



**Libreta:** Primera Libreta  
**Creado:** 09/07/2018 21:18  
**URLOrigen:** https://aithub.com/ethereum/wiki/wiki/Design-Rationale



search-shortcut-hint.svg  
09/07/2018 21:20, 413 B

Pull requests  
Issues  
Marketplace  
Explore



ethereum / wiki

Watch 1,169 1

Code Issues 6 Pull requests 0 Projects 0 Wiki Insights

# Design Rationale

Shun Usami edited this page 27 days ago · 30 revisions

Ed

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.

## Contents

- Principles
- Blockchain-level protocol
- Accounts and not UTXOs
- Merkle Patricia Trees
- RLP
- Compression algorithm
- Trie Usage
- Uncle incentivization
- Difficulty Update Algorithm
- Gas and Fees
- Virtual Machine

## Principles

The Ethereum protocol design process follows a number of principles:

- 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

Pages 205

### Basics

- Home
- Ethereum Whit
- Ethereum Intro
- Uses: DAOs an
- Getting Ether
- Releases
- FAQs
- Design Ratione
- EVM intro: Eth Paper, Beige P. EVM.
- Wiki for (old) v good introduc

### R&D

- Sharding intro: Compendium, roadmap
- Casper Proof-c compendium a
- Alternative blo randomness, e other research
- Hard Problems Cryptocurrency
- Governance

### Ethereum Virtual (EVM)

Ethereum clients wallets, dapp br other projects

App Developm

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

### Infrastructure

- Chain Spec For
- Inter-exchange Protocol
- URL Hint Proto
- NatSpec Deter
- Network Status
- Raspberry Pi
- Exchange Integ
- Mining
- Licensing
- Consortium Ch Development

### DEV Technologies

- RLP Encoding
- Node Discover
- EIP2p Wire f
- EIP2p White
- Web3 Secret S
- libp2p

### Ethereum Techno

- Patricia Tree
- Wire protocol
- Light client pro
- Subtleties
- Solidity Docum
- NatSpec Form:
- Contract ABI
- Bad Block Rep
- Bad Chain Can

### Ethash/Dashimot

- Ethash
- Ethash Yellow l
- Ethash C API
- Ethash DAG

### Whisper

- Whisper Propc
- Whisper Overv
- PoC-1 Wire pri
- PoC-2 Wire pri
- PoC-2 Whitepa

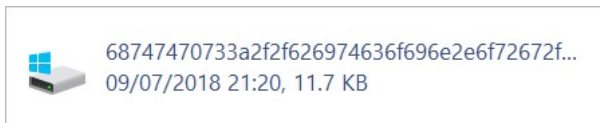
### Misc

- Glossary
- Chain Fibers (a sharding)
- Consensus Dat

Clone this wiki lo

Clone in Desktop

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.



(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

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  $\sim 2^{128}$  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 DoS 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:

[illegible]

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](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. For more details on Ethereum light clients and their use cases, see the [light client page \(principles section\)](#).

## 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 "intrinsic gas" fee for data is 4 gas per zero byte and 68 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



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 15000 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,  $x^1 \dots x^{255} = 2$  gas,  $x^{256} \dots x^{65535} = 3$  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  $\approx 50000^2$  computing effort, but only  $\sim 50000$  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 to 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  $\sim 0.0975\%$  ( $1/1024$ ) 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 if 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 see  $B.M[0] = 0$  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.
- **Having a 1024 call depth limit** - many programming 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,  $a * b \rightarrow (a * b) / 2^{32}$ ,  $a / b \rightarrow a * 2^{32} / b$ , 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,  $\text{mulmod}(a, b, c) = a * b \% c$ . 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/](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  $O(\log(\text{number of valid jump destinations}))$ , 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.
- **SELFDESTRUCT:** an opcode which allows a contract to quickly delete itself if it is no longer needed. The fact that SELFDESTRUCTs are processed at the end of transaction execution, and not immediately, is motivated by the fact that having the ability to revert SELFDESTRUCTs 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).

| [Deutsch](#) | [English](#) | [Español](#) | [Français](#) | [한국어](#) | [Italiano](#) | [Português](#) | [Română](#) | [فارسی](#) | [العربية](#) | [中文\(繁体\)](#) | [中文\(简体\)](#) | [日本語](#)

---

© 2018 GitHub, Inc.

[Terms](#)

[Privacy](#)

[Security](#)

[Status](#)

[Help](#)

[Contact GitHub](#)

[API](#)

[Training](#)

[Shop](#)

[Blog](#)

[About](#)