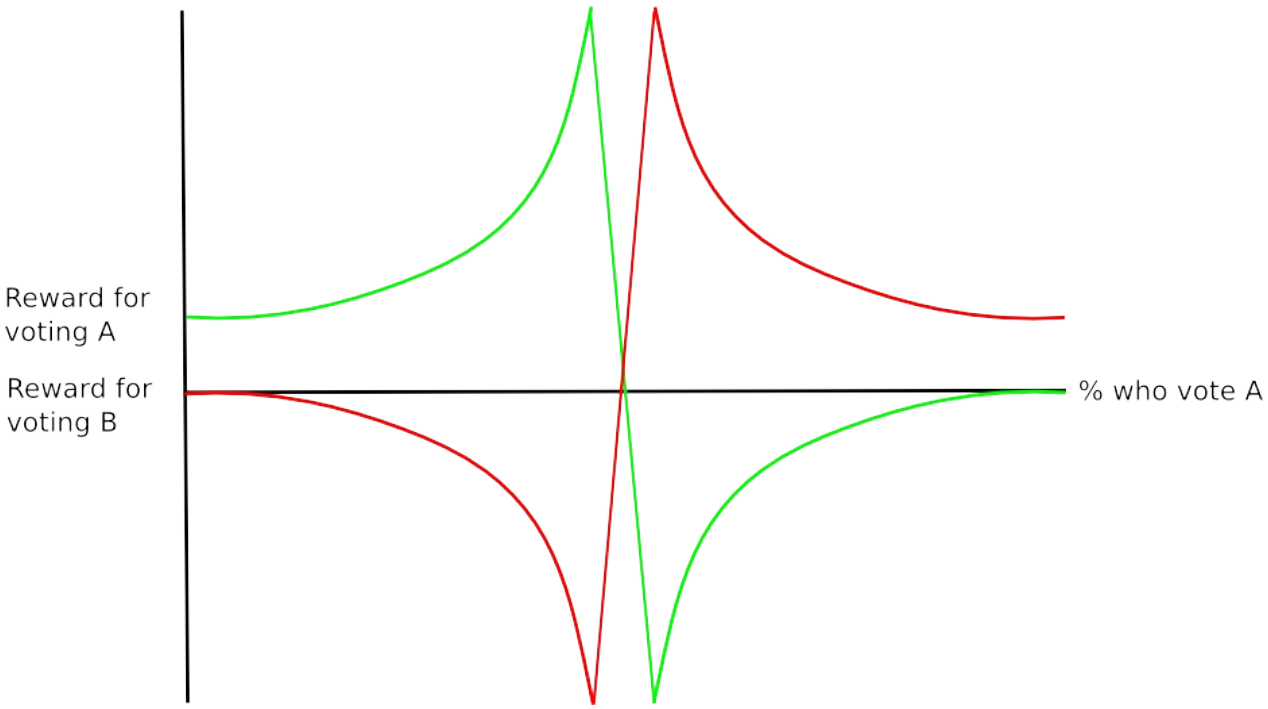
Schelling coin



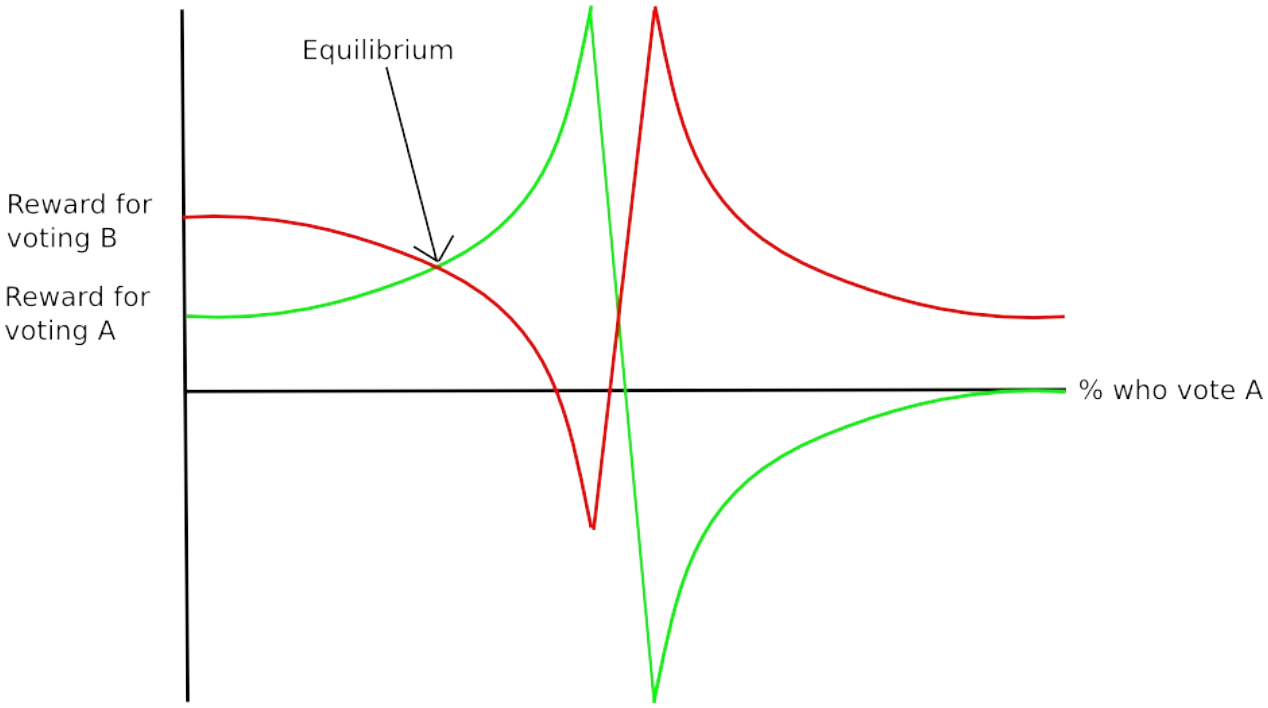
Schelling vote



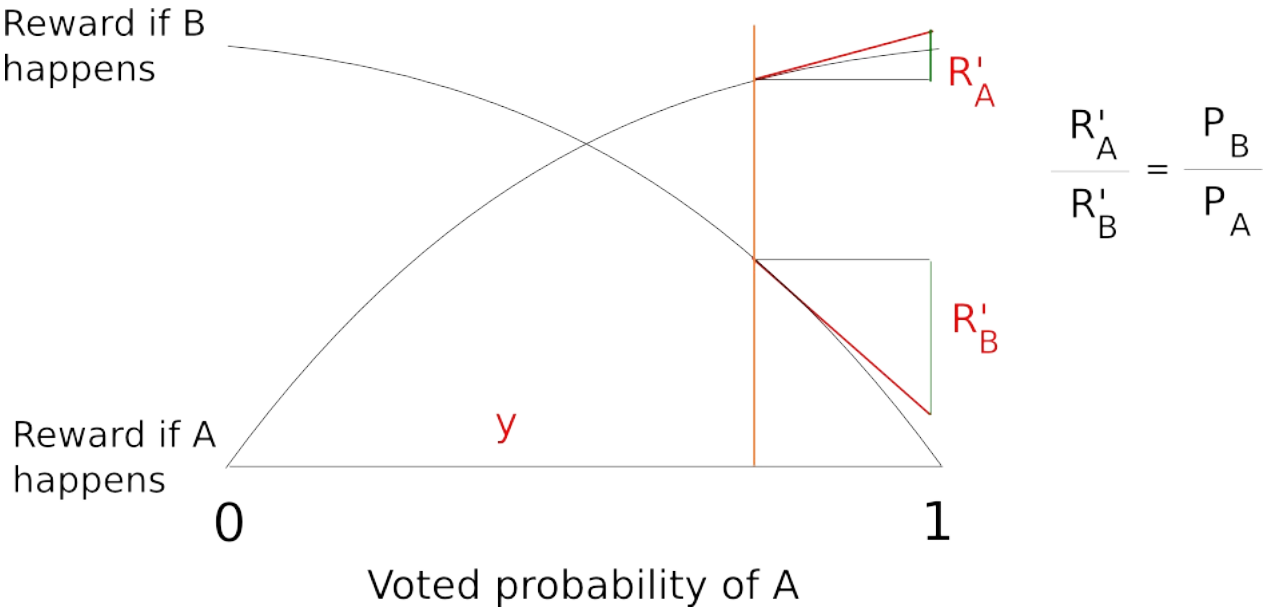
Schelling coin payoff 1



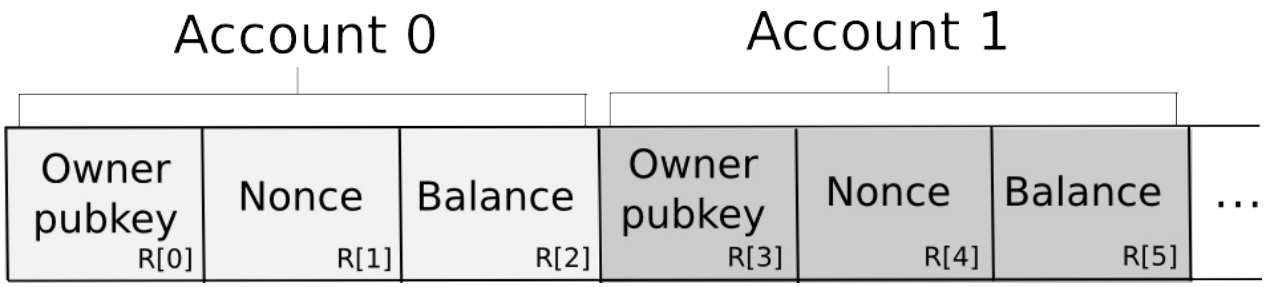
Schelling coin payoff 2



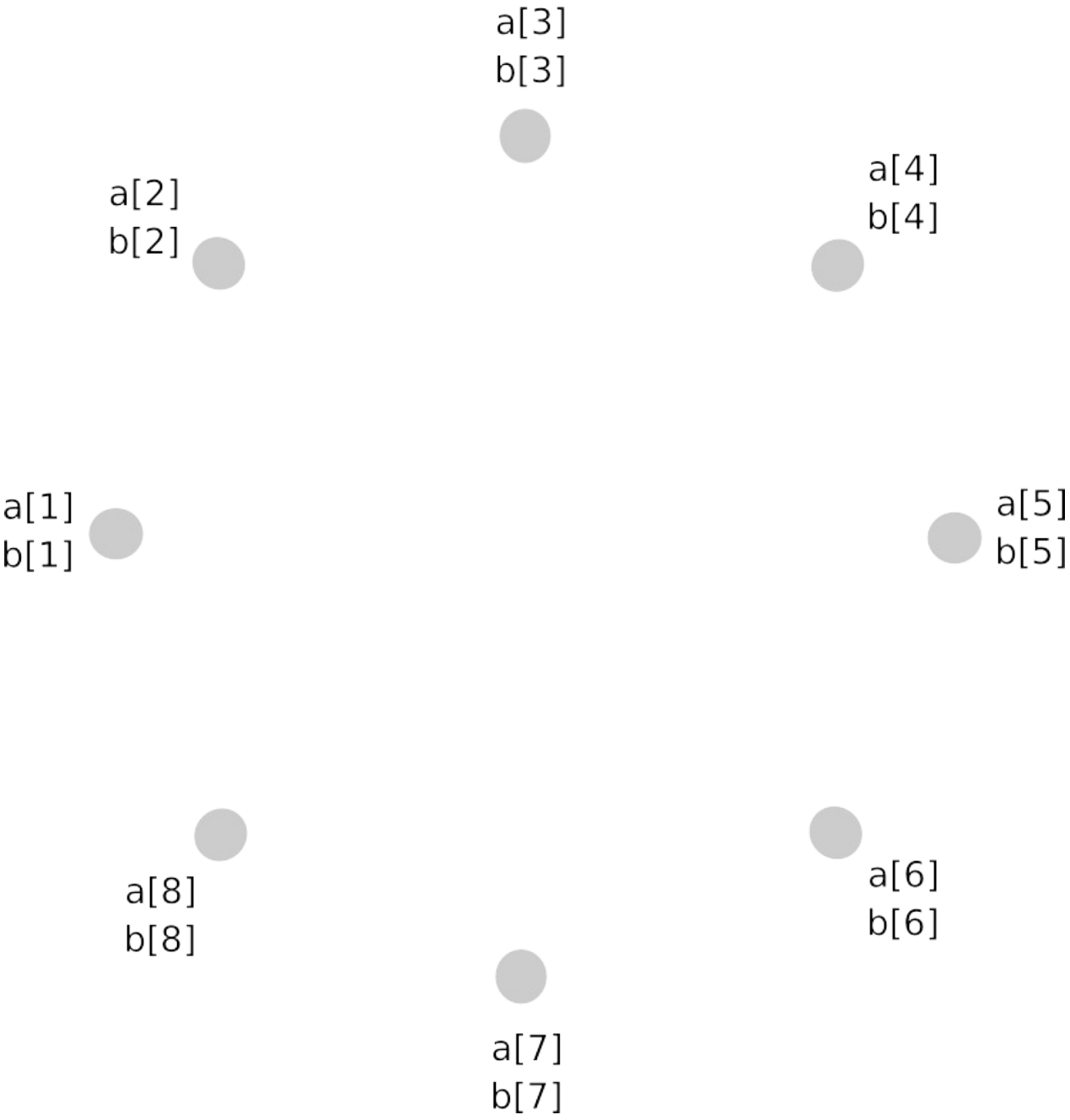
Scoring rule



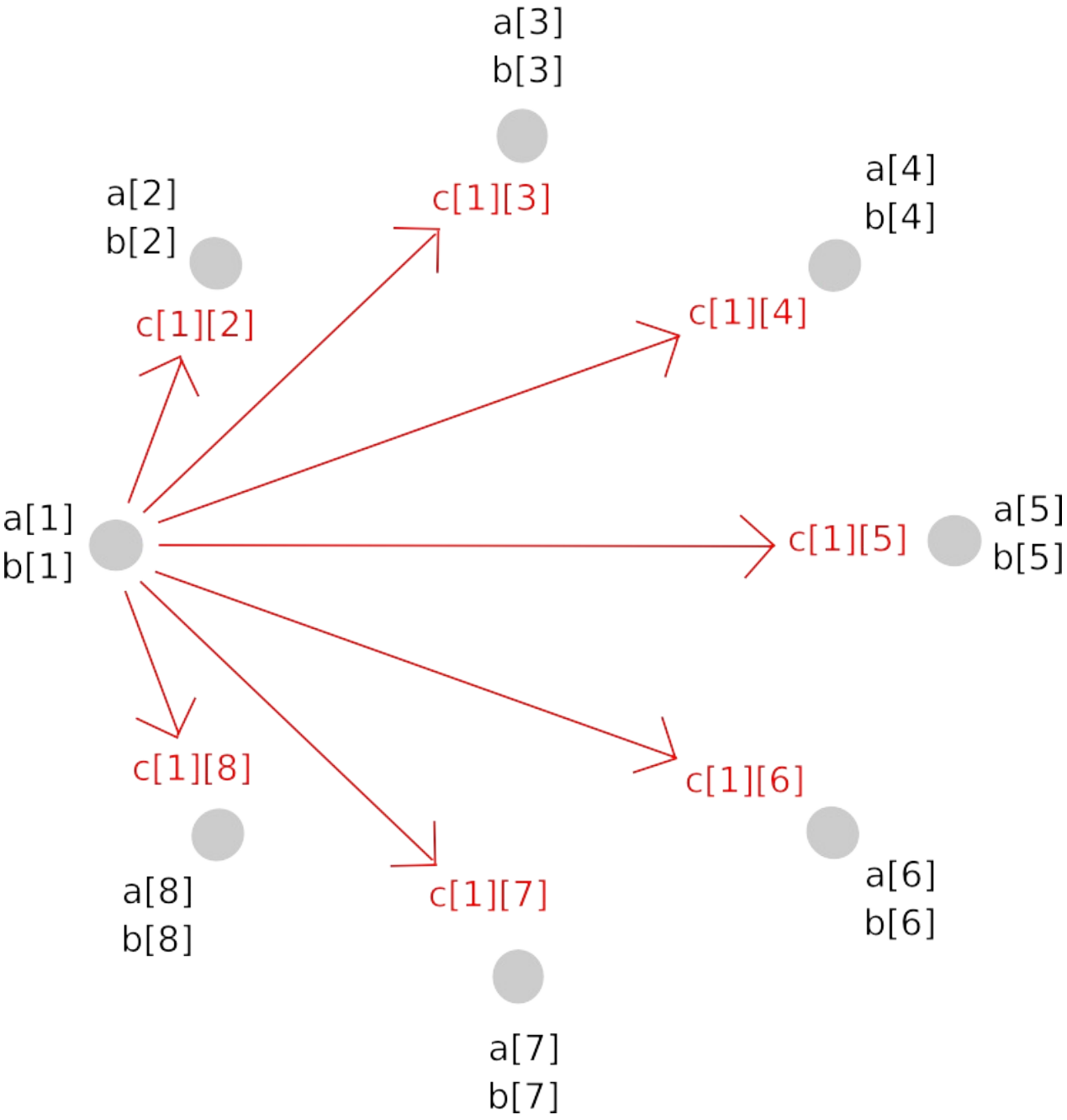
SecDAO accounts



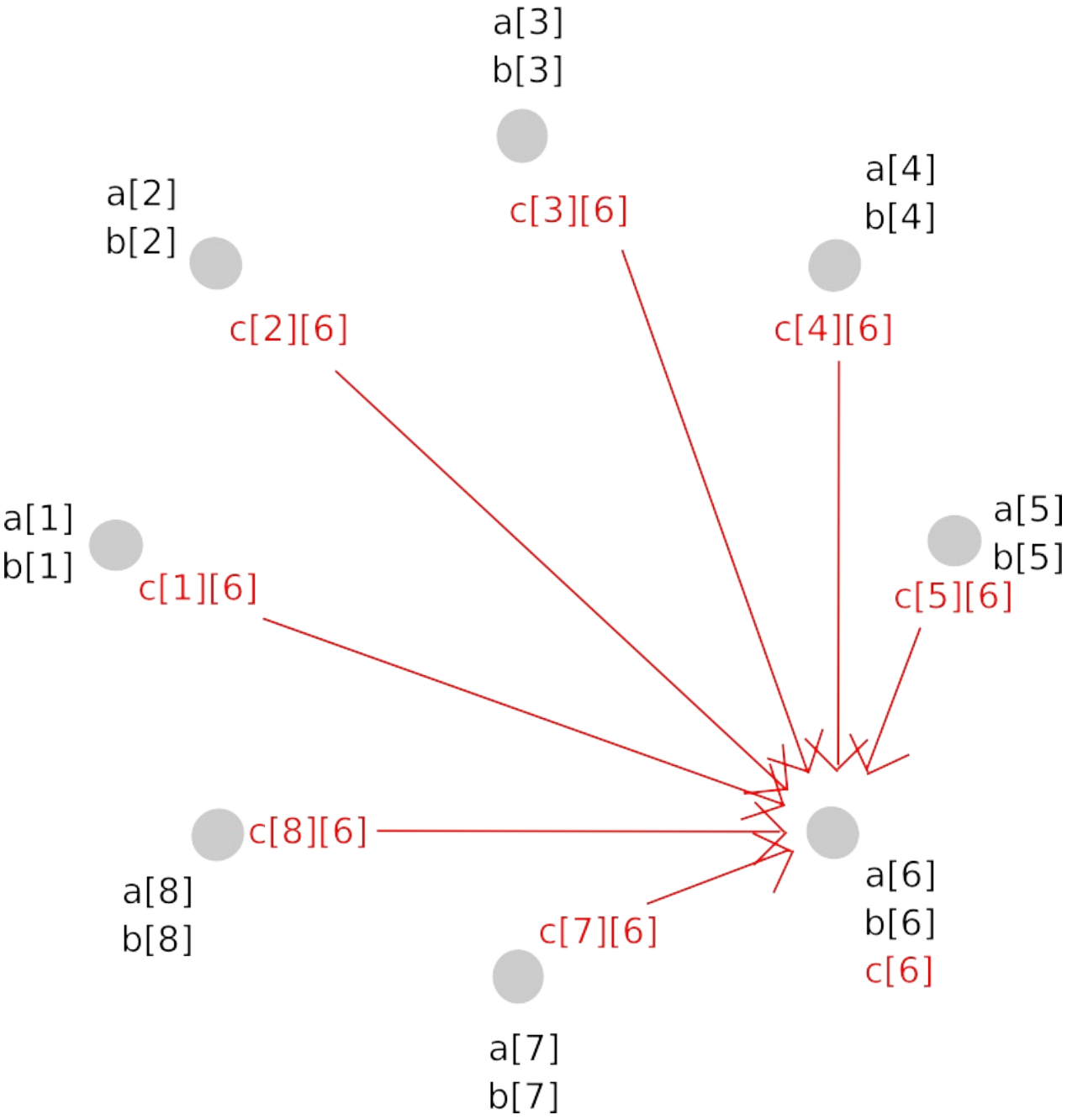
Secret multiply



Secret multiply 2



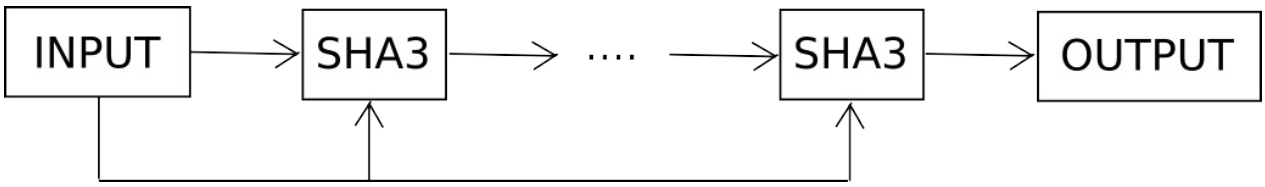
Secret multiply 3



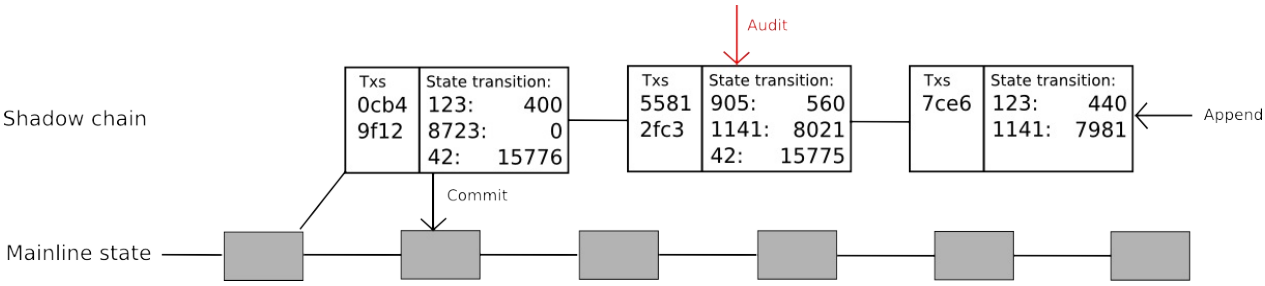
Serpent code

```
if !contract.storage[1000]:
    contract.storage[MYCREATOR] = 10^18
    contract.storage[1000] = 1
else:
    from = msg.sender
    to = msg.data[0]
    value = msg.data[1]
    if contract.storage[from] >= value:
        contract.storage[from] = contract.storage[from] - value
        contract.storage[to] = contract.storage[to] + value
```

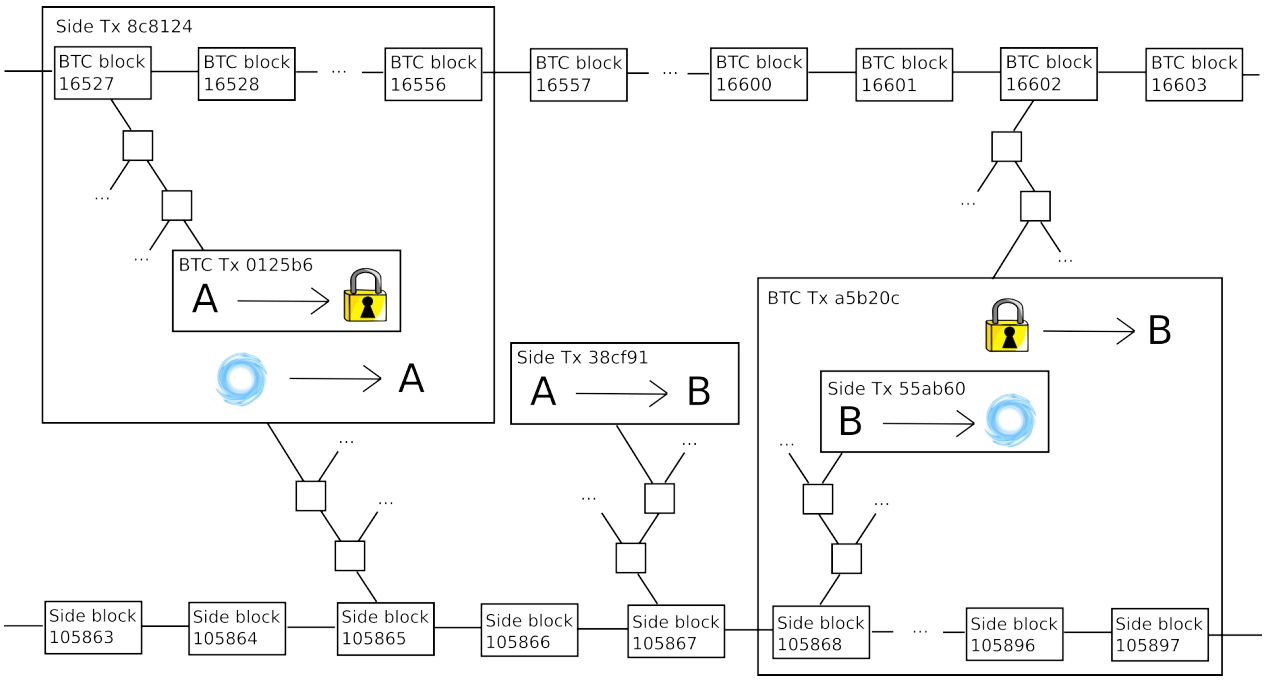
SHA3 repeat



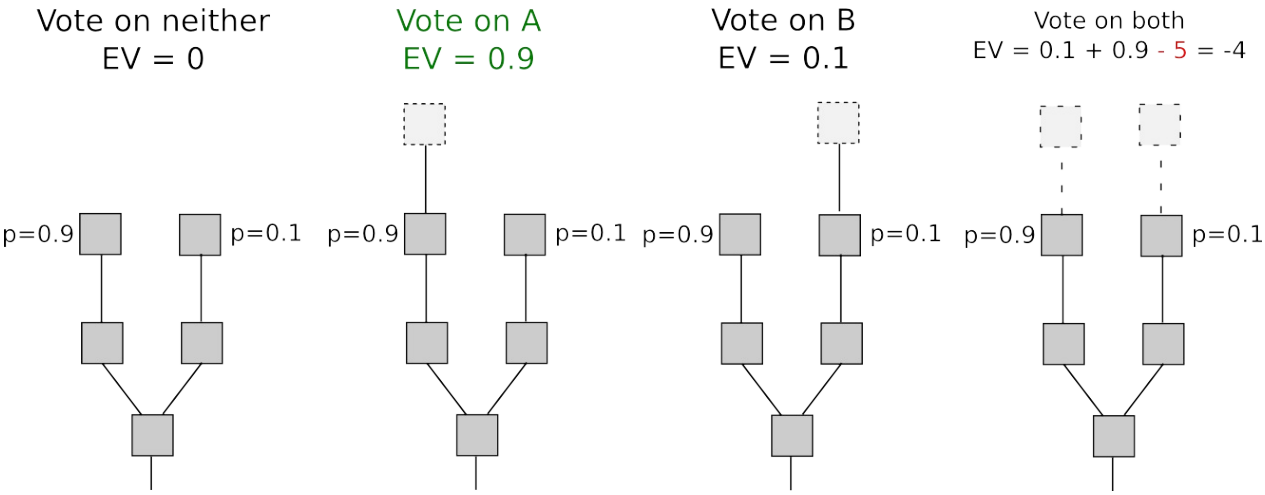
Shadowchain



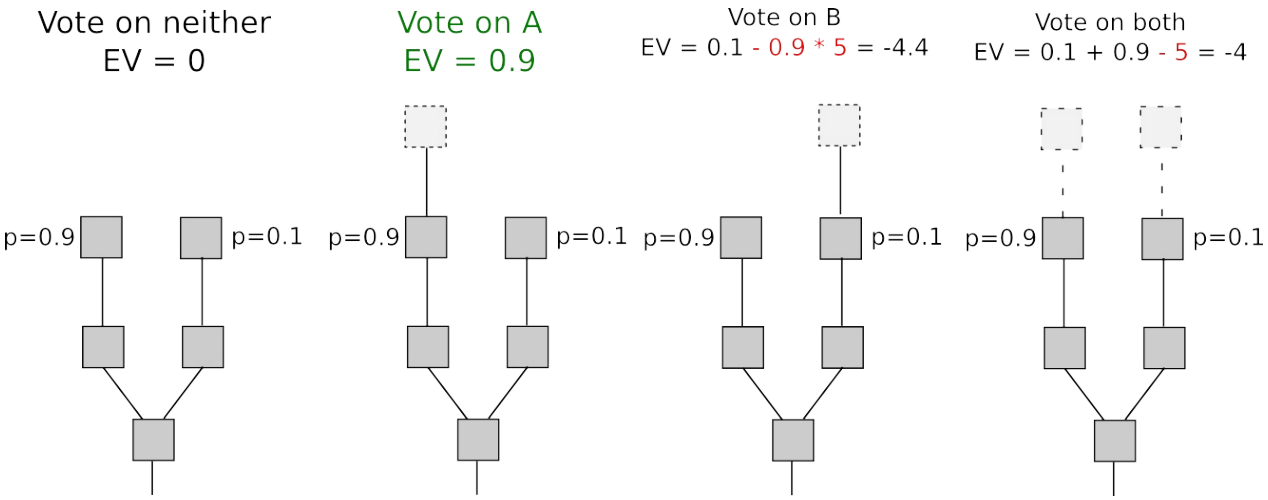
Sidechains



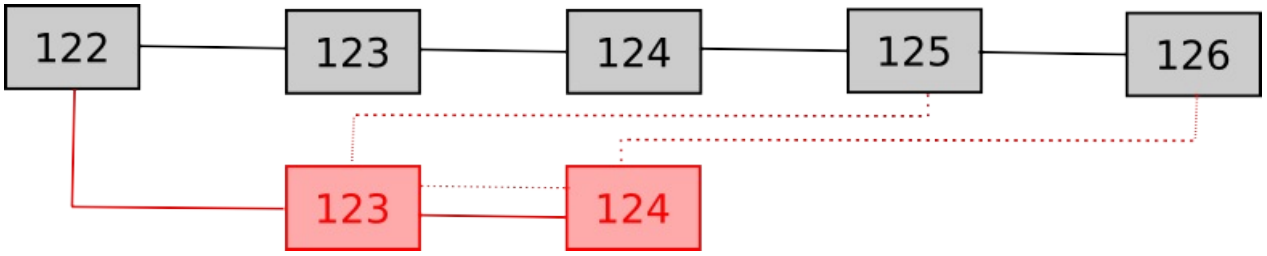
Slasher 1sec



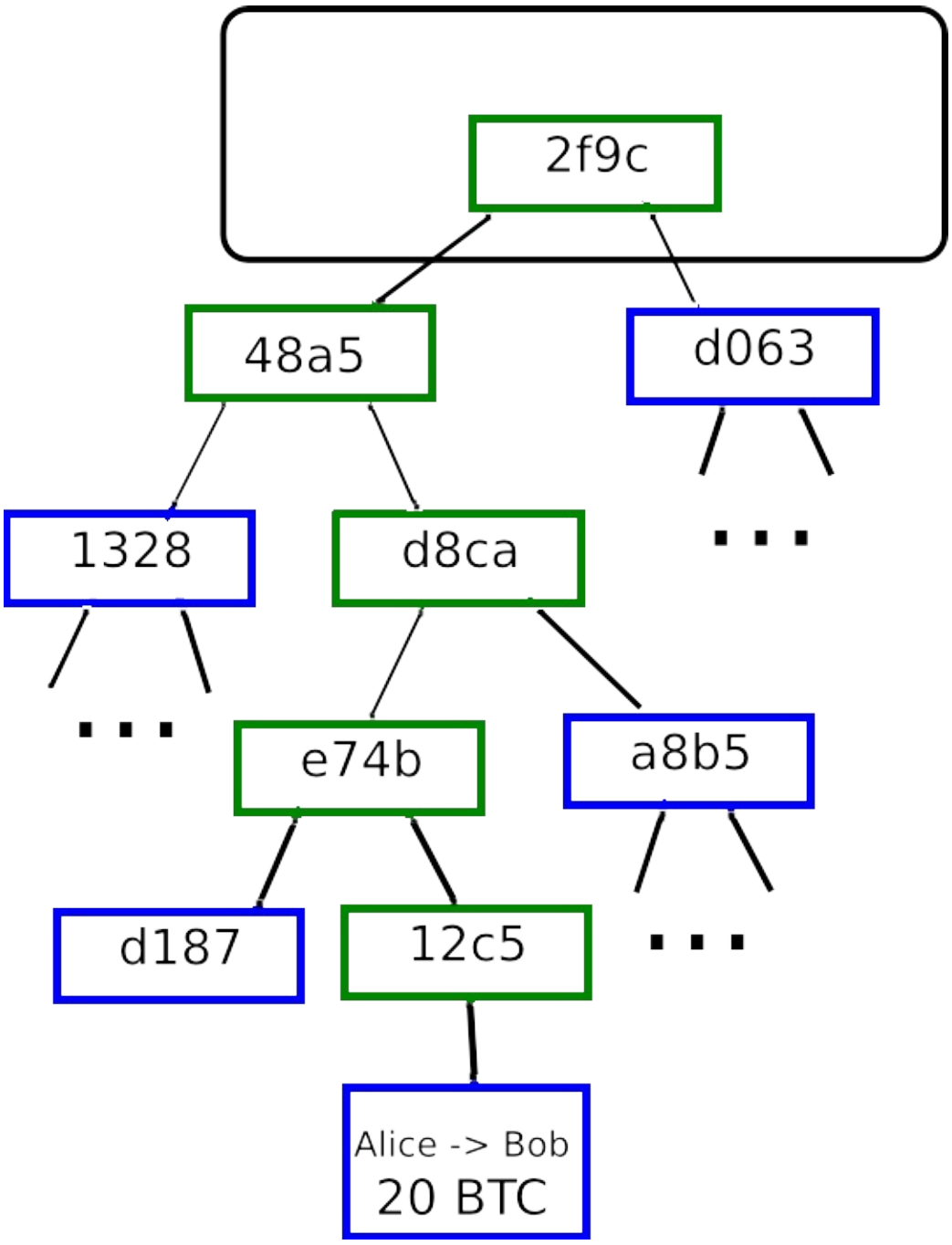
Slasher 2sec



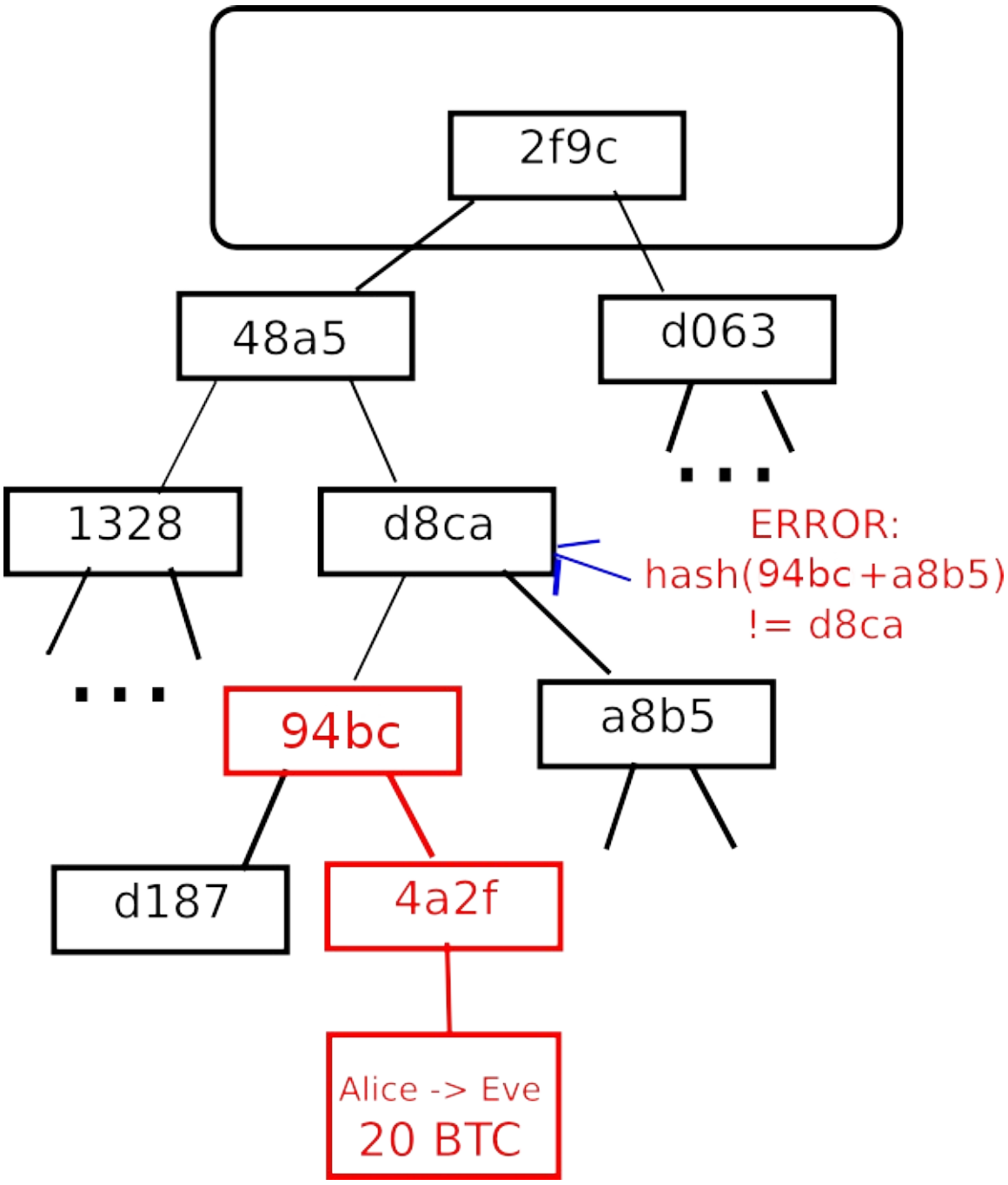
Slasher ghost



SPV1



SPV2


2f9c
48a5
d063
1328
d8ca
ERROR:
hash(94bc+a8b5)
!= d8ca
94bc
a8b5
d187
4a2f
Alice -> Eve
20 BTC

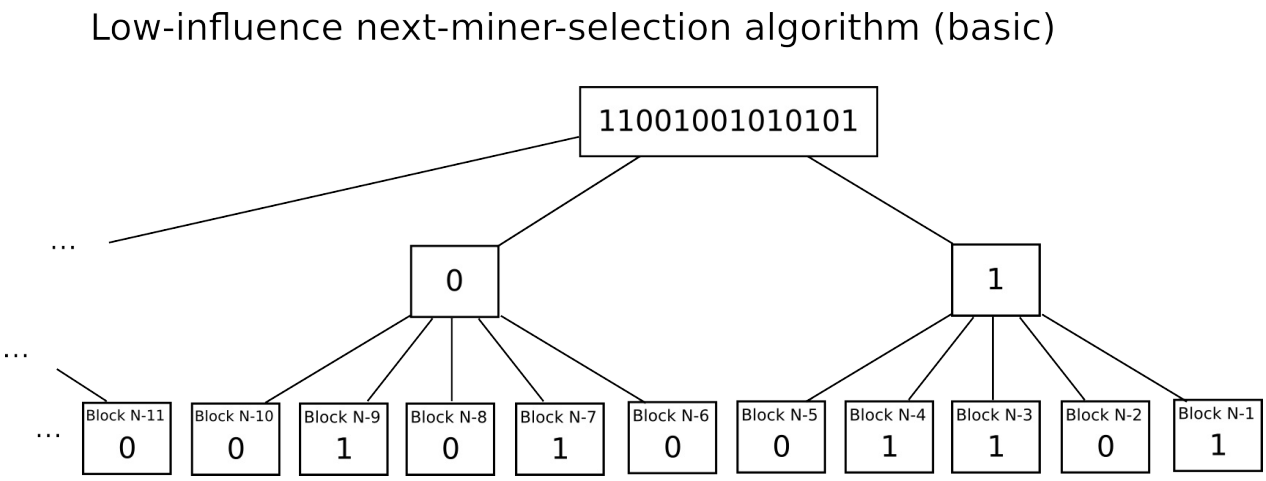
Stack trace

PC	45
STACK	32
	1664
	0
	0
	0
	0x57b4...
	1800
MEM	ac0981074060 0c31fa69f9db 79ed6fc3e328 1f758a950fe1 fb254a3a3ae5 71b66793a...

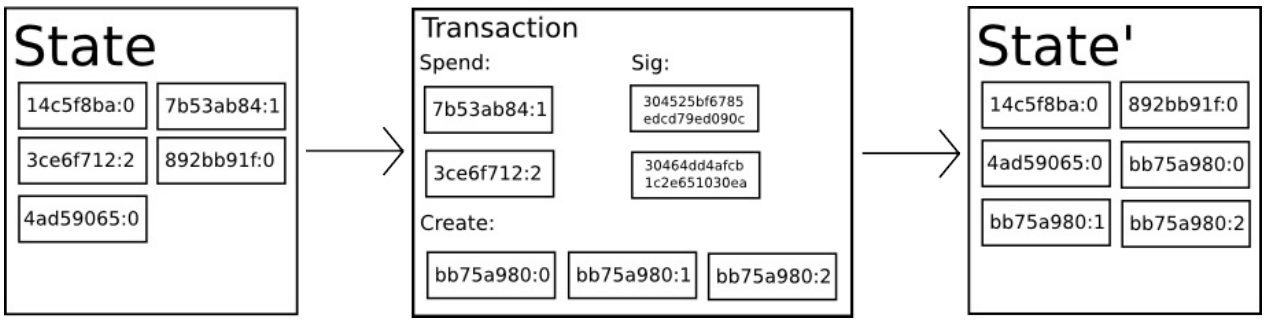
PC	208
STACK	111
	143
	0
MEM	631897788617 bc63fb0292c4 4f57a6c3824e 4264bc88fb4e 3c08b910f3f4 17fc12641...

PC	4
STACK	111
	111
MEM	

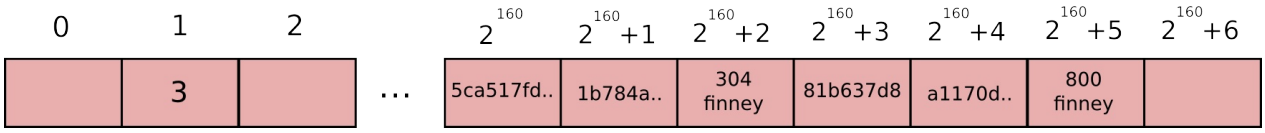
Stake low influence



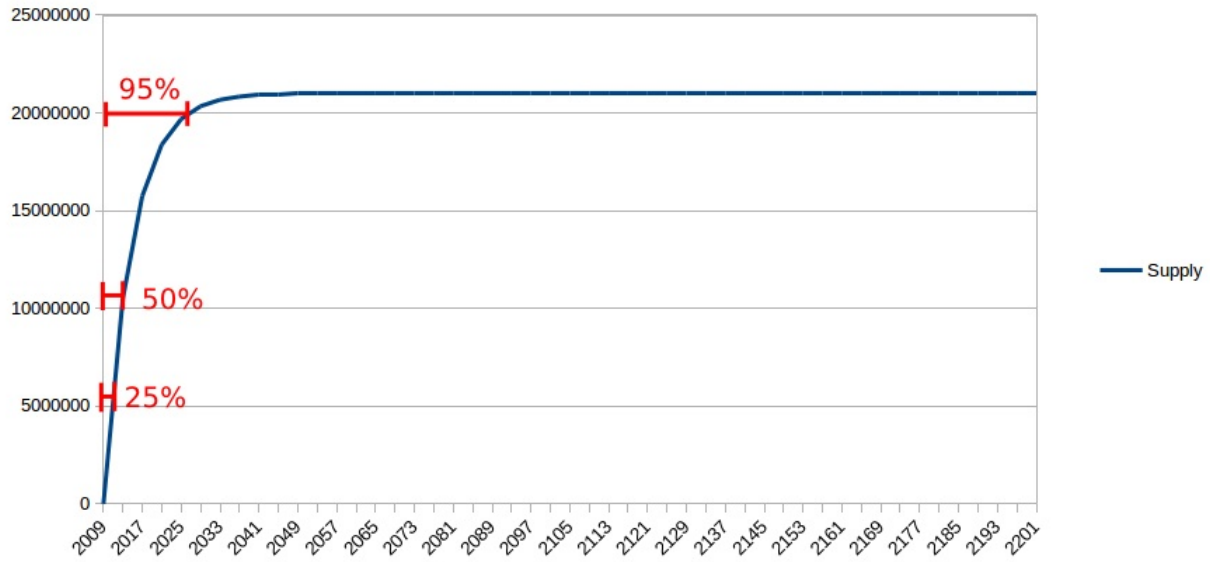
State transition



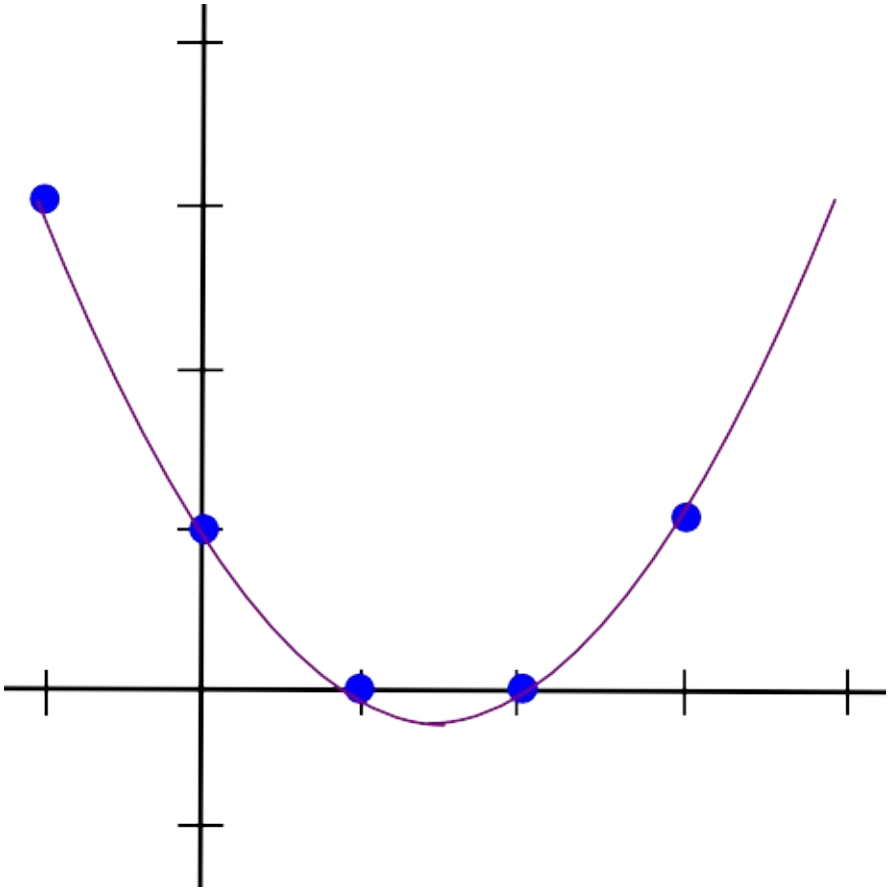
Strip



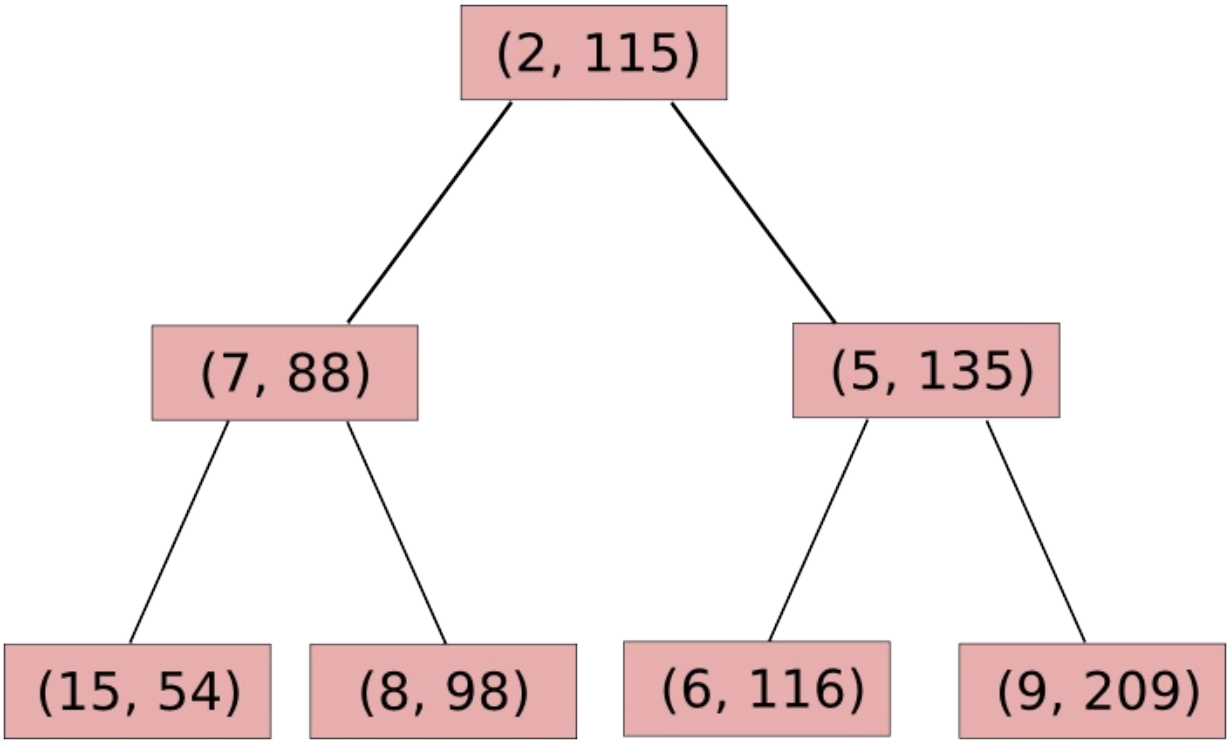
Supply chart



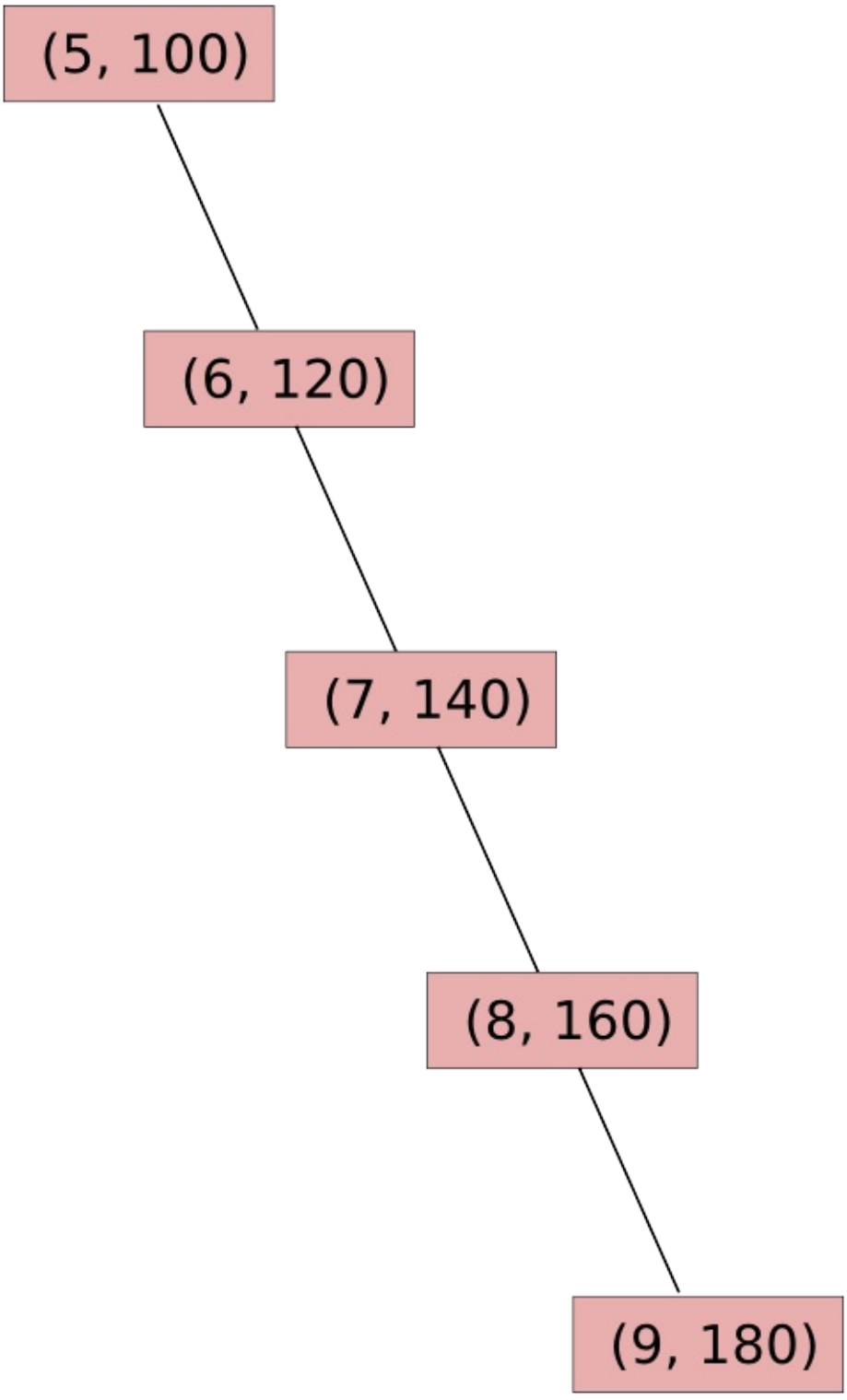
Three points



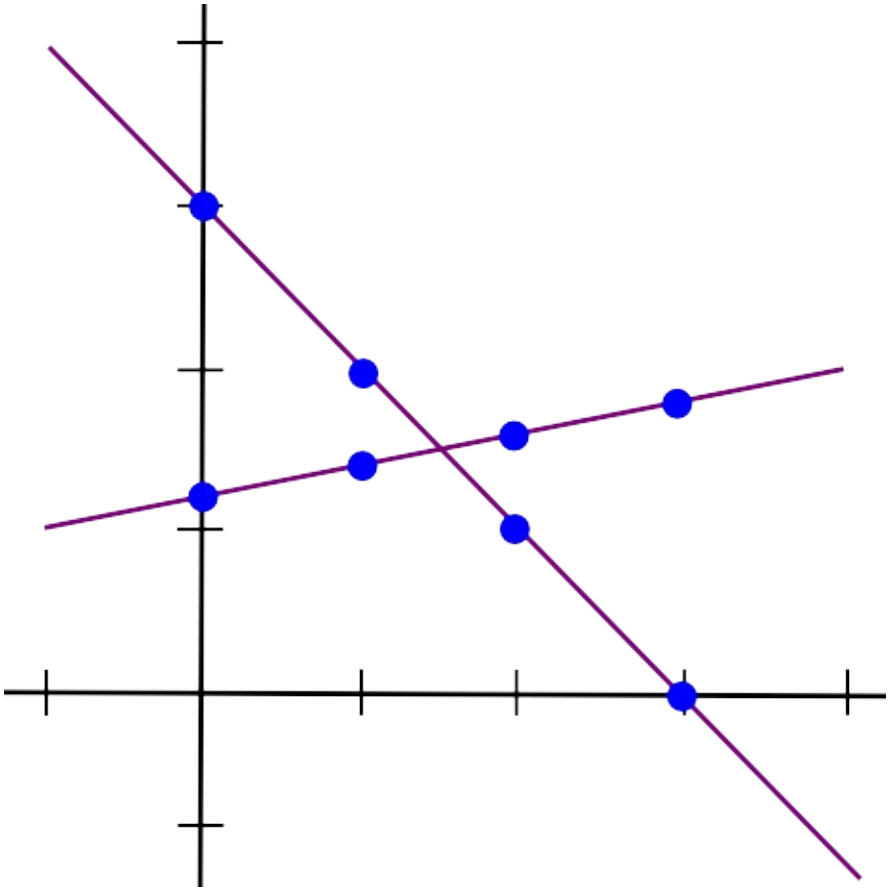
Treap



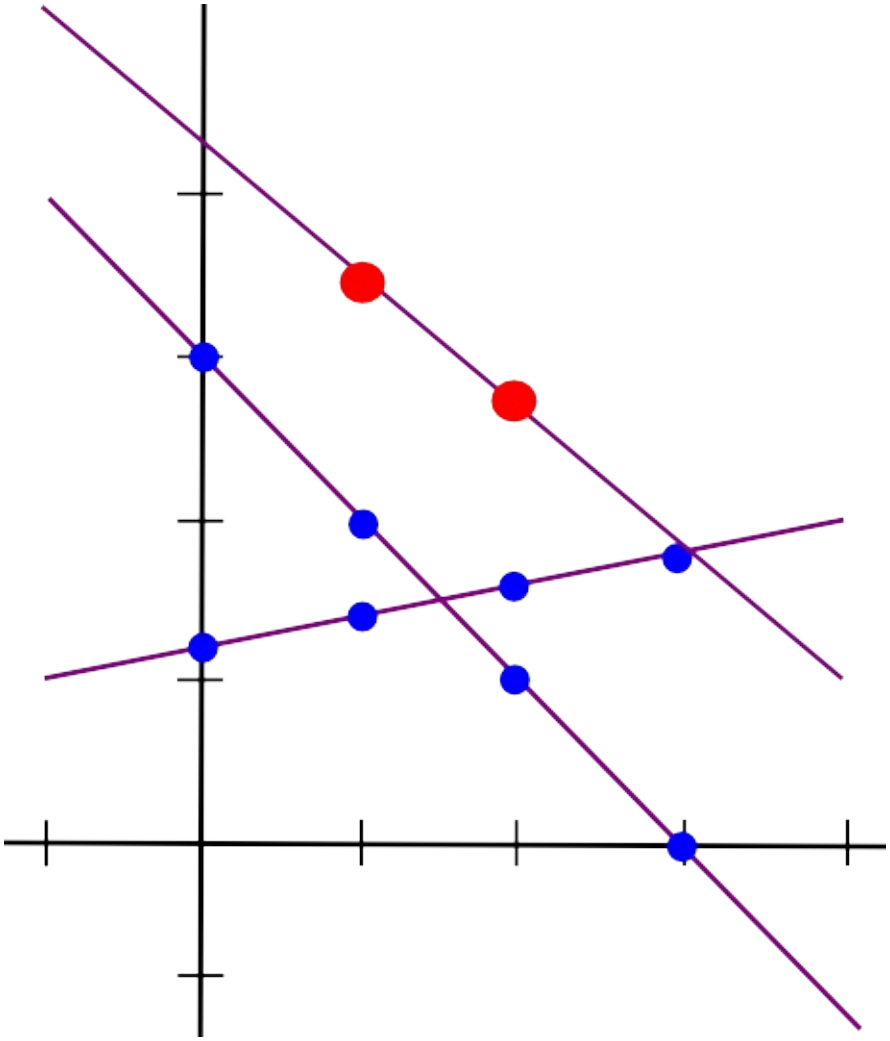
Treap 2



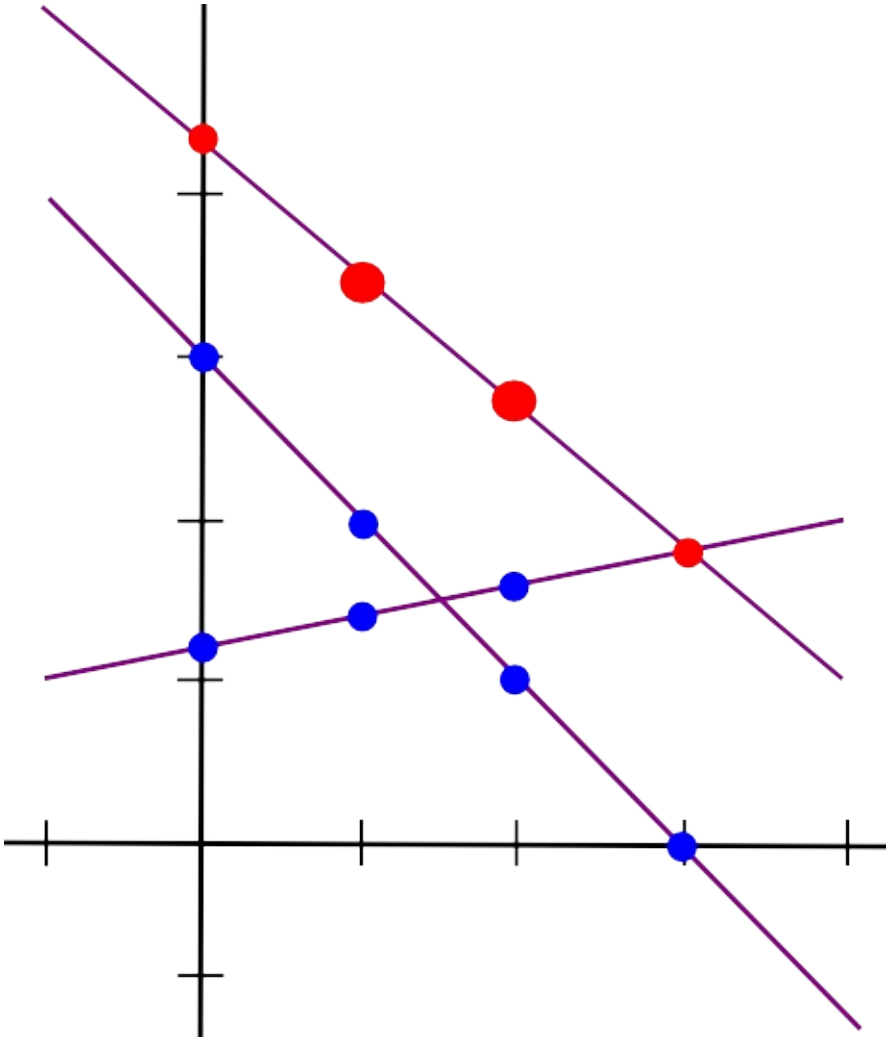
Twolines



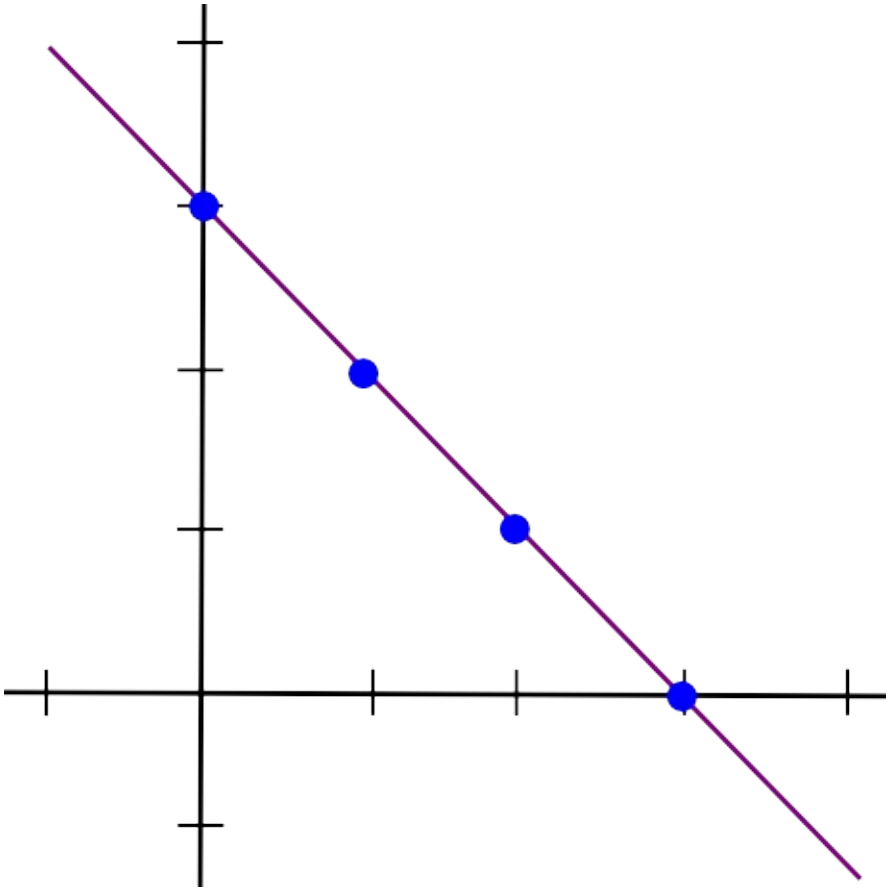
Twoline sum



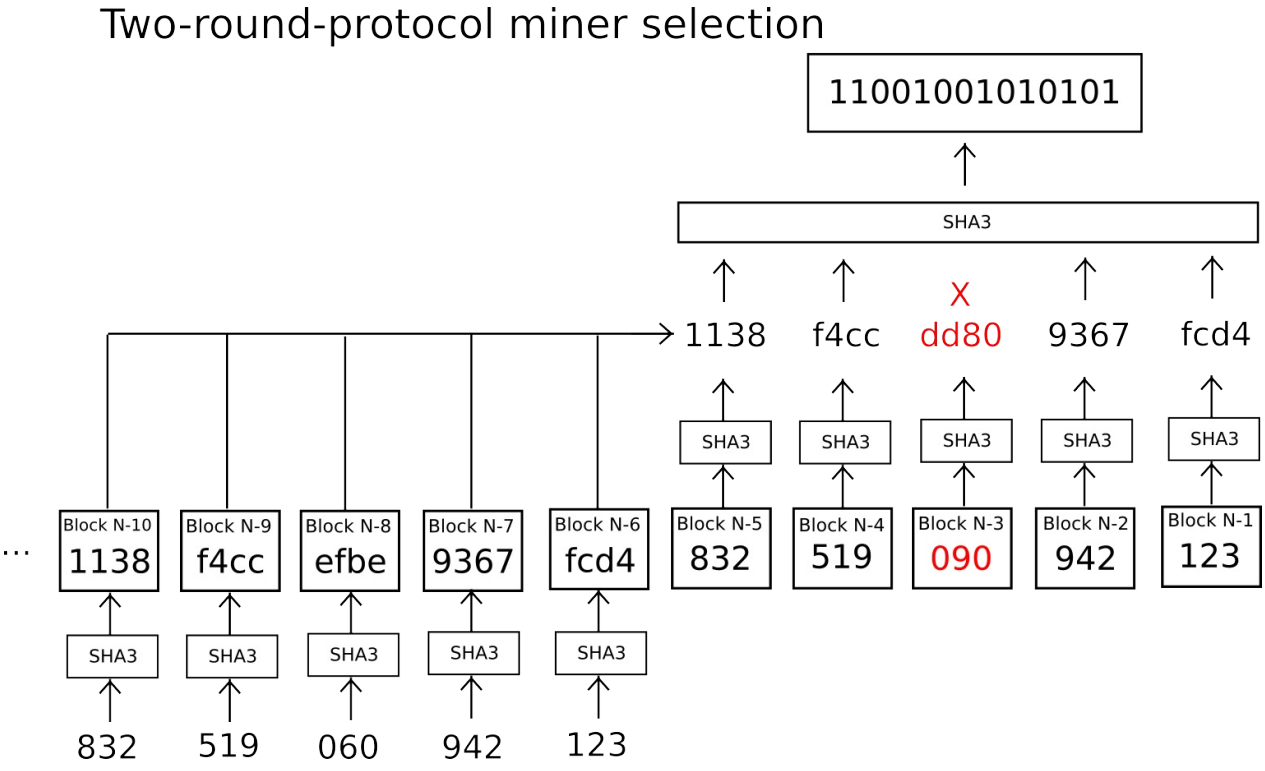
Twoline sum 2



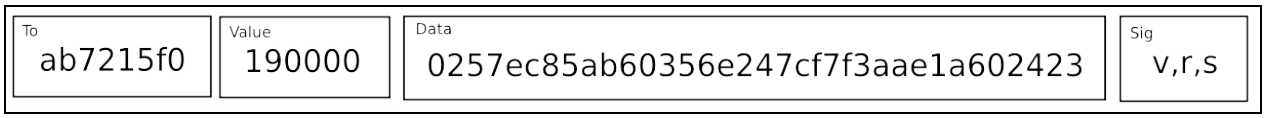
Twopoints



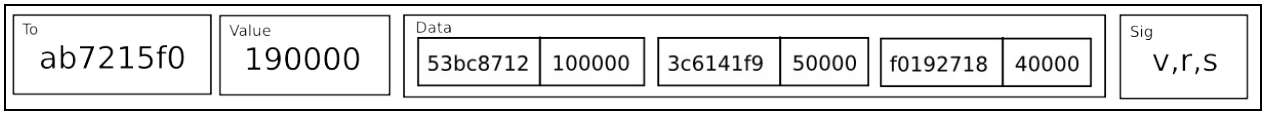
Tworound


Two-round-protocol miner selection
11001001010101
SHA3
1138
f4cc
X
dd80
9367
fcd4
SHA3
SHA3
SHA3
SHA3
SHA3
Block N-5
832
Block N-4
519
Block N-3
090
Block N-2
942
Block N-1
123
...
Block N-10
1138
Block N-9
f4cc
Block N-8
efbe
Block N-7
9367
Block N-6
fcd4
SHA3
SHA3
SHA3
SHA3
SHA3
832
519
060
942
123

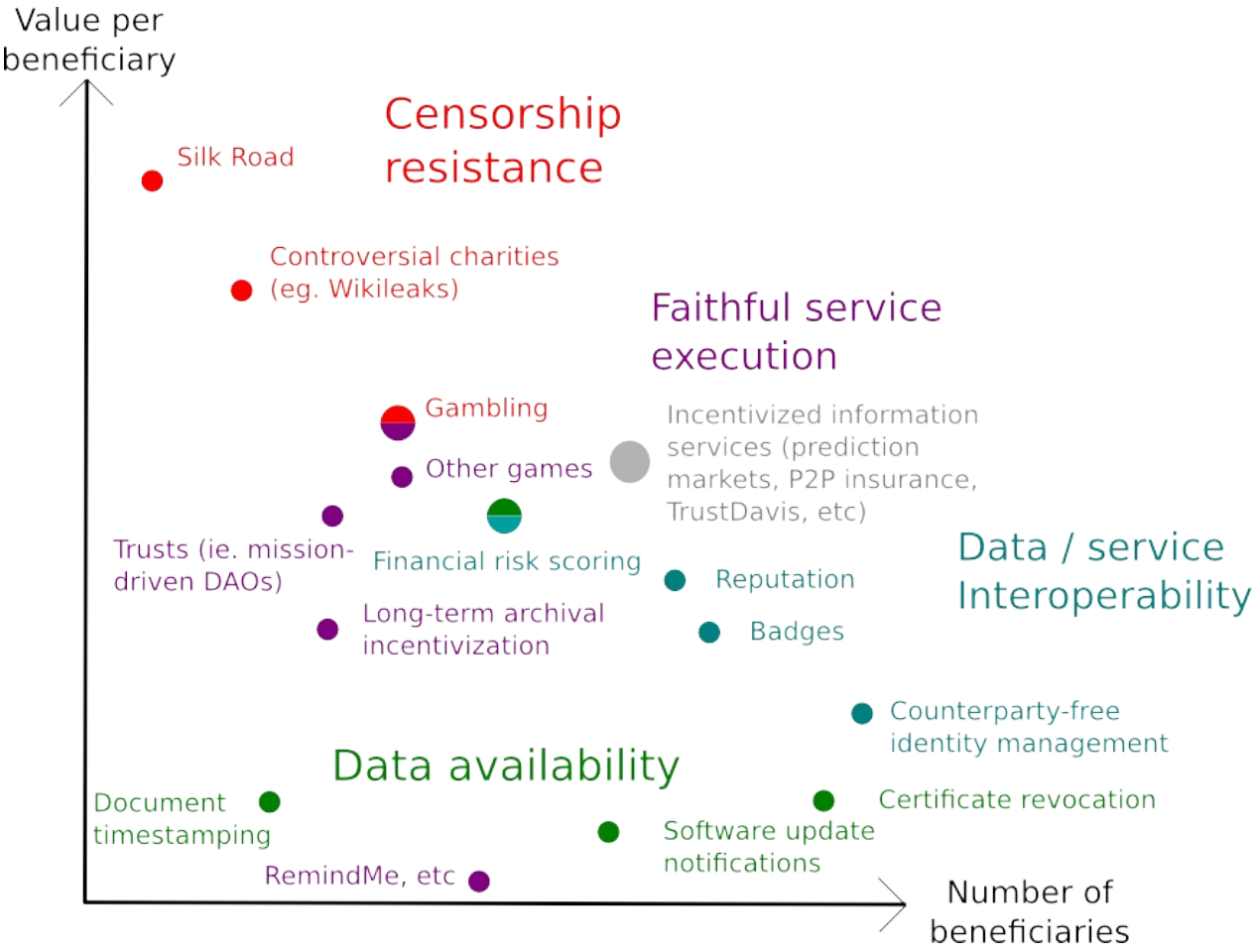
TX



TX batch



Utility chart



Where to mine