ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

FINAL DRAFT - UNDER REVIEW

DR. GAVIN WOOD
CO-FOUNDER & LEAD, ETHEREUM PROJECT
GAVIN@ETHEREUM.ORG

Abstract. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, not least Bitcoin. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. Introduction

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated, through the power of the default, consensus mechanisms and voluntary respect of the social contract that it is possible to use the internet to make a decentralised value-transfer system, shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original "currency application" of the technology into other applications albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. Driving Factors.

There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience or corruption of existing legal systems. By specifying a state-change system through a rich and unambiguous language, and furthermore architecting a system such that we can reasonably expect that an agreement will be thus enforced autonomously, we can provide a means to this end.

Dealings in this proposed system would have several attributes not often found in the real world. The incorruptibility of judgement, often difficult to find, comes naturally from a disinterested algorithmic interpreter. Transparency, or being able to see exactly how a state or judgement came about through the transaction log and rules or instructional codes, never happens perfectly in human-based systems since natural language is necessarily vague, information is often lacking, and plain old prejudices are difficult to shake.

Overall, I wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

1.2. Previous Work.

Buterin [2013] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

Hashcash, introduced by Back [2002] (in a five-year retrospective), provided the first work into the usage of a cryptographic proof of computational expenditure as a means of transmitting a value signal over the Internet. Though not widely adopted, the work was later utilised and expanded upon by Nakamoto [2008] in order to devise a cryptographically secure mechanism for coming to a decentralised social consensus over the order and contents of a series of cryptographically signed financial transactions. The fruits of this project, Bitcoin, provided a first glimpse into a decentralised transaction ledger.

Other projects built on Bitcoin's success; the alt-coins introduced numerous other currencies through alteration to the protocol. Some of the best known are Litecoin and Primecoin, discussed by Sprankel [2013]. Other projects sought to take the core value content mechanism of the protocol and repurpose it; Aron [2012] discusses, for example, the Namecoin project which aims to provide a decentralised name-resolution system.

Other projects still aim to build upon the Bitcoin network itself, leveraging the large amount of value placed in the system and the vast amount of computation that goes into the consensus mechanism. The Mastercoin project, first proposed by Willett [2013], aims to build a richer protocol involving many additional high-level features on top of the Bitcoin protocol through utilisation of a number of auxiliary parts to the core protocol. The Coloured Coins project, proposed by Rosenfeld [2012], takes a similar but more simplified strategy, embellishing the rules

of a transaction in order to break the fungibility of Bitcoin's base currency and allow the creation and tracking of tokens through a special "chroma-wallet"-protocol-aware piece of software.

Additional work has been done in the area with discarding the decentralisation foundation; Ripple, discussed by Boutellier and Heinzen [2014], has sought to create a "federated" system for currency exchange, effectively creating a new financial clearing system. It has demonstrated that high efficiency gains can be made if the decentralisation premise is discarded.

Early work on smart contracts has been done by Szabo [1997] and Miller [1997]. Around the 1990s it became clear that algorithmic enforcement of agreements could become a significant force in human cooperation. Though no specific system was proposed to implement such a system, it was proposed that the future of law would be heavily affected by such systems. In this light, Ethereum may be seen as a general implementation of such a *crypto-law* system.

2. The Blockchain Paradigm

Ethereum, taken as a whole, can be viewed as a transaction-based state machine: we begin with a genesis state and incrementally execute transactions to morph it into some final state. It is this final state which we accept as the canonical "version" of the world of Ethereum. The state can include such information as account balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissible. Transactions thus represent a valid arc between two states; the 'valid' part is important—there exist far more invalid state changes than valid state changes. Invalid state changes might, e.g. be things such as reducing an account balance without an equal and opposite increase elsewhere. A valid state transition is one which comes about through a transaction. Formally:

$$\boldsymbol{\sigma}_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

where Υ is the Ethereum state transition function. In Ethereum, Υ, together with $\boldsymbol{\sigma}$ are considerably more powerful then any existing comparable system; Υ allows components to carry out arbitrary computation, while $\boldsymbol{\sigma}$ allows components to store arbitrary state between transactions.

Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state (though do not store the final state itself—that would be far too big). They also punctuate the transaction series with incentives for nodes to *mine*. This incentivisation takes places as a state-transition function, adding value to a nominated account.

Mining is the process of dedicating effort (working) to bolster one series of transactions (a block) over any other potential competitor block. It is achieved thanks to a cryptographically secure proof. This scheme is known as a proof-of-work and is discussed in detail in section 11.5.

Formally, we expand to:

$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{2}$$

$$B \equiv (..., (T_0, T_1, ...)) \tag{3}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)...) \tag{4}$$

Where Ω is the block-finalisation state transition function (a function that rewards a nominated party); B is this block, which includes a series of transactions amongst some other components; and Π is the block-level state-transition function.

This is the basis of the blockchain paradigm, a model that forms the backbone of not only Ethereum, but all decentralised consensus-based transaction systems to date.

2.1. **Value.** In order to incentivise computation within the network, there needs to be an agreed method for transmitting value. To address this issue, Ethereum has an intrinsic currency, Ether, known also as ETH and sometimes referred to by the Old English Ð. The smallest subdenomination of Ether, and thus the one in which all integer values of the currency are counted, is the Wei. One Ether is defined as being 10^{18} Wei. There exist other subdenominations of Ether:

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

Throughout the present work, any reference to value, in the context of Ether, currency, a balance or a payment, should be assumed to be counted in Wei.

2.2. **Which History?** Since the system is decentralised and all parties have an opportunity to create a new block on some older pre-existing block, the resultant structure is necessarily a tree of blocks. In order to form a consensus as to which path, from root (the genesis block) to leaf (the block containing the most recent transactions) through this tree structure, known as the blockchain, there must be an agreed-upon scheme. If there is ever a disagreement between nodes as to which root-to-leaf path down the block tree is the 'best' blockchain, then a *fork* occurs.

This would mean that past a given point in time (block), multiple states of the system may coexist: some nodes believing one block to contain the canonical transactions, other nodes believing some other block to be canonical, potentially containing radically different or incompatible transactions. This is to be avoided at all costs as the uncertainty that would ensue would likely kill all confidence in the entire system.

The scheme we use in order to generate consensus is a simplified version of the GHOST protocol introduced by Sompolinsky and Zohar [2013]. This process is described in detail in section 10.

3. Conventions

I use a number of typographical conventions for the formal notation, some of which are quite particular to the present work:

The two sets of highly structured, 'top-level', state values, are denoted with bold lowercase Greek letters. They fall into those of world-state, which are denoted $\boldsymbol{\sigma}$ (or a variant thereupon) and those of machine-state, $\boldsymbol{\mu}$.

Functions operating on highly structured values are denoted with an upper-case greek letter, e.g. Υ, the Ethereum state transition function.

For most functions, an uppercase letter is used, e.g. C, the general cost function. These may be subscripted to denote specialised variants, e.g. C_{SSTORE}, the cost function for the SSTORE operation. For specialised and possibly externally defined functions, I may format as typewriter text, e.g. the Keccak-256 hash function (as per the winning entry to the SHA-3 contest) is denoted KEC (and generally referred to as plain Keccak).

Tuples are typically denoted with an upper-case letter, e.g. T, is used to denote an Ethereum transaction. This symbol may, if accordingly defined, be subscripted to refer to an individual component, e.g. T_n, denotes the nonce of said transaction. The form of the subscript is used to denote its type; e.g. uppercase subscripts refer to tuples with subscriptable components.

Scalars and fixed-size byte sequences (or, synonymously, arrays) are denoted with a normal lower-case letter, e.g. n is used in the document to denote a transaction nonce. Those with a particularly special meaning may be greek, e.g. δ, the number of items required on the stack for a given operation.

Arbitrary-length sequences are typically denoted as a bold lower-case letter, e.g. $\mathbf{o}$ is used to denote the byte-sequence given as the output data of a message call. For particularly important values, a bold uppercase letter may be used.

Throughout, we assume scalars are positive integers and thus belong to the set $\mathbb{P}$. The set of all byte sequences is $\mathbb{B}$, formally defined in Appendix B. If such a set of sequences is restricted to those of a particular length, it is denoted with a subscript, thus the set of all byte sequences of length 32 is named $\mathbb{B}_{32}$ and the set of all positive integers smaller than 2^{256} is named $\mathbb{P}_{256}$. This is formally defined in section 4.3.

Square brackets are used to index into and reference individual components or subsequences of sequences, e.g. $\boldsymbol{\mu}_{\mathbf{s}}[0]$ denotes the first item on the machine's stack. For subsequences, ellipses are used to specify the intended range, to include elements at both limits, e.g. $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ denotes the first 32 items of the machine's memory.

In the case of the global state $\boldsymbol{\sigma}$, which is a sequence of accounts, themselves tuples, the square brackets are used to reference an individual account.

When considering variants of existing values, I follow the rule that within a given scope for definition, if we assume that the unmodified 'input' value be denoted by the placeholder $\square$ then the modified and utilisable value is denoted as $\square'$, and intermediate values would be $\square^*$, $\square^{**}$ &c. On very particular occasions, in order to maximise readability and only if unambiguous in meaning, I may use alpha-numeric subscripts to denote intermediate values, especially those of particular note.

When considering the use of existing functions, given a function f, the function f^* denotes a similar, element-wise version of the function mapping instead between sequences. It is formally defined in section 4.3.

I define a number of useful functions throughout. One of the more common is ℓ, which evaluates to the last item in the given sequence:

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{5}$$

4. Blocks, State and Transactions

Having introduced the basic concepts behind Ethereum, we will discuss the meaning of a transaction, a block and the state in more detail.

4.1. World State.

The world state (*state*), is a mapping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix B). Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree (*trie*, see Appendix D). The trie requires a simple database backend that maintains a mapping of bytearrays to bytearrays; we name this underlying database the state database. This has a number of benefits; firstly the root node of this structure is cryptographically dependent on all internal data and as such its hash can be used as a secure identity for the entire system state. Secondly, being an immutable data structure, it allows any previous state (whose root hash is known) to be recalled by simply altering the root hash accordingly. Since we store all such root hashes in the blockchain, we are able to trivially revert to old states.

The account state comprises the following four fields:

nonce: A scalar value equal to the number of transactions sent from this address or, in the case of accounts with associated code, the number of contract-creations made by this account. For account of address a in state $\boldsymbol{\sigma}$, this would be formally denoted $\boldsymbol{\sigma}[a]_n$.

balance: A scalar value equal to the number of Wei owned by this address. Formally denoted $\boldsymbol{\sigma}[a]_b$.

storageRoot: A 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account (a mapping between 256-bit integer values), encoded into the trie as a mapping from the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted $\boldsymbol{\sigma}[a]_s$.

codeHash: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call; it is immutable and thus, unlike all other fields, cannot be changed after construction. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted $\boldsymbol{\sigma}[a]_c$, and thus the code may be denoted as $\mathbf{b}$, given that $\text{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$.

Since I typically wish to refer not to the trie's root hash but to the underlying set of key/value pairs stored within, I define a convenient equivalence:

$$\text{TRIE}\big(L_I^*(\boldsymbol{\sigma}[a]_{\mathbf{s}})\big) \equiv \boldsymbol{\sigma}[a]_s \tag{6}$$

The collapse function for the set of key/value pairs in the trie, L_I^*, is defined as the element-wise transformation of the base function L_I, given as:

$$L_I\big((k, v)\big) \equiv \big(k, \text{RLP}(v)\big) \tag{7}$$

where:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{P} \tag{8}$$

It shall be understood that $\boldsymbol{\sigma}[a]_{\mathbf{s}}$ is not a 'physical' member of the account and does not contribute to its later serialisation.

If the **codeHash** field is the Keccak-256 hash of the empty string, i.e. $\boldsymbol{\sigma}[a]_c = \texttt{KEC}(())$, then the node represents a simple account, sometimes referred to as a "non-contract" account.

Thus we may define a world-state collapse function L_S:

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{9}$$

where

$$p(a) \equiv \big(a, \texttt{RLP}\big((\boldsymbol{\sigma}[a]_n, \boldsymbol{\sigma}[a]_b, \boldsymbol{\sigma}[a]_s, \boldsymbol{\sigma}[a]_c)\big)\big) \tag{10}$$

This function, L_S, is used alongside the trie function to provide a short identity (hash) of the world state. We assume:

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{11}$$

where v is the account validity function:

$$v(x) \quad \equiv \quad x_n \in \mathbb{P}_{256} \wedge x_b \in \mathbb{P}_{256} \wedge x_s \in \mathbb{B}_{32} \wedge x_c \in \mathbb{B}_{32} \tag{12}$$

4.2. **The Transaction.** A transaction (formally, T) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. It is assumed that the external actor will be human in nature, though software tools will be used in its construction and dissemination. There are two types of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as 'contract creation'). Both types specify a number of common fields:

nonce: A scalar value equal to the number of transactions sent by the sender; formally T_n.

gasPrice: A scalar value equal to the number of Wei to be paid per unit of *gas* for all computation costs incurred as a result of the execution of this transaction; formally T_p.

gasLimit: A scalar value equal to the maximum amount of gas that should be used in executing this transaction. This is paid up-front, before any computation is done and may not be increased later; formally T_g.

to: The 160-bit address of the message call's recipient or, for a contract creation transaction, $\varnothing$, used here to denote the only member of $\mathbb{B}_0$; formally T_t.

value: A scalar value equal to the number of Wei to be transferred to the message call's recipient or, in the case of contract creation, as an endowment to the newly created account; formally T_v.

v, r, s: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally T_w, T_r and T_s. This is expanded in Appendix F.

Additionally, a contract creation transaction contains:

init: An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally $T_{\mathbf{i}}$.

init is an EVM-code fragment; it returns the **body**, a second fragment of code that executes each time the account receives a message call (either through a transaction or due to the internal execution of code). **init** is executed only once at account creation and gets discarded immediately thereafter.

In contrast, a message call transaction contains:

data: An unlimited size byte array specifying the input data of the message call, formally $T_{\mathbf{d}}$.

Appendix F specifies the function, S, which maps transactions to the sender, and happens through the ECDSA of the SECP-256k1 curve, using the hash of the transaction (excepting the latter three signature fields) as the datum to sign. For the present we simply assert that the sender of a given transaction T can be represented with $S(T)$.

$$L_T(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}, T_w, T_r, T_s) & \text{if } T_t = \varnothing \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}, T_w, T_r, T_s) & \text{otherwise} \end{cases} \tag{13}$$

Here, we assume all components are interpreted by the RLP as integer values, with the exception of the arbitrary length byte arrays $T_{\mathbf{i}}$ and $T_{\mathbf{d}}$.

$$\begin{array}{lllllll} T_n \in \mathbb{P}_{256} & \wedge & T_v \in \mathbb{P}_{256} & \wedge & T_p \in \mathbb{P}_{256} & \wedge \\ T_g \in \mathbb{P}_{256} & \wedge & T_w \in \mathbb{P}_5 & \wedge & T_r \in \mathbb{P}_{256} & \wedge \\ T_s \in \mathbb{P}_{256} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} & \end{array} \tag{14}$$

where

$$\mathbb{P}_n = \{P : P \in \mathbb{P} \wedge P < 2^n\} \tag{15}$$

The address hash $T_{\mathbf{t}}$ is slightly different: it is either a 20-byte address hash or, in the case of being a contract-creation transaction (and thus formally equal to $\varnothing$), it is the RLP empty byte-series and thus the member of $\mathbb{B}_0$:

$$T_t \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_t \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases} \tag{16}$$

4.3. **The Block.** The block in Ethereum is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$, and a set of other block headers $\mathbf{U}$ that are known to have a parent equal to the present block's parent's parent (such blocks are known as *uncles*). The block header contains several pieces of information:

parentHash: The Keccak 256-bit hash of the parent block's header, in its entirety; formally H_p.

unclesHash: The Keccak 256-bit hash of the uncles list portion of this block; formally H_u.

coinbase: The 160-bit address to which all fees collected from the successful mining of this block be transferred; formally H_c.

stateRoot: The Keccak 256-bit hash of the root node of the state trie, after all transactions are executed and finalisations applied; formally H_r.

transactionsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each transaction in the transactions list portion of the block; formally H_t.

receiptsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with the receipts of each transaction in the transactions list portion of the block; formally H_e.

logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of

each transaction in the transactions list; formally H_b.

difficulty: A scalar value corresponding to the difficulty level of this block. This can be calculated from the previous block's difficulty level and the timestamp; formally H_d.

number: A scalar value equal to the number of ancestor blocks. The genesis block has a number of zero; formally H_i.

gasLimit: A scalar value equal to the current limit of gas expenditure per block; formally H_l.

gasUsed: A scalar value equal to the total gas used in transactions in this block; formally H_g.

timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception; formally H_s.

extraData: An arbitrary byte array containing data relevant to this block. This must be 1024 bytes or fewer; formally H_x.

nonce: A 256-bit hash which proves that a sufficient amount of computation has been carried out on this block; formally H_n.

The other two components in the block are simply a list of uncle block headers (of the same format as above) and a series of the transactions. Formally, we can refer to a block B:

$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}}) \tag{17}$$

4.3.1. *Transaction Receipt.* In order to encode information about a transaction concerning which it may be useful to form a zero-knowledge proof, or index and search, we encode a receipt of each transaction containing certain information from concerning its execution. Each receipt, denoted $B_{\mathbf{R}}[i]$ for the ith transaction) is placed in an index-keyed trie and the root recorded in the header as H_e.

The transaction receipt is a tuple of four items comprising the post-transaction state, $R_{\boldsymbol{\sigma}}$, the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened, R_u, the set of logs created through execution of the transaction, $R_{\mathbf{l}}$ and the Bloom filter composed from information in those logs, R_b:

$$R \equiv (R_{\boldsymbol{\sigma}}, R_u, R_b, R_{\mathbf{l}}) \tag{18}$$

The function L_R trivially prepares a transaction receipt for being transformed into an RLP-serialised byte array:

$$L_R(R) \equiv (\texttt{TRIE}(L_S(R_{\boldsymbol{\sigma}})), R_u, R_b, R_{\mathbf{l}}) \tag{19}$$

thus the post-transaction state, $R_{\boldsymbol{\sigma}}$ is encoded into a trie structure, the root of which forms the first item.

We assert R_u, the cumulative gas used is a positive integer and that the logs Bloom, R_b, is a hash of size 512 bits (64 bytes):

$$R_u \in \mathbb{P} \quad \wedge \quad R_b \in \mathbb{B}_{64} \tag{20}$$

The log entries, $R_{\mathbf{l}}$, is a series of log entries, termed, for example, $(O_0, O_1, ...)$. A log entry, O, is a tuple of a logger's address, O_a, a series of 32-bytes log topics, $O_{\mathbf{t}}$ and some number of bytes of data, $O_{\mathbf{d}}$:

$$O \equiv (O_a, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{21}$$

$$O_a \in \mathbb{B}_{20} \quad \wedge \quad \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{22}$$

We define the Bloom filter function, M, to reduce a log entry include a single 64-byte hash:

$$M(O) \equiv \bigvee_{t \in \{O_a\} \cup O_{\mathbf{t}}} \left(M_{3:512}(t)\right) \tag{23}$$

where $M_{3:512}$ is a specialised Bloom filter that sets three bits out of 512, given an arbitrary byte series. It does this through taking the low-order 9 bits of each of the first three pairs of bytes in a Keccak-256 hash of the byte series. Formally:

$$M_{3:512}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{64} \quad \text{where:} \tag{24}$$

$$\mathbf{y} = (0, 0, ..., 0) \quad \text{except:} \tag{25}$$

$$\forall_{i \in \{0,2,4\}} : \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{26}$$

$$m(\mathbf{x}, i) \equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 512 \tag{27}$$

where $\mathcal{B}$ is the bit reference function such that $\mathcal{B}_j(\mathbf{x})$ equals the bit of index j (indexed from 0) in the byte array $\mathbf{x}$.

4.3.2. *Holistic Validity.* We can assert a block's validity if and only if it satisfies several conditions: it must be internally consistent with the uncle and transaction block hashes and the given transactions $B_{\mathbf{T}}$ (as specified in sec 11), when executed in order on the base state $\boldsymbol{\sigma}$ (derived from the final state of the parent block), result in a new state of the identity H_r:

$$\begin{aligned}
H_r &\equiv \texttt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) &\wedge\\
H_u &\equiv \texttt{KEC}(\texttt{RLP}(L^*_H(B_{\mathbf{U}}))) &\wedge\\
H_t &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P} : p(i, L_T(B_{\mathbf{T}}[i]))\}) &\wedge\\
H_e &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P} : p(i, L_R(B_{\mathbf{R}}[i]))\}) &\wedge\\
H_b &\equiv \textstyle\bigvee_{\mathbf{r} \in B_{\mathbf{R}}} \left(\mathbf{r}_b\right)
\end{aligned} \tag{28}$$

where $p(k, v)$ is simply the pairwise RLP transformation, in this case, the first being the index of the transaction in the block and the second being the transaction receipt:

$$p(k, v) \equiv \left(\texttt{RLP}(k), \texttt{RLP}(v)\right) \tag{29}$$

Furthermore:

$$\texttt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_r} \tag{30}$$

Thus $\texttt{TRIE}(L_S(\boldsymbol{\sigma}))$ is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of the state $\boldsymbol{\sigma}$ with values encoded using RLP, and $P(B_H)$ is the parent block of B, defined directly.

The values stemming from the computation of transactions, specifically the transaction receipts, $B_{\mathbf{R}}$, and that defined through the transactions state-accumulation function, Π, are formalised later in section 11.4.

4.3.3. *Serialisation.* The function L_B and L_H are the preparation functions for a block and block header respectively. Much like the transaction receipt preparation function L_R, we assert the types and order of the structure for when the RLP transformation is required:

$$\begin{aligned}
L_H(H) \equiv (\ & H_p, H_u, H_c, H_r, H_t, H_e, H_b, H_d,\\
& H_i, H_l, H_g, H_s, H_x, H_n\ )
\end{aligned} \tag{31}$$

$$L_B(B) \equiv \left(L_H(B_H), L^*_T(B_{\mathbf{T}}), L^*_H(B_{\mathbf{U}})\right) \tag{32}$$

With L^*_T and L^*_H being element-wise sequence transformations, thus:

$$f^*\left((x_0, x_1, ...)\right) \equiv \left(f(x_0), f(x_1), ...\right) \quad \text{for any function } f \tag{33}$$

The component types are defined thus:

$$(34)\quad \begin{array}{lllllll} H_p \in \mathbb{B}_{32} & \wedge & H_u \in \mathbb{B}_{32} & \wedge & H_c \in \mathbb{B}_{20} & \wedge \\ H_r \in \mathbb{B}_{32} & \wedge & H_t \in \mathbb{B}_{32} & \wedge & H_e \in \mathbb{B}_{32} & \wedge \\ H_b \in \mathbb{B}_{64} & \wedge & H_d \in \mathbb{P} & \wedge & H_i \in \mathbb{P} & \wedge \\ H_l \in \mathbb{P} & \wedge & H_g \in \mathbb{P} & \wedge & H_s \in \mathbb{P} & \wedge \\ H_x \in \mathbb{B} & \wedge & H_n \in \mathbb{B}_{32} & & & \end{array}$$

where

$$(35)\quad \mathbb{B}_n = \{B : B \in \mathbb{B} \wedge \|B\| = n\}$$

We now have a rigorous specification for the construction of a formal block structure. The RLP function RLP (see Appendix B) provides the canonical method for transforming this structure into a sequence of bytes ready for transmission over the wire or storage locally.

4.3.4. *Block Header Validity.* We define $P(B_H)$ to be the parent block of B, formally:

$$(36)\quad P(H) \equiv B' : \texttt{KEC}(\texttt{RLP}(B'_H)) = H_p$$

The block number is the parent's block number incremented by one:

$$(37)\quad H_i \equiv P(H)_{H_i} + 1$$

The canonical difficulty of a block of header H is defined as $D(H)$:

$$(38)\quad D(H) \equiv \begin{cases} 2^{22} & \text{if} \quad H_i = 0 \\ P(H)_{H_d} + \lfloor \frac{P(H)_{H_d}}{1024} \rfloor & \text{if} \quad H_s < P(H)_{H_s} + 8 \\ \max\{1024, x\} & \text{otherwise} \end{cases}$$

where:

$$(39)\quad x = P(H)_{H_d} - \lfloor \frac{P(H)_{H_d}}{1024} \rfloor$$

The canonical gas limit of a block of header H is defined as $L(H)$:

$$(40)\quad L(H) \equiv \begin{cases} 10^6 & \text{if} \quad H_i = 0 \\ 125000 & \text{if} \quad L'(H) < 125000 \\ L'(H) & \text{otherwise} \end{cases}$$

$$(41)\quad L'(H) \equiv \lfloor \frac{1023 P(H)_{H_l} + \lfloor \frac{6}{5} P(H)_{H_g} \rfloor}{1024} \rfloor$$

H_s is the timestamp of block H and must fulfil the relation:

$$(42)\quad H_s > P(H)_{H_s}$$

This mechanism enforces a homeostasis in terms of the time between blocks; a smaller period between the last two blocks results in an increase in the difficulty level and thus additional computation required, lengthening the likely next period. Conversely, if the period is too large, the difficulty, and expected time to the next block, is reduced.

The nonce, H_n, must satisfy the relation:

$$(43)\quad \texttt{PoW}(H, H_n) \leqslant \frac{2^{256}}{H_d}$$

Where PoW is the proof-of-work function (see section 11.5): this evaluates to an pseudo-random number cryptographically dependent on the parameters H and H_n. Given an approximately uniform distribution in the range $[0, 2^{256})$, the expected time to find a solution is proportional to the difficulty, H_d.

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 $V(H)$:

$$(44)\quad V(H) \equiv \texttt{PoW}(H, H_n) \leqslant \frac{2^{256}}{H_d} \quad \wedge$$
$$(45)\quad H_d = D(H) \quad \wedge$$
$$(46)\quad H_l = L(H) \quad \wedge$$
$$(47)\quad H_s > P(H)_{H_s} \quad \wedge$$
$$(48)\quad H_i = P(H)_{H_i} + 1 \quad \wedge$$
$$(49)\quad \|H_x\| \leq 1024$$

Noting additionally that **extraData** must be at most 1024 bytes.

5. Gas and Payment

In order to avoid issues of network abuse and to sidestep the inevitable questions stemming from Turing completeness, all programmable computation in Ethereum is subject to fees. The fee schedule is specified in units of *gas* (see Appendix G for the fees associated with various computation). Thus any given fragment of programmable computation (this includes creating contracts, making message calls, utilising and accessing account storage and executing operations on the virtual machine) has a universally agreed cost in terms of gas.

Every transaction has a specific amount of gas associated with it: **gasLimit**. This is the amount of gas which is implicitly purchased from the sender's account balance. The purchase happens at the according **gasPrice**, also specified in the transaction. The transaction is considered invalid if the account balance cannot support such a purchase. It is named **gasLimit** since any unused gas at the end of the transaction is refunded (at the same rate of purchase) to the sender's account. Gas does not exist outside of the execution of a transaction. Thus for accounts with trusted code associated, a relatively high gas limit may be set and left alone.

In general, Ether used to purchase gas that is not refunded is delivered to the *coinbase* address, the address of an account typically under the control of the miner. Transactors are free to specify any **gasPrice** that they wish, however miners are free to ignore transactions as they choose. A higher gas price on a transaction will therefore cost the sender more in terms of Ether and deliver a greater value to the miner and thus will more likely be selected for inclusion by more miners. Miners, in general, will choose to advertise the minimum gas price for which they will execute transactions and transactors will be free to canvas these prices in determining what gas price to offer. Since there will be a (weighted) distribution of minimum acceptable gas prices, transactors will necessarily have a trade-off to make between lowering the gas price and maximising the chance that their transaction will be mined in a timely manner.

6. Transaction Execution

The execution of a transaction is the most complex part of the Ethereum protocol: it defines the state transition function Υ. It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include:

(1) The transaction is well-formed RLP, with no additional trailing bytes;
(2) the transaction signature is valid;
(3) the transaction nonce is valid (equivalent to the sender account's current nonce);
(4) the gas limit is no smaller than the intrinsic gas, g_0, used by the transaction;
(5) the sender account balance contains at least the cost, v_0, required in up-front payment.

Formally, we consider the function Υ, with T being a transaction and $\boldsymbol{\sigma}$ the state:

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{50}$$

Thus $\boldsymbol{\sigma}'$ is the post-transactional state. We also define Υ^g to evaluate to the amount of gas used in the execution of a transaction and $\Upsilon^{\mathbf{l}}$ to evaluate to the transaction's accrued log items, both to be formally defined later.

6.1. Substate.

Throughout transaction execution, we accrue certain information that is acted upon immediately following the transaction. We call this *transaction substate*, and represent it as A, which is a tuple:

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_r) \tag{51}$$

The tuple contents include $A_{\mathbf{s}}$, the suicide set: a set of accounts that will be discarded following the transaction's completion. $A_{\mathbf{l}}$ is the log series: this is a series of archived and indexable 'checkpoints' in VM code execution that allow for contract-calls to be easily tracked by onlookers external to the Ethereum world (such as decentralised application front-ends). Finally there is A_r, the refund balance, increased through using the SSTORE instruction in order to reset contract storage to zero from some non-zero value. Though not immediately refunded, it is allowed to partially offset the total execution costs.

For brevity, we define the empty substate A^0 to have no suicides, no logs and a zero refund balance:

$$A^0 \equiv (\varnothing, (), 0) \tag{52}$$

6.2. Execution.

We define intrinsic gas g_0, the amount of gas this transaction requires to be paid prior to execution, as follows:

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{txdatazero} & \text{if} \quad i = 0 \\ G_{txdatanonzero} & \text{otherwise} \end{cases} + G_{transaction} \tag{53}$$

where $T_{\mathbf{i}}, T_{\mathbf{d}}$ means the series of bytes of the transaction's associated data and initialisation EVM-code, depending on whether the transaction is for contract-creation or message-call. G is defined in Appendix G.

The up-front cost v_0 is calculated as:

$$v_0 \equiv T_g T_p + T_v \tag{54}$$

The validity is determined as:

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_n &= \boldsymbol{\sigma}[S(T)]_n \quad \wedge \\ g_0 &\leqslant T_g \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_b \quad \wedge \\ T_g &\leqslant B_{Hl} - \ell(B_{\mathbf{R}})_u \end{aligned} \tag{55}$$

Note the final condition; the sum of the transaction's gas limit, T_g, and the gas utilised in this block prior, given by $\ell(B_{\mathbf{R}})_u$, must be no greater than the block's **gasLimit**, B_{Hl}.

The execution of a valid transaction begins with an irrevocable change made to the state: the nonce of the account of the sender, $S(T)$, is incremented by one and the balance is reduced by the up-front cost, v_0. The gas available for the proceeding computation, g, is defined as $T_g - g_0$. The computation, whether contract creation or a message call, results in an eventual state (which may legally be equivalent to the current state), the change to which is deterministic and never invalid: there can be no invalid transactions from this point.

We define the checkpoint state $\boldsymbol{\sigma}_0$:

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{56}$$

$$\boldsymbol{\sigma}_0[S(T)]_b \equiv \boldsymbol{\sigma}[S(T)]_b - T_g T_p \tag{57}$$

$$\boldsymbol{\sigma}_0[S(T)]_n \equiv \boldsymbol{\sigma}[S(T)]_n + 1 \tag{58}$$

Evaluating $\boldsymbol{\sigma}_P$ from $\boldsymbol{\sigma}_0$ depends on the transaction type; either contract creation or message call; we define the tuple of post-execution provisional state $\boldsymbol{\sigma}_P$, remaining gas g' and substate A:

$$(\boldsymbol{\sigma}_P, g', A) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad g, T_p, T_v, T_{\mathbf{i}}, 0) & \text{if} \quad T_t = \varnothing \\ \Theta_3(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad T_t, T_t, g, T_p, T_v, T_{\mathbf{d}}, 0) & \text{otherwise} \end{cases} \tag{59}$$

where g is the amount of gas remaining after deducting the basic amount required to pay for the existence of the transaction:

$$g \equiv T_g - g_0 \tag{60}$$

and T_o is the original transactor, which can differ from the sender in the case of a message call or contract creation not directly triggered by a transaction but coming from the execution of EVM-code.

Note we use Θ_3 to denote the fact that only the first three components of the function's value are taken; the final represents the message-call's output value (a byte array) and is unused in the context of transaction evaluation.

After the message call or contract creation is processed, the state is finalised by determining the amount to be refunded, g^* from the remaining gas, g', plus some allowance from the refund counter, to the sender at the original rate.

$$g^* \equiv g' + \min\{\left\lfloor \frac{T_g - g'}{2} \right\rfloor, A_r\} \tag{61}$$

The total refundable amount is the legitimately remaining gas g', added to A_r, with the latter component being capped up to a maximum of half (rounded down) of the total amount used $T_g - g'$.

The Ether for the gas is given to the miner, whose address is specified as the coinbase of the present block B. So

we define the pre-final state $\boldsymbol{\sigma}^*$ in terms of the provisional state $\boldsymbol{\sigma}_P$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{62}$$
$$\boldsymbol{\sigma}^*[S(T)]_b \equiv \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \tag{63}$$
$$\boldsymbol{\sigma}^*[m]_b \equiv \boldsymbol{\sigma}_P[m]_b + (T_g - g^*)T_p \tag{64}$$
$$m \equiv B_{H\,c} \tag{65}$$

The final state, $\boldsymbol{\sigma}'$, is reached after deleting all accounts that appear in the suicide list:

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{66}$$
$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] \equiv \varnothing \tag{67}$$

And finally, we specify Υ^g, the total gas used in this transaction and $\Upsilon^{\mathbf{l}}$, the logs created by this transaction:

$$\Upsilon^g(\boldsymbol{\sigma}, T) \equiv T_g - g' \tag{68}$$
$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{69}$$

These are used to help define the transaction receipt, discussed later.

7. Contract Creation

There are number of intrinsic parameters used when creating an account: sender (s), original transactor (o), available gas (g), gas price (p), endowment (v) together with an arbitrary length byte array, $\mathbf{i}$, the initialisation EVM code and finally the present depth of the message-call/contract-creation stack (e).

We define the creation function formally as the function Λ, which evaluates from these values, together with the state $\boldsymbol{\sigma}$ to the tuple containing the new state, remaining gas and accrued transaction substate $(\boldsymbol{\sigma}', g', A)$, as in section 6:

$$(\boldsymbol{\sigma}', g', A) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e) \tag{70}$$

The address of the new account is defined as being the rightmost 160 bits of the Keccak hash of RLP encoding of the structure containing only the sender and the nonce. Thus we define the resultant address for the new account a:

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_n - 1)\ \big)\Big)\Big) \tag{71}$$

where KEC is the Keccak 256-bit hash function, RLP is the RLP encoding function, $\mathcal{B}_{a..b}(X)$ evaluates to binary value containing the bits of indices in the range $[a, b]$ of the binary data X and $\boldsymbol{\sigma}[x]$ is the address state of x or $\varnothing$ if none exists. Note we use one fewer than the sender's nonce value; we assert that we have incremented the sender account's nonce prior to this call, and so the value used is the sender's nonce at the beginning of the responsible transaction or VM operation.

The account's nonce is initially defined as zero, the balance as the value passed, the storage as empty and the code hash as the Keccak 256-bit hash of the empty string; the sender's balance is also reduced by the value passed. Thus the mutated state becomes $\boldsymbol{\sigma}^*$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{72}$$

$$\boldsymbol{\sigma}^*[a] \equiv \big(0, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\big) \tag{73}$$
$$\boldsymbol{\sigma}^*[s]_b \equiv \boldsymbol{\sigma}^*[s]_b - v \tag{74}$$

where v' is the account's pre-existing value, in the event it was previously in existence:

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_b & \text{otherwise} \end{cases} \tag{75}$$

Finally, the account is initialised through the execution of the initialising EVM code $\mathbf{i}$ according to the execution model (see section 9). Code execution can effect several events that are not internal to the execution state: the account's storage can be altered, further accounts can be created and further message calls can be made. As such, the code execution function Ξ evaluates to a tuple of the resultant state $\boldsymbol{\sigma}^{**}$, available gas remaining g^{**}, the accrued substate A and the body code of the account $\mathbf{b}$.

Code execution depletes gas; thus it may exit before the code has come to a natural halting state. In this (and several other) exceptional cases we say an Out-of-Gas exception has occurred: The evaluated state is defined as being the empty set $\varnothing$ and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation. The gas remaining should be zero in any such exceptional condition. If the creation was conducted as the reception of a transaction, then this doesn't affect payment of the intrinsic cost: it is paid regardless.

If such an exception does not occur, then the remaining gas is refunded to the originator and the now-altered state is allowed to persevere. Thus formally, we may specify the resultant state, gas and substate as $(\boldsymbol{\sigma}', g', A)$ where:

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I) \tag{76}$$

$$g' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ g^{**} & \text{if} \quad g^{**} < c \\ g^{**} - c & \text{otherwise} \end{cases} \tag{77}$$

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{if} \quad g^{**} < c \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_c = \texttt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases} \tag{78}$$

Where I contains the parameters of the execution environment as defined in section 9.

$$I_a \equiv a \tag{79}$$
$$I_o \equiv o \tag{80}$$
$$I_p \equiv p \tag{81}$$
$$I_{\mathbf{d}} \equiv () \tag{82}$$
$$I_s \equiv s \tag{83}$$
$$I_v \equiv v \tag{84}$$
$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{85}$$
$$I_e \equiv e \tag{86}$$

where c is the code-deposit cost:

$$c \equiv G_{createdata} \times |\mathbf{o}| \tag{87}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_H has no special treatment and is determined from the blockchain. The exception in the determination of $\boldsymbol{\sigma}'$ dictates that the resultant byte sequence from the execution of the initialisation code specifies the final body code for the newly-created account, with $\boldsymbol{\sigma}'[a]_c$

being the Keccak 256-bit hash of the newly created account's body code and $\mathbf{o}$ the output byte sequence of the code execution.

No code is deposited in the state if the gas does not cover the additional per-byte contract deposit fee, however, the value is still transferred and the execution side-effects take place.

7.1. **Subtleties.** Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code. Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SUICIDE instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

8. Message Call

In the case of executing a message call, several parameters are required: sender (s), transaction originator (o), recipient (r), the account whose code is to be executed (c, usually the same as recipient), available gas (g), value (v) and gas price (p) together with an arbitrary length byte array, $\mathbf{d}$, the input data of the call and finally the present depth of the message-call/contract-creation stack (e).

Aside from evaluating to a new state and transaction substate, message calls also have an extra component—the output data denoted by the byte array $\mathbf{o}$. This is ignored when executing transactions, however message calls can be initiated due to VM-code execution and in this case this information is used.

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \mathbf{d}, e) \tag{88}$$

We define $\boldsymbol{\sigma}_1$, the first transitional state as the original state but with the value transferred from sender to recipient:

$$\boldsymbol{\sigma}_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{89}$$

Throughout the present work, it is assumed that if $\boldsymbol{\sigma}_1[r]$ was originally undefined, it will be created as an account with no code or state and zero balance and nonce. Thus the previous equation should be taken to mean:

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{90}$$

$$\boldsymbol{\sigma}'_1[s]_b \equiv \boldsymbol{\sigma}_1[s]_b - v \tag{91}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{92}$$

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (v, 0, \texttt{KEC}(()), \texttt{TRIE}(\varnothing)) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \\ \boldsymbol{\sigma}'_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v & \text{otherwise} \end{cases} \tag{93}$$

The account's associated code (identified as the fragment whose Keccak hash is $\boldsymbol{\sigma}[c]_c$) is executed according to the execution model (see section 9). Just as with contract creation, if the execution halts in an exceptional fashion (i.e. due to an exhausted gas supply, stack underflow, invalid jump destination or invalid instruction), then no gas is refunded to the caller and the state is reverted to the point immediately prior to balance transfer (i.e. $\boldsymbol{\sigma}$).

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases} \tag{94}$$

$$(\boldsymbol{\sigma}^{**}, g', \mathbf{s}, \mathbf{o}) \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad a = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad a = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad a = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad a = 4 \\ \Xi(\boldsymbol{\sigma}_1, g, I) & \text{otherwise} \end{cases} \tag{95}$$

$$I_a \equiv a \tag{96}$$

$$I_o \equiv o \tag{97}$$

$$I_p \equiv p \tag{98}$$

$$I_d \equiv d \tag{99}$$

$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{100}$$

$$I_s \equiv s \tag{101}$$

$$I_v \equiv v \tag{102}$$

$$I_e \equiv e \tag{103}$$

$$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_c \tag{104}$$

It is assumed that the client will have stored the pair $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$ at some point prior in order to make the determination of $I_{\mathbf{b}}$ feasible.

As can be seen, there are four exceptions to the usage of the general execution framework Ξ for evaluation of the message call: these are four so-called 'precompiled' contracts, meant as a preliminary piece of architecture that may later become *native extensions*. The four contracts in addresses 1, 2, 3 and 4 execute the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD 160-bit hash scheme and the identity function respectively.

Their full formal definition is in Appendix E.

9. Execution Model

The execution model specifies how the system state is altered given a series of bytecode instructions and a small tuple of environmental data. This is specified through a formal model of a virtual state machine, known as the Ethereum Virtual Machine (EVM). It is a *quasi*-Turing-complete machine; the *quasi* qualification comes from the fact that the computation is intrinsically bounded through a parameter, *gas*, which limits the total amount of computation done.

9.1. **Basics.** The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack 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 an unlimited size. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction.

The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas (OOG) 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) and which will deal with it separately.

9.2. **Fees Overview.** Fees (denominated in gas) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation. Most operations require a single gas fee to be paid for their execution; exceptions include SSTORE, SLOAD, CALL, CREATE, BALANCE, EXP, LOG and SHA3. Secondly, gas may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for CREATE, CALL and CALLCODE. Finally, gas may be paid due to an increase in the usage of the memory.

Over an account's execution, the total fee for memory-usage payable is proportional to smallest multiple of 32 bytes that are required such that all memory indices (whether for read or write) are included in the range. This is paid for on a just-in-time basis; as such, referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional memory usage fee. Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds. That said, implementations must be able to manage this eventuality.

Storage fees have a slightly nuanced behaviour—to incentivise minimisation of the use of storage (which corresponds directly to a larger state database on all nodes), the execution fee for an operation that clears an entry in the storage is not only waived, a qualified refund is given; in fact, this refund is effectively paid up-front since the initial usage of a storage location costs substantially more than normal usage.

See Appendix H for a rigorous definition of the EVM gas cost.

9.3. **Execution Environment.** 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:

- I_a, the address of the account which owns the code that is executing.
- I_o, the sender address of the transaction that originated this execution.
- I_p, the price of gas in the transaction that originated this execution.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- I_s, 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.
- I_v, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- $I_{\mathbf{b}}$, the byte array that is the machine code to be executed.
- I_H, the block header of the present block.
- I_e, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATEs being executed at present).

The execution model defines the function Ξ, which can compute the resultant state $\boldsymbol{\sigma}'$, the remaining gas g', the suicide list $\mathbf{s}$, the log series $\mathbf{l}$, the refunds r and the resultant output, $\mathbf{o}$, given these definitions:

$$(\boldsymbol{\sigma}', g', \mathbf{s}, \mathbf{l}, r, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{105}$$

9.4. **Execution Overview.** We must now define the Ξ function. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, $\boldsymbol{\sigma}$ and the machine state, $\boldsymbol{\mu}$. Formally, we define it recursively with a function X. This uses an iterator function O (which defines the result of a single cycle of the state machine) together with functions Z which determines if the present state is an exceptional halting state of the machine and H, specifying the output data of the instruction if and only if the present state is a normal halting state of the machine.

The empty sequence, denoted (), is not equal to the empty set, denoted $\varnothing$; this is important when interpreting the output of H, which evaluates to $\varnothing$ when execution is to continue but a series (potentially empty) when execution should halt.

$$\Xi(\boldsymbol{\sigma}, g, I) \equiv X_{0,1,2,4}\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{106}$$
$$\boldsymbol{\mu}_g \equiv g \tag{107}$$
$$\boldsymbol{\mu}_{pc} \equiv 0 \tag{108}$$
$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{109}$$
$$\boldsymbol{\mu}_i \equiv 0 \tag{110}$$
$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{111}$$
(112)

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} \big(\varnothing, \boldsymbol{\mu}, A^0, I, ()\big) & \text{if } Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if } \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{otherwise} \end{cases}$$

where

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{113}$$
$$(a, b, c) \cdot d \equiv (a, b, c, d) \tag{114}$$

Note that we must drop the fourth value in the tuple returned by X to correctly evaluate Ξ, hence the subscript $X_{0,1,2,4}$.

X is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either Z 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.

9.4.1. *Machine State.* The machine state $\boldsymbol{\mu}$ is defined as the tuple $(g, pc, \mathbf{m}, i, \mathbf{s})$ which are the gas available, the program counter $pc \in \mathbb{P}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents $\boldsymbol{\mu}_{\mathbf{m}}$ 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 in Appendix H.

For the purposes of defining Z, H and O, we define w as the current operation to be executed:

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{115}$$

We also assume the fixed amounts of δ and α, specifying the stack items removed and added, both subscriptable on the instruction and an instruction cost function C evaluating to the full cost, in gas, of executing the given instruction.

9.4.2. *Exceptional Halting.* The exceptional halting function Z is defined as:

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w \in \{\text{JUMP}, \text{JUMPI}\} \quad \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \end{aligned} \tag{116}$$

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid or if a CALL, CALLCODE or CREATE instruction is executed when the call stack limit of 1024 is reached. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt.

9.4.3. *Jump Destination Validity.* We previously used D as the function to determine the set of valid jump destinations given the code that is being run. We define this as any position in the code occupied by a JUMPDEST instruction.

All such positions must be on valid instruction boundaries, rather than sitting in the data portion of PUSH operations and must appear within the explicitly defined portion of the code (rather than in the implicitly defined STOP operations that trail it).

Formally:

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, \mathbf{0}) \cap Y(\mathbf{c}, 0) \tag{117}$$

where:

$$Y(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup Y(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{118}$$

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{119}$$

where N is the next valid instruction position in the code, skipping the data of a PUSH instruction, if any:

$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \text{if} \quad w \in [\text{PUSH1}, \text{PUSH32}] \\ i + 1 & \text{otherwise} \end{cases} \tag{120}$$

9.4.4. *Normal Halting.* The normal halting function H is defined:

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{RETURN} \\ () & \text{if} \quad w \in \{\text{STOP}, \text{SUICIDE}\} \\ \varnothing & \text{otherwise} \end{cases} \tag{121}$$

The data-returning halt operation, RETURN, has a special function H_{RETURN}, defined in Appendix H.

9.5. **The Execution Cycle.** Stack items are added or removed from the left-most, lower-indexed portion of the series; all other items remain unchanged:

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{122}$$

$$\Delta \equiv \alpha_w - \delta_w \tag{123}$$

$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{124}$$

$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x + \Delta] \tag{125}$$

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:

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{126}$$

$$\boldsymbol{\mu}'_{pc} \equiv \begin{cases} J_{\text{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_{pc}, w) & \text{otherwise} \end{cases} \tag{127}$$

In general, we assume the memory, suicide list and system state don't change:

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{128}$$

$$\boldsymbol{\mu}'_i \equiv \boldsymbol{\mu}_i \tag{129}$$

$$A' \equiv A \tag{130}$$

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{131}$$

However, instructions do typically alter one or several components of these values. Altered components listed by instruction are noted in Appendix H, alongside values for α and δ and a formal description of the gas requirements.

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 akin to existing schemes, such as that employed in Bitcoin-derived protocols.

This scheme notably ignores so-called *stale* blocks: valid, mined blocks, which were propagated too late into the network and thus were beaten to network consensus by a sibling block (one with the same parent). Such blocks become more common as the network propagation time approaches the ideal inter-block time. However, by counting the computation work of stale block headers, we are able to do better: we can utilise this otherwise wasted computation and put it to use in helping to buttress the

more popular blockchain making it a stronger choice over less popular (though potentially longer) competitors.

This increases overall network security by making it much harder for an adversary to silently mine a canonical blockchain (which, it is assumed, would contain different transactions to the current consensus) and dump it on the network with the effect of overriding existing blocks and reversing the transactions within.

In order to validate the extra computation, a given block B may include the block headers from any known uncle blocks (i.e. blocks whose parent is equivalent to the Nth generation grandparent of B, for a limited set of N). Since a block header includes the nonce, a proof-of-work, then the header alone is enough to validate the computation done. Any such blocks contribute toward the total computation or *total difficulty* of a chain that includes them. To incentivise computation and inclusion, a reward is given both to the miner of the stale block and the miner of the block that references it.

Thus we define the total difficulty of block B recursively as:

$$B_t \equiv B'_t + B_d + \sum_{U \in B_{\mathbf{U}}} U_d \tag{132}$$

$$B' \equiv P(B_H) \tag{133}$$

As such given a block B, B_t is its total difficulty, B' is its parent block, B_d is its difficulty and $B_{\mathbf{U}}$ is its set of uncle blocks.

11. Block Finalisation

The process of finalising a block involves four stages:

(1) Validate (or, if mining, determine) uncles;
(2) validate (or, if mining, determine) transactions;
(3) apply rewards;
(4) verify (or, if mining, compute a valid) state and nonce.

11.1. **Uncle Validation.** The validation of uncle headers means nothing more than verifying that each uncle header is both a valid header and satisfies the relation of Nth-generation uncle to the present block where $N \leq 6$. Formally:

$$\bigwedge_{U \in B_{\mathbf{U}}} V(U) \ \wedge \ k(U, P(B_H), 6) \tag{134}$$

where k denotes the "is-kin" property:

$$k(U, H, n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U, H) & \\ \quad \vee\, k(U, P(H), n-1) & \text{otherwise} \end{cases} \tag{135}$$

and s denotes the "is-sibling" property:

$$s(U, H) \equiv (P(H) = P(U) \ \wedge \ H \neq U) \tag{136}$$

11.2. **Transaction Validation.** The given **gasUsed** must correspond faithfully to the transactions listed: $B_{H\,u}$, the total gas used in the block, must be equal to the accumulated gas used according to the final transaction:

$$B_{H\,g} = \ell(\mathbf{R})_u \tag{137}$$

11.3. **Reward Application.** The application of rewards to a block involves raising the balance of the accounts of the coinbase address of the block and each uncle by a certain amount. We raise the block's coinbase account by R_b; for each uncle, we raise the block's coinbase by an additional $\frac{1}{32}$ of the block reward and the coinbase of the uncle by $\frac{15}{16}$ of the reward. Formally we define the function Ω:

$$\Omega(B, \boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:} \tag{138}$$

$$\boldsymbol{\sigma}'[B_{H\,c}]_b = \boldsymbol{\sigma}[B_{H\,c}]_b + (1 + \frac{|B_{\mathbf{U}}|}{32})R_b \tag{139}$$

$$\forall_{U \in B_{\mathbf{U}}}\ \boldsymbol{\sigma}'[U_c]_b = \boldsymbol{\sigma}[U_c]_b + \frac{15}{16}R_b \tag{140}$$

If there are collisions of the coinbase addresses between uncles and the block (i.e. two uncles with the same coinbase address or an uncle with the same coinbase address as the present block), additions are applied cumulatively.

We define the block reward as 1500 Finney:

$$\text{Let} \quad R_b = 1.5 \times 10^{18} \tag{141}$$

11.4. **State & Nonce Validation.** We may now define the function, Γ, that maps a block B to its initiation state:

$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_i : \texttt{TRIE}(L_S(\boldsymbol{\sigma}_i)) = P(B_H)_{H_r} & \text{otherwise} \end{cases} \tag{142}$$

Here, $\texttt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ means the hash of the root node of a trie of state $\boldsymbol{\sigma}_i$; it is assumed that implementations will store this in the state database, trivial and efficient since the trie is by nature an immutable data structure.

And finally define Φ, the block transition function, which maps an incomplete block B to a complete block B':

$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{except:} \tag{143}$$

$$B'_n = n : \quad \texttt{PoW}(B^*, n) < \frac{2^{256}}{H_d} \tag{144}$$

$$B^* \equiv B \quad \text{except:} \quad B'_r = r(\Pi(\Gamma(B), B)) \tag{145}$$

As specified at the beginning of the present work, Π is the state-transition function, which is defined in terms of Ω, the block finalisation function and Υ, the transaction-evaluation function, both now well-defined.

As previously detailed, $\mathbf{R}[n]_{\boldsymbol{\sigma}}$, $\mathbf{R}[n]_{\mathbf{l}}$ and $\mathbf{R}[n]_u$ are the nth corresponding states, logs and cumulative gas used after each transaction ($\mathbf{R}[n]_b$, the fourth component in the tuple, has already been defined in terms of the logs). The former is defined simply as the state resulting 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 such transaction):

$$\mathbf{R}[n]_{\boldsymbol{\sigma}} = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases} \tag{146}$$

In the case of $B_{\mathbf{R}}[n]_u$, 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:

$$\mathbf{R}[n]_u = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_u & \text{otherwise} \end{cases} \tag{147}$$

For $\mathbf{R}[n]_\mathbf{l}$, we utilise the $\Upsilon^\mathbf{l}$ function that we conveniently defined in the transaction execution function.

$$\mathbf{R}[n]_\mathbf{l} = \Upsilon^\mathbf{l}(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_\mathbf{T}[n]) \tag{148}$$

Finally, we define Π as the new state given the block reward function Ω applied to the final transaction's resultant state, $\ell(B_\mathbf{R})_{\boldsymbol{\sigma}}$:

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\mathbf{R})_{\boldsymbol{\sigma}}) \tag{149}$$

Thus the complete block-transition mechanism, less PoW, the proof-of-work function is defined.

11.5. **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 determination 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 skewing 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 function, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

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 memory and 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 hardware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to "specialised hardware" for the task.

More formally, the proof-of-work function takes the form of PoW:

$$\mathtt{PoW}(H_{\not n}, n) < \frac{2^{256}}{H_d} \tag{150}$$

Where $H_{\not n}$ is the new block's header H, but *without* the nonce component; H_d is the new block's difficulty value (i.e. the block difficulty from section 10).

As of the proof-of-concept (PoC) series of the Ethereum software, the proof-of-work function is simplistic and does not attempt to secure these goals. It will be described here for completeness.

11.5.1. *PoC Series.* For the PoC series, we use a simplified proof-of-work. This is not ASIC resistant and is meant merely as a placeholder. It utilises the bare Keccak hash function to secure the block chain by requiring the Keccak hash of the concatenation of the nonce and the header's Keccak hash to be sufficiently low.

It is formally defined as PoW:

$$\mathtt{PoW}(H, n) \equiv \mathtt{BE}(\mathtt{KEC}(\mathtt{KEC}(\mathtt{RLP}(H_{\not n})) \circ n)) \tag{151}$$

where: $\mathtt{RLP}(H_{\not n})$ is the RLP encoding of the block header H, not including the final nonce component; KEC is the Keccak hash function accepting an arbitrary length series of bytes and evaluating to a series of 32 bytes (i.e. 256-bit); n is the nonce, a series of 32 bytes; $\circ$ is the series concatenation operator; $\mathtt{BE}(X)$ evaluates to the value equal to X when interpreted as a big-endian-encoded integer.

11.5.2. *Release Series.* For the release series, we use a more complex proof-of-work. This has yet to be formally defined, but involves two components; firstly that it concerns the evaluation of programs on the EVM. Secondly that it concerns the utilisation of either the blockchain or the full state trie.

As an overview, the output of the function is based upon the system state, defined as the hash of the root node of the state trie. A set of transactions, pseudo-randomly determined from the nonce value and selected from the last N blocks is taken. N is large enough and the selection criteria are such that execution of the transactions requires some non-negligible amount of processing by the EVM. Whenever code is executed on the EVM, it is pseudo-randomly (seeded again by the nonce) corrupted before alteration. Corruption could involve switching addresses with other transactions or rotating them through in the state trie (perhaps to the next address with the same order of magnitude of funds), rotating through instructions that have equivalent stack behaviour (e.g. swapping ADD for SUB or GT for EQ), or more destructive techniques such as randomly changing opcodes. This results in a problem that both require generalised computation hardware and is sequentially memory (and perhaps even disk) hard.

Any specialised hardware to perform this task could also be leveraged to speed up (and thus drive down costs) of general Ethereum transaction processing.

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 provides 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 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 phenomenon. 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 coinbase address. For a series of such numbers, a trivial solution would be to work from the previous pseudo-random number, adding some constant amount and hashing the result.

This strategy does have a downside: a miner with sufficient power could alter any of the above values in order to deliver a seed in order to alter the outcome of the pseudorandom-based executions. For a more secure pseudo-random offering, all involved parties could agree on a number of data feed contracts; these could be combined along with the block timestamp and hashed to produce the first number in the series. By spreading the inputs and thus the trust between numerous parties the likelihood of a malicious miner altering the outcome becomes increasingly less likely.

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 maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain.

Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 10000th block) could be maintained by the peer network, effectively recasting the genesis block. This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks.

Finally, blockchain compression could perhaps be conducted: nodes in state trie that haven't sent/received a transaction in some constant amount of blocks could be thrown out, reducing both Ether-leakage and the growth of the state database.

13.1. **Scalability.** Scalability remains an eternal concern. With a generalised state transition function, it becomes 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 provide a considerably more scalable protocol.

Some form of hierarchical structure, achieved by either consolidating smaller lighter-weight chains into the main block or building the main block through the incremental combination and adhesion (through proof-of-work) of smaller transaction sets may allow parallelisation of transaction combination and block-building. Parallelism 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 available 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

Useful corrections and suggestions were provided by a number of others from the Ethereum community including Aeron Buchanan, Christoph Jentzsch, Paweł Bylica, Jutta Steiner, Gustav Simonsson, Nick Savers, Viktor Trón, Marko Simovic and, of course, Vitalik Buterin.

References

Jacob Aron. BitCoin software finds new life. *New Scientist*, 213(2847):20, 2012.

Adam Back. Hashcash - Amortizable Publicly Auditable Cost-Functions. 2002. URL `{http://www.hashcash.org/papers/amortizable.pdf}`.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers, Innovators and Imitators. In *Growth Through Innovation*, pages 85–96. Springer, 2014.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. 2013. URL `{http://ethereum.org/ethereum.html}`.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1:2012, 2008.

Meni Rosenfeld. Overview of Colored Coins. 2012. URL `{https://bitcoil.co.il/BitcoinX.pdf}`.

Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains, 2013. URL {CryptologyePrintArchive,Report2013/881}. http://eprint.iacr.org/.

Simon Sprankel. Technical Basis of Digital Currencies, 2013.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

J. R. Willett. MasterCoin Complete Specification. 2013. URL {https://github.com/mastercoin-MSC/spec}.

Appendix A. Terminology

External Actor: A person or other entity able to interface to an Ethereum node, but external to the world of Ethereum. It can interact with Ethereum through depositing signed Transactions and inspecting the blockchain and associated state. Has one (or more) intrinsic Accounts.

Address: A 160-bit code used for identifying Accounts.

Account: Accounts have an intrinsic balance and transaction count maintained as part of the Ethereum state. They also have some (possibly empty) EVM Code and a (possibly empty) Storage State associated with them. Though homogenous, it makes sense to distinguish between two practical types of account: those with empty associated EVM Code (thus the account balance is controlled, if at all, by some external entity) and those with non-empty associated EVM Code (thus the account represents an Autonomous Object). Each Account has a single Address that identifies it.

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.

Autonomous Object: A notional object existent only within the hypothetical state of Ethereum. Has an intrinsic address and thus an associated account; the account will have non-empty associated EVM Code. Incorporated only as the Storage State of that account.

Storage State: The information particular to a given Account that is maintained between the times that the Account's associated EVM Code runs.

Message: Data (as a set of bytes) and Value (specified as Ether) that is passed between two Accounts, either through the deterministic operation of an Autonomous Object or the cryptographically secure signature of the Transaction.

Message Call: The act of passing a message from one Account to another. If the destination account is associated with non-empty EVM Code, then the VM will be started with the state of said Object and the Message acted upon. If the message sender is an Autonomous Object, then the Call passes any data returned from the VM operation.

Gas: The fundamental network cost unit. Paid for exclusively by Ether (as of PoC-4), which is converted freely to and from Gas as required. Gas does not exist outside of the internal Ethereum computation engine; its price is set by the Transaction and miners are free to ignore Transactions whose Gas price is too low.

Contract: Informal term used to mean both a piece of EVM Code that may be associated with an Account or an Autonomous Object.

Object: Synonym for Autonomous Object.

App: An end-user-visible application hosted in the Ethereum Browser.

Ethereum Browser: (aka Ethereum Reference Client) A cross-platform GUI of an interface similar to a simplified browser (a la Chrome) that is able to host sandboxed applications whose backend is purely on the Ethereum protocol.

Ethereum Virtual Machine: (aka EVM) The virtual machine that forms the key part of the execution model for an Account's associated EVM Code.

Ethereum Runtime Environment: (aka ERE) 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 EVM relies for certain I/O instructions including CALL & CREATE.

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

EVM Assembly: The human-readable form of EVM-code.

LLL: The Lisp-like Low-level Language, a human-writable language used for authoring simple contracts and general low-level language toolkit for trans-compiling to.

Appendix B. Recursive Length Prefix

This is a serialisation method for encoding arbitrarily structured binary data (byte arrays).

We define the set of possible structures $\mathbb{T}$:

$$\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B} \tag{152}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \wedge \forall_{n<\|\mathbf{t}\|}\ \mathbf{t}[n] \in \mathbb{T}\} \tag{153}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \wedge \forall_{n<\|\mathbf{b}\|}\ \mathbf{b}[n] \in \mathbb{Y}\} \tag{154}$$

Where $\mathbb{Y}$ is the set of bytes. Thus $\mathbb{B}$ is the set of all sequences of bytes (otherwise known as byte-arrays, and a leaf if imagined as a tree), $\mathbb{L}$ is the set of all tree-like (sub-)structures that are not a single leaf (a branch node if imagined as a tree) and $\mathbb{T}$ is the set of all byte-arrays and such structural sequences.

We define the RLP function as $\texttt{RLP}$ through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

$$\texttt{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases} \tag{155}$$

If the value to be serialised is a byte-array, the RLP serialisation takes one of three forms:

- If the byte-array contains solely a single byte and that single byte is less than 128, then the input is exactly equal to the output.
- If the byte-array contains fewer than 56 bytes, then the output is equal to the input prefixed by the byte equal to the length of the byte array plus 128.
- Otherwise, the output is equal to the input prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the input byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 183.

Formally, we define R_b:

$$R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\texttt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \texttt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{otherwise} \end{cases} \tag{156}$$

$$\texttt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge \sum_{n=0}^{n<\|\mathbf{b}\|-1} b_n 256^{\|\mathbf{b}\|-1-n} \tag{157}$$

$$(a) \cdot (b, c) \cdot (d, e) = (a, b, c, d, e) \tag{158}$$

Thus $\texttt{BE}$ is the function that expands a positive integer value to a big-endian byte array of minimal length and the dot operator performs sequence concatenation.

If instead, the value to be serialised is a sequence of other items then the RLP serialisation takes one of two forms:

- If the concatenated serialisations of each contained item is less than 56 bytes in length, then the output is equal to that concatenation prefixed by the byte equal to the length of this byte array plus 192.
- Otherwise, the output is equal to the concatenated serialisations prefixed by the minimal-length byte-array which when interpreted as a big-endian integer is equal to the length of the concatenated serialisations byte array, which is itself prefixed by the number of bytes required to faithfully encode this length value plus 247.

Thus we finish by formally defining R_l:

$$R_l(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\texttt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \texttt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \tag{159}$$

$$s(\mathbf{x}) \equiv \texttt{RLP}(\mathbf{x}_0) \cdot \texttt{RLP}(\mathbf{x}_1)... \tag{160}$$

If RLP is used to encode a scalar, defined only as a positive integer, it must be specified as the shortest byte array such that the big-endian interpretation of it is equal. Thus the RLP of some positive integer i is defined as:

$$\texttt{RLP}(i : i \in \mathbb{P}) \equiv \texttt{RLP}(\texttt{BE}(i)) \tag{161}$$

When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

There is no specific canonical encoding format for signed or floating-point values.

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

It is defined as the function $\texttt{HP}$ which maps from a sequence of nibbles (represented by the set $\mathbb{Y}$) together with a boolean value to a sequence of bytes (represented by the set $\mathbb{B}$):

$$\texttt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if} \quad \|\mathbf{x}\| \text{ is even} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{162}$$

$$f(t) \equiv \begin{cases} 2 & \text{if} \quad t \\ 0 & \text{otherwise} \end{cases} \tag{163}$$

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag t. The low nibble of the first byte is zero in the case of an even number of nibbles and the

first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

Appendix D. Modified Merkle Patricia Tree

The modified Merkle Patricia tree (trie) provides a persistent data structure to map between arbitrary-length binary data (byte arrays). It is defined in terms of a mutable data structure to map between 256-bit binary fragments and arbitrary-length binary data, typically implemented as a database. The core of the trie, and its sole requirement in terms of the protocol specification is to provide a single value that identifies a given set of key-value pairs, which may either a 32 byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner the allows effective and efficient realisation of the protocol.

Formally, we assume the input value $\mathfrak{I}$, a set containing pairs of byte sequences:

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{164}$$

When considering such a sequence, we use the common numeric subscript notation to refer to a tuple's key or value, thus:

$$\forall_{I \in \mathfrak{I}} I \equiv (I_0, I_1) \tag{165}$$

Any series of bytes may also trivially be viewed as a series of nibbles, given an endian-specific notation; here we assume big-endian. Thus:

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{166}$$

$$\forall_n \quad \forall_{i:i<2\|\mathbf{k}_n\|} \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases} \tag{167}$$

We define the function TRIE, which evaluates to the root of the trie that represents this set when encoded in this structure:

$$\texttt{TRIE}(\mathfrak{I}) \equiv \texttt{KEC}(c(\mathfrak{I}, 0)) \tag{168}$$

We also assume a function n, the trie's node cap function. When composing a node, we use RLP to encode the structure. As a means of reducing storage complexity, for nodes whose composed RLP is fewer than 32 bytes, we store the RLP directly; for those larger we assert prescience of the byte array whose Keccak hash evaluates to our reference. Thus we define in terms of c, the node composition function:

$$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{if } \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if } \|c(\mathfrak{I}, i)\| < 32 \\ \texttt{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases} \tag{169}$$

In a manner similar to a radix tree, when the trie is traversed from root to leaf, one may build a single key-value pair. The key is accumulated through the traversal, acquiring a single nibble from each branch node (just as with a radix tree). Unlike a radix tree, in the case of multiple keys shared the same prefix or in the case of a single key having a unique suffix, two optimising nodes are provided. Thus while traversing, one may potentially acquire multiple nibbles from each of the other two node types, extension and leaf. There are three kinds of nodes in the trie:

Leaf: A two-item structure whose first item corresponds to the nibbles in the key not already accounted for by the accumulation of keys and branches traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be $true$.

Extension: A two-item structure whose first item corresponds to a series of nibbles of size greater than one that are shared by at least two distinct keys past the accumulation of nibbles keys and branches as traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be $false$.

Branch: A 17-item structure whose first sixteen items correspond to each of the sixteen possible nibble values for the keys at this point in their traversal. The 17th item is used in the case of this being a terminator node and thus a key being ended at this point in its traversal.

A branch is then only used when necessary; no branch nodes may exist that contain only a single non-zero entry. We may formally define this structure with the structural composition function c:

$$c(\mathfrak{I}, i) \equiv \begin{cases} \texttt{RLP}\Big(\big(\texttt{HP}(I_0[i..(\|I_0\|-1)], true), I_1\big)\Big) & \text{if } \|\mathfrak{I}\| = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \texttt{RLP}\Big(\big(\texttt{HP}(I_0[i..(j-1)], false), n(\mathfrak{I}, j)\big)\Big) & \text{if } i \neq j \quad \text{where } j = \arg\max_x : \exists \mathbf{l} : \|\mathbf{l}\| = x : \forall_{I \in \mathfrak{I}} : I_0[0..(x-1)] = \mathbf{l} \\ \texttt{RLP}\Big((u(0), u(1), ..., u(15), v)\Big) & \text{otherwise} \quad \text{where } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad\qquad v = \begin{cases} I_1 & \text{if } \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () & \text{otherwise} \end{cases} \end{cases} \tag{170}$$

D.1. **Trie Database.** Thus no explicit assumptions are made concerning what data is stored and what is not, since that is an implementation-specific consideration; we simply define the identity function mapping the key-value set $\mathfrak{I}$ to a 32-byte hash and assert that only a single such hash exists for any $\mathfrak{I}$, which though not strictly true is accurate within acceptable precision given the Keccak hash's collision resistance. In reality, a sensible implementation will not fully recompute the trie root hash for each set.

A reasonable implementation will maintain a database of nodes determined from the computation of various tries or, more formally, it will memoise the function c. This strategy uses the nature of the trie to both easily recall the contents of any previous key-value set and to store multiple such sets in a very efficient manner. Due to the dependency relationship, Merkle-proofs may be constructed with an $O(\log N)$ space requirement that can demonstrate a particular leaf must exist within a trie of a given root hash.

Appendix E. Precompiled Contracts

For each precompiled contract, we make use of a template function, Ξ_{PRE}, which implements the out-of-gas checking.

$$\Xi_{\text{PRE}}(\boldsymbol{\sigma}, g, I) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{if} \quad g < g_r \\ (\boldsymbol{\sigma}, g - g_r, A^0, \mathbf{o}) & \text{otherwise} \end{cases} \tag{171}$$

The precompiled contracts each use these definitions and provide specifications for the $\mathbf{o}$ (the output data) and g_r, the gas requirements.

For the elliptic curve DSA recover VM execution function, we also define $\mathbf{d}$ to be the input data, well-defined for an infinite length by appending zeroes as required. Importantly in the case of an invalid signature ($\texttt{ECDSARECOVER}(h, v, r, s) = \varnothing$), then we have no output.

$$\Xi_{\text{ECREC}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{172}$$

$$g_r = 500 \tag{173}$$

$$|\mathbf{o}| = \begin{cases} 0 & \text{if} \quad \texttt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases} \tag{174}$$

$$\text{if} \quad |\mathbf{o}| = 32: \tag{175}$$

$$\mathbf{o}[0..11] = 0 \tag{176}$$

$$\mathbf{o}[12..31] = \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{where:} \tag{177}$$

$$\mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] = I_{\mathbf{d}} \tag{178}$$

$$\mathbf{d}[|I_{\mathbf{d}}|..] = (0, 0, ...) \tag{179}$$

$$h = \mathbf{d}[0..31] \tag{180}$$

$$v = \mathbf{d}[32..63] \tag{181}$$

$$r = \mathbf{d}[64..95] \tag{182}$$

$$s = \mathbf{d}[96..127] \tag{183}$$

The two hash functions, RIPEMD-160 and SHA2-256 are more trivially defined as an almost pass-through operation. Their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words.

$$\Xi_{\text{SHA256}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{184}$$

$$g_r = 50 + 50\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{185}$$

$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{186}$$

$$\Xi_{\text{RIP160}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{187}$$

$$g_r = 50 + 50\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{188}$$

$$\mathbf{o}[0..11] = 0 \tag{189}$$

$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{190}$$

(191)

For the purposes here, we assume we have well-defined standard cryptographic functions for RIPEMD-160 and SHA2-256 of the form:

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{192}$$

$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{193}$$

Finally, the fourth contract, the identity function Ξ_{ID} simply defines the output as the input:

$$\Xi_{\text{ID}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{194}$$

$$g_r = 1 + \left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{195}$$

$$\mathbf{o} = I_{\mathbf{d}} \tag{196}$$

Appendix F. Signing Transactions

The method of signing transactions is similar to the 'Electrum style signatures'; it utilises the SECP-256k1 curve as described by Gura et al. [2004].

It is assumed that the sender has a valid private key p_r, a randomly selected positive integer in the range $(0, 2^{256})$ represented as a byte array of length 32 in big-endian form.

We assert the functions $\texttt{ECDSASIGN}$, $\texttt{ECDSARESTORE}$ and $\texttt{ECDSAPUBKEY}$. These are formally defined in the literature.

$$\texttt{ECDSAPUBKEY}(p_r \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{197}$$

$$\texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_r \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{198}$$

$$\texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{199}$$

Where p_u is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each $< 2^{256}$) and p_r is the private key, a byte array of size 32 (or a single positive integer $< 2^{256}$). It is assumed that v is the 'recovery id', a 1 byte value specifying the sign and finiteness of the curve point; this value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid.

We declare that a signature is invalid unless the following is true:

$$v \in \{27, 28\} \quad \wedge \quad r < \texttt{secp256k1n} \quad \wedge \quad s < \texttt{secp256k1p} \tag{200}$$

where:

$$\texttt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{201}$$

$$\texttt{secp256k1p} = 2^{256} - 2^{32} - 977 \tag{202}$$

(203)

For a given private key, p_r, the Ethereum address $A(p_r)$ (a 160-bit value) to which it corresponds is defined as the right most 160-bits of the Keccak hash of the corresponding ECDSA public key:

$$A(p_r) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_r)\big)\big) \tag{204}$$

The message hash, $h(T)$, to be signed is the Keccak hash of the transaction without the latter three signature components, formally described as T_r, T_s and T_w:

$$L_S(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}) & \text{if } T_t = 0 \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}) & \text{otherwise} \end{cases} \tag{205}$$

$$h(T) \equiv \texttt{KEC}(L_S(T)) \tag{206}$$

The signed transaction $G(T, p_r)$ is defined as:

$$G(T, p_r) \equiv T \quad \text{except:} \tag{207}$$

$$(T_w, T_r, T_s) = \texttt{ECDSASIGN}(h(T), p_r) \tag{208}$$

We may then define the sender function S of the transaction as:

$$S(T) \equiv \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSARECOVER}(h(T), T_w, T_r, T_s)\big)\big) \tag{209}$$

The assertion that the sender of the a signed transaction equals the address of the signer should be self-evident:

$$\forall T : \forall p_r : S(G(T, p_r)) \equiv A(p_r) \tag{210}$$

Appendix G. Fee Schedule

The fee schedule G is a tuple of 10 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{step}	1	Default amount of gas to pay for execution cycle.
$G_{balance}$	20	Paid for a BALANCE operation.
G_{stop}	0	Nothing paid for the STOP operation.
$G_{suicide}$	0	Nothing paid for the SUICIDE operation.
G_{sload}	20	Paid for a SLOAD operation.
G_{sset}	300	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	100	Paid for an SSTORE operation when the storage value's zeroness remains unchanged.
G_{sclear}	0	Nothing paid for an SSTORE operation when the storage value is set to zero from non-zero.
R_{sclear}	100	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
G_{create}	100	Paid for a CREATE operation.
$G_{createdata}$	5	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	20	Paid for a CALL operation.
G_{exp}	1	Partial payment for an EXP operation.
$G_{expbyte}$	1	Partial payment when multiplied by $\lceil \log_{256}(\mathit{exponent}) \rceil$ for the EXP operation.
G_{memory}	1	Paid for every additional word when expanding memory.
$G_{txdatazero}$	1	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	5	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	500	Paid for every transaction.
G_{log}	1	Partial payment for a LOG operation.
$G_{logdata}$	1	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	1	Paid for each topic of a LOG operation.
G_{sha3}	10	Paid for each SHA3 operation.
$G_{sha3word}$	10	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	1	Partial payment for *COPY operations, multiplied by words copied, rounded up.

Appendix H. Virtual Machine Specification

When interpreting 256-bit binary values as integers, the representation is big-endian.

When a 256-bit machine datum is converted to and from a 160-bit address or hash, the rightwards (low-order for BE) 20 bytes are used and the left most 12 are discarded or filled with zeroes, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.

H.1. **Gas Cost.** The general gas cost function, C, is defined as:

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv C_{memory}(\boldsymbol{\mu}'_i) - C_{memory}(\boldsymbol{\mu}_i) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SSTORE} \\ G_{exp} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{step} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CALLDATACOPY} \\ G_{step} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CODECOPY} \\ G_{step} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{logtopic} & \text{if} \quad w = \text{LOG1} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{logtopic} & \text{if} \quad w = \text{LOG2} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{logtopic} & \text{if} \quad w = \text{LOG3} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{logtopic} & \text{if} \quad w = \text{LOG4} \\ G_{call} + \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if} \quad w = \text{CALL} \\ G_{create} & \text{if} \quad w = \text{CREATE} \\ G_{sha3} + G_{sha3word} \lceil \mathbf{s}[1] \div 32 \rceil & \text{if} \quad w = \text{SHA3} \\ G_{sload} & \text{if} \quad w = \text{SLOAD} \\ G_{balance} & \text{if} \quad w = \text{BALANCE} \\ G_{stop} & \text{if} \quad w = \text{STOP} \\ G_{suicide} & \text{if} \quad w = \text{SUICIDE} \\ G_{step} & \text{otherwise} \end{cases} \tag{211}$$

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{212}$$

where:

$$C_{memory}(a) \equiv G_{memory}(a + \left\lfloor \frac{a^2}{1024} \right\rfloor) \tag{213}$$

and C_{SSTORE} is specified in the appropriate section below.

Note the memory cost component, given as the product of G_{memory} and the maximum of 0 & the ceiling of the number of words in size that the memory must be over the current number of words, $\boldsymbol{\mu}_i$ in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes.

Referencing a zero length range (e.g. by attempting to pass it as the input range to a CALL) does not require memory to be extended to the beginning of the range. $\boldsymbol{\mu}'_i$ is defined as this new maximum number of words of active memory; special-cases are given where these two are not equal.

Note also that C_{memory} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 32KB of memory used, after which it costs substantially more.

While defining the instruction set, we defined the memory-expansion for range function, M, thus:

$$M(s, f, l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f + l) \div 32 \rceil) & \text{otherwise} \end{cases} \tag{214}$$

H.2. **Instruction Set.** As previously specified in section 9, these definitions take place in the final context there. In particular we assume O is the EVM state-progression function and define the terms pertaining to the next cycle's state $(\boldsymbol{\sigma}', \boldsymbol{\mu}')$ such that:

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted} \tag{215}$$

Here given are the various exceptions to the state transition rules given in section 9 specified for each instruction, together with the additional instruction-specific definitions of J and C. For each instruction, also specified is α, the additional items placed on the stack and δ, the items removed from stack, as defined in section 9.

TODO: Whenever a stack item is used as an address, it should be assumed it is the stack item modulo 2^{160}.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted.

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution. $\boldsymbol{\mu}'_R = []$
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$
0x05	SDIV	2	1	Signed integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers. Note the overflow semantic when -2^{255} is negated.
0x06	MOD	2	1	Modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{otherwise} \end{cases}$
0x07	SMOD	2	1	Signed modulo remainder operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0])\lvert\boldsymbol{\mu}_{\mathbf{s}}[0]\rvert \bmod \lvert\boldsymbol{\mu}_{\mathbf{s}}[1]\rvert & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \bmod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x0a	EXP	2	1	Exponential operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]^{\boldsymbol{\mu}_{\mathbf{s}}[1]}$
0x0b	SIGNEXTEND	2	1	Extend length of two's complement signed integer. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_t & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_i & \text{otherwise} \end{cases}$

10s: Comparison & Bitwise Logic Operations

$\boldsymbol{\mu}_\mathbf{s}[0]_i$ gives the ith bit (counting from zero) of $\boldsymbol{\mu}_\mathbf{s}[0]$

Value	Mnemonic	δ	α	Description
0x10	LT	2	1	Less-than comparision. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] < \boldsymbol{\mu}_\mathbf{s}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparision. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\mu}_\mathbf{s}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparision. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] < \boldsymbol{\mu}_\mathbf{s}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparision. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\mu}_\mathbf{s}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparision. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] = \boldsymbol{\mu}_\mathbf{s}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_\mathbf{s}[0]_i \equiv \boldsymbol{\mu}_\mathbf{s}[0]_i \wedge \boldsymbol{\mu}_\mathbf{s}[1]_i$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_\mathbf{s}[0]_i \equiv \boldsymbol{\mu}_\mathbf{s}[0]_i \vee \boldsymbol{\mu}_\mathbf{s}[1]_i$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_\mathbf{s}[0]_i \equiv \boldsymbol{\mu}_\mathbf{s}[0]_i \oplus \boldsymbol{\mu}_\mathbf{s}[1]_i$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_\mathbf{s}[0]_i \equiv \begin{cases} 1 & \text{if } \boldsymbol{\mu}_\mathbf{s}[0]_i = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_\mathbf{s}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_\mathbf{s}[1]_{(i+8\boldsymbol{\mu}_\mathbf{s}[0])} & \text{if } i < 8 \wedge \boldsymbol{\mu}_\mathbf{s}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

20s: SHA3

Value	Mnemonic	δ	α	Description
0x20	SHA3	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \ldots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)])$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$

				30s: Environmental Information
Value	**Mnemonic**	δ	α	**Description**
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_a$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_b & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_o$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_s$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_v$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{d}}\|$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad i < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases}$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{b}}\|$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad i < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$
0x3a	GASPRICE	0	1	Get price of gas in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_p$ This is gas price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\sigma}[\boldsymbol{\mu}_s[0] \mod 2^{160}]_c\|$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad i < \|\mathbf{c}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_s[0] \mod 2^{160}]_c$

40s: Block Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{H_p}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ where P is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than the current block number or more than 256 blocks behind the current block. $P(h,n,a) \equiv \begin{cases} 0 & \text{if} \quad n > H_i \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_i \\ P(H_p, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined as its hash is the parent hash in the block following it.
0x41	COINBASE	0	1	Get the block's coinbase address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hc}$
0x42	TIMESTAMP	0	1	Get the block's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hs}$
0x43	NUMBER	0	1	Get the block's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hi}$
0x44	DIFFICULTY	0	1	Get the block's difficulty. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hd}$
0x45	GASLIMIT	0	1	Get the block's gas limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{Hl}$

50s: Stack, Memory, Storage and Flow Operations				
Value	**Mnemonic**	δ	α	**Description**
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \wedge \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sclear} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \wedge \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ G_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \wedge \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\text{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 9.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{pc} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 9.
0x58	PC	0	1	Get the program counter. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{pc}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
0x5a	GAS	0	1	Get the amount of available gas. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_g$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{pc}+1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if } x < \|I_{\mathbf{b}}\| \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+2)\big)$ where c is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+32)\big)$ where c is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

Value	Mnemonic	δ	α	Description
0x80	DUP1	1	2	Duplicate 1st stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x81	DUP2	2	3	Duplicate 2nd stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$
⋮	⋮	⋮	⋮	⋮
0x8f	DUP16	16	17	Duplicate 16th stack item. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[15]$

90s: Exchange Operations

Value	Mnemonic	δ	α	Description
0x90	SWAP1	2	2	Exchange 1st and 2nd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathbf{s}}[1] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
0x91	SWAP2	3	3	Exchange 1st and 3rd stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[2]$ $\boldsymbol{\mu}'_{\mathbf{s}}[2] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$
⋮	⋮	⋮	⋮	⋮
0x9f	SWAP16	17	17	Exchange 1st and 17th stack items. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[16]$ $\boldsymbol{\mu}'_{\mathbf{s}}[16] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_a, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

Value	Mnemonic	δ	α	Description
0xa0	LOG0	2	0	Append log record with no topics. $\mathbf{t} \equiv ()$
0xa1	LOG1	3	0	Append log record with one topic. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2])$
⋮	⋮	⋮	⋮	⋮
0xa4	LOG4	6	0	Append log record with four topics. $\mathbf{t} \equiv (\boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[3], \boldsymbol{\mu}_{\mathbf{s}}[4], \boldsymbol{\mu}_{\mathbf{s}}[5])$

f0s: System operations

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[1] \dots (\boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A^+) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_a, I_o, \boldsymbol{\mu}_g, I_p, \boldsymbol{\mu}_\mathbf{s}[0], \mathbf{i}, I_e + 1) & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[0] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_g, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}$ except $\boldsymbol{\sigma}^*[I_a]_n = \boldsymbol{\sigma}[I_a]_n + 1$ $A' \equiv A \Cup A^+$ which implies: $A'_\mathbf{s} \equiv A_\mathbf{s} \cup A^+_\mathbf{s} \quad \wedge \quad A'_\mathbf{l} \equiv A_\mathbf{l} \cdot A^+_\mathbf{l} \quad \wedge \quad A'_\mathbf{r} \equiv A_\mathbf{r} + A^+_\mathbf{r}$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to lack of gas or $I_e = 1024$ (the maximum call depth limit is reached) or $\boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\sigma}[I_a]_b$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_a, \boldsymbol{\sigma}[I_a]_n)$, the address of the newly created account, otherwise. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[1], \boldsymbol{\mu}_\mathbf{s}[2])$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $\mathbf{o} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + \boldsymbol{\mu}_\mathbf{s}[6] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1]$ where $x = 0$ if the code execution for this operation failed due to lack of gas or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_a]_b$ (not enough funds) or $I_e = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_i \equiv M(M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.
0xf2	CALLCODE	7	1	Message-call into this account with alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_a, I_o, I_a, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \text{if} \quad \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, \mathbf{o}) & \text{otherwise} \end{cases}$ Note the change in the fourth parameter to the call Θ from the 2nd stack value $\boldsymbol{\mu}_\mathbf{s}[1]$ (as in CALL) to the present address I_a. This means that the recipient is in fact the same account as at present, simply that the code is overridden altered.
0xf3	RETURN	2	0	Halt execution returning output data. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 9. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$
0xff	SUICIDE	1	0	Halt execution and register account for later deletion. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_a\}$ $\boldsymbol{\sigma}'[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b + \boldsymbol{\sigma}[I_a]_b$ $\boldsymbol{\sigma}'[I_a]_b \equiv 0$

Appendix I. Genesis Block

The genesis block is 13 items, and is specified thus:

$$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 2^{22}, 0, 0, 1000000, 0, 0, 0, \texttt{KEC}\big((42)\big)\big), (), ()\big) \tag{216}$$

Where 0_{256} refers to the parent hash, a 256-bit hash which is all zeroes; 0_{160} refers to the coinbase address, a 160-bit hash which is all zeroes; 2^{22} refers to the difficulty; 0 refers to the timestamp (the Unix epoch); the transaction trie root and extradata are both 0, being equivalent to the empty byte array. The sequences of both uncles and transactions are empty and represented by (). $\texttt{KEC}\big((42)\big)$ refers to the Keccak hash of a byte array of length one whose first and only byte is of value 42. $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ value refers to the hash of the uncle lists in RLP, both empty lists.

The proof-of-concept series include a development premine, making the state root hash some value *stateRoot*. The latest documentation should be consulted for the value of the state root.