

Beigepaper: An 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 utilizes the distributed ledger model that originated with Bitcoin and repurposes it to model a virtual computer, giving machine level opcodes the same level of certainty as Bitcoin transactions. Just as sure as you can be certain that Bitcoin's ledger is accurate and that timestamps are correct through the Bitcoin *consensus mechanism*, just so sure is it that machine instructions initiated on Ethereum *will* execute.

In other words, programs executed on the Ethereum Blockchain are *basically* unstoppable. This doesn't mean that Ethereum programs can't have bugs. This means that Ethereum programs can be trusted to execute without any interference from *external* non-network forces. This property arises from the inherent security of the blockchain which is built by, and maintained upon, cryptographic proofs.

1.1. Native Currency

Because Ethereum strives not primarily at the currency application, but at all applications, there is a fundamental *network cost unit* used to mitigate the possibility of abusing the network with excessive computational ex-

penditures. This is called gas, and is explained fully in §3. Gas is paid for exclusively in ether. The smallest unit of currency in Ethereum is the Wei, which is equal to $\Xi 10^{-18}$, where Ξ stands for 1 ether. All currency transactions in Ethereum, at the machine level, are counted in Wei. There is also the Szabo, which is $\Xi 10^{-6}$, and the Finney, which is $\Xi 10^{-3}$.

The Ethereum network is subservient to others in terms of one thing only: **ether**, the native currency for Ethereum. Everything the system can do is bounded up in its ability to expend ether in exchange for gas, which buys a particular amount of system performance in some desired direction.

| 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. Memory and Storage

2.1. World State

The *world state* is a mapping of Ethereum addresses to accounts. Such a mapping consists of two things:

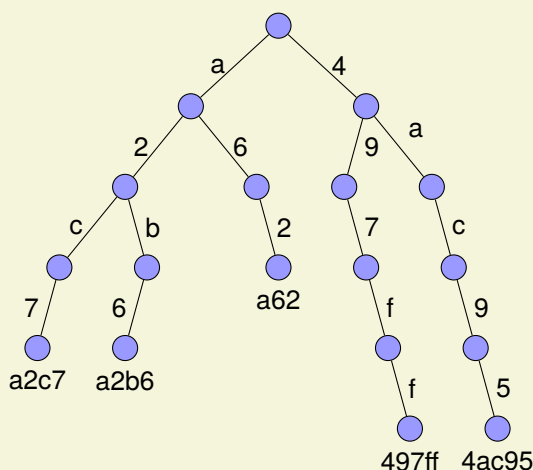
1. addresses
2. account states

through the use of the recursive length prefix standard (RLP). This information is stored as a merkle patricia tree in a DATABASE BACKEND.^a that maintains a mapping of bytearrays to bytearrays.^{b,c} As a whole, the state is the sum total of database relationships in the **state database**.

2.1.1. Merkle Patricia Trees

Merkle patricia trees 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:



2.2. Tree Terminology¹

- 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.
- 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.

^aThe database backend is accessed by users through an external application, most likely an Ethereum client; see also: **state database**

^bA 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.

^cThis 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.

2.2.1. Recursive Length Prefix Encoding

Recursive Length Prefix Encoding (RLP) imposes a structure on data that intrinsically considers a prefixed hex value to position the data in the state database tree. This hex value determines the depth of a certain piece of data. There are two types of fundamental items one can encode in RLP:²

1. Strings of bytes
2. Lists of other items

RLP encodes arrays of nested binary data to an arbitrary depth; it is the main serialization method for data in Ethereum. RLP encodes structure of data only, so it does not pay heed to the particular types of data being encoded.

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 zeroes it is invalid.³

The global state database is encoded as RLP for fast traversal and inspection of data. RLP encoding creates a mapping between *addresses* and *account states*. Since it is stored on node operator's computers, the tree can be indexed and searched without network delay. RLP encodes values as byte-arrays, or as sequences of further values, which are subsequently encoded as byte-arrays.⁴

2.3. The Block

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

2.3.1. 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.

2.3.2. The Other 2 Block Components

Ommmer Block Headers – These are ommmer block headers (15 components listed above) of this block.

Transaction Series – This is a list of transactions in this block and the only non-header content in the block.

2.3.3. 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 the time difference between blocks, in order to allow more time to develop proof-of-stake and preventing the network from “freezing” up.⁴

2.3.4. 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.

^aA particular path from root to leaf in the **state database**

2.3.5. 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.^a
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.

2.3.6. 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.

2.3.7. Transaction Receipts

3. Processing and Computation

3.1. 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.

3.1.1. 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.^a
5. **Value** – The number of Wei to be transferred to the recipient of a message call.^b
6. **v, r, s** – Values for the transaction's signature from which the public key can be derived.

3.2. State Transition Function

State Transitions come about through the State Transition Function; this is a high-level abstraction of several operations in Ethereum which comprise the overall act of taking changes from the *machine state* and adding them to the world state.

^aIn the case of a contract creation this is 0x00000000000000000000.

^bIn the case of a contract creation, an endowment to the newly created contract account.

3.3. Mining

The Block Beneficiary is the 160-bit (20-byte) 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. This is a scalar value corresponding to the difficulty level of a current block.

3.4. Verification

The process in The EVM that verifies Ommmer Headers.

3.5. Sender Function

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

3.6. 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.

3.7. 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 specialized instructions.

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.

3.7.1. Fees

Fees (denominated in gas) are charged under `THREE` distinct circumstances, all three as prerequisite to the execution of an operation.⁴ The **first** and most common is *the fee intrinsic to the computation of the operation*. **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 the `CREATE`, `CALL` and `CALLCODE` operations. **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 that addresses will trend above 32-bit bounds.⁴

Implementations must be able to manage this eventuality. Storage fees have a slightly nuanced behaviour to incentivize minimization 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.⁴

3.8. Execution

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.

3.8.1. 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.

3.8.2. Transaction Receipt

While the amount of gas used in the execution and the accrued log items belonging to the transaction are stored, 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. As a result, 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`^a 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, which is a pre-final state. Gas used for the execution of individual EVM opcodes prior to their potential addition to the `world_state` creates:

- Provisional state.
- `intrinsic gas`, and
- an associated substate.
- 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.^b

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.

3.8.3. Code Deposit

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.

3.8.4. 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 and the remaining gas for computation there are several pieces of important information used in the execution environment that the execution agent must provide:

- `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.⁴
- `code array` the byte array that is the machine code to be executed.⁴
- `block_header` the block header of the present block.
- `stack_depth` the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).⁴

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

^aDesignated "intrinsic_gas" in the Yellowpaper

^bThe `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

where the accrued substate is defined as the tuple of the `self-destructs_set`, the `log_series`, the `touched_accounts` and the `refunds`.⁴

3.8.5. 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^{256} .⁴

3.8.6. 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.

Provisional State A smaller, temporary state that is generated during transaction execution. It contains three sets of data.^a

3.8.7. Message Calls

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.) Messages can come in the form of `msg_calls` which give output data. If it is 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 another account. If the message call is an account creation transaction then the value given takes on the role of an endowment towards the new account. Every time an account receives a message call it returns the `body`, something which is triggered by the `init` function. User data input to a `message_call`, structured as an unlimited size byte-array.

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.⁴ There is also a message call/contract creation *stack*. This stack has a depth, depending on how many transactions are in

^aThe final state is reached after deleting all accounts that either appear in the self-destruct list or are touched and empty.

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.

3.8.8. Contract Creation

To initiate contract creation you need to send transaction to `nothing`. This executes `INIT` and returns the `BODY`. `Init` is executed only once at `ACCOUNT_CREATION`, and permanently discarded after that.

3.8.9. Execution Environment

The Ethereum Runtime Environment is the environment under which Autonomous Objects execute in the EVM: the EVM runs as a part of this environment.

3.8.10. 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.

3.9. Gas

Gas is 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 total network's miners decision to charge certain fees. Each miner choose individually which gas prices they want to accept and which they want to reject.

3.9.1. Gas Price/Gas Limit

`Gas price` is a value equal to the current limit of gas expenditure per block, according to the miners. Any unused gas is refunded to the sender. The canonical gas limit of a block is expressed 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 prove that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on it. Given an approximately uniform distribution in the range the expected time to find a solution is proportional to the difficulty.⁴

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

3.9.2. 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

3.9.3. 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 loop⁴ with a Boolean value:
 - **true** indicating that in the run of computation, an exception was signaled

- **false** indicating in the run of computation, no 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 A smaller, temporary state that is generated during transaction execution and runs parallel to machine state. 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.^a

3.9.4. EVM Code

The bytecode that the EVM can natively execute. Used to explicitly specify the meaning of a message to an account. A contract is a piece of EVM Code that may be associated with an Account or an Autonomous Object. **EVM Assembly** is the human readable version of EVM Code.

3.10. 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

^aThe `SSTORE` operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

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`) `ommers`; `validate` (or, if mining, `determine`) `transactions`; `apply rewards`; `verify` (or, if mining, `compute a valid`) `state` and `nonce`.

3.11. 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.

3.12. 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.

3.13. 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; 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 ommers and the block two ommers with the same beneficiary address

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

additions are applied cumulatively. The block reward is three ether per block.

State & Nonce Validation The function that maps a block B to its initiation state, that is, the hash of the root node of a trie of state x. This value is stored 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 there is 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. Finally new state exists in the context of 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.

3.14. 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 `pow_token`. It is utilised to enforce the security of the blockchain. Since mined blocks produce a reward, the proof-of-work also serves as a wealth distribution mechanism. For this reason, the proof of work function is designed to be as accessible as possible to as many people as possible.

A very basic application of this principle of accessibility is found in combining the traditional Proof-of-Work function with a *Memory-Hardness function*. By forcing the hashing algorithm to use memory as well as CPU, miners are more likely to use computers than ASICs, meaning that ASIC efficiency will not obsolete the person who wants to mine on their home computer from participating in the mining process. To make the Ethereum Blockchain ASIC resistant, the Proof-of-Work mechanism has been designed to be sequential and memory-hard. This means that the nonce requires high amounts 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} the new block's header but without the nonce and mix-hash components. There is the `header_nonce`, and `data_set` which are required to compute the mix hash and `block_difficulty`, the difficulty value of the new block. The proof-of-work function 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 the `header_nonce` and the `data_set`. The name for this algorithm is **Ethash**.

3.14.1. Ethash: Seed→Cache→Dataset→Slice

Ethash is the Proof-of-Work algorithm which was used to launch the Ethereum network and bring it through its first few releases. It is in the process of being gradually phased out and replaced with a Proof-of-Stake model. For now 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`, that is `CACHE_INIT` bytes in initial size. Light clients store the cache. From the cache, a dataset is generated, `dataset_size` 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 (10,000) blocks, so the vast majority of a miner's effort is spent on reading the dataset, rather than on making changes to it.

3.14.2. Difficulty Mechanism

This mechanism enforces a relative predictability in terms of the time-window 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 next time-window. Conversely, if the time-window is too large, the difficulty is reduced, reducing the amount of time to the next block. The `total_difficulty`^a is the `difficulty_state` of the entire Ethereum blockchain. The `block_difficulty`, in contrast, is not a state of the blockchain, but is local—particular to each specific block. You reach the total difficulty by summing the individual difficulty of all previous blocks and then adding the difficulty of the present block.

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 a single block in order to function. For that reason, in the Bitcoin protocol, it's impossible to submit a "double-spend" block without having at

^aAlternatively known as `total_computation`

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.

The "GHOST Protocol" (which stands for Greedy Heaviest Object subTree) rather requires that miners begin mining whichever chain the most other miners are on. Because of differences in network propagation of data about which miners are mining which block, this has a tendency to create more uncles. Nevertheless, in spite of the increased amount of uncle blocks, the chain itself is equally secure, and this method allows for higher throughput of transactions than Satoshi's solution to double-spending does.

3.15. Pseudorandom Numbers

Pseudo-random numbers may be generated by utilizing data which is generally unknowable at the time of transacting. This constitutes anything based off of factors which are unknowable under regular circumstances, but become knowable through the regular operation and growth of the chain. Such data might include a current (or relatively current) block's hash, timestamp, or beneficiary address. The BLOCKHASH opcode uses the previous 256 blocks as pseudo-random numbers. One could further automate this randomness by adding two BLOCKHASH operations and hashing the result.

3.16. Chainsize Limits

The **state database** usually won't store every single tree structure in the history of the blockchain. One idea has been proposed to simply maintain node checkpoints for each age (10,000 blocks) and eventually discard checkpoints which no longer contain necessary

state data. This would be a variation on a compression scheme.⁴

3.17. Scalability

Scalability is a constant concern. Because Ethereum's state transitions are so broad in terms of possible content, and because its applications and use-cases are so numerous in the number of potential transactions required, scalability is inherently necessary for increased transaction throughput and for more efficient storage and traversal of the chain.

3.17.1. Sharding

Parallelization of transaction combination and block building.

3.17.2. Casper

3.17.3. Plasma

A. EVM Opcodes⁵

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 | RETURNDATA-SIZE | 2 | 0 | 1 | |
| 0x3e | RETURNDATA-COPY | 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. ^a |
| 0x41 | COINBASE | 2 | 0 | 1 | Look up a block's beneficiary address by its hash. |

^aA 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.

| | | | | | |
|------|------------|---|---|---|---|
| 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 |
|------|----------|----------------|-------|--------|--|
| 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 | 5,000 – 20,000 | 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. ^a |

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. |

^aThis operation has no effect on the `machine_state` during execution.

| | | | | | |
|------|--------|---|---|---|------------------------------------|
| 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. |
| 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. |
| 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. |

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_\(data_structure\)%5C&oldid=813972413](https://en.wikipedia.org/w/index.php?title=Tree_(data_structure)%5C&oldid=813972413) (cit. on p. 2).
- [2] E. Foundation, *Pyrlp tutorial*, 2015. [Online]. Available: <https://github.com/jnnk/pyrlp/blob/master/docs/tutorial.rst> (cit. on p. 3).
- [3] —, *Ethereum whitepaper*, <https://github.com/ethereum/wiki/wiki/White-Paper>, 2017 (cit. on p. 3).
- [4] D. G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, <https://github.com/ethereum/yellowpaper>, 2017 (cit. on pp. 3, 4, 6–10, 13).
- [5] V. Buterin, *Pyethereum source*, <https://www.github.com/ethereum/pyethereum/ethereum/opcodes.py>, 2017 (cit. on pp. 14–19).

Glossary

account state The state of a particular account—a section of the total world state. Comprises: the nonce, balance, storage root, and code hash of the account. [21](#)

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](#)

beneficiary The 20-character (160-bit) address to which all fees collected from the successful mining of a block are 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 are limited, usually only one, possible output yet that output is unique. [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](#)

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](#)

Message Data (as a set of bytes) and Value (specified in Wei) that is passed between two accounts. [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. [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 radix tree mapping bytarrays (organized chunks of binary data) to other bytarrays. The *relationships* between each node on this trie constitutes a MAPPING of Ethereum's state. [1](#), [4](#), [13](#), [21](#)

storage root One aspect of an ACCOUNT'S STATE: this is the hash of the trie^a that decides the STORAGE CONTENTS of the account. [21](#)

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. [21](#)

Acronyms

ERE Ethereum Runtime Environment. [21](#)

EVM Ethereum Virtual Machine. [21](#)

RLP Recursive Length Prefix. [21](#)

^aA particular path from root to leaf in the state database

Index

160 bit, 5
256 bit, 4, 5
50% attack, 13

abstract state-machine, 1
account, 9
account address, 7
account addresses, 2
account balance, 4
account body, 4
account code hash, 4
account creation, 4, 9
account init, 4
account nonce, 4
account state, 4
account states, 2
account storage root, 4
accrued substate, 8
accumulated gas used, 11
age, 13
ancestor node, 2
apply rewards, 5, 11
arbitrarily determined, 9
arbitrary length byte-array, 9
asic resistant, 12
autonomous objects, 9
available gas, 9

balance, 1
beneficiary, 3
beneficiary address, 11
big endian, 3
big endian function, 9
Bitcoin, 1
Bitcoin Whitepaper, 13
block, 9
block beneficiary, 5
block composition, 3
block contents, 9
block difficulty, 12
block finalization state transition function, 11
block header, 3, 7
block header validity function, 9
block number, 4, 11
block reward, 5, 11
block reward function, 11
block rewards, 3
BLOCKHASH, 13
body, 9
branch node, 2
byte array, 5
byzantium, 4

cache, 12
canonical blockchain, 11
canonical gas, 9
casper, 13
certainty, 1
checkpoint nodes, 13
checkpoints, 10
child node, 2
code array, 7
collisions, 11
complete block, 11
computation, 9
computation of operation, 6
compute valid nonce, 11
compute valid state, 11
contract creation, 4, 9
contract creation stack, 9
contract creation transactions, 9
controlled halt, 10
correct DAG, 9
cumulative difficulty, 12
cumulative gas, 11

DAG, 9
data structure, 2
dataset, 12
dataset slice, 13
descendant node, 2
deserialization, 5, 9

deterministic, 1
difficulty, 3, 9
difficulty bomb, 4
difficulty mechanism, 12
discard nodes, 13
double-spend problem, 13
dynamic difficulty homeostasis, 4

EIP 100, 4
EIP 2, 4
EIP 649, 4
elliptic curve, 5
elliptic curve computation, 5
elliptic curve cryptography, 5
empty byte-array, 3
empty byte-sequence, 9
empty set, 10
ere, 9
ethash, 12
ether, 1
ethereum runtime environment, 9
EVM, 5
EVM assembly, 10
EVM code, 10
evm computation instances, 7
exceptional halt, 6, 8
executed, 1
execution, 6
execution environment, 7
execution function, 8
execution model, 8
explicitly specify meaning, 10
exponential difficulty increase, 4
extra data, 3
extract remaining gas, 8

fees, 6
finalization function, 7
Finney, 1
forest, 2

gas, 6, 9
gas available, 10
gas deducted, 6
gas expenditure per block, 9
gas limit, 3, 5, 6
gas paid for increased use of memory, 6
gas price, 5, 6, 9
gas refund clearing space, 6
gas used, 3, 6, 9, 11
genesis block, 11
genesis difficulty, 4
GHOST protocol, 13
global state database, 3

halting function, 8
halting state, 8
hash scheme, 5
heaviest path, 11
homestead, 4
homestead difficulty parameter, 4

ice age, 4
incomplete block, 11
init, 8, 9
input data, 7
inspection of data, 3
instantial runtime, 7
intrinsic validity, 6
invalid instruction, 6
iterative progression, 8
iterator function, 8

keccak 256, 3, 5

leaf node, 2, 4
ledger, 1
log events, 6
log items, 6
log series, 8
logs bloom, 3, 10
logs series, 10
logs set, 10
longest chain, 13

machine halt, 6
machine instructions, 1

machine state, 5, 10
machine storage, 5
mapping, 2
mapping between account states, 3
mapping between addresses, 3
memory, 5
memory contents, 10
memory model, 5
memory model volatility of, 5
memory size, 5
memory stack, 5
memory usage fee, 6
memory word count, 10
merkle-patricia trees, 2
merkle-patricia tries, 2
merkletries, 2
message call, 6, 8, 9
message call transactions, 9
miner choice, 9
miners, 9
minimize storage use, 6
mining, 1
mining effort, 11
mix hash, 3, 9, 12
modified merkletries, 2

native currency, 1
natively execute, 10
nested binary data, 3
network cost unit, 9
newstate value, 7
no leading zeroes, 3
node depth, 2
node height, 2
node operator computer, 3
non empty deserialized integer, 3
non-standard architecture, 6
nonce, 3, 9
nonce validation, 11
number, 3

ommer, 11
ommer block headers, 4
ommer headers, 11
ommer validation, 11
ommers hash, 3
opcodes, 1
originator price, 7
out-of-gas, 6

parent hash, 3
parent node, 2
payment, 6
plasma, 13
positive integer, 9
post transaction state, 6
present depth, 9
probability statement, 12
program counter, 10
proof-of-work, 12
pseudocode, 1
pseudorandom number generation, 13

receipts root, 3
recipient, 9
refunded, 9
refunds, 8
remaining gas, 7, 8
report exception, 6
resultant output, 8
resultant state, 8
RLP, 2, 3
rlp, 9
RLP encodes as byte-arrays, 3
RLP integers, 3
root node, 2, 4

Satoshi Nakamoto, 13
scalability, 13
seed, 12
self-destructs set, 8
sender, 9
sender account, 6
sender address, 7
sequence concatenation, 9
serialization, 5, 9

sharding, 13
sibling node, 2
singleton, 1
slice, 12
speedy traversal of data, 3
stack based, 5, 7
stack based architecture, 5
stack contents, 10
stack underflow, 6
state database, 4, 13
state machine cycle, 8
state root, 3
state transition, 5
state transition function, 8
state unchanged, 6
status code, 11
storage model, 5
substates, 7
system state, 5, 7
Szabo, 1

tagged for self destruction, 10
time stamp, 9
timestamp, 3
timestamped, 1
to, 5
to execute, 10
total difficulty, 12
total fee, 6
total gas used, 11
totaly difficulty, 11
touched accounts, 8
transaction, 5
transaction execution, 6
transaction execution function, 11
transaction nonce, 6
transaction originator, 9
transaction receipt, 6, 10

transaction series, 10
transaction signature, 6
transaction validation, 11
transactions, 11
transactions root, 3
tree arbitrary depth, 3
tree database, 2
tree degree, 2
tree edge, 2
tree height, 2
tree level, 2
tree path, 2
trie database, 2
trusted, 1

universal gas, 9
unstoppable, 1
unused gas, 9
upfront payment, 6

valid header, 11
valid state, 9
value, 5, 9
verification, 5
virtual machine, 1, 7
virtual ROM, 6

Wei, 1
wei, 4
well defined memory, 6
well defined storage, 6
well-formed RLP, 6
word addressable, 5
word addressed, 5
word array, 5
word size, 5
world state, 2

Yellowpaper, 1