Off-Whitepaper

Ethereum

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 people can execute machine codes with the same veracity and certainty that one has when sending a Bitcoin transaction. Just as sure that 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. 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, and are recorded, and are 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 Ξ 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	Ξ**1**.000000000000000000	1,000,000,000,000,000,000
Finney	Ξ0.001**0**00000000000000	1,000,000,000,000,000
Szabo	Ξ0.000001**0**00000000000	1,000,000,000,000
Wei	Ξ0.000000000000000001	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. Total Difficulty

The *Total Difficulty* of a block is defined recursively by a function which calculates the difficulty of all blocks prior to the header in the present block.

2.1.2. 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.[3]

Pseudocode	Definition
`total(difficulty)`	Total difficulty at *this block*.
`block(difficulty)`	*This block's* difficulty.

2.2. GHOST Protocol

Notation : `ghost`

Description : Stands for greedy heaviest object subtree; the GHOST Protocol determines the probable correctness of the next block based on the number of miners attaching to it.

3. Memory and Storage

3.1. Data Structures

3.1.1. Merkle-Patricia Trees

Notation : `merkle_p_t`

Description : Generalized Merkle DHT

As blockchain technologies move beyond the "1.0" model of every node processing every transaction, and a more diverse ecosystem including "light clients" that achieve security by downloading only a small portion of the blockchain and extracting the rest of the data on-demand through hash-based authentication comes into play, and particularly in the long term as scalability models essentially turn _every_ node into a light client, there arises the need to develop a strong, robust and effective networking infrastructure to handle the load. Ideally, the core technology should be built to be maximally generalized, so that the same core code and network can be used for multiple blockchain, as well as non-blockchain, applications.

3.1.2. World State

Also known as *Actual State*, 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.[a] that maintains a mapping of bytearrays to bytearrays.[b] 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*.[c]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.

3.1.3. The Block

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

3.1.4. Block Header

Notation : `header`

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

[a]The database backend is accessed by users through an external application, most likely an Ethereum client; see also: state database

[b]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.

[c]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.

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. **Ommer Block Headers** – These are the same components listed above for any ommers.

3.1.5. Block Footer

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

3.1.6. Block Number

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 be- tween blocks varies, as discussed below, as implemented in EIP-2. In the Homestead release, the expo- nential 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 above, and the denominator 9, in order to target the mean block time including uncle blocks Buterin [2016]. Finally, in the Byzantium release, with EIP-649, the ice age was delayed by creating a fake block number, which is obtained by substracting 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.[3]

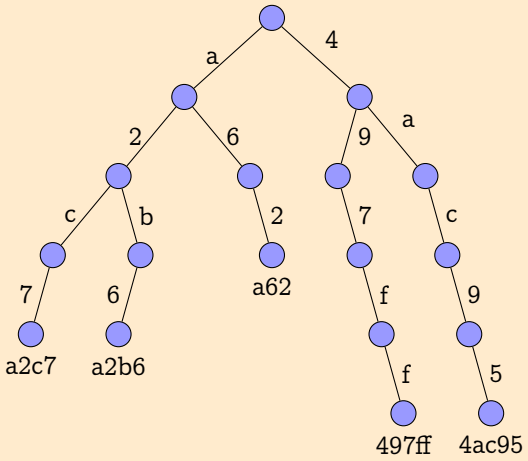
3.2. State Database

3.2.1. Tries

Notation : `trie`

Description : Also known as a search tree, a radix tree, or a prefix tree, a trie is tree structure for organizing sequential data hierarchically. Members of a previous generation can spawn infinitely many members of the next generation, but the data cannot be read out of sequence. In other words a parent can spawn any number of children, but cannot skip a generation. This ensures that all the data in a trie are related to each other properly.

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



3.2.2. Tree Terminology[1]

a) Root– The top (first) node in a tree.

b) Child– A node directly connected to another node when moving away from the Root.

c) Parent– The converse notion of a child.

d) Siblings– A group of nodes with the same parent.

e) Descendant– A node reachable by repeated proceeding from parent to child.

f) Ancestor– A node reachable by repeated proceeding from child to parent.

g) Leaf[a]– A node with no children.

h) Branch[b]– 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 ≥ 0 disjoint trees.

[a] a.k.a. External Node
[b] a.k.a. Internal Node

3.2.3. RLP

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 adddress (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.[2]

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.[3]

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.

In other words:

3.2.4. Create Account

Notation : `create`

Description : Account creation definitively accours in the YP with contract creation. Does account creation also apply to regular account creation, or does it only apply to contract accounts? Also, what about protocol accounts?

Related : `init`

3.2.5. Account State

Notation : `body`

Description : The EVM-code fragment that executes each time an account receives a message call.

Description : 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 number of Wei owned by this address.
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.
4. The storage root aspect of an account's state is the hash of the trie
5. **code_hash** The hash of the EVM code of this account's contract.

The account state is the state of any particular account during some specified world state.

Nonce The **nonce** aspect of an ACCOUNT'S STATE is the number of transactions sent from, or the number of contract-creations by, the address of that account.[a]

Storage Root The **storage root** aspect of an ACCOUNT'S STATE is the hash of the trie[b]

Code Hash The **code hash** aspect of an ACCOUNT'S STATE is the HASH OF THE EVM CODE of this account. 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.

Balance The amount of **Wei** OWNED by this account.

- Key/value pair stored inside the root hash.
- L_I^*, is defined as the element-wise transformation of the base function
- The *element-wise transformation of the base-function* refers to all of the key/value pairs in L_I
- L_I refers to a particular trie.

3.3. Bloom Filter

Notation : `logs_bloom`

Description : 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.

3.3.1. Transaction Receipts

4. Processing and Computation

4.1. The Transaction

The basic method for Ethereum accounts to interact with eachother. 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 temporary changes in computation that must be made before a block is finalized and added to the world state.

[a] σ is the world state at a certain given time, and n is the number of transactions or contract creations by that account.
[b] A particular path from root to leaf in the **state database** that encodes the STORAGE CONTENTS of the account.

Description : A single cryptographically signed instruction sent to the Ethereum network. There are two types of transactions: MESSAGE CALLS and CONTRACT CREATIONS. Transactions are ubiquitous on the Ethereum network, and represent several common fields.

4.1.1. Transactions Root

Notation : `listhash`

Alternatively: Transactions Root

Description : The `K256` hash of the root node[a] 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.[b]
5. **Value** The number of Wei to be transferred to the recipient of a message call.[c]
6. **v, r, s**
7.
8.

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 computing changes to the *machine state* prior to adding them to the *world state*, that is, through them being finalized and rewards applied to a given miner. `apply_rewards` and `block_beneficiary` are here. subsectionMining

Block Beneficiary 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 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. This can be calculated from the previous block's difficulty level and the timestamp.

4.2.1. Ethash

GHOST Protocol

4.3. Verification

Notation : `verification`

Description : The process in The EVM that verifies Ommer Headers

4.3.1. Ommers

Ommershash

Notation : `sender`

Description : A function that maps transactions to their sender using `ECDSA` of the SECP-256k1 curve, (excepting the latter three signature fields) as the datum to sign. The sender of a given transaction can be represented: `transaction.sender`

4.4. Serialization/Deserialization

Notation : `big_endian`

Description : 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.

[a] A root node is a type of progenitor node
[b] In the case of a contract creation this is 0x000000000000000000.
[c] In the case of a contract creation, an endowment to the newly created contract account.

4.5. Ethereum Virtual Machine

The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack BYZANTIUM VERSION 3475aa8 - 2018-01-26 11 item) is 256-bit. This was chosen to facilitate the Keccak- 256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. 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 Neu- mann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored sepa- rately in a virtual ROM interactable only through a specialised instruction. The machine can have exceptional execution for several reasons, including stack underflows and invalid instruc- tions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immedi- ately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning ex- ecution environment) which will deal with it separately. 9.2. Fees Overview. Fees (denominated in gas) are charged under three distinct circumstances, all three as pre-requisite 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 previ- ously 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 in- centivise minimisation of the use of storage (which corre- sponds 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.[3]

4.5.1. 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.

4.5.2. 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.

4.5.3. 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 concering 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`[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:

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 effeffect 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.

[a] Designated "`intrinsic_gas`" in the Yellowpaper

4.5.4. 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 instantial 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 $\boldsymbol{\sigma}$, 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.[3]
- `code.array` the byte array that is the machine code to be executed.[3]
- `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).[3]

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`.

4.5.5. 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. 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` evaluating to the full cost, in gas, of executing the given instruction.

4.5.6. 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.

4.6. 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 `l ogs_bloom` from the `tx_receipt`.
- The refund balance.[a]

4.6.1. 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

[a] 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 takes 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.[3] 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.

4.6.2. 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.

4.6.3. Account Creation

4.7. 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.

4.8. 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.

4.8.1. 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 can- not propagate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because its satisfaction de- pends on the contents of the block and in turn its com- posed 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.

4.8.2. 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

4.8.3. 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 are not 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[3] 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 `l ogs_bloom` from the `tx_receipt`.
- The refund balance.[a]

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.

4.8.4. EVM Code

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

Notation : `contract`

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

4.9. 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.

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

4.10. Ommer Validation

The validation of ommer headers means nothing more than verifying that each om- mer header is both a valid header and satisfies the rela- tion of N th-generation ommer to the present block where N 6. 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.

4.11. 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.

4.12. 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 ommers and the block two ommers with the same beneficiary address or an ommer with the same beneficiary address as the present block,

[a] The SSTORE operation increases the amount refunded by resetting contract storage to zero from some non-zero state.

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 trans- action (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.

4.13. Mining Proof-of-Work

The mining proof-of- work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the deter- mination of some token value n. It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism. For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as possible, and, ideally, makes the act of mining a simple swap from electricity to Ether at roughly the same rate for anyone around the world. Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network's total mining power and as such gives them a super-linear reward (thus skew- ing distribution in their favour) as well as reducing the network security. One plague of the Bitcoin world is ASICs. These are specialised pieces of compute hardware that exist only to do a single task. In Bitcoin's case the task is the SHA256 hash function. While ASICs exist for a proof-of-work func- tion, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. diffi- cult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the nonce requires a lot of mem- ory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of "specialised hard- ware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop com- puters are likely to be pretty close to "specialised hard- ware" for the task. For Ethereum 1.0 we have chosen the first path. More formally, the proof-of-work function takes the form of 2 256 m = H m is the new block's header but without the nonce and mix-hash compo-

nents; H n is the nonce of the header; d is a large data set needed to compute the mix- Hash and H d is the new block's difficulty value (i.e. the block difficulty from section 10). PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudo-random number cryptographically dependent on H and d. The underlying algorithm is called Ethash and is described below. 11.5.1. Ethash. Ethash is the PoW algorithm for Ethereum 1.0. It is the latest version of Dagger- Hashimoto, introduced by Buterin [2013b] and Dryja [2014], although it can no longer appropriately be called that since many of the original features of both algorithms have been drastically changed in the last month of research and development. 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, J datasetinit 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 ev- ery J 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. A data feed contract is one which pro- vides a single service: it gives access to information from the external world within Ethereum. The accuracy and timeliness of this information is not guaranteed and it is BYZANTIUM VERSION 3475aa8 - 2018-01-26 15 the task of a secondary contract author—the contract that utilises the data feed—to determine how much trust can be placed in any single data feed. The general pattern involves a single contract within Ethereum which, when given a message call, replies with some timely information concerning an external phenom- enon. An example might be the local temperature of New York City. This would be implemented as a contract that returned that value of some known point in storage. Of course this point in storage must be maintained with the correct such temperature, and thus the second part of the pattern would be for an external server to run an Ethereum node, and immediately on discovery of a new block, creates a new valid transaction, sent to the contract, updating said value in storage. The contract's code would accept such updates only from the identity contained on said server. 12.2. Random Numbers. Providing random numbers within a deterministic system is, naturally, an impossible task. However, we can approximate with pseudo-random numbers by utilising data which is generally unknowable at the time of transacting. Such data might include the block's hash, the block's timestamp and the block's benefi- ciary address. In order to make it hard for malicious miner to control those values, one should use the BLOCKHASH operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result. 13. Future Directions 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 maxi- mum limit on the amount of computation needed in order to retrieve any state throughout the blockchain. Blockchain consolidation could be used in order to re- duce 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 con- ducted: 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. 13.1. Scalability. Scalability remains an eternal con- cern. With a generalised state transition function, it be- comes difficult to partition and parallelise transactions to apply the divide-and-conquer strategy. Unaddressed, the dynamic value-range of the system remains essentially fixed and as the average transaction value increases, the less valuable of them become ignored, being economically pointless to include in the main ledger. However, several strategies exist that may potentially be exploited to pro- vide a considerably more scalable protocol.ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER Some form of hierarchical structure, achieved by ei- ther consolidating smaller lighter-weight chains into the main block or building the main block through the in- cremental combination and adhesion (through proof-of- work) of smaller transaction sets may allow parallelisa- tion of transaction combination and block-building. Par- allelism could also come from a prioritised set of parallel blockchains, consolidated each block and with duplicate or invalid transactions thrown out accordingly. Finally, verifiable computation, if made generally avail- able and efficient enough, may provide a route to allow the proof-of-work to be the verification of final state. 14. Conclusion I have introduced, discussed and formally defined the protocol of Ethereum. Through this protocol the reader may implement a node on the Ethereum network and join others in a decentralised secure social operating system. Contracts may be authored in order to algorithmically specify and autonomously enforce rules of interaction. 15. Acknowledgements Many thanks to Aeron Buchanan for authoring the Homestead revisions, Christoph Jentzsch for authoring the Ethash algorithm and Yoichi Hirai for doing most of the EIP-150 changes. Important maintenance, useful corrections and suggestions were provided by a number of oth- ers from the Ethereum DEV organisation and Ethereum community at large including Gustav Simonsson, Pawe l Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari and, of course, Vitalik Buterin.

4.13.1. EVM Assembly

The human readable version of EVM Code (opcodes).

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. [a]
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

[a] 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. [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.
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.

[a] 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 *and* 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

B.4. Viper

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 (cit. on p. 4).

[2] E. Foundation, *Ethereum whitepaper*, https://github.com/ethereum/wiki/wiki/White-Paper, 2017 (cit. on p. 4).

[3] D. G. Wood, *Ethereum: A secure decentralised generalised transaction ledger*, https://github.com/ethereum/yellowpaper, 2017 (cit. on pp. 2–4, 7, 8, 10, 11).

Glossary

addresses 20 character strings, specifically the rightmost 20 characters of the Keccak-256 hash of the RLP-derived mapping which contains the sender's address and the nonce of the block.. 23

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. 23

beneficiary The 20-character (160-bit) address to which all fees collected from the successful mining of this block be transferred. 23

block header All the information in a block besides transaction information. 23

Contract A piece of EVM Code that may be associated with an Account or an Autonomous Object. 23

Cryptographic hashing functions Hash functions make secure blockchains possible by establishing universal inputs for which there is typically only one possible output.. 23

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. 23

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.. 23

EVM Assembly The human readable version of EVM code. 23

EVM Code The bytecode that the EVM can natively execute. Used to formally specify the meaning and ramifications of a message to an Account. 23

Gas The fundamental network cost unit; gas is paid for exclusively by Ether. 23

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. 23

Message Data (as a set of bytes) and Value (specified in Wei) that is passed between two accounts.. 23

Recursive Length Prefix Recursive Length Prefix. 23

root node the uppermost node in a particular tree, of blocks, representing a single world state at a particular time. 2, 23

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**]. 23

state machine The term *State Machine* is reserved for any simple or complex process that moves deterministically from one discrete state to the next.. 23

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 MAPPING of all previous *world states* which a client might need to reference. 2, 5, 23

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. 23

[a] 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. 23

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**]. 23

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, 5, 23

Acronyms

ERE Ethereum Runtime Environment. 23

EVM Ethereum Virtual Machine. 23

RLP Recursive Length Prefix. 23

Index