

# Ethereum Technical Specification

Micah Dameron

## Abstract

The Ethereum Protocol is a deterministic but practically unbounded state-machine with two basic functions; the first being a globally accessible singleton state, and the second being a virtual machine that applies changes to that state. This paper explains the individual parts that make up these two factors.

## 1. Imagining Bitcoin as a Computer

Ethereum borrows the crypto-economic model of a cryptographically-secure and instantiated ledger that originated with Bitcoin and repurposes it to model not just a ledger, but a stack-based virtual machine, complete with an assembly language, its own byte code, and many higher-level languages as well. With this machine one can execute machine codes with the same veracity and certainty that one has when sending a Bitcoin transaction. Just as sure as you can be certain of a Bitcoin transaction being the right balance, the right transfer, that things in the record are timestamped and actually *did in fact happen*, just as certain as the 1st, 2nd, 1000th, and 10,000th transactions on the Bitcoin network are the same transactions that happened nine years ago, without anyone altering the record, (for no alteration of the record is possible, nor has ever been possible) just so sure are the *machine instructions in assembly language that are given to the Ethereum interpreter*.

This has profound implications for all areas of society and cannot be explored fully in this context. But in short, *programs written and executed on the Ethereum Blockchain are unstoppable*. This does not mean that no loopholes in the code exist that could cause undesirable outcomes, rather it means that at the base level, the outcomes, whether desirable or not, happened, remain verifiable, are recorded, and are entirely uncensorable.

## 2. Native Currency & Mining

Because Ethereum aims not at being a currency, but at modeling a computer, there needs to be a *network cost unit* to mitigate the possibility of abusing the network with excessive computational expenditures. The smallest unit of currency in Ethereum is the Wei, which is equal to  $\Xi 10^{-18}$ , where  $\Xi$  stands for 1 Ether. All currency transactions in Ethereum are counted in Wei. There is also the Szabo, which is  $\Xi 10^{-6}$ , and the Finney, which is  $\Xi 10^{-3}$ .

Unit	Ether	Wei
Ether	$\Xi 1.000000000000000000$	1,000,000,000,000,000,000
Finney	$\Xi 0.001000000000000000$	1,000,000,000,000,000
Szabo	$\Xi 0.000001000000000000$	1,000,000,000,000
Wei	$\Xi 0.000000000000000001$	1

### 2.1. Mining

#### Apply Rewards

**Notation** : `apply_rewards`

**Description** : The third process in `block_finalization` that sends the mining reward to an account's address.

#### 2.1.1. Difficulty Mechanism

This mechanism enforces a homeostasis in terms of the time between blocks; a smaller period between

the last two blocks results in an increase in the difficulty level and thus additional computation required, lengthening the likely next period. Conversely, if the period is too large, the difficulty, and expected time to the next block, is reduced. The *Total Computation* is the difficulty state of the entire Ethereum blockchain. The *Block Difficulty* is not a state of the blockchain, but is local-particular to each specific block. You reach the total difficulty by summing the difficulty of all previous blocks and then adding the present one.

The **GHOST Protocol** provides an alternative solution to double-spend attacks from the original solution in Satoshi Nakamoto's Bitcoin Whitepaper. Nakamoto solved the problem of double-spending by requiring the network to agree on the first valid block. impossible to submit a "double-spend" block without having at least 50% of the network's mining power to force the longest chain. This is because the network automatically chooses the longest chain. So even if one wanted to submit two spend transactions in a row, the network simply picks whichever one comes first, ignoring the second because it no longer pertains to the longest chain (which now contains the first block that was sent) so the would-be hacker needs to submit a new block, as the first double block is no longer feasible.

## 3. Memory and Storage

### 3.1. World State

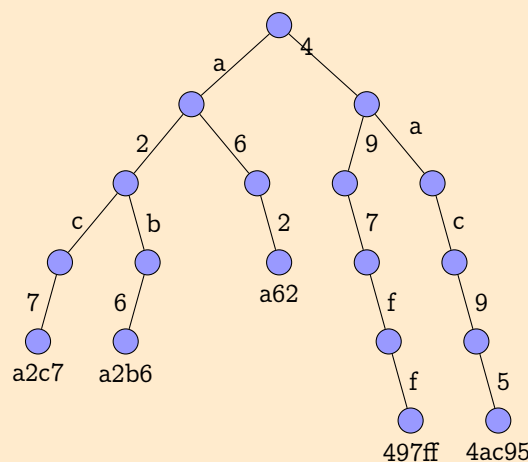
The actual state of the Ethereum Blockchain, after all transactions have been finalized and applied, this is a MAPPING of addresses and account states through the use of RLP. The mapping is stored as a Merkle-Patricia trie in a DATABASE BACKEND.<sup>a</sup> that maintains a mapping of bytearrays to bytearrays.<sup>b</sup> The cryptographic internal data going back to the **root node** represents the *State* of the Blockchain at any given root, i.e. at any given *time*.<sup>c</sup> As a whole, the

state is the sum total of database relationships in the **state database**. The state is an inert position on the chain, a position between prior state and post state; a block's frame of reference, and a defined set of relationships to that frame of reference.

#### 3.1.1. Merkle-Patricia Trees

Merkle-Patricia Trees<sup>d</sup> are modified Merkle trees where nodes represent individual characters from hashes rather than each node representing an entire hash. This allows the state data structure itself to represent not only the intrinsically correct paths in the data, but also the requisite cryptographic proofs which go into making sure that a piece of data was valid in the first place. In other words, it keeps the blockchain valid by combining the structure of a standard Merkle-tree with the structure of a Radix Tree. Since all searching and sorting algorithms in Ethereum must be filtered through this stringently correct database, accuracy of information is guaranteed.

The following is a search tree beginning with hexadecimal values a and 4:



#### 3.1.2. Tree Terminology<sup>1</sup>

- a) **Root Node** – The top (first) node in a tree.
- b) **Child Node** – A node directly connected to another node when moving away from the Root.

<sup>a</sup>The database backend is accessed by users through an external application, most likely an Ethereum client; see also: **state database**

<sup>b</sup>A bytearray is specific set of bytes [data] that can be loaded into memory. It is a structure for storing binary data, e.g. the contents of a file.

<sup>c</sup>This permanent data structure makes it possible to easily recall any previous state with its root hash keeping the resources off-chain and minimizing on-chain storage needs.

<sup>d</sup>Also known as "tries"

- c) **Parent Node** – The converse notion of a child.
- d) **Sibling Nodes** – A group of nodes with the same parent.
- e) **Descendant Node** – A node reachable by repeated proceeding from parent to child.
- f) **Ancestor Node** – A node reachable by repeated proceeding from child to parent.
- g) **Leaf Node** – A node with no children.
- h) **Branch Node** – A node with at least one child.
- i) **Degree** – The number of subtrees of a node.
- j) **Edge** – The connection between one node and another.
- k) **Path** – A sequence of nodes and edges connecting a node with a descendant.
- l) **Level** – The level of a node is defined by 1 + (the number of connections between the node and the root).
- m) **Node Height** – The height of a node is the number of edges on the longest path between that node and a leaf.
- n) **Tree Height** – The height of a tree is the height of its root node.
- o) **Depth** – The depth of a node is the number of edges from the tree's root node to the node.
- p) **Forest** – A forest is a set of  $n \geq 0$  disjoint trees.

### 3.1.3. Recursive Length Prefixes

**Notation** : rlp

**Description** : RLP encodes arrays of nested binary data to an arbitrary depth; it is the main serialization method for data in Ethereum. RLP encodes mainly structure and does not pay heed to what type of data it is encoding.

Positive RLP integers are represented with the most significant value stored at the lowest memory address (big endian) and without any leading zeroes. As a result, the RLP integer value for 0 is represented by an empty byte-array. If a non-empty deserialized integer begins with leading zeros it is invalid.<sup>2</sup>

The global state database is encoded as RLP for fast traversal and inspection of data. In structure it

constitutes a mapping between *addresses* and *account states*. Since it is stored on node operator's computers, the tree can be traversed speedily and without network delay. RLP encodes values as byte-arrays, or as sequences of further values.<sup>3</sup>

This means that:

```

if   rlp(x)           = bytearray
then rlp(bytearray)  = true
elif rlp(x)           = value
then rlp(value)      = true
elif rlp(x)           = null
then rlp(x)          = false

```

1. If the RLP-serialized byte-array contains a single byte integer value less than 128, then the output is exactly equal to the input.

## 3.2. The Block

A block is made up of 17 different elements. The first 15 elements are part of what is called the *block header*.

### 3.2.1. The Block Header

**Description** : The information contained in a block besides the transactions list. This consists of:

1. **Parent Hash** – This is the Keccak-256 hash of the parent block's header.
2. **Ommers Hash** – This is the Keccak-256 hash of the ommer's list portion of this block.
3. **Beneficiary** – This is the 20-byte address to which all block rewards are transferred.
4. **State Root** – This is the Keccak-256 hash of the root node of the state trie, after a block and its transactions are finalized.
5. **Transactions Root** – This is the Keccak-256 hash of the root node of the trie structure populated with each transaction from a Block's transaction list.
6. **Receipts Root** – This is the Keccak-256 hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block.

7. **Logs Bloom** – This is the bloom filter composed from indexable information (log address and log topic) contained in the receipt for each transaction in the transactions list portion of a block.
8. **Difficulty** – This is the difficulty of this block – a quantity calculated from the previous block’s difficulty and its timestamp.
9. **Number** – This is a quantity equal to the number of ancestor blocks behind the current block.
10. **Gas Limit** – This is a quantity equal to the current maximum gas expenditure per block.
11. **Gas Used** – This is a quantity equal to the total gas used in transactions in this block.
12. **Timestamp** – This is a record of Unix’s time at this block’s inception.
13. **Extra Data** – This byte-array of size 32 bytes or less contains extra data relevant to this block.
14. **Mix Hash** – This is a 32-byte hash that verifies a sufficient amount of computation has been done on this block.
15. **Nonce** – This is an 8-byte hash that verifies a sufficient amount of computation has been done on this block.
16. **Ommers Block Headers** – These are the same components listed above for any ommers.

### 3.2.2. Block Footer

**Transaction Series** – This is the only non-header content in the block.

### 3.2.3. Block Number and Difficulty

Note that is the difficulty of the genesis block. The Homestead difficulty parameter, is used to affect a dynamic homeostasis of time between blocks, as the time between blocks varies, as discussed below, as implemented in EIP-2. In the Homestead release, the exponential difficulty symbol, causes the difficulty to slowly increase (every 100,000 blocks) at an exponential rate, and thus increasing the block time difference, and putting time pressure on transitioning

to proof-of-stake. This effect, known as the “difficulty bomb”, or “ice age”, was explained in EIP-649 and delayed and implemented earlier in EIP-2, was also modified in EIP-100 with the use of  $x$ , the adjustment factor, and the denominator 9, in order to target the mean block time including uncle blocks. Finally, in the Byzantium release, with EIP-649, the ice age was delayed by creating a fake block number, which is obtained by subtracting three million from the actual block number, which in other words reduced and the time difference between blocks, in order to allow more time to develop proof-of-stake and preventing the network from “freezing” up.<sup>3</sup>

### 3.2.4. Account Creation

Account creation definitively occurs with contract creation. Is related to: `init`. Lastly, there is the body which is the EVM-code that executes if/when the account containing it receives a message call.

### 3.2.5. Account State

The account state contains details of any particular account during some specified world state. The account state is made up of four variables:

1. **nonce** The number of transactions sent from this address, or the number of contract creations made by the account associated with this address.
2. **balance** The amount of **Wei** OWNED by this account. Stored as a key/value pair inside the state database.
3. **storage\_root** A 256-bit (32-byte) hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account.<sup>a</sup>
4. **code\_hash** The hash of the EVM code of this account’s contract. Code hashes are STORED in the **state database**. Code hashes are permanent and they are executed when the address belonging to that account RECEIVES a message call.

<sup>a</sup>A particular path from root to leaf in the **state database**

### 3.2.6. Bloom Filter

The Bloom Filter is composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list.

### 3.2.7. Transaction Receipts

## 4. Processing and Computation

### 4.1. The Transaction

The basic method for Ethereum accounts to interact with each other. The transaction is a single cryptographically signed instruction sent to the Ethereum network. There are two types of transactions: MESSAGE CALLS and CONTRACT CREATIONS. Transactions lie at the heart of Ethereum, and are entirely responsible for the dynamism and flexibility of the platform. Transactions are the bread and butter of state transitions, that is of block additions, which contain all of the computation performed in one block. Each transaction applies the execution changes to the *machine state*, a temporary state which consists of all the required changes in computation that must be made before a block is finalized and added to the world state.

#### 4.1.1. Transactions Root

**Notation** : `listhash`

**Alternatively:** Transactions Root

**Description** : The Keccak-256 hash of the root node that precedes the transactions in the `transactions_list` section of a Block.

1. **Nonce** – The number of transactions sent by the sender.
2. **Gas Price** – The number of Wei to pay the network for unit of gas.
3. **Gas Limit** – The maximum amount of gas to be used in while executing a transaction.

4. **To** – The 20-character recipient of a message call.<sup>a</sup>
5. **Value** The number of Wei to be transferred to the recipient of a message call.<sup>b</sup>
6. **v, r, s**

### 4.2. State Transition Function

State Transitions come about through a what is known as the State Transition Function; this is an abstraction of several operations in Ethereum which comprise the overall act of taking changes from the *machine state* and confirming them to add them to the world state.

### 4.3. Mining

The Block Beneficiary is the 160-bit (20-byte, or 20-character) address to which all fees collected from the successful mining of a block are transferred. Apply Rewards is the third process in `block_finalization` that sends the mining reward to an account's address. A scalar value corresponding to the difficulty level of a current block.

### 4.4. Verification

The process in The EVM that verifies Ommer Headers

### 4.5. Sender Function

A description that maps transactions to their sender using ECDSA of the SECP-256k1 curve,

### 4.6. Serialization/Deserialization

This function expands a positive-integer value to a big-endian byte array of minimal length. When accompanied by a `.` operator, it signals sequence concatenation. The `big_endian` function accompanies RLP serialization and deserialization.

<sup>a</sup>In the case of a contract creation this is 0x00000000000000000000.

<sup>b</sup>In the case of a contract creation, an endowment to the newly created contract account.



## 4.7. Ethereum Virtual Machine

The EVM has a simple stack-based architecture. The word size of the machine and thus size of stack is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve based computation. The memory model is a simple word-addressed byte array. The memory stack has a maximum size of 1024-bits. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is non-volatile and is maintained as part of the system state.

All locations in both storage and memory are well-defined initially as zero. The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction. The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

**9.2. Fees Overview.** Fees (denominated in gas) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation (see Appendix G). Secondly, gas may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for `CREATE`, `CALL` and `CALLCODE`. Finally, gas may be paid due to an increase in the usage of the memory. Over an account's execution, the total fee for memory-usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range. This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional memory usage fee. Due to this fee it

is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality. Storage fees have a slightly nuanced behaviour—to incentivise minimisation of the use of storage (which corresponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given; in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more than normal usage. See Appendix H for a rigorous definition of the EVM gas cost.<sup>3</sup>

### 4.7.1. Execution

**Description** : The execution of a transaction defines the state transition function: `stf`. However, before any transaction can be executed it needs to go through the initial tests of intrinsic validity.

### 4.7.2. Code Deposit

**Notation** : `code_deposit`

**Description** : If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, `c`, proportional to the size of the created contract's code.

### 4.7.3. Intrinsic Validity

The criteria for intrinsic validity are as follows:

- The transaction follows the rules for *well-formed RLP* (recursive length prefix.)
- The *signature* on the transaction is valid.
- The *nonce* on the transaction is valid, i.e. it is equivalent to the sender account's current nonce.
- The `gas_limit` is greater than or equal to the `intrinsic_gas` used by the transaction.
- The sender's account balance contains the cost required in up-front payment.

Accordingly, the post-transactional state of Ethereum is expressed thus:

```
transaction(post.state) = stf(present.state,
transaction)
```

While the amount of gas used in the execution

is expressed: `stf(gas_used)` and the accrued log items belonging to the transaction are expressed: `stf(logsbloom, content)(logsbloom, set)` Information concerning the result of a transaction's execution is stored in the transaction receipt `tx_receipt`. The set of log events which are created through the execution of the transaction, `logs_set` in addition to the bloom filter which contains the actual information from those log events `logs_bloom` are located in the transaction receipt. In addition, the post-transaction state `post_transaction(state)` and the amount of gas used in the block containing the transaction receipt `post(gas_used)` are stored in the transaction receipt. Thusly the transaction receipt is a record of any given execution.

A valid transaction execution begins with a permanent change to the state: the nonce of the sender account is increased by one and the balance is decreased by the `collateral_gas`<sup>a</sup> which is the amount of gas a transaction is required to pay prior to its execution. The original transactor will differ from the sender if the message call or contract creation comes from a contract account executing code.

After a transaction is executed, there comes a PROVISIONAL STATE:

**Notation** : `pv_state`

**Description** : Used to define the PRE-FINAL STATE, the PROVISIONAL STATE. Gas used for the execution of individual EVM opcodes prior to their potential addition to the `world_state` creates the provisional state. `productive_gas`, and an associated substate `substate_a`.

Code execution always depletes gas. If gas runs out, an out-of-gas error is signaled (`oog`) and the resulting state defines itself as an empty set; it has no effect on the world state. This describes the transactional nature of Ethereum. In order to affect the WORLD STATE, a transaction must go through completely or not at all.

<sup>a</sup>Designated "intrinsic\_gas" in the Yellowpaper

#### 4.7.4. Execution Model

**Basics** : The stack-based *virtual machine* which lies at the heart of the Ethereum and performs the actions of a computer. This is actually an instancial runtime that executes several substates, as EVM computation instances, before adding the finished result, all calculations having been completed, to the final state via the finalization function.

In addition to the system state  $\sigma$ , and the remaining gas for computation  $g$ , there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple  $I$ :

- `account_address`, the address of the account which owns the code that is executing.
- `sender_address` the sender address of the transaction that originated this execution.
- `originator_price` the price of gas in the transaction that originated this execution.
- `input_data`, a byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- `account_address` the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- `newstate_value` the value, in Wei, passed to this account if the execution agent is a transaction, this would be the transaction value.<sup>3</sup>
- `code.array` the byte array that is the machine code to be executed.<sup>3</sup>
- `samestate_header` the block header of the present block.
- `the stack depth` the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).<sup>3</sup>

The execution model defines the `state_transition` function, which can compute the resultant state, the remaining\_gas, the accrued\_substate and the resultant\_output, given these definitions. For the

present context, we will define it where the accrued substate is defined as the tuple of the `suicides_set`, the `log_series`, the `touched_accounts` and the `refunds`.

#### 4.7.5. Execution Overview

The `execution_function`, in most practical implementations, will be modeled as an iterative progression of the pair comprising the full `system_state` and the `machine_state`. It's defined recursively with the `iterator_function`, which defines the result of a single cycle of the state machine, together with the `halting_check` function, which determines if the present state is an exceptional halting state of the machine and `output_data` of the instruction if the present state is a `controlled_halt` of the machine. An empty sequence/series indicates that execution should halt, while the empty set indicates that execution should continue.

When evaluating execution, we extract the remaining gas from the resultant machine state. It is thus cycled (recursively or with an iterative loop) until either `exceptional_halt` becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until `H` becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

The machine state is defined as the tuple which are the gas available, the program counter, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents are a series of zeroes of size 2<sup>256</sup>. For the ease of reading, the instruction mnemonics, written in small-caps (e.g. `ADD`), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given For the purposes of defining we define `w` as the current operation to be executed: `STOP` otherwise We also assume the fixed amounts of and, specifying the stack items removed and added, both subscriptable on the instruction and an instruction `cost_function` evalu-

ating to the full cost, in gas, of executing the given instruction.

#### 4.7.6. The Execution Cycle

Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged: The gas is reduced by the instruction's gas cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function `J`, subscripted by one of two instructions, which evaluates to the according value: otherwise In general, we assume the memory, self-destruct set and system state don't change: However, instructions do typically alter one or several components of these values.

### 4.8. Provisional State

**Description** : A smaller, temporary state that is generated during transaction execution. It contains three sets of data:

- The accounts tagged for self-destruction following the transaction's completion. `self_destruct(accounts)`
- The `logs_series`, which creates checkpoints in EVM code execution for frontend applications to explore, and is made up of the `logs_set` and `logs_bloom` from the `tx_receipt`.
- The refund balance.<sup>a</sup>

#### 4.8.1. Message Calls

**Description** :

**Notation** : `message_call`

**Description** : A message call can come from a transaction or internally from contract code execution. It contains the field `DATA`, which consists of user data input to a message call. Messages allow communication between accounts (whether contract or external,) and are a carryover from established concepts in Computer Science, most notably the *MPI: Message-Passing Framework*. Messages can come in the form of

<sup>a</sup>The `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.



`msg_calls` which give output data. If an account has EVM code in it (a contract account,) this code gets executed when the account receives a message call. Message calls and contract creations are both *transactions*, but contract creations are never considered the same as message calls. Message calls always transfer some amount of value to an account. If the message call is an account creation transaction then the value given is taken on the role of an endowment toward the new account. Every time an account receives a message call it returns the body, something which is triggered by the `init` function. A message call can come through a transaction, or through the internal execution of code. Message call transactions only contain data. They are separate from regular, standard *transactions*.

**Universal Gas** Message calls always have a universally agreed-upon cost in gas. There is a strong distinction between contract creation transactions and message call transactions. Computation performed, whether it is a contract creation or a message call, represents the currently legal valid state. There can be no invalid transactions from this point.<sup>3</sup> There is also a message call/contract creation *stack*. This stack has a depth, depending on how many transactions are in it. Contract creations and message calls have entirely different ways of executing, and are entirely different in their roles in Ethereum. The concepts can be conflated. Message calls can result in computation that occurs in the next state rather than the current one. If an account that is currently executing receives a message call, no code will execute, because the account might exist but has no code in it yet. To execute a message call transactions are required:

- Sender
- Transaction\_Originator
- Recipient
- Account (usually the same as the recipient)
- Available\_Gas
- Value
- Gas\_Price
- An arbitrary length byte-array. `arb_array`

- Present\_Depth of the message call/contract creation stack.

**Notation** : data

**Description** : User data input to a `message_call`, structured as an unlimited size byte-array.

#### 4.8.2. Contract Creation

**Notation** : `init`

**Description** : When `INIT` is executed it returns the `BODY`. `Init` is executed only once at `ACCOUNT_CREATION`, and permanently discarded after that. Contract creation transactions are equal the recursive length prefix of an empty byte-sequence.

#### 4.8.3. Account Creation

### 4.9. Halting

#### Execution Environment

**Notation** : `ERE`

**Description** : The environment under which an Autonomous Object executes in the EVM: the EVM runs as a part of this environment.

**Notation** : `big_endian_f`

**Description** : `BIG_ENDIAN_FUNCTION` This function expands a positive-integer value to a big-endian byte array of minimal length. When accompanied by a `·` operator, it signals sequence concatenation. The `big_endian` function accompanies RLP serialization and deserialization.

#### 4.10. Gas

**Description** : The fundamental network cost unit converted to and from Ether as needed to complete the transaction while it is sent. Gas is arbitrarily determined at the moment it is needed, by the block and according to the miners decision to charge certain fees.

**Miner Choice** Miners choose which gas prices they want to accept.

#### 4.10.1. Gasprice

**Notation** : `gas_limit`

**Description** : A value equal to the current limit of gas expenditure per block, according to the miners.

**Gaslimit** Any unused gas is refunded to the user. The canonical gas limit of a block is expressed `canonical_gas`, and is stabilized by the `time_stamp` of the block.

**Gas Price Stability** Where `new_header` is the new block's header, but without the nonce and mix-hash components, `d` being the current DAG, a large data set needed to compute the mix-hash, and PoW is the proof-of-work function this evaluates to an array with the first item being the mix-hash, to proof that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on. Given an approximately uniform distribution in the range the expected time to find a solution is proportional to the difficulty. This is the foundation of the security of the blockchain and is the fundamental reason why a malicious node cannot propagate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because its satisfaction depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult and, over time, requires approximately the total compute power of the trustworthy portion of the mining peers. Thus we are able to define the block header validity function.

#### Gasused

**Description** : A value equal to the total gas used in transactions in this block.

#### 4.10.2. Machine State

The machine state is a tuple consisting of five elements:

1. `gas_available`
2. `program_counter`
3. `memory_contents` A series of zeroes of size  $2^{256}$
4. `memory_words.count`
5. `stack_contents`

There is also, `[to_execute]`: the current operation to be executed

#### 4.10.3. Exceptional Halting

An exceptional halt may be caused by four conditions existing on the stack with regard to the next opcode in line for execution:

```
if
out_of_gas = true
or
bad_instruction = true
or
bad_stack_size = true
or
bad_jumpdest = true
then throw exception
else exec opcode x
then init control_halt
```

Exceptional halts are reserved for opcodes that fail to execute. They can never be caused through an opcode's actual execution.

- The amount of remaining gas in each transaction is extracted from information contained in the `machine_state`
- A simple iterative recursive loop<sup>3</sup> with a boolean value:
  - `true` indicating that in the run of computation, an exception was signaled
  - `false` indicating in the run of computation, exceptions were signaled. If this value remains false for the duration of the execution until the set of transactions becomes a series (rather than an empty set.) This means that the machine has reached a controlled halt.

#### Substate

**Notation** : substate

**Description** : A smaller, temporary state that is generated during transaction execution. It contains three sets of data:

- The accounts tagged for self-destruction following the transaction's completion. `self_destruct(accounts)`
- The `logs_series`, which creates checkpoints in EVM code execution for frontend applications to explore, and is made up of `the_logs_set` and `logs_bloom` from the `tx_receipt`.
- The refund balance.<sup>a</sup>

The substate is an emergent, ever-changing ball of computational energy that is about to be applied to the main state. It is the *meta state* by which transactions are decided valid and to be added to the blockchain.

#### 4.10.4. EVM Code

The bytecode that the EVM can natively execute. Used to explicitly specify the meaning of a message to an account.

**Description** : A contract is a piece of EVM Code that may be associated with an Account or an Autonomous Object.

#### 4.10.5. EVM Assembly

The human readable version of EVM Code (opcodes).

### 4.11. Blocktree to Blockchain

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the heaviest path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined genesis block, in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive

at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols. Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any block contributes toward the total computation or total difficulty of a chain. Thus we define the total difficulty of `this_block` recursively by the difficulty of its parent block and the block itself.

The jobs of miners and validators are as follows: Validate (or, if mining, determine) omers; validate (or, if mining, determine) transactions; apply rewards; verify (or, if mining, compute a valid) state and nonce.

### 4.12. Ommer Validation

The validation of ommer headers means nothing more than verifying that each ommer header is both a valid header and satisfies the relation of Nth-generation ommer to the present block. The maximum of ommer headers is two. Formally: where `k` denotes the “is-kin” property: otherwise and `s` denotes the “is-sibling” property: is the block of the corresponding header `H`.

### 4.13. Transaction Validation

The given `gasUsed` must correspond faithfully to the transactions listed, the total gas used in the block, must be equal to the accumulated gas used according to the final transaction.

### 4.14. Reward Application

The application of rewards to a block involves raising the balance of the accounts of the beneficiary address of the block and each ommer by a certain amount. We raise the block's beneficiary account by `R b`; for each ommer, we raise the block's beneficiary by 1 an additional 32 of the block reward and the beneficiary of the ommer gets rewarded depending on the block number. This constitutes the `block_finalization_state_transition_function`. If there are collisions of the beneficiary addresses between omers and the block two omers with the same beneficiary address

<sup>a</sup>The `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

or an ommer with the same beneficiary address as the present block,

additions are applied cumulatively. We define the block reward as 3 Ether: State & Nonce Validation. We may now define the function, that maps a block B to its initiation state: otherwise Here, that means the hash of the root node of a trie of state x; it is assumed that implementations will store this in the state database, trivial and efficient since the trie is by nature a resilient data structure. And finally define the `block_transition_function`, which maps an incomplete block to a complete block with a specified dataset. As specified at the beginning of the present work, the `state_transition_function`, which is defined in terms of, the `block_finalisation_function` and, the `transaction_evaluation_function`. As previously detailed, there is the `nth` corresponding status code, logs and cumulative gas used after each transaction, the fourth component in the tuple, has already been defined in terms of the logs).

The `nth` state is given from applying the corresponding transaction to the state resulting from the previous transaction (or the block's initial state in the case of the first BYZANTIUM VERSION 3475aa8 - 2018-01-26 14 such transaction): otherwise In certain cases we take a similar approach defining each item as the gas used in evaluating the corresponding transaction summed with the previous item (or zero, if it is the first), giving us a running total: the function is used that was defined in the transaction execution function. We define `R[n]` a similar manner. Finally, we define new state given the block reward function applied to the final transaction's resultant state, thus the complete block-transition mechanism, less PoW, the proof-of-work function is defined.

#### 4.15. Mining Proof-of-Work

Proof that a certain amount of mining has been done exists as a cryptographic probability statement which asserts beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value `n`. It is utilised to en-

force the uncompromisable security of the blockchain. Since mining blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence in the future and past state of the machine, but also as a wealth distribution mechanism. The proof of work function should be as accessible as possible to as many people as possible.

To make the Ethereum Blockchain ASIC resistant, the Proof-of-Work mechanism has been designed to be sequential memory-hard. This means that the nonce requires a lot of memory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. Therefore, the proof-of-work function takes the form of  $2^{256}m = Hm$  is the new block's header but without the nonce and mix-hash components;  $Kn$  is the nonce of the header;  $d$  is a large data set needed to compute the mix hash and  $Hd$  is the new block's difficulty value. PoW is the proof-of-work function which evaluates to an array with the first item being the mix hash and the second item being a pseudorandom number which is cryptographically dependent on  $H$  and  $d$ . The name for this algorithm is Ethash.

#### 4.16. Ethash

Ethash is the PoW algorithm for Ethereum 1.0. It is the latest version of Dagger-Hashimoto, introduced by Vitalik Buterin. The general route that the algorithm takes is as follows: There exists a seed which can be computed for each block by scanning through the block headers up until that point. From the seed, one can compute a pseudorandom cache,  $J$  cacheinit bytes in initial size. Light clients store the cache. From the cache, we can generate a dataset,  $ds$  bytes in initial size, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time. Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache. The large dataset is updated once every 1 epoch blocks, so the vast majority of a miner's effort will be reading the dataset, not making

changes to it. The mentioned parameters as well as the algorithm is explained in detail in appendix J. 12. Implementing Contracts There are several patterns of contracts engineering that allow particular useful behaviours; two of these that I will briefly discuss are data feeds and random numbers. 12.1. Data Feeds.

#### 4.17. Pseudorandom Numbers

Pseudo-random numbers be generated by utilizing data which is generally unknowable at the time of transacting. Such data might include the block's hash, the block's timestamp or the block's beneficiary address. The BLOCKHASH opcode uses the previous 256 blocks as pseudo-random numbers. One could automate this randomness by adding a fixed value and hashing the result.

#### 4.18. Chainsize Limits

The state database won't be forced to maintain all past state trie structures into the future. It should maintain an age for each node and eventually discard nodes that are neither recent enough nor checkpoints; checkpoints, or a set of nodes in the database that allow a particular block's state trie to be traversed, could be used to place a maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain. Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 10,000th block) could be maintained by the peer network, effectively recasting the genesis block. This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks. Finally, blockchain compression could perhaps be conducted: nodes in state trie that haven't sent/received a transaction in some constant amount of blocks could be thrown out, reducing both Ether-leakage and the growth of the state database.

#### 4.19. Scalability

Scalability remains an eternal concern. With a generalized state transition function, it becomes difficult to partition space, but several strategies exist that may provide for scaling.

##### 4.19.1. Sharding

Parallelization of transaction combination and block building.

##### 4.19.2. Casper

##### 4.19.3. Plasma



## A. Opcodes

### A.1. 0x10's: Comparisons and Bitwise Logic Operations

Data	Opcode	Gas	Input	Output	Description
0x00	STOP	0	0	0	Halts execution.
0x01	ADD	3	2	1	Addition operation.
0x02	MUL	5	2	1	Multiplication operation.
0x03	SUB	3	2	1	Subtraction operation.
0x04	DIV	5	2	1	Integer division operation.
0x05	SDIV	5	2	1	Signed integer division operation (truncated.)
0x06	MOD	5	2	1	Modulo remainder operation.
0x07	SMOD	5	2	1	Signed modulo remainder operation.
0x08	ADDMOD	8	3	1	Modulo addition operation.
0x09	MULMOD	8	3	1	Modulo multiplication operation.
0x0a	EXP	10	2	1	Exponential operation.
0x0b	SIGNEXTEND	5	2	1	Extend the length of two's complementary signed integer.
0x10	LT	3	2	1	Less-than comparison.
0x11	GT	3	2	1	Greater-than comparison.
0x12	SLT	3	2	1	Signed less-than comparison.
0x13	SGT	3	2	1	Signed greater-than comparison.
0x14	EQ	3	2	1	Equality comparison.
0x15	ISZERO	3	1	1	Simple not operator.
0x16	AND	3	2	1	Bitwise AND operation.
0x17	OR	3	2	1	Bitwise OR operation.
0x18	XOR	3	2	1	Bitwise XOR operation.
0x19	NOT	3	1	1	Bitwise NOT operation.
0x1a	BYTE	3	2	1	Retrieve single byte from word.

### A.2. 0x20's: SHA3

Data	Opcode	Gas	Input	Output	Description
0x20	SHA3	30	2	1	Compute a Keccak-256 hash.

### A.3. 0x30's: Environmental Information

Data	Opcode	Gas	Input	Output	Description
0x30	ADDRESS	2	0	1	Get the address of the currently executing account.
0x31	BALANCE	400	1	1	Get the balance of the given account.
0x32	ORIGIN	2	0	1	Get execution origination address. This is always the original sender of a transaction, never a contract account.

0x33	CALLER	2	0	1	Get caller address. This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	2	0	1	Get deposited value by the instruction/transaction responsible for this execution.
0x35	CALLDATALOAD	3	1	1	Get input data of the current environment.
0x36	CALLDATASIZE	2	0	1	Get size of input data in current environment. This refers to the optional data field that can be passed with a message call instruction or transaction.
0x37	CALLDATACOPY	3	3	0	Copy input data in the current environment to memory. This refers to the optional data field passed with the message call instruction or transaction.
0x38	CODESIZE	2	0	1	Get size of code running in the current environment.
0x39	CODECOPY	3	3	0	Copy the code running in the current environment to memory.
0x3a	GASPRICE	2	0	1	Get the price of gas in the current environment. This is the gas price specified by the originating transaction.
0x3b	EXTCODESIZE	700	1	1	Get the size of an account's code.
0x3c	EXTCODECOPY	700	4	0	Copy an account's code to memory.
0x3d	RETURNDATASIZE	2	0	1	
0x3e	RETURNDATACOPY	3	3	0	

#### A.4. 0x40's: Block Data

Data	Opcode	Gas	Input	Output	Description
0x40	BLOCKHASH	20	1	1	Get the hash of one of the 256 most recent blocks. <sup>a</sup>
0x41	COINBASE	2	0	1	Look up a block's beneficiary address by its hash.
0x42	TIMESTAMP	2	0	1	Look up a block's timestamp by its hash.
0x43	NUMBER	2	0	1	Look up a block's number by its hash.
0x44	DIFFICULTY	2	0	1	Look up a block's difficulty by its hash.
0x45	GASLIMIT	2	0	1	Look up a block's gas limit by its hash.

#### A.5. 0x50's: Stack, memory, storage, and flow operations.

Data	Opcode	Gas	Input	Output	Description
------	--------	-----	-------	--------	-------------

<sup>a</sup>A value of 0 is left on the stack if the block number is more than 256 in number behind the current one, or if it is a number greater than the current one.

0x50	POP	2	1	0	Removes an item from the stack.
0x51	MLOAD	3	1	1	Load a word from memory.
0x52	MSTORE	3	2	0	Save a word to memory.
0x53	MSTORE8	3	2	0	Save a byte to memory.
0x54	SLOAD	200	1	1	Load a word from storage.
0x55	SSTORE	0	2	0	Save a word to storage.
0x56	JUMP	8	1	0	Alter the program counter.
0x57	JUMPI	10	2	0	Conditionally alter the program counter.
0x58	PC	2	0	1	Look up the value of the program counter prior to the increment resulting from this instruction.
0x59	MSIZE	2	0	1	Get the size of active memory in bytes.
0x5a	GAS	2	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction.
0x5b	JUMPDEST	1	0	0	Mark a valid destination for jumps. <sup>a</sup>

## A.6. 0x60-70's: Push Operations

Data	Opcode	Gas	Input	Output	Description
0x60	PUSH1	-	0	1	Place a 1-byte item on the stack.
0x61	PUSH2	-	0	1	Place a 2-byte item on the stack.
0x62	PUSH3	-	0	1	Place a 3-byte item on the stack.
0x63	PUSH4	-	0	1	Place a 4-byte item on the stack.
0x64	PUSH5	-	0	1	Place a 5-byte item on the stack.
0x65	PUSH6	-	0	1	Place a 6-byte item on the stack.
0x66	PUSH7	-	0	1	Place a 7-byte item on the stack.
0x67	PUSH8	-	0	1	Place a 8-byte item on the stack.
0x68	PUSH9	-	0	1	Place a 9-byte item on the stack.
0x69	PUSH10	-	0	1	Place a 10-byte item on the stack.
0x6a	PUSH11	-	0	1	Place a 11-byte item on the stack.
0x6b	PUSH12	-	0	1	Place a 12-byte item on the stack.
0x6c	PUSH13	-	0	1	Place a 13-byte item on the stack.
0x6d	PUSH14	-	0	1	Place a 14-byte item on the stack.
0x6e	PUSH15	-	0	1	Place a 15-byte item on the stack.
0x6f	PUSH16	-	0	1	Place a 16-byte item on the stack.
0x70	PUSH17	-	0	1	Place a 17-byte item on the stack.
0x71	PUSH18	-	0	1	Place a 18-byte item on the stack.
0x72	PUSH19	-	0	1	Place a 19-byte item on the stack.
0x73	PUSH20	-	0	1	Place a 20-byte item on the stack.
0x74	PUSH21	-	0	1	Place a 21-byte item on the stack.
0x75	PUSH22	-	0	1	Place a 22-byte item on the stack.

<sup>a</sup>This operation has no effect on the machine\_state during execution.

0x76	PUSH23	-	0	1	Place a 23-byte item on the stack.
0x77	PUSH24	-	0	1	Place a 24-byte item on the stack.
0x78	PUSH25	-	0	1	Place a 25-byte item on the stack.
0x79	PUSH26	-	0	1	Place a 26-byte item on the stack.
0x7a	PUSH27	-	0	1	Place a 27-byte item on the stack.
0x7b	PUSH28	-	0	1	Place a 28-byte item on the stack.
0x7c	PUSH29	-	0	1	Place a 29-byte item on the stack.
0x7d	PUSH30	-	0	1	Place a 30-byte item on the stack.
0x7e	PUSH31	-	0	1	Place a 31-byte item on the stack.
0x7f	PUSH32	-	0	1	Place a 32-byte item on the stack.

### A.7. 0x80's: Duplication Operations

Data	Opcode	Gas	Input	Output	Description
0x80	DUP1	-	1	2	Duplicate the 1st item in the stack.
0x81	DUP2	-	2	3	Duplicate the 2nd item in the stack.
0x82	DUP3	-	3	4	Duplicate the 3rd item in the stack.
0x83	DUP4	-	4	5	Duplicate the 4th item in the stack.
0x84	DUP5	-	5	6	Duplicate the 5th item in the stack.
0x85	DUP6	-	6	7	Duplicate the 6th item in the stack.
0x86	DUP7	-	7	8	Duplicate the 7th item in the stack.
0x87	DUP8	-	8	9	Duplicate the 8th item in the stack.
0x88	DUP9	-	9	10	Duplicate the 9th item in the stack.
0x89	DUP10	-	10	11	Duplicate the 10th item in the stack.
0x8a	DUP11	-	11	12	Duplicate the 11th item in the stack.
0x8b	DUP12	-	12	13	Duplicate the 12th item in the stack.
0x8c	DUP13	-	13	14	Duplicate the 13th item in the stack.
0x8d	DUP14	-	14	15	Duplicate the 14th item in the stack.
0x8e	DUP15	-	15	16	Duplicate the 15th item in the stack.
0x8f	DUP16	-	16	17	Duplicate the 16th item in the stack.

### A.8. 0x90's: Swap Operations

Data	Opcode	Gas	Input	Output	Description
0x90	SWAP1	-	2	2	Exchange the 1st and 2nd stack items.
0x91	SWAP2	-	3	3	Exchange the 1st and 3rd stack items.
0x92	SWAP3	-	4	4	Exchange the 1st and 4th stack items.
0x93	SWAP4	-	5	5	Exchange the 1st and 5th stack items.
0x94	SWAP5	-	6	6	Exchange the 1st and 6th stack items.
0x95	SWAP6	-	7	7	Exchange the 1st and 7th stack items.
0x96	SWAP7	-	8	8	Exchange the 1st and 8th stack items.
0x97	SWAP8	-	9	9	Exchange the 1st and 9th stack items.
0x98	SWAP9	-	10	10	Exchange the 1st and 10th stack items.
0x99	SWAP10	-	11	11	Exchange the 1st and 11th stack items.
0x9a	SWAP11	-	12	12	Exchange the 1st and 12th stack items.

0x9b	SWAP12	-	13	13	Exchange the 1st and 13th stack items.
0x9c	SWAP13	-	14	14	Exchange the 1st and 14th stack items.
0x9d	SWAP14	-	15	15	Exchange the 1st and 15th stack items.
0x9e	SWAP15	-	16	16	Exchange the 1st and 16th stack items.
0x9f	SWAP16	-	17	17	Exchange the 1st and 17th stack items.

### A.9. 0xa0's: Logging Operations

Data	Opcode	Gas	Input	Output	Description
0xa0	LOG0	375	2	0	Append log record with 0 topics.
0xa1	LOG1	750	3	0	Append log record with 1 topic.
0xa2	LOG2	1125	4	0	Append log record with 2 topic.
0xa3	LOG3	1500	5	0	Append log record with 3 topic.
0xa4	LOG4	1875	6	0	Append log record with 4 topic.

### A.10. 0xf0's: System Operations

Data	Opcode	Gas	Input	Output	Description
0xf0	CREATE	32000	3	1	Create a new contract account. Operand order is: value, input offset, input size.
0xf1	CALL	700	7	1	Message-call into an account. The operand order is: gas, to, value, in offset, in size, out offset, out size.
0xf2	CALLCODE	700	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL, except the recipient is the same account as at present, but the code is overwritten.
0xf3	RETURN	0	2	0	Halt execution, then return output data. This defines the output at the moment of the halt.
0xf4	DELEGATECALL	700	6	1	Message-call into this account with an alternative account's code, but with persisting values for sender and value. DELEGATECALL takes one less argument than CALL. This means that the recipient is in fact the same account as at present, but that the code is overwritten <i>and</i> the context is almost entirely identical.
0xf5	CALLBLACKBOX	40	7	1	-
0xfa	STATICCALL	40	6	1	-
0xfd	REVERT	0	2	0	-
0xfe	INVALID	-	1	0	Designated invalid instruction.



0xff	SELFDESTRUCT	5000	1	0	Halt execution and register the account for later deletion.
------	--------------	------	---	---	---

## B. Higher Level Languages

### B.1. Lower-Level Lisp

The Lisp-Like low level language: a human-writable language used for authoring simple contracts and trans-compiling to higher-level languages.

### B.2. Solidity

A language similar in syntax to Javascript, and the most commonly used language for creating smart contracts in Ethereum.

### B.3. Serpent

A deprecated language.

### B.4. Vyper

A newer language for developing smart contracts – still under development.

## References

- [1] W. contributors, *Tree (data structure)* — *wikipedia, the free encyclopedia*, [Online; accessed 15-December-2017], 2017. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=Tree\\_\(data\\_structure\)&oldid=813972413](https://en.wikipedia.org/w/index.php?title=Tree_(data_structure)&oldid=813972413) (cit. on p. 2).
- [2] E. Foundation, *Ethereum whitepaper*, <https://github.com/ethereum/wiki/wiki/White-Paper>, 2017 (cit. on p. 3).
- [3] D. G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, <https://github.com/ethereum/yellowpaper>, 2017 (cit. on pp. 3, 4, 6, 7, 9, 10).

## Glossary

**addresses** 20 character strings, specifically the right-most 20 characters of the Keccak-256 hash of the RLP-derived mapping which contains the sender's address and the nonce of the block.. 21

**balance** A value which is intrinsic to accounts; the quantity of Wei in the account. All EVM operations are associated with changes in account balance. 21

**beneficiary** The 20-character (160-bit) address to which all fees collected from the successful mining of this block be transferred. 21

**block header** All the information in a block besides transaction information. 21

**Contract** A piece of EVM Code that may be associated with an Account or an Autonomous Object. 21

**Cryptographic hashing functions** Hash functions make secure blockchains possible by establishing universal inputs for which there is typically only one possible output.. 21

**Ethereum Runtime Environment** The environment which is provided to an Autonomous Object executing in the EVM. Includes the EVM but also the structure of the world state on which the relies for certain I/O instructions including CALL & CREATE. 21

**Ethereum Virtual Machine** A sub-process of the *State Transition Function* which initializes and executes all of the transactions (ergo computations) in a block, prior to their finalization into the state.. 21

**EVM Assembly** The human readable version of EVM code. 21

**EVM Code** The bytecode that the EVM can natively execute. Used to formally specify the meaning

and ramifications of a message to an Account. 21

**Gas** The fundamental network cost unit; gas is paid for exclusively by Ether. 21

**leaf node** the bottom-most node in a particular tree, of blocks, one half of the “key” the other half being the root node, which creates the path between. 21

**Message** Data (as a set of bytes) and Value (specified in Wei) that is passed between two accounts.. 21

**Recursive Length Prefix** Recursive Length Prefix. 21

**root node** the uppermost node in a particular tree, of blocks, representing a single world state at a particular time. 2, 21

**serialization** Serialization is the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the machine state of an object in order to be able to recreate it when needed. [billwagner]. 21

**state machine** The term *State Machine* is reserved for any simple or complex process that moves deterministically from one discrete state to the next.. 21

**state database** A database stored off-chain, [i.e. on the computer of some user running an Ethereum client] which contains a trie structure mapping bytearrays [i.e. organized chunks of binary data] to other bytearrays [other organized chunks of binary data]. The *relationships* between each node on this trie constitute a MAP, a.k.a. a MAP-PING of all previous *world states* which a client might need to reference. 2, 4, 21

**storage root** One aspect of an ACCOUNT'S STATE: this is the hash of the trie<sup>a</sup> that decides the STORAGE CONTENTS of the account. 21

<sup>a</sup>A particular path from root to leaf in the state database

**Storage State** The information particular to a given account that is maintained between the times that the account's associated EVM Code runs. [21](#)

**transaction** A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain. Transactions are regarded as a single unit of work, and must be processed completely or not at all.[[Ngondi2016](#)]. [21](#)

**trie** A tree-structure for organizing data, the position of data in the tree contains the particular path from root to leaf node that represents the key (the path from root to leaf is "one" key) you are searching the trie structure for. The data of the key is contained in the trie relationships that emerge from related nodes in the trie structure. [2](#), [21](#)

## Acronyms

**ERE** Ethereum Runtime Environment. [21](#)

**EVM** Ethereum Virtual Machine. [21](#)

**RLP** Recursive Length Prefix. [21](#)

## Index

- 50% attack, 2
- 160
  - bit, 5
- 256
  - bit, 4, 6
- 1024
  - bit, 6
- abstract
  - state-machine, 1
- account
  - addresses, 2
  - balance, 4
  - body, 4
  - code hash, 4
  - creation, 4
  - init, 4
  - nonce, 4
  - state, 4
  - states, 2
  - storage root, 4
- ancestor node, 3
- apply
  - rewards, 5
- apply rewards, 1
- assembly language, 1
- balance, 1
- beneficiary, 3
- big endian, 3
- Bitcoin, 1
- Bitcoin Whitepaper, 2
- block
  - beneficiary, 5
  - composition, 3
  - header, 3
  - reward, 5
  - rewards, 3
- block difficulty, 2
- block number, 4
- branch node, 3
- byte
  - array, 6
- bytecode, 1
- byzantium, 4
- certainty, 1
- child node, 3
- contract
  - creation, 4
- cryptoeconomic
  - model, 1
- cryptographically
  - instantiated, 1
  - secure, 1
- cumulative difficulty, 2
- data
  - structure, 2
- descendant node, 3
- deserialization, 5
- deterministic, 1
- difficulty, 4
  - bomb, 4
- difficulty mechanism, 2
- double-spend problem, 2
- dynamic difficulty homeostasis, 4
- EIP
  - 2, 4
  - 100, 4
  - 649, 4
- elliptic curve, 6
  - computation, 6
  - cryptography, 6
- empty byte-array, 3
- Ether, 1
- ethereum interpreter, 1
- EVM, 5
- excessive computational expenditures, 1
- executed, 1
- exponential difficulty increase, 4
- extra data, 4
- Finney, 1
- forest, 3
- gas
  - limit, 4, 5
  - price, 5



- used, 4
- genesis difficulty, 4
- GHOST protocol, 2
- global state database, 3
- hash scheme, 6
- higher-level languages, 1
- homestead, 4
  - difficulty parameter, 4
- ice age, 4
- inspection
  - of data, 3
- keccak
  - 256, 6
- Keccak-256, 3, 4
- leaf
  - node, 3, 4
- ledger, 1
- logs bloom, 4
- longest chain, 2
- loopholes, 1
- machine
  - codes, 1
  - state, 5
  - storage, 6
- mapping, 2
- mapping between
  - account states, 3
  - addresses, 3
- memory, 6
  - model, 6
    - volatility of, 6
  - size, 6
  - stack, 6
- merkle-patricia
  - trees, 2
  - tries, 2
- merkletries, 2
- mining, 1
- mix hash, 4
- models
  - computer, 1
- modified merkletries, 2
- native currency, 1
- nested binary data, 3
- network cost unit, 1
- no leading zeroes, 3
- node
  - depth, 3
  - height, 3
- node operator computer, 3
- non
  - empty deserialized integer, 3
- nonce, 4
- number, 4
- ommer block headers, 4
- ommers
  - hash, 3
- parent
  - hash, 3
- parent node, 3
- pseudocode, 1
- r, 5
- receipts root, 4
- recorded, 1
- RLP, 2, 3
  - encodes as
    - byte-arrays, 3
  - integers, 3
  - serialized
    - byte-array, 3
- root
  - node, 4
- root node, 3
- s, 5
- Satoshi Nakamoto, 2
- serialization, 5
- sibling node, 3
- single byte integer, 3
- singleton, 1
- speedy
  - traversal
    - of data, 3
- stack
  - based, 6
  - architecture, 6

- stack-based
  - virtual machine, [1](#)
- state
  - database, [4](#)
  - transition, [5](#)
- state root, [3](#)
- storage
  - model, [6](#)
- system state, [6](#)
- Szabo, [1](#)
- timestamp, [4](#)
- timestamped, [1](#)
- to, [5](#)
- total difficulty, [2](#)
- transaction, [5](#)
- transactions root, [4](#)
- transfer, [1](#)
- tree
  - arbitrary depth, [3](#)
  - database, [2](#)
  - degree, [3](#)
  - edge, [3](#)
  - height, [3](#)
  - level, [3](#)
  - path, [3](#)
- trie
  - database, [2](#)
- uncensorable, [1](#)
- undesirable outcomes, [1](#)
- unstoppable, [1](#)
- v, [5](#)
- value, [5](#)
- veracity, [1](#)
- verifiable, [1](#)
- verification, [5](#)
- virtual machine, [1](#)
- Wei, [1](#)
- wei, [4](#)
- word
  - addressable, [6](#)
  - addressed, [6](#)
  - array, [6](#)
  - word size, [6](#)
- World State, [2](#)
- Yellowpaper, [1](#)