ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
PARIS VERSION 71beac3 – 2024-06-26

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

ABSTRACT. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. 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 that can be 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.

Dealsings 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, we 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.

Dwork and Naor [1992] provided the first work into the usage of a cryptographic proof of computational expenditure ("proof-of-work") as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a strong economic signal, allowing a receiver to make a physical assertion without having to rely upon trust. Back [2002] later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by Vishnumurthy et al. [2003]. In this instance, the token was used to keep peer-to-peer file trading in check, providing “consumers” with the ability to make micro-payments to “suppliers” for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record couldn’t be corrupted and that malicious actors could not spoof payment or unjustly complain about service delivery. Five years later, Nakamoto [2008] introduced another such proof-of-work-secured value token, somewhat wider in scope. The fruits of this project, Bitcoin, became the first widely adopted global 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 Sprinkel [2013]. Other projects sought to take the core value content mechanism of the protocol and repurpose it; Aon [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 et al. [2012], takes a similar but more simplified strategy, emboldening 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.

For a list of terms used in this paper, refer to Appendix A.

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 current state. It is this current state which we accept as the canonical “version” of the world of Ethereum. The state can include such information as account balance, incurring value. To address this issue, Ethereum has an intrinsic reward, exchange, currency, a balance or a payment, that would be far too big).

Formally, we expand to:

\[
\begin{align*}
\sigma_{t+1} & \equiv \Pi(\sigma, B) \\
B & \equiv \left(\ldots, T_0, T_1, \ldots\right) \\
\Pi(\sigma, B) & \equiv \left(\Pi(\mathcal{Y}(\sigma, T_0), T_1)\right) \ldots
\end{align*}
\]

Where \(B\) is this block, which includes a series of transactions amongst some other components and \(\Pi\) 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 reward, also as ETH and sometimes referred to by the Old English D. 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:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^6)</td>
<td>Wei</td>
</tr>
<tr>
<td>(10^{12})</td>
<td>Szabo</td>
</tr>
<tr>
<td>(10^{15})</td>
<td>Finney</td>
</tr>
<tr>
<td>(10^{18})</td>
<td>Ether</td>
</tr>
</tbody>
</table>

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.

Since the Paris hard fork, reaching consensus on new blocks is managed by a protocol called the Beacon Chain. It is known as the consensus layer of Ethereum, and it defines the rules for determining the canonical history of Ethereum blocks. This document describes the execution layer of Ethereum. The execution layer defines the rules for interacting with and updating the state of the Ethereum virtual machine. The consensus layer is described in greater detail in the consensus specifications. How the consensus layer is used to determine the canonical state of Ethereum is discussed in section 11.
There are many versions of Ethereum, as the protocol has undergone a number of updates. These updates can be specified to occur:

- at a particular block number in the case of pre-Paris updates,
- after reaching a terminal total difficulty in the case of the Paris update, or
- at a particular block timestamp in the case of post-Paris updates.

This document describes the Paris version.

In order to follow back the history of a path, one must reference multiple versions of this document. Here are the block numbers of protocol updates on the Ethereum main network:

<table>
<thead>
<tr>
<th>Name</th>
<th>First Block Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Homestead</td>
<td>1150000</td>
</tr>
<tr>
<td>TangerineWhistle</td>
<td>2463000</td>
</tr>
<tr>
<td>SpuriousDragon</td>
<td>2675000</td>
</tr>
<tr>
<td>Byzantium</td>
<td>4370000</td>
</tr>
<tr>
<td>Constantinople</td>
<td>7280000</td>
</tr>
<tr>
<td>Peterburg</td>
<td>7280000</td>
</tr>
<tr>
<td>Istanbul</td>
<td>9069000</td>
</tr>
<tr>
<td>MuirGlacier</td>
<td>9200000</td>
</tr>
<tr>
<td>Berlin</td>
<td>12244000</td>
</tr>
<tr>
<td>London</td>
<td>12965000</td>
</tr>
<tr>
<td>ArrowGlacier</td>
<td>13773000</td>
</tr>
<tr>
<td>GrayGlacier</td>
<td>15050000</td>
</tr>
<tr>
<td>Paris</td>
<td>15537394</td>
</tr>
</tbody>
</table>

Occasionally actors do not agree on a protocol change, and a permanent fork occurs. In order to distinguish between diverged blockchains, EIP-155 by Buterin [2016] introduced the concept of chain ID, which we denote by $\beta$. For the Ethereum main network

\[
\beta = 1
\]

3. Conventions

We 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 $\sigma$ (or a variant thereupon) and those of machine-state, $\mu$.

Functions operating on highly structured values are denoted with an upper-case Greek letter, e.g. $\Sigma$, 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. $C_{\text{store}}$; the cost function for the `STORE` operation. For specialised and possibly externally defined functions, we may format as typewriter text, e.g. the Keccak-256 hash function (as per version 3 of the winning entry to the SHA-3 contest by Bertoni et al. [2011], rather than the final SHA-3 specification), is denoted $\text{KEC}$ (and generally referred to as plain Keccak). Also, $\text{KEC512}$ refers to the Keccak-512 hash function.

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. $T^n_1$; 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. $\delta$, 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. $o$ 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 non-negative integers and thus belong to the set $\mathbb{N}$. 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 non-negative integers smaller than $2^{256}$ is named $\mathbb{N}_{256}$. 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. $\mu_0[0..31]$ denotes the first 32 items of the machine’s memory. For subsequences, ellipses are used to specify the intended range, to include elements at both limits, e.g. $\mu_0[0..31]$ denotes the first 32 items of the machine’s memory.

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

When considering variants of existing values, we 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, we 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.

We define a number of useful functions throughout. One of the more common is $\ell$, which evaluates to the last item in the given sequence:

\[
\ell(x) = x[\|x\| - 1]
\]

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

\[\text{Note that while the Paris fork activated at block 15,537,394, the trigger was not the block number, but rather reaching a specified total difficulty. The trigger for the Paris hard fork is discussed in greater detail in section 3.0.}\]
the implementation will maintain this mapping in a modified Merkle Patricia tree (\textit{trie}, see Appendix F). The trie requires a simple database backend that maintains a mapping of byte arrays to byte arrays; 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, \( \sigma[a] \), 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 \( \sigma \), this would be formally denoted \( \sigma[a]_{\text{nonce}} \).

- **balance**: A scalar value equal to the number of Wei owned by this address. Formally denoted \( \sigma[a]_{\text{balance}} \).

- **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 Keccak 256-bit hash of the 256-bit integer keys to the RLP-encoded 256-bit integer values. The hash is formally denoted \( \sigma[a]_{\text{storageRoot}} \).

- **codeHash**: The hash of the EVM code of this account—this is the code that gets executed should this address receive a message call. All such code fragments are contained in the state database under their corresponding hashes for later retrieval. This hash is formally denoted \( \sigma[a]_{\text{codeHash}} \), and thus the code may be denoted as \( b \), given that \( \text{KEC}(b) = \sigma[a]_{\text{codeHash}} \).

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

\[
\text{TRIE}(L_1((\sigma[a])) = \sigma[a]
\]

The collapse function for the set of key/value pairs in the trie, \( L_1 \), is defined as the element-wise transformation of the base function \( L_1 \), given as:

\[
L_1((k, v)) = (\text{KEC}(k), \text{RLP}(v))
\]

where:

\[
k \in \mathbb{B}_{32} \land v \in \mathbb{N}
\]

It shall be understood that \( \sigma[a]_{\text{storageRoot}} \) 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. \( \sigma[a]_{\text{codeHash}} = \text{KEC}() \), then the node represents a simple account, sometimes referred to as a “non-contract” account.

Thus we may define a world-state collapse function \( L_8 \):

\[
L_8(\sigma) = \{(p(a) : \sigma[a] \neq \varnothing)\}
\]

where

\[
p(a) \equiv (\text{KEC}(a), \text{RLP}((\sigma[a]_{\text{nonce}}, \sigma[a]_{\text{balance}}, \sigma[a]_{\text{storageRoot}}, \sigma[a]_{\text{codeHash}})))
\]

This function, \( L_8 \), is used alongside the trie function to provide a short identity (hash) of the world state. We assume:

\[
\forall a : \sigma[a] = \varnothing \lor (a \in \mathbb{B}_{20} \land v(\sigma[a]))
\]

where \( v \) is the account validity function:

\[
v(x) = x_n \in \mathbb{N}_{256} \land x_v \in \mathbb{B}_{32} \land x_{s} \in \mathbb{B}_{32}
\]

An account is empty when it has no code, zero nonce and zero balance:

\[
\text{EMPTY}(\sigma, a) \equiv \sigma[a]_{\text{nonce}} = \text{KEC}() \wedge \sigma[a]_{\text{balance}} = 0 \wedge \sigma[a]_{\text{storageRoot}} = 0
\]

Even callable precompiled contracts can have an empty account state. This is because their account states do not usually contain the code describing its behavior.

An account is dead when its account state is non-existent or empty:

\[
\text{DEAD}(\sigma, a) \equiv \sigma[a] = \varnothing \lor \text{EMPTY}(\sigma, a)
\]

4.2. The Transaction. A transaction (formally, \( T \)) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. The sender of a transaction cannot be a contract. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemination\(^2\); EIP-2718 by Zoltu [2020] introduced the notion of different transaction types. As of the London version of the protocol, there are three transaction types: 0 (legacy), 1 (EIP-2930 by Buterin and Swende [2020]), and 2 (EIP-1559 by Buterin et al. [2019]). Further, there are two subtypes 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’). All transaction types specify a number of common fields:

- **type**: EIP-2718 transaction type; formally \( T_a \).
- **nonce**: A scalar value equal to the number of transactions sent by the sender; formally \( T_n \).
- **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 \( T_v \).
- **r, s**: Values corresponding to the signature of the transaction and used to determine the sender of the transaction; formally \( T_r \) and \( T_s \). This is expanded in Appendix F; EIP-2030 (type 1) and EIP-1559 (type 2) transactions also have:
accessList: List of access entries to warm up; formally $T_a$. Each access list entry $E$ is a tuple of a contract address and a list of storage keys: $E \equiv (E_\text{addr}, E_\text{keys})$.

chainId: Chain ID; formally $T_c$. Must be equal to the network chain ID $\mathcal{T}$.

yParity: Signature Y parity; formally $T_y$.

Legacy transactions do not have an accessList ($T_a = ()$), while chainId and yParity for legacy transactions are combined into a single value:

$w$: A scalar value encoding Y parity and possibly chain ID; formally $T_w$. $T_w = 27 + T_y$ or $T_w = 2 \beta + 35 + T_y$ (see EIP-155 by Buterin [2016]).

There are differences in how one’s acceptable gas price is specified in type 2 transactions versus type 0 and type 1 transactions. Type 2 transactions take better advantage of the gas market improvements introduced in EIP-1559 by explicitly limiting the priority fee\(^3\); that is paid. Type 2 transactions have the following two fields related to gas:

maxFeePerGas: A scalar value equal to the maximum 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 $T_m$.

maxPriorityFeePerGas: A scalar value equal to the maximum number of Wei to be paid to the block’s fee recipient as an incentive to include the transaction; formally $T_i$.

In contrast, type 0 and type 1 transactions specify gas price as a single value:

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 $T_p$.

Additionally, a contract creation transaction (regardless of transaction type) contains:

init: An unlimited size byte array specifying the EVM-code for the account initialisation procedure, formally $T_i$.

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 $T_d$.

Appendix $^3$ 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 $S(T)$.

\(^3\)The priority fee is discussed in greater detail in sections \(^5\) and \(^6\).

\(^4\)Type 0 and type 1 transactions will get the same gas price behavior as a type 2 transaction with $T_m$ and $T_i$ set to the value of $T_p$.

\(^5\)ommers is a gender-neutral term to mean “sibling of parent”; see https://nonbinary.miraheze.org/wiki/Gender_neutral_language_in_English#Aunt/Uncle.
The other two components in the block are a series of bytes in a Keccak-256 hash of the byte sequence.

The transaction receipt, \( R \), is a tuple of five items comprising: the type of the transaction, \( R_t \), the status code of the transaction, \( R_s \), the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened, \( R_u \), and the Bloom filter composed from information in those logs, \( R_b \):

\[
R \equiv (R_t, R_s, R_u, R_b)
\]

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

\[
L_R(R) \equiv (R_t, R_u, R_s, R_b)
\]

We assert that the status code \( R_s \) is a non-negative integer:

\[
R_s \in \mathbb{N}
\]

We assert that \( R_u \), the cumulative gas used, is a non-negative integer and that the logs Bloom, \( R_b \), is a hash of size 2048 bits (256 bytes):

\[
\begin{align*}
R_t & \in \mathbb{N} \quad \land \\
R_s & \in \mathbb{N} \quad \land \\
R_u & \in \mathbb{B}_{256}
\end{align*}
\]

The sequence \( R_t \) is a series of log entries, \((O_0, O_1, \ldots)\). A log entry, \( O \), is a tuple of the logger’s address, \( O_a \), a possibly empty series of 32-byte log topics, \( O_t \) and some number of bytes of data, \( O_d \):

\[
O \equiv (O_a, (O_t, O_0, O_1, \ldots), O_d)
\]

We define the Bloom filter function, \( M \), to reduce a log entry into a single 256-byte hash:

\[
M(O) \equiv \sum_{x \in \{(O_a,):O_t \}} (M_{2048}(x))
\]

where \( M_{2048} \) is a specialised Bloom filter that sets three bits out of 2048, given an arbitrary byte sequence. It does this through taking the low-order 11 bits of each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence:

Formally:

\[
M_{3:2048}(x : x \in \mathbb{B}) \equiv y : y \in \mathbb{B}_{256} \quad \text{where:}
\]

\[
\begin{align*}
y &= (0, 0, \ldots, 0) \quad \text{except:} \\
\forall i \in \{0, 2, 4\} & : B_{2048:m(x,i)}(y) = 1
\end{align*}
\]

\[
m(x, i) \equiv \text{KEC}(x_i)[i, i + 1] \mod 2048
\]

where \( B \) is the bit reference function such that \( B_t(x) \) equals the bit of index \( j \) (indexed from 0) in the byte array \( x \). Notably, it treats \( x \) as big-endian (more significant bits will have smaller indices).

4.3.2. Holistic Validity. We can assert a block’s validity if and only if it satisfies several conditions: the block’sommers field \( B_T \) must be an empty array and the block’s header must be consistent with the given transactions \( B_T \). For the header to be consistent with the transactions \( B_T \), \( \text{stateRoot} (H_s) \) must match the resultant state after executing all transactions in order on the base state \( \sigma \) (as specified in section 12), and \( \text{transactionsRoot} (H_t) \), \( \text{receiptsRoot} (H_r) \), and \( \text{logsBloom} (H_b) \) must be correctly derived from the transactions themselves, the transaction receipts resulting from execution, and the resulting logs, respectively.

\[
\begin{align*}
B_T & \equiv () \\
H_s & \equiv \text{TRIE}(L_s(\Pi(\sigma, B_T))) \\
H_t & \equiv \text{TRIE}(\{i < ||B_T||, i \in \mathbb{N} : pR(i, B_T[i]))} \\
H_r & \equiv \text{TRIE}(\{i < ||B_R||, i \in \mathbb{N} : pR(i, B_R[i]))} \\
H_b & \equiv \bigvee_{i \in B_R} (r_b)
\end{align*}
\]

where \( pR(k, v) \) and \( pR(k, v) \) are pairwise RLP transformations, but with a special treatment for EIP-2718 transactions:

\[
\begin{align*}
pr(k, T) & \equiv \begin{cases} \\
(\text{RLP}(k), L_T(T)) & \text{if } T = 0 \\
(T_k) : \text{RLP}(L_T(T)) & \text{otherwise}
\end{cases}
\end{align*}
\]

and

\[
\begin{align*}
pr(k, R) & \equiv \begin{cases} \\
(\text{RLP}(k), L_R(R)) & \text{if } R = 0 \\
(R_k) : \text{RLP}(L_R(R)) & \text{otherwise}
\end{cases}
\end{align*}
\]
Thus $\text{TRIE}(L_(\sigma))$ is the root node hash of the Merkle Patricia tree structure containing the key-value pairs of the state $\sigma$ 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, $B_R$, and that defined through the transaction’s state-accumulation function, $T$, are formalised later in section 4.2.

4.3.3. Serialisation. The function $L_B$ and $L_H$ are the preparation functions for a block and block header respectively. We assert the types and order of the structure for when the RLP transformation is required:

$$L_H(H) \equiv (H_p, H_a, H_c, H_l, H_w, H_v, H_i, H_l, H_w, H_c, H_a, H_i, H_v)$$

$$L_B(B) \equiv (L_H(B_H), L_T(B_T), L_H(B_R))$$

where $L_T$ takes a special case of EIP-2718 transactions:

$$L_T(T) = \begin{cases} \text{RLP}(\text{TRIE}(T)) & \text{if } T_k = 0 \\ \text{RLP}(T_k) & \text{otherwise} \end{cases}$$

with $L_T$ and $L_H$ being element-wise sequence transformations, thus:

$$f^*(x_0, x_1, \ldots) \equiv (f(x_0), f(x_1), \ldots) \text{ for any function } f$$

The component types are defined thus:

$$H_p, H_a, H_c, H_l, H_w, H_v, H_i, H_l, H_w, H_c, H_a, H_i, H_v \in \mathbb{B}_{32}$$

$$H_r \in \mathbb{B}_{256}$$

$$H_s \in \mathbb{N}$$

$$H_t \in \mathbb{B}$$

where

$$\mathbb{B}_n = \{B : B \in \mathbb{B} \land \|B\| = n\}$$

We now have a rigorous specification for the construction of a formal block structure. The RLP function $\text{RLP}$ (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 $P(B_m)$ to be the parent block of $B$, formally:

$$P(H) \equiv B' : \text{KEC}(\text{RLP}(B_m')) H_1$$

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

$$H_i \equiv P(H)_{H_1} + 1$$

The London release introduced the block attribute base fee per gas $H'_f$ (see EIP-1559 by Buterin et al. [2019]). The base fee is the amount of wei burned per unit of gas consumed while executing transactions within the block. The value of the base fee is a function of the difference between the gas used by the parent block and the parent block’s gas target.

The expected base fee per gas is defined as $F(H)$:

$$F(H) \equiv \begin{cases} 1000000000 & \text{if } H_i = F_{\text{London}} \\ P(H)_{H_1} & \text{if } P(H)_{H_1} = \tau \\ P(H)_{H_1} - \nu & \text{if } P(H)_{H_1} < \tau \\ P(H)_{H_1} + \nu & \text{if } P(H)_{H_1} > \tau \end{cases}$$

where:

$$\tau \equiv \frac{P(H)_{H_1}}{\rho}$$

$$\rho \equiv 2$$

$$\nu \equiv \begin{cases} \frac{P(H)_{H_1} \times (\tau - P(H)_{H_1})}{\tau} & \text{if } P(H)_{H_1} < \tau \\ \frac{P(H)_{H_1} \times (P(H)_{H_1} - \tau)}{\tau} & \text{if } P(H)_{H_1} > \tau \end{cases}$$

$$\max(\lfloor \frac{\nu_i}{\tau} \rfloor, 1)$$

The gas target, $\tau$, is defined as the gas limit $H_i$ divided by the elasticity multiplier, $\rho$, a global constant set to 2. So while blocks can consume as much gas as the gas limit, the base fee is adjusted so that on average, blocks consume as much gas as the gas target. The base fee is increased in the current block when the parent block’s gas usage exceeds the gas target, and, conversely, the base fee is decreased in the current block when the parent block’s gas usage is less than the gas target.

The magnitude of the increase or decrease in the base fee, defined as $\nu$, is proportional to the difference between the amount of gas the parent block consumed and the parent block’s gas target. The effect on the base fee is dampened by a global constant called the base fee max change denominator, formally $\xi$, set to 8. A value of 8 entails that the base fee can increase or decrease by at most 12.5% from one block to the next.

The canonical gas limit $H_i$ of a block of header $H$ must fulfill the relation:

$$H_i < P(H)_{H_1} + P(H)_{H_1} \times \frac{1}{1024}$$

$$H_i > P(H)_{H_1} - P(H)_{H_1} \times \frac{1}{1024}$$

$$H_i \geq 5000$$

To avoid a discontinuity in gas usage, the value of the parent block gas limit for the purpose of validating the current block’s gas limit is modified at the London fork block by multiplying it by the elasticity multiplier, $\rho$. We call this modified value $P(H)_{H_1}$. This ensures that the gas target for post-London blocks can be set roughly in line with the gas limit of pre-London blocks.

$H_s$ is the timestamp (in Unix’s time()) of block $H$ and must fulfill the relation:

$$H_s \geq P(H)_{H_1}$$

The Paris hard fork changed Ethereum’s consensus from proof of work to proof of stake, and thus deprecated many block header properties related to proof of work. These deprecated properties include nonce ($H_n$), ommersHash ($H_o$), difficulty ($H_d$), and mixHash ($H_m$).
**mixHash** has been replaced with a new field **prevRandao** ($H_n$). The other header fields related to proof of work have been replaced with constants:

\[(54) \quad H_n \equiv \text{KEC}(\text{RLP}(i)) \]
\[(55) \quad H_A = 0 \]
\[(56) \quad H_n = \text{0x0000000000000000} \]

The value of **prevRandao** must be determined using information from the Beacon Chain. While the details of generating the **RANDAO** value on the Beacon Chain is beyond the scope of this paper, we refer to the expected **RANDAO** value for the previous block as **PREVRAANDAO**().

Thus we are able to define the block header validity function **V(H)**:

\[(57) \quad V(H) \equiv H_n \leq H_1 \land H_1 < P(H)_{H||} + \frac{P(H)_{H||}}{1024} \land H_1 > P(H)_{H||} - \frac{P(H)_{H||}}{1024} \land H_1 \geq 5000 \land H_i > P(H)_{H_k} \land H_i = P(H)_{H_k} + 1 \land \|H_k\| \leq 32 \land H_i = F(H) \land H_0 = \text{KEC}(\text{RLP}(i)) \land H_4 = 0 \land H_6 = \text{0x0000000000000000} \land H_n = \text{PREVRAANDAO}() \]

Note additionally that **extraData** must be at most 32 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 effective gas price defined in section 4. 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.

Since the introduction of EIP-1559 by Buterin et al. [2019] in the London hard fork, every transaction included in a block must pay a **base fee**, which is specified as wei per unit of gas consumed and is constant for each transaction within a block. The ether that is paid to meet the base fee is burned (taken out of circulation). The base fee adjusts dynamically as a function of the previous block’s gas consumption relative to its gas target (a value that is currently half the block’s gas limit, which can be adjusted by validators). If the previous block’s total gas consumption exceeds the gas target, this indicates excess demand for block space at the current base fee, and the base fee is increased. Conversely, if the gas consumed in the previous block is lower than the gas target, demand for block space is lower than the gas target at the current base fee, and thus the base fee is decreased. This process of adjusting the base fee should bring the average block’s gas consumption in line with the gas target. Refer to section 4.3 for greater detail on how the base fee is set.

To incentivize validators to include transactions, there is an additional fee known as a **priority fee**, which is also specified as wei per unit of gas consumed. The total fee paid by the transaction therefore is the sum of the base fee per gas and the priority fee per gas multiplied by the total gas consumed. Ether used to satisfy the priority fee is delivered to the beneficiary address, the address of an account typically under the control of the validator.

Transactions using type 2 transactions can specify the maximum priority fee they are willing to pay (**maxPriorityFeePerGas**) as well as the max total fee they are willing to pay (**maxFeePerGas**), inclusive of both the priority fee and the base fee. **maxFeePerGas** must at least as high as the base fee for the transaction to be included in a block. Type 0 and type 1 transactions have only one field for specifying a gas price- **gasPrice**—which also must be at least as high as the base fee for inclusion in a block. The amount by which **gasPrice** is higher than the base fee constitutes the priority fee in the case of a type 0 or type 1 transaction.

Transactions are free to select any priority fee that they wish, however validators are free to ignore transactions as they choose. A higher priority fee on a transaction will therefore cost the sender more in terms of Ether and deliver a greater value to the validator and thus will more likely be selected for inclusion. Since there will be a (weighted) distribution of minimum acceptable priority fees, transactions will necessarily have a trade-off to make between lowering the priority fee and maximising the chance that their transaction will be included in a block 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 $\Gamma$. 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 sender account has no contract code deployed (see EIP-3607 by Feist et al. [2021]);
5. the gas limit is no smaller than the intrinsic gas, $g_0$, used by the transaction;
(6) the sender account balance contains at least the cost, \(v_0\), required in up-front payment;
(7) the \texttt{maxFeePerGas}, \(T_m\), in the case of type 2 transactions, or \texttt{gasPrice}, \(T_p\), in the case of type 0 and type 1 transactions, is greater than or equal to the block’s base fee, \(H_i\); and
(8) for type 2 transactions, \texttt{maxPriorityFeePerGas}, \(T_f\), must be no larger than \texttt{maxFeePerGas}, \(T_m\).

Formally, we consider the function \(T\), with \(T\) being a transaction and \(\sigma\) the state:

\[
\sigma' = T(\sigma, T)
\]

Thus \(\sigma'\) is the post-transactional state. We also define \(T^*\) to evaluate to the amount of gas used in the execution of a transaction, \(T\), to evaluate to the transaction’s accrued log items and \(\overline{T^*}\) to evaluate to the status code resulting from the transaction. These will 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 the \texttt{accrued transaction state}, or \texttt{accrued substate} for short, and represent it as \(A\), which is a tuple:

\[
A = (A_s, A_t, A_i, A_r, A_A, A_K)
\]

The tuple contents include \(A_s\), the self-destruct set: a set of accounts that will be discarded following the transaction’s completion. \(A_t\) 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). \(A_i\) is the set of touched accounts, of which the empty ones are deleted at the end of a transaction. \(A_r\) is the refund balance, increased through using the \texttt{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. Finally, EIP-2929 by Buterin and Swende [2020a] introduced \(A_A\), the set of accessed account addresses, and \(A_K\), the set of accessed storage keys (more accurately, each element of \(A_K\) is a tuple of a 20-byte account address and a 32-byte storage slot).

We define the empty accrued substate \(A^0\) to have no self-destructs, no logs, no touched accounts, zero refund balance, all precompiled contracts in the accessed addresses, and no accessed storage:

\[
A^0 = (\emptyset, (), \emptyset, 0, \pi, \emptyset)
\]

where \(\pi\) is the set of all precompiled addresses.

6.2. Execution. We define intrinsic gas \(g_0\), the amount of gas this transaction requires to be paid prior to execution, as follows:

\[
g_0 = \sum_{i \in T_1, T_4} \begin{cases} 
\text{supersatAddress} & \text{if } i = 0 \\
\text{ugsatAddress} & \text{otherwise} 
\end{cases} 
+ \begin{cases} 
\text{Gaccesslistaddress} & \text{if } T_i = \emptyset \\
0 & \text{otherwise} 
\end{cases} 
+ \sum_{j=0}^{\lvert T_A \rvert - 1} (\text{Gaccesslistaddress} + \lvert T_A \rvert \text{Gaccessliststorage})
\]

where \(T_1, T_4\) 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.

The validity is determined as:

\[
m = \begin{cases} 
T_p & \text{if } T_x = 0 \lor T_x = 1 \\
T_m & \text{if } T_x = 2
\end{cases}
\]

Note the final condition; the sum of the transaction’s gas limit, \(T_m\), and the gas utilised in this block prior, given by \(\sum (B_{hn})_u\), must be no greater than the block’s \texttt{gasLimit}, \(B_{Hn}\). Also, with a slight abuse of notation, we assume that \(\sigma[S(T)]_c = \text{KEC}()\), \(\sigma[S(T)]_n = 0\), and \(\sigma[S(T)]_b = 0\) if \(\sigma[S(T)] = \emptyset\).

For type 2 transactions, we add an additional check that \texttt{maxPriorityFeePerGas} is no larger than \texttt{maxFeePerGas}:

\[
T_m \geq T_i
\]

The execution of a valid transaction begins with an irreversible 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 part of the up-front cost, \(T_p\). The gas available for the proceeding computation, \(g\), is defined as \(T_x = g_0\). 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 \( \sigma_0 \):

\[
\begin{align*}
\sigma_0 & \equiv \sigma & \text{except:} \\
\sigma_0[S(T)]_i & \equiv \sigma[S(T)]_{b-} - T_a p \\
\sigma_0[S(T)]_n & \equiv \sigma[S(T)]_{n+1} 
\end{align*}
\]

Evaluating \( \sigma_T \) from \( \sigma_0 \) depends on the transaction type; either contract creation or message call; we define the tuple of post-execution provisional state \( \sigma_P \), remaining gas \( g' \), accrued state \( A \) and status code \( z \):

\[
\begin{align*}
(\sigma_P, g', A, z) & \equiv \\
\begin{cases}
A_0(\sigma_0, A^*, S(T), S(T), g, & p, T_v, T_i, 0, \varnothing, \top) & \text{if } T_i = \varnothing \\
\Theta(\sigma_0, A^*, S(T), S(T), T_v, & T_i, 0, \varnothing, \top) & \text{otherwise}
\end{cases}
\end{align*}
\]

where

\[
\begin{align*}
A^* & \equiv A^0 & \text{except} \\
A^0 & \equiv \bigcup_{E \in E_T} \{ \{ \varnothing < \ll E_a[i], i \in N \colon (E_a, E_b[i]) \} \}
\end{align*}
\]

and \( g \) is the amount of gas remaining after deducting the basic amount required to pay for the existence of the transaction:

\[
g \equiv T_b - g_0
\]

Note we use \( A_0 \) and \( A^0 \) to denote the fact that only the first four components of the functions’ values are taken; the final represents the message-call’s output value (a byte array) and is unused in the context of transaction evaluation.

Then 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.

\[
g^* \equiv g' + \min \left\{ \frac{T_b - g'}{5}, A_i \right\}
\]

The total refundable amount is the legitimately remaining gas \( g' \), added to \( A_i \) with the latter component being capped up to a maximum of one fifth \((rounded\ down)\) of the total amount used \( T_b - g' \). Therefore, \( g^* \) is the total gas that remains after the transaction has been executed.

The validator, whose address is specified as the beneficiary of the present block \( B \), receives the gas consumed multiplied by the transaction’s priority fee per gas, defined as \( f \) in this section. The ether that is paid by the transaction that goes toward the base fee is debited from the transaction’s account but credited to no other account, so it is burned.

We define the pre-final state \( \sigma^* \) in terms of the provisional state \( \sigma_T \):

\[
\begin{align*}
\sigma^* & \equiv \sigma_T & \text{except} \\
\sigma^*[S(T)]_i & \equiv \sigma_T[S(T)]_{b-} + g' p \\
\sigma^*[B_{hi}]_n & \equiv \sigma_T[B_{hi}]_{n+1} + (T_b - g')f
\end{align*}
\]

The max refundable proportion of gas was reduced from one half to one fifth by EIP-3529 by Buterin and Swende [2021] in the London release.

For \( \epsilon \) 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.
The address of the new account is added to the set of accessed accounts:

$$A' \equiv A$$ except $$A_0' \equiv A_0 \cup \{a\}$$

The account’s nonce is initially defined as one, the balance as the value passed 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 $$\sigma^*$$:

$$\sigma^* = \sigma$$ except:

$$\sigma^*[a] = (v, v', \text{TRIE}(\sigma), \text{KEC}(\sigma))$$

Thus the mutated state becomes $$\sigma^*$$:

$$\sigma^*[s] = \begin{cases} \emptyset & \text{if } \sigma[s] = \emptyset \land v = 0 \\ a^* & \text{otherwise} \end{cases}$$

$$a^* \equiv (\sigma[s], \sigma[s]_b, \sigma, \sigma[s], \sigma[s])$$

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

$$v' = \begin{cases} 0 & \text{if } \sigma[a] = \emptyset \\ \sigma[a]_b & \text{otherwise} \end{cases}$$

Finally, the account is initialised through the execution of the initialising EVM code $$i$$ 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 $$\Sigma_i$$ evaluates to a tuple of the resultant state $$\sigma^{**}$$, available gas remaining $$g^{**}$$, the resultant accrued substate $$A^{**}$$ and the body code of the account $$o$$.

$$\{\sigma^{**}, g^{**}, A^{**}, o\} \equiv \Sigma_i(\sigma^*, g, A', I)$$

where $$I$$ contains the parameters of the execution environment; that is:

$$I_a \equiv a$$

$$I_o \equiv o$$

$$I_p \equiv p$$

$$I_i \equiv i$$

$$I_s \equiv s$$

$$I_v \equiv v$$

$$I_{o[0]} \equiv 0$$

$$I_e \equiv e$$

$$I_w \equiv w$$

$$I_a$$ 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.

Code execution depletes gas, and gas may not go below zero, thus execution 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 (OOG) exception has occurred: The evaluated state is defined as being the empty set, $$\emptyset$$, and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation.

If the initialization code completes successfully, a final contract-creation cost is paid, the code-deposit cost, $$c$$, proportional to the size of the created contract’s code:

$$c \equiv G_{\text{codedeposit}} \times ||o||$$

If there is not enough gas remaining to pay this, i.e. $$g^{**} < c$$, then we also declare an out-of-gas exception.

The gas remaining will be zero in any such exceptional condition, i.e. if the creation was conducted as the reception of a transaction, then this doesn’t affect payment of the intrinsic cost of contract creation; it is paid regardless. However, the value of the transaction is not transferred to the aborted contract’s address when we are out-of-gas, thus the contract’s code is not stored.

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

$$g' \equiv \begin{cases} 0 & \text{if } F \\ g^{**} - c & \text{otherwise} \end{cases}$$

$$\sigma' \equiv \begin{cases} \sigma^{**} & \text{if } F \land \sigma^{**} = \emptyset \\ \sigma^{**} & \text{otherwise} \end{cases}$$

$$A' \equiv \begin{cases} A^{**} & \text{if } F \land \sigma^{**} = \emptyset \\ A^{**} & \text{otherwise} \end{cases}$$

$$z \equiv \begin{cases} 0 & \text{if } F \land \sigma^{**} = \emptyset \\ 1 & \text{otherwise} \end{cases}$$

where

$$F \equiv \left(\sigma'[a] \neq \emptyset \land (\sigma'[a]_c \neq \text{KEC}(a)) \lor (s|a|_c \neq 0)\right) \lor (\sigma^{**} = \emptyset \land o = \emptyset) \lor (g^{**} < c) \lor (||o|| > 24576) \lor (o[0] = 0xef)$$

Note the last condition of $$F$$ indicates that contract code cannot begin with the byte $$0xef$$ (refer to EIP-3541 by Beregszaszi et al. [2021]).

The exception in the determination of $$\sigma'$$ dictates that $$o$$, the resultant byte sequence from the execution of the initialisation code, specifies the final body code for the newly-created account.

Note that intention is that the result is either a successfully created new contract with its endowment, or no new contract with no transfer of value. In addition, observe that if the execution of the initialising code reverts $$(\sigma^{**} = \emptyset \land o \neq \emptyset)$$, the resultant gas $$g'$$ is not depleted (provided there was no other exception), but no new account is created.
7.1. Subtleties. Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code\textsuperscript{10}; Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SELFDESTRUCT 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 effective gas price (p) together with an arbitrary length byte array, d, the input data of the call, the present depth of the message-call/contract-creation stack (e) and finally the permission to make modifications to the state (w).

Aside from evaluating to a new state and accrued transaction substate, message calls also have an extra component—the output data denoted by the byte array 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.

(114) $(\sigma', g', A', z, o) \equiv \Theta(\sigma, A, s, o, r, c, g, p, v, \tilde{v}, d, e, w)$

Note that we need to differentiate between the value that is to be transferred, v, from the value apparent in the execution context, $\tilde{v}$, for the DELEGATECALL instruction.

We define $\sigma_1$, the first transitional state as the initialisation execution ends with a SELFDESTRUCT 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.

(122) $\sigma' \equiv \begin{cases} \sigma & \text{if } \sigma^{**} = \emptyset \\ \sigma^{**} & \text{otherwise} \end{cases}$

(123) $g' \equiv \begin{cases} o = \emptyset & \text{if } \sigma^{**} = \emptyset \\ g^{**} & \text{otherwise} \end{cases}$

(124) $A' \equiv \begin{cases} A & \text{if } \sigma^{**} = \emptyset \\ A^{**} & \text{otherwise} \end{cases}$

(125) $z \equiv \begin{cases} 0 & \text{if } \sigma^{**} = \emptyset \\ 1 & \text{otherwise} \end{cases}$

(126) $(\sigma^{**}, g^{**}, A^{**}, o) \equiv \Xi$

where

(135) $\Xi \equiv \begin{cases} \Xi_{\text{RECEIVED}}(\sigma_1, g, A, I) & \text{if } c = 1 \\ \Xi_{\text{SHA256}}(\sigma_1, g, A, I) & \text{if } c = 2 \\ \Xi_{\text{RIPEMD160}}(\sigma_1, g, A, I) & \text{if } c = 3 \\ \Xi_{\text{ID}}(\sigma_1, g, A, I) & \text{if } c = 4 \\ \Xi_{\text{EXPMOD}}(\sigma_1, g, A, I) & \text{if } c = 5 \\ \Xi_{\text{BLAKE2}}(\sigma_1, g, A, I) & \text{if } c = 6 \\ \Xi_{\text{SHA512}}(\sigma_1, g, A, I) & \text{if } c = 7 \\ \Xi_{\text{SHA384}}(\sigma_1, g, A, I) & \text{if } c = 8 \\ \Xi_{\text{BLAKE2}}(\sigma_1, g, A, I) & \text{if } c = 9 \\ \Xi(\sigma_1, g, A, I) & \text{otherwise} \end{cases}$

and

(136) $\Xi(\sigma_1, g, A, I) = \sigma[e]$

It is assumed that the client will have stored the pair $(\Xi(\sigma_1, g, A, I), I_h)$ at some point prior in order to make the determination of $I_h$ feasible.

As can be seen, there are nine exceptions to the usage of the general execution framework $\Xi$ for evaluation of the message call: these are so-called ‘precompiled’ contracts, meant as a preliminary piece of architecture that may later become native extensions. The contracts in addresses 1 to 9 execute the elliptic curve public key recovery function, the SHA2 256-bit hash scheme, the RIPEMD160-bit hash scheme, the identity function, arbitrary precision modular exponentiation, elliptic curve addition, elliptic curve scalar multiplication, an elliptic curve pairing check, and the BLAKE2 compression function $\mathcal{F}$ respectively. Their full formal definition is in Appendix E. We denote the set of

\textsuperscript{10}During initialization code execution, $\text{EXTCODESIZE}$ on the address should return zero, which is the length of the code of the account while $\text{CODESIZE}$ should return the length of the initialization code (as defined in $\text{HEW}$).
the addresses of the precompiled contracts by $\pi$:

$$\pi \equiv \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

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 items) is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word-addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

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

The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately.

9.2. Fees Overview. Fees (denominated in gas) are charged under three distinct circumstances, all three as prerequisite to the execution of an operation. The first and most common is the fee intrinsic to the computation of the operation (see Appendix C2). 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, CREATE2, 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 $\sigma$, the remaining gas for computation $g$, and the accrued substate $A$, 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.
- $I_s$, the sender address of the transaction that originated this execution.
- $I_p$, the price of gas paid by the signer of the transaction that originated this execution. This is defined as the effective gas price $\frac{p}{s}$ in section 6;
- $I_d$, 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.
- $I_b$, the byte array that is the machine code to be executed.
- $I_H$, the block header of the present block.
- $I_c$, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATE(2)s being executed at present).
- $I_w$, the permission to make modifications to the state.

The execution model defines the function $\Xi$, which can compute the resultant state $\sigma'$, the remaining gas $g'$, the resultant accrued substate $A'$ and the resultant output, $o$, given these definitions. For the present context, we will define it as:

$$\Xi(\sigma', g', A', o) \equiv \Xi(\sigma, g, A, I)$$

where we will remember that $A$, the accrued substate, is defined in section 6.1;

9.4. Execution Overview. We must now define the $\Xi$ function. In most practical implementations this will be modelled as an iterative progression of the pair comprising the full system state, $\sigma$ and the machine state, $\mu$. 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 $\emptyset$; this is important when interpreting the output of $H$, which evaluates to $\emptyset$ when execution is to continue but a series (potentially empty) when execution...
9.4.2. Exceptional Halting. The exceptional halting function \( Z \) is defined as:

\[
Z(\sigma, \mu, A, I) \equiv \mu_g < C(\sigma, \mu, A, I) \vee \\
\delta_w \equiv \emptyset \vee \\
||\mu|| < \delta_w \vee \\
w = JUMP \land \mu[0] \notin D(I_b) \vee \\
w = JUMPI \land \mu[1] \neq 0 \land \\
\mu[0] \notin D(I_b) \vee \\
w = RETURNDATACOPY \land \\
\mu[1] + \mu[2] > ||\mu|| \vee \\
||\mu|| - \delta_w + \alpha_w > 1024 \vee \\
-I_w \land W(w, \mu) \vee \\
w = SSTORE \land \mu_g \leq G_{\text{callstipend}}
\]

where

\[
W(w, \mu) \equiv w \in \{\text{CREATE, CREATE2, SSTORE, SELFDESTRUCT}\} \lor \\
\text{LOG0} \leq w \land w \leq \text{LOG4} \lor \\
w = \text{CALL} \land \mu[2] \neq 0
\]

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its \( \delta \) subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid, the new stack size would be larger than 1024 or state modification is attempted during a static call. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt. Also, the execution is in an exceptional halting state if the gas left prior to executing an SSTORE instruction is less than or equal to the call stipend \( G_{\text{callstipend}} \) — see EIP-2200 by Tang [2019] for more information.

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 JUMPDST 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 \( \text{STOP} \) operations that trail it).

Formally:

\[
D(c) \equiv D_1(c, 0)
\]

where:

\[
D_1(c, i) \equiv \begin{cases} 
\{\} & \text{if } i \geq ||c|| \\
\{i\} \cup D_1(c, N(i, c[i])) & \text{if } c[i] = \text{JUMPDST} \\
D_1(c, N(i, c[i])) & \text{otherwise}
\end{cases}
\]

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

\[
N(i, w) \equiv \begin{cases} 
i + w - \text{PUSH1} + 2 & \text{if } w \in \{\text{PUSH1, PUSH32}\} \\
i + 1 & \text{otherwise}
\end{cases}
\]
9.4.4. Normal Halting. The normal halting function $H$ is defined:

$$H(\mu; I) \equiv \begin{cases} H_{\text{RETURN}}(\mu) & \text{if } w \in \{\text{RETURN, REVERT}\} \\ () & \text{if } w \in \{\text{STOP, SELFDESTRUCT}\} \\ \emptyset & \text{otherwise} \end{cases}$$

The data-returning halt operations, RETURN and REVERT, have a special function $H_{\text{RETURN}}$. Note also the difference between the empty sequence and the empty set as discussed here.

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:

$$O((\sigma, \mu, A, I)) \equiv (\sigma', \mu', A', I)$$

$$\Delta \equiv \alpha_w - \delta_w$$

$$\forall x \in [\alpha_w, ||\mu||_w] : \mu'_w[x] \equiv \mu_w[x - \Delta]$$

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:

$$\mu'_s \equiv \mu_s - C(\sigma, \mu, A, I);$$

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

In general, we assume the memory, accrued substate and system state do not change:

$$\mu'_m \equiv \mu_m$$

$$\mu'_i \equiv \mu_i$$

$$\sigma' \equiv \sigma$$

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 $\alpha$ and $\delta$ and a formal description of the gas requirements.

10. Transition to Proof of Stake

The Paris hard fork changed the underlying consensus mechanism of Ethereum from proof of work to proof of stake.

Unlike all previous hard forks of Ethereum, Paris was not defined to occur at a particular block height, but rather after a specified terminal total difficulty was reached. Total difficulty was used instead of block height to avoid a scenario in which a minority of hash power could create a malicious fork that could race to satisfy the block height requirement and claim the first proof of stake block.

Thus the terminal block, the last proof of work block before the Paris fork takes effect, is defined as having:

$$B_t \geq 587'500'000'000'000'000'000'000$$

$$P(B_t) < 587'500'000'000'000'000'000'000$$

where $B_t$ is the total difficulty of block $B$ and $P(B_t)$ is the total difficulty of its parent.

Total difficulty for a proof of work (pre-Paris) block $B$ is defined recursively as:

$$B_t \equiv P(B_{t+1}) + H_t$$

where $H_t$ is the difficulty of the current block $B$.

Upon reaching the terminal block, new blocks are processed by the Beacon Chain.

11. Blocktree to Blockchain

Prior to the transition to proof of stake at the Paris hard fork, the canonical blockchain was defined as the block path with the greatest total difficulty, defined in section 10, as $B_t$.

After reaching the terminal block described in section 10, the greatest total difficulty rule must be removed in favor of a rule known as LMD Ghost.

Note that in order to determine what blocks comprise the canonical Ethereum blockchain after Paris, one must have additional information from the Beacon Chain, which is not described herein. We denote events emitted by the Beacon Chain with the prefix POS.

On each occurrence of a POS_FORK选择_UPDATED event, starting with the first at the transition block described in section 10, the canonical chain is defined as the chain beginning with the genesis block and ending at the block nominated by the event as the head of the chain.

The head of the chain should be updated if and only if a POS_FORK选择_UPDATED is emitted, in which case the head should set to the block specified by the event. No optimistic updates to the head of the chain should be made.

The POS_FORK选择_UPDATED event additionally references a finalized block. The most recent finalized block should be set to this block.

The canonical blockchain must also contain a block with the hash and number of the terminal block defined in section 10.

12. Block Finalisation

The process of finalising a block involves two stages:

1. validate transactions;
2. verify state.

12.1. Transaction Validation. The given gasUsed must correspond faithfully to the transactions listed: $\Gamma(B)$, the total gas used in the block, must be equal to the accumulated gas used according to the final transaction:

$$\text{gasUsed} = \Gamma(B)$$

12.2. State Validation. We may now define the function, $\Gamma$, that maps a block $B$ to its initiation state:

$$\Gamma(B) \equiv \begin{cases} \sigma_0 & \text{if } P(B) = \emptyset \\ \sigma_1 : \text{TRIE}(L_{\sigma_1}) & \text{otherwise} \end{cases}$$

Here, $\text{TRIE}(L_{\sigma_1})$ means the hash of the root node of a trie of state $\sigma_1$; it is assumed that implementations will store this in the state database, which is trivial and efficient since the trie is by nature an immutable data structure.

And finally we define $\Phi$, the block transition function, which maps an incomplete block $B$ to a complete block.
\[ B' : \]  
\[
\Phi(B) \equiv B' : B' = B \text{ except:} \\
\Phi(B) = \left\{ \begin{array}{ll}
\text{true} & \text{if } n < 0 \\
\text{false} & \text{otherwise}
\end{array} \right.
\]

As specified at the beginning of the present work, \( \Phi \) is the state-transition function, which is defined in terms of \( \Upsilon \), the transaction-evaluation function.

As previously detailed, \( R[n], R[n] \) and \( R[n] \) are the nth corresponding status code, logs and cumulative gas used after each transaction \( (R[n], \text{the fourth component}) \), which has already been defined in terms of the logs). We also define the nth state \( \sigma[n] \), which 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):

\[
\sigma[n] = \left\{ \begin{array}{ll}
\Upsilon(B) & \text{if } n < 0 \\
\Upsilon(\sigma[n-1], B_T[n]) & \text{otherwise}
\end{array} \right.
\]

In the case of \( B_T[n] \), 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:

\[
R[n] = \left\{ \begin{array}{ll}
0 & \text{if } n < 0 \\
\Upsilon(\sigma[n-1], B_T[n]) + R[n-1] & \text{otherwise}
\end{array} \right.
\]

For \( R[n] \), we utilise the \( \Upsilon' \) function that we conveniently defined in the transaction execution function:

\[
R[n] = \Upsilon'(\sigma[n-1], B_T[n])
\]

We define \( R[n] \) in a similar manner:

\[
R[n] = \Upsilon'(\sigma[n-1], B_T[n])
\]

Finally, we define \( \Pi \) as the final transaction’s resultant state, \( \ell(\sigma) \):

\[
\Pi(\sigma, B) \equiv \ell(\sigma)
\]

Thus the complete block-transition mechanism (before consensus) is defined.

13. IMPLEMENTING CONTRACTS

There are several patterns of contracts engineering that allow particular useful behaviours; two of these that we will briefly discuss are data feeds and random numbers.

13.1. 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 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.

13.2. 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 and the block’s beneficiary address. In order to make it hard for malicious validators to control those values, one should use the BLOCKHASH operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result.

14. FUTURE DIRECTIONS

The state database won’t be forced to maintain all past 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 node. A compressed archive of the trie structure at given points in time (perhaps one in every 10,000th block) could be maintained by the peer network, effectively recasting the genesis block. This would reduce the amount to be downloaded to a single archive plus a hard maximum limit of blocks.

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

15. CONCLUSION

We 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.

16. ACKNOWLEDGEMENTS

Many thanks to Aeron Buchanan for authoring the Homestead revisions, Christoph Jentzsch for authoring the Ethash algorithm and Yoichi Hirai for doing most of the EIP-150 changes. Important maintenance, useful corrections and suggestions were provided by a number of others from the Ethereum DEV organisation and Ethereum community at large including Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari and, of course, Vitalik Buterin.

17. AVAILABILITY

The source of this paper is maintained at \text{https://github.com/ethereum/yellowpaper/}. An auto-generated PDF is located at \text{https://ethereum.github.io/yellowpaper/paper.pdf}. 

PARIS VERSION
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 validators 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 $\mathbb{T}$:

\[
\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B}
\]

\[
\mathbb{L} \equiv \{(t : t = (t[0], t[1], \ldots) \land \forall n < ||t|| : t[n] \in \mathbb{T})\}
\]

\[
\mathbb{B} \equiv \{b : b = (b[0], b[1], \ldots) \land \forall n < ||b|| : b[n] \in \mathbb{O}\}
\]

Where $\mathbb{O}$ is the set of (8-bit) 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. The disjoint union $\cup$ is needed only to distinguish the empty byte array (\(\varepsilon\) \in \mathbb{B} from the empty list (\(\varepsilon\) \in \mathbb{L}, which are encoded differently as defined below; as common, we will abuse notation and leave the disjoint union indices implicit, inferable from context.
We define the RLP function as $\text{RLP}$ 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:

$$\text{RLP}(x) \equiv \begin{cases} R_0(x) & \text{if } x \in \mathbb{B} \\ R_1(x) & \text{otherwise}\end{cases}$$

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, provided that it contains fewer than 2^{64} bytes, 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.

Byte arrays containing 2^{64} or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding of a byte array is always below 192, and thus it can be readily distinguished from the encodings of sequences in L.

Formally, we define $R_0$:

$$R_0(x) \equiv \begin{cases} x & \text{if } \|x\| = 1 \land x[0] < 128 \\ (128 + \|x\|) \cdot x & \text{if } \|x\| < 56 \\ (183 + \text{BE}(\|x\|)) \cdot \text{BE}(\|x\|) \cdot x & \text{if } \|x\| < 2^{64} \\ \emptyset & \text{otherwise}\end{cases}$$

$$\text{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \land x = \sum_{n=0}^{\|b\| - 1} b_n \cdot 256^{\|b\| - 1 - n}$$

Thus $\text{BE}$ is the function that expands a non-negative 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, provided that they contain fewer than 2^{64} bytes, 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.

Sequences whose concatenated serialized items contain 2^{64} or more bytes cannot be encoded. This restriction ensures that the first byte of the encoding does not exceed 255 (otherwise it would not be a byte).

Thus we finish by formally defining $R_1$:

$$R_1(x) \equiv \begin{cases} (192 + \|s(x)\|) \cdot s(x) & \text{if } s(x) \neq \emptyset \land \|s(x)\| < 56 \\ (247 + \text{BE}(\|s(x)\|)) \cdot \text{BE}(\|s(x)\|) \cdot s(x) & \text{if } s(x) \neq \emptyset \land \|s(x)\| < 2^{64} \\ \emptyset & \text{otherwise}\end{cases}$$

$$s(x) \equiv \text{RLP}(x[0]) \cdot \text{RLP}(x[1]) \cdot \ldots \text{ if } \forall i : \text{RLP}(x[i]) \neq \emptyset \ldots \text{ otherwise}$$

If RLP is used to encode a scalar, defined only as a non-negative integer (in $\mathbb{N}$ or in $\mathbb{N}_o$ for any $x$), it must be encoded as the shortest byte array whose big-endian interpretation is the scalar. Thus the RLP of some non-negative integer $i$ is defined as:

$$\text{RLP}(i : i \in \mathbb{N}) \equiv \text{RLP}(\text{BE}(i))$$

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 $\text{HP}$ 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 $\mathbb{B}$):

$$\text{HP}(y, b)$$
The key is accumulated through the traversal, acquiring a single nibble from each branch node (just as with a radix tree).

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 be either arbitrary-length binary data, typically implemented as a database.

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.

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

\[
(191) \quad \mathbb{HP}(x, t) : x \in \mathcal{Y} \equiv \begin{cases} 
(16f(t), 16x[0] + x[1], 16x[2] + x[3], \ldots) & \text{if } ||x|| \text{ is even} \\
(16f(t) + 1) + x[0], 16x[1] + x[2], 16x[3] + x[4], \ldots & \text{otherwise}
\end{cases}
\]

\[
(192) \quad f(t) \equiv \begin{cases} 
2 & \text{if } t \neq 0 \\
0 & \text{otherwise}
\end{cases}
\]

The modified Merkle Patricia tree (trie) provides a persistent data structure to map between arbitrary-length binary data. 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 be either arbitrary-length binary data, typically implemented as a database.

Formally, we assume the input value \( J \), a set containing pairs of byte sequences with unique keys:

\[
(193) \quad J = \{(k_0 \in \mathbb{B}, v_0 \in \mathbb{B}), (k_1 \in \mathbb{B}, v_1 \in \mathbb{B}), \ldots\}
\]

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

\[
(194) \quad \forall I \in J : I \equiv (I_0, I_1)
\]

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:

\[
(195) \quad y(\mathcal{J}) = \{(k'_0 \in \mathbb{Y}, v_0 \in \mathbb{B}), (k'_1 \in \mathbb{Y}, v_1 \in \mathbb{B}), \ldots\}
\]

\[
(196) \quad \forall n : \forall i < 2||k_n|| : k'_n[i] \equiv \begin{cases} 
[k_n[i / 2] \div 16] & \text{if } i \text{ is even} \\
[k_n[(i + 2)] \mod 16] & \text{otherwise}
\end{cases}
\]

We define the function \( \text{TRIE} \), which evaluates to the root of the trie that represents this set when encoded in this structure:

\[
(197) \quad \text{TRIE}(\mathcal{J}) \equiv \text{KEC(RLP(c(\mathcal{J}, 0)))}
\]

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, we store nodes whose composed RLP is fewer than 32 bytes directly; for those larger we assert prescience of the byte array whose Keccak-256 hash evaluates to our reference. Thus we define in terms of \( c \), the node composition function:

\[
(198) \quad n(J, i) \equiv \begin{cases} 
() \in \mathbb{B} & \text{if } J = \emptyset \\
c(J, i) & \text{if } ||\text{RLP}(c(J, i))|| < 32 \\
\text{KEC(RLP}(c(J, i)))) & \text{otherwise}
\end{cases}
\]

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 sharing 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 1.

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 the keys of nibbles and the keys of branches as traversed from the root. The hex-prefix encoding method is used and the second parameter to the function is required to be 0.

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 $c$:

$$c(\mathcal{J}, i) \equiv \begin{cases} (\mathcal{H}(I_0[i..(\|I_0\| - 1)]), I_1) & \text{if } \|I\| = 1 \text{ where } \mathcal{H}: I \in \mathcal{J} \\ (\mathcal{H}(I_0[i..(j - 1)]), 0, n(\mathcal{J}, j)) & \text{if } i \neq j \text{ where } j = \max\{x : \exists I : \|I\| = x \land \forall I \in \mathcal{J} : I_0[0..(x - 1)] = I\} \end{cases}$$

(199)

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 $\mathcal{J}$ to a 32-byte hash and assert that only a single such hash exists for any $\mathcal{J}$, 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, $\Xi_{\text{PRE}}$, which implements the out-of-gas checking.

$$\Xi_{\text{PRE}}(\sigma, g, A, I) \equiv \begin{cases} (\sigma, 0, A,()) & \text{if } g < g_r \\ (\sigma, g - g_r, A, o) & \text{otherwise} \end{cases}$$

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

We define $\Xi_{\text{ECREC}}$ as a precompiled contract for the elliptic curve digital signature algorithm (ECDSA) public key recovery function (ecrecover). See Appendix F for the definition of the function $\Xi_{\text{ECDSARECOVER}}$ and the constant $\text{secp256k1n}$. We also define $d$ to be the input data, well-defined for an infinite length by appending zeroes as required. In the case of an invalid signature, we return no output.

$$\Xi_{\text{ECREC}} \equiv \Xi_{\text{PRE}} \text{ where:}$$

(201)

$$g_r = 3000$$

(202)

$$\|o\| = \begin{cases} 0 & \text{if } v \notin \{27, 28\} \lor r = 0 \lor r \geq \text{secp256k1n} \lor s = 0 \lor s \geq \text{secp256k1n} \\ 32 & \text{otherwise} \end{cases}$$

(203)

$$d[0..(\|I_a\| - 1)] = I_a$$

(204)

$$d[\|I_a\|..] = (0, 0, \ldots)$$

(205)

$$h = d[0..31]$$

(206)

$$r = d[64..95]$$

(207)

$$s = d[96..127]$$

We define $\Xi_{\text{SHA256}}$ and $\Xi_{\text{RIPEMD160}}$ as precompiled contracts implementing the SHA2-256 and RIPEMD-160 hash functions respectively. Their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words.

$$\Xi_{\text{SHA256}} \equiv \Xi_{\text{PRE}} \text{ where:}$$

(208)

$$g_r = 60 + 12\lfloor\frac{\|I_a\|}{32}\rfloor$$

(209)

$$\Xi_{\text{RIPEMD160}} \equiv \Xi_{\text{PRE}} \text{ where:}$$

(210)

$$g_r = 600 + 120\lfloor\frac{\|I_a\|}{32}\rfloor$$

(211)

$$o[0..11] = 0$$

(212)

$$o[12..31] = \text{RIPEMD160}(I_a)$$

(213)
For the purposes here, we assume we have well-defined standard cryptographic functions for RIPEMD-160 and SHA2-256 of the form:

\[(220)\] \[
\text{SHA256}(i \in \mathbb{B}) \equiv o \in \mathbb{B}_{32}
\]

\[(221)\] \[
\text{RIPEMD160}(i \in \mathbb{B}) \equiv o \in \mathbb{B}_{20}
\]

The fourth contract, the identity function \(\Xi_B\) simply defines the output as the input:

\[(222)\] \[
\Xi_B \equiv \Xi_{\text{PRE}} \quad \text{where:}
\]

\[(223)\] \[
g_r = 15 + 3 \left\lceil \frac{\|I_d\|}{32} \right\rceil
\]

\[(224)\] \[
o = I_d
\]

The fifth contract performs arbitrary-precision exponentiation under modulo. Here, 0\(^0\) is taken to be one, and \(x \mod 0\) is zero for all \(x\). The first word in the input specifies the number of bytes that the first non-negative integer \(B\) occupies. The second word in the input specifies the number of bytes that the second non-negative integer \(E\) occupies. The third word in the input specifies the number of bytes that the third non-negative integer \(M\) occupies. These three words are followed by \(B\), \(E\) and \(M\). The rest of the input is discarded. Whenever the input is too short, the missing bytes are considered to be zero. The output is encoded big-endian into the same format as \(M\)’s.

\[(225)\] \[
\Xi_{\text{EXPADD}} \equiv \Xi_{\text{PRE}} \quad \text{except:}
\]

\[(226)\] \[
g_r = \max\left(200, \left\lceil \frac{f(\max(\ell_M, \ell_B)) \max(\ell_E, 1)}{\ell_{\text{quaddivisor}}} \right\rceil \right)
\]

\[(227)\] \[
G_{\text{quaddivisor}} \equiv 3
\]

\[(228)\] \[
f(x) \equiv \left\lceil \frac{x^2}{8} \right\rceil
\]

\[(229)\] \[
\ell_E = \begin{cases} 
0 & \text{if } \ell_E \leq 32 \land E = 0 \\
[\log_2(E)] & \text{if } \ell_E \leq 32 \land E \neq 0 \\
8(\ell_E - 32) + [\log_2(i([96 + \ell_B]..(127 + \ell_B)])] & \text{if } 32 < \ell_E \land i([96 + \ell_B]..(127 + \ell_B)] \neq 0 \\
8(\ell_E - 32) & \text{otherwise}
\end{cases}
\]

\[(230)\] \[
o = (B^{E^o} \mod M) \in N_{0,M}
\]

\[(231)\] \[
\ell_B \equiv i[0..31]
\]

\[(232)\] \[
\ell_E \equiv i[32..63]
\]

\[(233)\] \[
\ell_M \equiv i[64..95]
\]

\[(234)\] \[
B \equiv i[96..(95 + \ell_B)]
\]

\[(235)\] \[
E \equiv i[(96 + \ell_B]..(95 + \ell_B + \ell_E)]
\]

\[(236)\] \[
M \equiv i[(96 + \ell_B + \ell_E]..(95 + \ell_B + \ell_E + \ell_M)]
\]

\[(237)\] \[
i[x] \equiv \begin{cases} 
I_d[x] & \text{if } x < \|I_d\| \\
0 & \text{otherwise}
\end{cases}
\]

E.1. zkSNARK Related Precompiled Contracts. We choose two numbers, both of which are prime.

\[(238)\] \[
p \equiv 2188824287183927522246405745257275088696311157297823662689037894645226208583
\]

\[(239)\] \[
q \equiv 21888242871839275222464057452572750885483644004160343698204186575808495617
\]

Since \(p\) is a prime number, \(\{0, 1, \ldots, p - 1\}\) forms a field with addition and multiplication modulo \(p\). We call this field \(F_p\).

We define a set \(C_1\) with

\[(240)\] \[
C_1 \equiv \{(X, Y) \in F_p \times F_p \mid Y^2 = X^3 + 3\} \cup \{(0, 0)\}
\]

We define a binary operation \(+\) on \(C_1\) for distinct elements \((X_1, Y_1), (X_2, Y_2)\) with

\[(241)\] \[
(X_1, Y_1) + (X_2, Y_2) = \begin{cases} 
(X, Y) & \text{if } X_1 \neq X_2 \\
(0, 0) & \text{otherwise}
\end{cases}
\]

\[
\lambda \equiv \frac{Y_2 - Y_1}{X_2 - X_1}
\]

\[
X \equiv \lambda^2 - X_1 - X_2
\]

\[
Y \equiv \lambda(X_1 - X) - Y_1
\]
We define a binary operation $+$ and scalar multiplication $\cdot$ with

\begin{equation}
(X_1, Y_1) + (X_2, Y_2) \equiv \begin{cases} 
(X, Y) & \text{if } Y_1 \neq 0 \\
(0, 0) & \text{otherwise}
\end{cases}
\end{equation}

\begin{align*}
\lambda & \equiv \frac{3X_1^2}{2Y_1} \\
X & \equiv \lambda^2 - 2X_1 \\
Y & \equiv \lambda(X_1 - X) - Y_1
\end{align*}

$(C_1, +)$ is known to form a group. We define scalar multiplication $\cdot$ with

\begin{equation}
n \cdot P \equiv (0, 0) + P + \ldots + P
\end{equation}

for a natural number $n$ and a point $P$ in $C_1$.

We define $P_1$ to be a point $(1, 2)$ on $C_1$. Let $G_1$ be the subgroup of $(C_1, +)$ generated by $P_1$. $G_1$ is known to be a cyclic group of order $q$. For a point $P$ in $G_1$, we define log$_P$($P$) to be the smallest natural number $n$ satisfying $n \cdot P = P$. log$_P$($P$) is at most $q - 1$.

Let $F_{p^2}$ be a field $F_p[i]/(i^2 + 1)$. We define a set $C_2$ with

\begin{equation}
C_2 \equiv \{(X, Y) \in F_{p^2} \times F_{p^2} \mid Y^2 = X^3 + 3(i + 9)^{-1} \} \cup \{(0, 0)\}
\end{equation}

We define a binary operation $+$ and scalar multiplication $\cdot$ with the same equations (241), (242) and (243). $(C_2, +)$ is also known to be a group. We define $P_2$ in $C_2$ with

\begin{equation}
P_2 \equiv (11559732032986387107991004021392285783925812861821192530917403151452391805634 \\
+1085704690902305713594457076723292481370756535975815086990519993285655852781, \\
4082367858634336813322034031453568316851327593401208105741076214120093531 \times i \\
+849565923123431417604973247489272438418190587263600148770280649306958101930)
\end{equation}

We define $G_2$ to be the subgroup of $(C_2, +)$ generated by $P_2$. $G_2$ is known to be the only cyclic group of order $q$ on $C_2$. For a point $P$ in $G_2$, we define log$_P$($P$) to be the smallest natural number $n$ satisfying $n \cdot P_2 = P$. With this definition, log$_P$($P$) is at most $q - 1$.

Let $G_T$ be the multiplicative abelian group underlying $F_{p^{12}}$. It is known that a non-degenerate bilinear map $e : G_1 \times G_2 \rightarrow G_T$ exists. This bilinear map is a type three pairing. There are several such bilinear maps, it does not matter which is chosen to be $e$. Let $P_T = e(P_1, P_2)$, $a$ be a set of $k$ points in $G_1$, and $b$ be a set of $k$ points in $G_2$. It follows from the definition of a pairing that the following are equivalent

\begin{equation}
\prod_{i=0}^{k} e(a_i, b_i) = P_T
\end{equation}

Thus the pairing operation provides a method to verify (246).

A 32 byte number $x \in P_{256}$ might and might not represent an element of $F_p$.

\begin{equation}
\delta_p(x) \equiv \begin{cases} 
x & \text{if } x < p \\
\emptyset & \text{otherwise}
\end{cases}
\end{equation}

A 64 byte data $x \in B_{512}$ might and might not represent an element of $G_1$.

\begin{equation}
\delta_1(x) \equiv \begin{cases} 
g_1 & \text{if } g_1 \in G_1 \\
\emptyset & \text{otherwise}
\end{cases}
\end{equation}

\begin{equation}
g_1 \equiv \begin{cases} 
(x, y) & \text{if } x \neq \emptyset \land y \neq \emptyset \\
\emptyset & \text{otherwise}
\end{cases}
\end{equation}

\begin{equation}
x \equiv \delta_p(x[0..31])
\end{equation}

\begin{equation}
y \equiv \delta_p(x[32..63])
\end{equation}
A 128 byte data $x \in \mathbb{B}_{1024}$ might and might not represent an element of $G_2$.

$$
\delta_2(x) \equiv \begin{cases} 
  g_2 & \text{if } g_2 \in G_2 \\
  \emptyset & \text{otherwise}
\end{cases}
$$

\[(253)\]

$$
g_2 \equiv \begin{cases} 
  ((x_0 + y_0), (x_1 + y_1)) & \text{if } x_0 \neq \emptyset \land y_0 \neq \emptyset \land x_1 \neq \emptyset \land y_1 \neq \emptyset \\
  \emptyset & \text{otherwise}
\end{cases}
$$

\[(254)\]

$$
x_0 \equiv \delta_\emptyset(x[0..31])
$$

\[(255)\]

$$
y_0 \equiv \delta_\emptyset(x[32..63])
$$

\[(256)\]

$$
x_1 \equiv \delta_\emptyset(x[64..95])
$$

\[(257)\]

$$
y_1 \equiv \delta_\emptyset(x[96..127])
$$

\[(258)\]

We define $\Xi_{\text{EIP-152}}$ as a precompiled contract which checks if (246) holds, for intended use in zkSNARK verification.

$$
\Xi_{\text{EIP-152}} \equiv \Xi_{\text{PKE}} \text{ except:}
$$

\[(259)\]

$$
\Xi_{\text{EIP-152}}(\sigma, g, A, I) = (\emptyset, 0, A, ()) \text{ if } F
$$

\[(260)\]

$$
k \equiv \left\lfloor \|Ia\| \right\rfloor
$$

\[(261)\]

$$
g_r \equiv 34000 + 45000
$$

\[(262)\]

$$
o[0..31] \equiv \begin{cases} 
  0x0000000000000000000000000000000000000000000000000000000000000001 & \text{if } v \land \neg F \\
  0x0000000000000000000000000000000000000000000000000000000000000000 & \text{if } \neg v \land \neg F
\end{cases}
$$

\[(263)\]

$$
v \equiv \log_{P_1}(a_1) \times \log_{P_1}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_1}(b_k) \equiv 1 \mod q
$$

\[(264)\]

$$
a_1 \equiv \delta_1(Ia[0..63])
$$

\[(265)\]

$$
b_1 \equiv \delta_2(Ia[64..191])
$$

\[(266)\]

$$
: 
$$

\[(267)\]

$$
a_k \equiv \delta_1(Ia[\|Ia\| - 192]..\|Ia\| - 129])
$$

\[(268)\]

$$
b_k \equiv \delta_2(Ia[\|Ia\| - 128]..\|Ia\| - 1])
$$

\[(269)\]

We define a precompiled contract for addition on $G_1$.

$$
\Xi_{\text{SH-ADD}} \equiv \Xi_{\text{PKE}} \text{ except:}
$$

\[(270)\]

$$
\Xi_{\text{SH-ADD}}(\sigma, g, A, I) = (\emptyset, 0, A, ()) \text{ if } x = \emptyset \lor y = \emptyset
$$

\[(271)\]

$$
g_r \equiv 150
$$

\[(272)\]

$$
o \equiv \delta_1^{-1}(x + y) \text{ where } + \text{ is the group operation in } G_1
$$

\[(273)\]

$$
x \equiv \delta_1 \left(Ia[0..63]\right)
$$

\[(274)\]

$$
y \equiv \delta_1 \left(Ia[64..127]\right)
$$

\[(275)\]

$$
Ia[x] \equiv \begin{cases} 
  Ia[x] & \text{if } x < \|Ia\| \\
  0 & \text{otherwise}
\end{cases}
$$

\[(276)\]

We define a precompiled contract for scalar multiplication on $G_1$, where $Ia$ is defined in (276).

$$
\Xi_{\text{SH-MUL}} \equiv \Xi_{\text{PKE}} \text{ except:}
$$

\[(277)\]

$$
\Xi_{\text{SH-MUL}}(\sigma, g, A, I) = (\emptyset, 0, A, ()) \text{ if } x = \emptyset
$$

\[(278)\]

$$
g_r \equiv 6000
$$

\[(279)\]

$$
o \equiv \delta_1^{-1}(n \cdot x) \text{ where } \cdot \text{ is the scalar multiplication in } G_1
$$

\[(280)\]

$$
x \equiv \delta_1 \left(Ia[0..63]\right)
$$

\[(281)\]

$$
n \equiv Ia[64..95]
$$

\[(282)\]

E.2. **BLAKE2 Precompiled Contract.** EIP-152 by [Hess et al., 2016](#) defines $\Xi_{\text{BLAKE2-F}}$ as a precompiled contract implementing the compression function $F$ used in the BLAKE2 cryptographic hashing algorithm. The $F$ compression
where

\[ \forall i : h_i \in \mathbb{B}_{64}, v_i \in 0...15 : m_i \in \mathbb{B}_{64}, t_{low} \in \mathbb{B}_{64}, t_{high} \in \mathbb{B}_{64}, f \in \mathbb{B}_8, \text{BE}_k \text{ is the k-byte big-endian representation—compare with (180)}: \]

\[ \text{BE}_k(x) \equiv (b_0, b_1, ..., b_{k-1}) : x = \sum_{n=0}^{k-1} b_n \cdot 256^{k-1-n} \]

and \( \text{LE}_k \) is the \( k \)-byte little-endian representation:

\[ \text{LE}_k(x) \equiv (b_0, b_1, ..., b_{k-1}) : x = \sum_{n=0}^{k-1} b_n \cdot 256^n \]

Appendix F. Signing Transactions

Transactions are signed using recoverable ECDSA signatures. This method utilises the SECP-256k1 curve as described by Courtois et al. [2014], and is implemented similarly to as described by Gura et al. [2004] on p. 9 of 15, para. 3.

It is assumed that the sender has a valid private key \( p_k \), which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range \([1, \text{secp256k1n} - 1]\).

We assume the existence of functions \texttt{ECDSAPUBKEY}, \texttt{ECDSASIGN} and \texttt{ECDSARECOVER}. These are formally defined in the literature, e.g. by Johnson et al. [2001].

\[ \text{ECDSAPUBKEY}(p_k \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \]

\[ \text{ECDSASIGN}(e \in \mathbb{B}_{32}, p_k \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \]

\[ \text{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \]

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 < \( 2^{256} \)), \( p_k \) is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range) and \( e \) is the hash of the transaction, \( h(T) \). It is assumed that \( v \) is the ‘recovery identifier’. The recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which \( r \) is the x-value; this value is in the range of \([0,3]\), however we declare the upper two possibilities, representing infinite values, invalid. The value 0 represents an even \( y \) value and 1 represents an odd \( y \) value.

We declare that an ECDSA signature is invalid unless all the following conditions are true:

\[ 0 < r < \text{secp256k1n} \]

\[ 0 < s < \text{secp256k1n} \div 2 + 1 \]

\[ m \in \{0, 1\} \]

where:

\[ \text{secp256k1n} = 115792089237316195423570985008686790752837564279074904382605163141518161494337 \]

Note that this restriction on \( s \) is more stringent than restriction 203 in the \( \text{Z} \) precompile; see EIP-2 by Buterin [2015] for more detail.

For a given private key, \( p_k \), the Ethereum address \( A(p_k) \) (a 160-bit value) to which it corresponds is defined as the rightmost 160-bits of the Keccak-256 hash of the corresponding ECDSA public key:

\[ A(p_k) = \mathbb{B}_{96..255}(\text{KEC(ECDSAPUBKEY}(p_k))) \]
The message hash, \( h(T) \), to be signed is the Keccak-256 hash of the transaction. Four different flavours of signing schemes are available:

\[
L_X(T) \equiv \begin{cases} 
(T_n, T_p, T_g, T_t, T_v, p) & \text{if } T_x = 0 \land T_w \in \{27, 28\} \\
(T_n, T_p, T_g, T_t, T_v, p, \beta, (, ()) & \text{if } T_x = 0 \land T_w \in \{2\beta + 35, 2\beta + 36\} \\
(T_c, T_n, T_p, T_g, T_t, T_v, p, p, A) & \text{if } T_x = 1 \\
(T_c, T_n, T_m, T_g, T_t, T_v, p, A) & \text{if } T_x = 2
\end{cases}
\]

where

\[
P \equiv \begin{cases} 
T_i & \text{if } T_t = \emptyset \\
T_d & \text{otherwise}
\end{cases}
\]

\[
h(T) \equiv \begin{cases} 
\text{KEC(RLP}(L_X(T))) & \text{if } T_x = 0 \\
\text{KEC}(T_x \cdot \text{RLP}(L_X(T))) & \text{otherwise}
\end{cases}
\]

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

\[
G(T, p_r) \equiv T \text{ except:}
\]

\[
(T_y, T_r, T_s) = \text{ECDSASIGN}(h(T), p_r)
\]

Reiterating from previously:

\[
T_r \equiv r
\]

\[
T_s \equiv s
\]

and \( T_y \) of legacy transactions is either \( 27 + T_y \) or \( 2\beta + 35 + T_y \).

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

\[
S(T) \equiv B_{96..255}(\text{KEC(ECDSSARECOVER}(h(T), v, T_t, T_s))
\]

\[
v \equiv \begin{cases} 
T_w - 27 & \text{if } T_x = 0 \land T_w \in \{27, 28\} \\
(T_w - 35) \mod 2 & \text{if } T_x = 0 \land T_w \in \{2\beta + 35, 2\beta + 36\} \\
T_y & \text{if } T_x = 1 \lor T_s = 2
\end{cases}
\]

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

\[
\forall T : \forall p_r : S(G(T, p_r)) \equiv A(p_r)
\]
## Appendix G. Fee Schedule

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G_{\text{zero}}$</td>
<td>0</td>
<td>Nothing paid for operations of the set $W_{\text{zero}}$.</td>
</tr>
<tr>
<td>$G_{\text{jumpdest}}$</td>
<td>1</td>
<td>Amount of gas to pay for a JUMPDEST operation.</td>
</tr>
<tr>
<td>$G_{\text{base}}$</td>
<td>2</td>
<td>Amount of gas to pay for operations of the set $W_{\text{base}}$.</td>
</tr>
<tr>
<td>$G_{\text{verylow}}$</td>
<td>3</td>
<td>Amount of gas to pay for operations of the set $W_{\text{verylow}}$.</td>
</tr>
<tr>
<td>$G_{\text{low}}$</td>
<td>5</td>
<td>Amount of gas to pay for operations of the set $W_{\text{low}}$.</td>
</tr>
<tr>
<td>$G_{\text{mid}}$</td>
<td>8</td>
<td>Amount of gas to pay for operations of the set $W_{\text{mid}}$.</td>
</tr>
<tr>
<td>$G_{\text{high}}$</td>
<td>10</td>
<td>Amount of gas to pay for operations of the set $W_{\text{high}}$.</td>
</tr>
<tr>
<td>$G_{\text{warmaccess}}$</td>
<td>100</td>
<td>Cost of a warm account or storage access.</td>
</tr>
<tr>
<td>$G_{\text{accesslistaddress}}$</td>
<td>2400</td>
<td>Cost of warming up an account with the access list.</td>
</tr>
<tr>
<td>$G_{\text{accessliststorage}}$</td>
<td>1900</td>
<td>Cost of warming up a storage with the access list.</td>
</tr>
<tr>
<td>$G_{\text{coldaccountaccess}}$</td>
<td>2600</td>
<td>Cost of a cold account access.</td>
</tr>
<tr>
<td>$G_{\text{coldload}}$</td>
<td>2100</td>
<td>Cost of a cold storage access.</td>
</tr>
<tr>
<td>$G_{\text{set}}$</td>
<td>20000</td>
<td>Paid for an SSTORE operation when the storage value is set to non-zero from zero.</td>
</tr>
<tr>
<td>$G_{\text{reset}}$</td>
<td>2900</td>
<td>Paid for an SSTORE operation when the storage value’s zeroness remains unchanged or is set to zero.</td>
</tr>
<tr>
<td>$R_{\text{clear}}$</td>
<td>4800</td>
<td>Refund given (added into refund counter) when the storage value is set to zero from non-zero. The refund amount is defined as $G_{\text{reset}} + G_{\text{accessliststorage}}$.</td>
</tr>
<tr>
<td>$G_{\text{selfdestruct}}$</td>
<td>5000</td>
<td>Amount of gas to pay for a SELFDESTRUCT operation.</td>
</tr>
<tr>
<td>$G_{\text{create}}$</td>
<td>32000</td>
<td>Paid for a CREATE operation.</td>
</tr>
<tr>
<td>$G_{\text{codedeposit}}$</td>
<td>200</td>
<td>Paid per byte for a CREATE operation to succeed in placing code into state.</td>
</tr>
<tr>
<td>$G_{\text{callvalue}}$</td>
<td>9000</td>
<td>Paid for a non-zero value transfer as part of the CALL operation.</td>
</tr>
<tr>
<td>$G_{\text{callstipend}}$</td>
<td>2300</td>
<td>A stipend for the called contract subtracted from $G_{\text{callvalue}}$ for a non-zero value transfer.</td>
</tr>
<tr>
<td>$G_{\text{newaccount}}$</td>
<td>25000</td>
<td>Paid for a CALL or SELFDESTRUCT operation which creates an account.</td>
</tr>
<tr>
<td>$G_{\text{exp}}$</td>
<td>10</td>
<td>Partial payment for an EXP operation.</td>
</tr>
<tr>
<td>$G_{\text{expbyte}}$</td>
<td>50</td>
<td>Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.</td>
</tr>
<tr>
<td>$G_{\text{memory}}$</td>
<td>3</td>
<td>Paid for every additional word when expanding memory.</td>
</tr>
<tr>
<td>$G_{\text{txcreate}}$</td>
<td>32000</td>
<td>Paid by all contract-creating transactions after the Homestead transition.</td>
</tr>
<tr>
<td>$G_{\text{txdatazero}}$</td>
<td>4</td>
<td>Paid for every zero byte of data or code for a transaction.</td>
</tr>
<tr>
<td>$G_{\text{txdatanonzero}}$</td>
<td>16</td>
<td>Paid for every non-zero byte of data or code for a transaction.</td>
</tr>
<tr>
<td>$G_{\text{transaction}}$</td>
<td>21000</td>
<td>Paid for every transaction.</td>
</tr>
<tr>
<td>$G_{\text{log}}$</td>
<td>375</td>
<td>Partial payment for a LOG operation.</td>
</tr>
<tr>
<td>$G_{\text{logdata}}$</td>
<td>8</td>
<td>Paid for each byte in a LOG operation’s data.</td>
</tr>
<tr>
<td>$G_{\text{logtopic}}$</td>
<td>375</td>
<td>Paid for each topic of a LOG operation.</td>
</tr>
<tr>
<td>$G_{\text{keccak256}}$</td>
<td>30</td>
<td>Paid for each KECCAK256 operation.</td>
</tr>
<tr>
<td>$G_{\text{keccak256word}}$</td>
<td>6</td>
<td>Paid for each word (rounded up) for input data to a KECCAK256 operation.</td>
</tr>
<tr>
<td>$G_{\text{copy}}$</td>
<td>3</td>
<td>Partial payment for *COPY operations, multiplied by words copied, rounded up.</td>
</tr>
<tr>
<td>$G_{\text{blockhash}}$</td>
<td>20</td>
<td>Payment for each BLOCKHASH operation.</td>
</tr>
</tbody>
</table>

## 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 leftmost 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:

\[(317)\]

\[C(\sigma, \mu, A, I) = C_{selfdestruct}(\sigma, \mu) - C_{mem}(\mu) + C_{store}(\sigma, \mu, A, I)\]

where:

\[(318)\]

\[w = \begin{cases} b_0[\mu_{pc}] & \mu_{pc} < ||b_0|| \\ \text{STOP} & \text{otherwise} \end{cases}\]

\[(319)\]

\[C_{mem}(a) = G_{memory} \cdot a + \left\lceil \frac{a^2}{512} \right\rceil\]

\[(320)\]

\[C_{access}(x, A) = \begin{cases} G_{warmaccess} & \text{if } x \in A_0 \\ G_{coldaccess} & \text{otherwise} \end{cases}\]

with $C_{call}$, $C_{selfdestruct}$, $C_{load}$, and $C_{store}$ as specified in the appropriate section below. We define the following subsets of instructions:

- $W_{zero} = \{\text{STOP, RETURN, REVERT}\}$
- $W_{base} = \{\text{ADDRESS, ORIGIN, CALLER, CALVALUE, CALDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, PREVRANDAO, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS, BASEFEE}\}$
- $W_{verylow} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR, CALLDATALOAD, MLOAD, MSLOAD, MSSTORE, PUSH*, DUP*, SWAP*, CALLDATACOPY, CODECOPY, RETURNDATACOPY\}$
- $W_{low} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEFFECT, SELFbalance\}$
- $W_{mid} = \{\text{ADDMOD, MULMOD, JUMP\}$
- $W_{high} = \{\text{JUMPI\}$
- $W_{copy} = \{\text{CALLDATACOPY, CODECOPY, RETURNDATACOPY\}$
- $W_{call} = \{\text{CALL, CALLCODE, DELEGATECALL, STATICCALL\}$
- $W_{extaccount} = \{\text{BALANCE, EXTCODESIZE, EXTCODEHASH\}$

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, $\mu$, 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. $\mu$ 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 $C_{\text{mem}}$ 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 704B of memory used, after which it costs substantially more.

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

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

Another useful function is “all but one 64th” function $L$ defined as:

\[
L(n) \equiv n - \lfloor n/64 \rfloor
\]

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 $(\sigma', \mu')$ such that:

\[
O(\sigma, \mu, A, I) \equiv (\sigma', \mu', A', I) \text{ with exceptions, as noted}
\]

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 $\alpha$, the additional items placed on the stack and $\delta$, the items removed from stack, as defined in section 9.
### 0s: Stop and Arithmetic Operations

All arithmetic is modulo $2^{256}$ unless otherwise noted. The zero-th power of zero $0^0$ is defined to be one.

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>STOP</td>
<td>0</td>
<td>0</td>
<td>Halts execution.</td>
</tr>
<tr>
<td>0x01</td>
<td>ADD</td>
<td>2</td>
<td>1</td>
<td>Addition operation. $\mu_s[0] \equiv \mu_s[0] + \mu_s[1]$</td>
</tr>
<tr>
<td>0x02</td>
<td>MUL</td>
<td>2</td>
<td>1</td>
<td>Multiplication operation. $\mu_s[0] \equiv \mu_s[0] \times \mu_s[1]$</td>
</tr>
<tr>
<td>0x03</td>
<td>SUB</td>
<td>2</td>
<td>1</td>
<td>Subtraction operation. $\mu_s[0] \equiv \mu_s[0] - \mu_s[1]$</td>
</tr>
<tr>
<td>0x04</td>
<td>DIV</td>
<td>2</td>
<td>1</td>
<td>Integer division operation. $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[1] = 0 \ \lfloor \mu_s[0] \div \mu_s[1] \rfloor &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x05</td>
<td>SDIV</td>
<td>2</td>
<td>1</td>
<td>Signed integer division operation (truncated). $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[1] = 0 \ -2^{255} &amp; \text{if } \mu_s[0] = -2^{255} \land \mu_s[1] = -1 \ \sgn(\mu_s[0]) \lfloor \mu_s[0] \div \mu_s[1] \rfloor &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x06</td>
<td>MOD</td>
<td>2</td>
<td>1</td>
<td>Modulo remainder operation. $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[1] = 0 \ \mu_s[0] \mod \mu_s[1] &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x07</td>
<td>SMOD</td>
<td>2</td>
<td>1</td>
<td>Signed modulo remainder operation. $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[1] = 0 \ \sgn(\mu_s[0])(\mu_s[0] \mod</td>
</tr>
<tr>
<td>0x08</td>
<td>ADDMOD</td>
<td>3</td>
<td>1</td>
<td>Modulo addition operation. $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[2] = 0 \ (\mu_s[0] + \mu_s[1]) \mod \mu_s[2] &amp; \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x09</td>
<td>MULMOD</td>
<td>3</td>
<td>1</td>
<td>Modulo multiplication operation. $\mu_s'[0] \equiv \begin{cases} 0 &amp; \text{if } \mu_s[2] = 0 \ (\mu_s[0] \times \mu_s[1]) \mod \mu_s[2] &amp; \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x0a</td>
<td>EXP</td>
<td>2</td>
<td>1</td>
<td>Exponential operation. $\mu_s'[0] \equiv \mu_s[0]^{\mu_s[1]}$</td>
</tr>
<tr>
<td>0x0b</td>
<td>SIGNEXTEND</td>
<td>2</td>
<td>1</td>
<td>Extend length of two’s complement signed integer. $\forall i \in [0..255]: \mu_s'[i], \equiv \begin{cases} \mu_s[1] &amp; \text{if } i \leq t \text{ where } t = 256 - 8(\mu_s[0] + 1) \ \mu_s[0] &amp; \text{otherwise} \end{cases}$ $\mu_s[x]_i$ gives the $i$th bit (counting from zero) of $\mu_s[x]$</td>
</tr>
</tbody>
</table>

Where all values are treated as two’s complement signed 256-bit integers. Note the overflow semantic when $-2^{255}$ is negated.
10s: Comparison & Bitwise Logic Operations

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>LT</td>
<td>2</td>
<td>1</td>
<td>Less-than comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] &lt; \mu_s[1] \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x11</td>
<td>GT</td>
<td>2</td>
<td>1</td>
<td>Greater-than comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] &gt; \mu_s[1] \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x12</td>
<td>SLT</td>
<td>2</td>
<td>1</td>
<td>Signed less-than comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] &lt; \mu_s[1] \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Where all values are treated as two’s complement signed 256-bit integers.</td>
</tr>
<tr>
<td>0x13</td>
<td>SGT</td>
<td>2</td>
<td>1</td>
<td>Signed greater-than comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] &gt; \mu_s[1] \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Where all values are treated as two’s complement signed 256-bit integers.</td>
</tr>
<tr>
<td>0x14</td>
<td>EQ</td>
<td>2</td>
<td>1</td>
<td>Equality comparison.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] = \mu_s[1] \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x15</td>
<td>ISZERO</td>
<td>1</td>
<td>1</td>
<td>Simple not operator.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \begin{cases} 1 &amp; \mu_s[0] = 0 \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x16</td>
<td>AND</td>
<td>2</td>
<td>1</td>
<td>Bitwise AND operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\forall i \in [0..255]: \mu'_s[i] \equiv \mu_s[i] \land \mu_s[1]$,</td>
</tr>
<tr>
<td>0x17</td>
<td>OR</td>
<td>2</td>
<td>1</td>
<td>Bitwise OR operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\forall i \in [0..255]: \mu'_s[i] \equiv \mu_s[i] \lor \mu_s[1]$,</td>
</tr>
<tr>
<td>0x18</td>
<td>XOR</td>
<td>2</td>
<td>1</td>
<td>Bitwise XOR operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\forall i \in [0..255]: \mu'_s[i] \equiv \mu_s[i] \oplus \mu_s[1]$,</td>
</tr>
<tr>
<td>0x19</td>
<td>NOT</td>
<td>1</td>
<td>1</td>
<td>Bitwise NOT operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\forall i \in [0..255]: \mu'_s[i] \equiv \begin{cases} 1 &amp; \mu_s[i] = 0 \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x1a</td>
<td>BYTE</td>
<td>2</td>
<td>1</td>
<td>Retrieve single byte from word.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\forall i \in [0..255]: \mu'<em>{s[i]} \equiv \begin{cases} \mu</em>{i}[1] &amp; \mu_s[0] &lt; 32 \ 0 &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>For the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).</td>
</tr>
<tr>
<td>0x1b</td>
<td>SHL</td>
<td>2</td>
<td>1</td>
<td>Left shift operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv (\mu_s[1] \times 2^{\mu_s[0]}) \mod 2^{256}$</td>
</tr>
<tr>
<td>0x1c</td>
<td>SHR</td>
<td>2</td>
<td>1</td>
<td>Logical right shift operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv [\mu_s[1] \div 2^{\mu_s[0]}]$</td>
</tr>
<tr>
<td>0x1d</td>
<td>SAR</td>
<td>2</td>
<td>1</td>
<td>Arithmetic (signed) right shift operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv [\mu_s[1] \div 2^{\mu_s[0]}]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Where $\mu'_s[0]$ and $\mu_s[1]$ are treated as two’s complement signed 256-bit integers, while $\mu_s[0]$ is treated as unsigned.</td>
</tr>
</tbody>
</table>

20s: KECCAK256

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x20</td>
<td>KECCAK256</td>
<td>2</td>
<td>1</td>
<td>Compute Keccak-256 hash.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv \text{KEC}(\mu_s[0], \mu_s[1])$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_s[0] \equiv M(\mu_s[0], \mu_s[1])$</td>
</tr>
</tbody>
</table>
## 30s: Environmental Information

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x30</td>
<td>ADDRESS</td>
<td>0</td>
<td>1</td>
<td>Get address of currently executing account. $\mu_s[0] \equiv I_s$</td>
</tr>
<tr>
<td>0x31</td>
<td>BALANCE</td>
<td>1</td>
<td>1</td>
<td>Get balance of the given account. $\mu_s[0] \equiv \left{ \begin{array}{ll} \sigma \mu_s[0] \bmod 2^{160} &amp; \text{if } \sigma \mu_s[0] \bmod 2^{160} \neq \emptyset \ 0 &amp; \text{otherwise} \end{array} \right.$ $A'_s \equiv A_s \cup { \mu_s[0] \bmod 2^{160} }$</td>
</tr>
<tr>
<td>0x32</td>
<td>ORIGIN</td>
<td>0</td>
<td>1</td>
<td>Get execution origination address. $\mu_s[0] \equiv I_s$ This is the sender of original transaction; it is never an account with non-empty associated code.</td>
</tr>
<tr>
<td>0x33</td>
<td>CALLER</td>
<td>0</td>
<td>1</td>
<td>Get caller address. $\mu_s[0] \equiv I_s$ This is the address of the account that is directly responsible for this execution.</td>
</tr>
<tr>
<td>0x34</td>
<td>CALVALUE</td>
<td>0</td>
<td>1</td>
<td>Get deposited value by the instruction/transaction responsible for this execution. $\mu_s[0] \equiv I_s$</td>
</tr>
<tr>
<td>0x35</td>
<td>CALLDATASIZE</td>
<td>0</td>
<td>1</td>
<td>Get size of input data in current environment. $\mu_s[0] \equiv |I_s|$ This pertains to the input data passed with the message call instruction or transaction.</td>
</tr>
<tr>
<td>0x36</td>
<td>CALLDATACOPY</td>
<td>3</td>
<td>0</td>
<td>Copy input data in current environment to memory. $\forall i \in {0 \ldots \mu_s[2] - 1} : \mu'_s[\mu_s[0] + i] \equiv \left{ \begin{array}{ll} I_s[\mu_s[1] + i] &amp; \text{if } \mu_s[1] + i &lt; |I_s| \ 0 &amp; \text{otherwise} \end{array} \right.$ The additions in $\mu_s[1] + i$ are not subject to the $2^{256}$ modulo. $\mu'_s \equiv M(\mu_s[0], \mu_s[2])$ This pertains to the input data passed with the message call instruction or transaction.</td>
</tr>
<tr>
<td>0x37</td>
<td>CODESIZE</td>
<td>0</td>
<td>1</td>
<td>Get size of code running in current environment. $\mu_s[0] \equiv |I_s|$</td>
</tr>
<tr>
<td>0x38</td>
<td>CODECOPY</td>
<td>3</td>
<td>0</td>
<td>Copy code running in current environment to memory. $\forall i \in {0 \ldots \mu_s[2] - 1} : \mu'_s[\mu_s[0] + i] \equiv \left{ \begin{array}{ll} I_s[\mu_s[1] + i] &amp; \text{if } \mu_s[1] + i &lt; |I_s| \ \text{STOP} &amp; \text{otherwise} \end{array} \right.$ The additions in $\mu_s[1] + i$ are not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x39</td>
<td>GASPRICE</td>
<td>0</td>
<td>1</td>
<td>Get price of gas in current environment. Note that as of the London hard fork, this value no longer represents what is received by the validator, but rather just what is paid by the sender. $\mu_s[0] \equiv I_p$</td>
</tr>
<tr>
<td>0x3a</td>
<td>EXTCODESIZE</td>
<td>1</td>
<td>1</td>
<td>Get size of an account’s code. $\mu_s[0] \equiv \left{ \begin{array}{ll} |b| &amp; \text{if } \sigma \mu_s[0] \bmod 2^{160} \neq \emptyset \ 0 &amp; \text{otherwise} \end{array} \right.$ where $KEC(b) \equiv \sigma \mu_s[0] \bmod 2^{160}$ $A'_s \equiv A_s \cup { \mu_s[0] \bmod 2^{160} }$</td>
</tr>
<tr>
<td>0x3b</td>
<td>EXTCODECOPY</td>
<td>4</td>
<td>0</td>
<td>Copy an account’s code to memory. $\forall i \in {0 \ldots \mu_s[3] - 1} : \mu'_s[\mu_s[0] + i] \equiv \left{ \begin{array}{ll} \sigma b[\mu_s[2] + i] &amp; \text{if } \mu_s[2] + i &lt; |b| \ \text{STOP} &amp; \text{otherwise} \end{array} \right.$ where $KEC(b) \equiv \sigma \mu_s[0] \bmod 2^{160}$. We assume $b \equiv ()$ if $\sigma \mu_s[0] \bmod 2^{160} = \emptyset$. $\mu'_s \equiv M(\mu_s[0], \mu_s[3])$ The additions in $\mu_s[2] + i$ are not subject to the $2^{256}$ modulo. $A'_s \equiv A_s \cup { \mu_s[0] \bmod 2^{160} }$</td>
</tr>
</tbody>
</table>
ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

PARIS VERSION

0x3d  RETURNDATASIZE  0 1 Get size of output data from the previous call from the current environment.

\[ \mu_s'[0] \equiv ||\mu_o|| \]

0x3e  RETURNDATACOPY  3 0 Copy output data from the previous call to memory.

\[ \forall i \in \{0 \ldots \mu_s[2] - 1\} : \mu_m'[\mu_s[0] + i] \equiv \begin{cases} \mu_o[\mu_s[1] + i] & \text{if } \mu_s[1] + i < ||\mu_o|| \\ 0 & \text{otherwise} \end{cases} \]

The additions in \( \mu_s[1] + i \) are not subject to the \( 2^{256} \) modulo.

\[ \mu_i' \equiv M(\mu_i, \mu_s[0], \mu_s[2]) \]

0x3f  EXTCODEHASH  1 1 Get hash of an account’s code.

\[ \mu_s'[0] \equiv \begin{cases} 0 & \text{if } \text{DEAD}(\sigma, \mu_s[0]) \text{ mod } 2^{160} \\ \sigma | \mu_s[0] \text{ mod } 2^{160} & \text{otherwise} \end{cases} \]

\[ A_n' \equiv A_n \cup \{\mu_s[0] \text{ mod } 2^{160}\} \]

40s: Block Information

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>( \delta )</th>
<th>( \alpha )</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x40</td>
<td>BLOCKHASH</td>
<td>1 1</td>
<td></td>
<td>Get the hash of one of the 256 most recent complete blocks.</td>
</tr>
</tbody>
</table>

\[ \mu_s'[0] \equiv P(H_{f_e}, \mu_s[0], 0) \]

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 or equal to the current block number or more than 256 blocks behind the current block.

\[ P(h, n, a) \equiv \begin{cases} 0 & \text{if } n > H_i \lor a = 256 \lor h = 0 \\ h & \text{if } n = H_i \\ P(H_{f_e}, n, a + 1) & \text{otherwise} \end{cases} \]

and we assert the header \( H \) can be determined from its hash \( h \) unless \( h \) is zero (as is the case for the parent hash of the genesis block).

| 0x41  | COINBASE  | 0 1         |             | Get the current block’s beneficiary address. |

\[ \mu_s'[0] \equiv I_{f_e} \]

| 0x42  | TIMESTAMP | 0 1         |             | Get the current block’s timestamp. |

\[ \mu_s'[0] \equiv I_{f_s} \]

| 0x43  | NUMBER    | 0 1         |             | Get the current block’s number. |

\[ \mu_s'[0] \equiv I_{f_i} \]

| 0x44  | PREVRANDAO| 0 1         |             | Get the latest RANDAO mix of the post beacon state of the previous block. |

\[ \mu_s'[0] \equiv I_{f_a} \]

| 0x45  | GASLIMIT  | 0 1         |             | Get the current block’s gas limit. |

\[ \mu_s'[0] \equiv I_{f_l} \]

| 0x46  | CHAINID   | 0 1         |             | Get the chain ID. |

\[ \mu_s'[0] \equiv I_{f_l} \]

| 0x47  | SELFBALANCE| 0 1        |             | Get balance of currently executing account. |

\[ \mu_s'[0] \equiv \sigma | I_{a}\] |

| 0x48  | BASEFEE   | 0 1         |             | Get the current block’s base fee. |

\[ \mu_s'[0] \equiv I_{f_l} \]
50s: Stack, Memory, Storage and Flow Operations

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x50</td>
<td>POP</td>
<td>1</td>
<td>1</td>
<td>Remove item from stack.</td>
</tr>
<tr>
<td>0x51</td>
<td>MLOAD</td>
<td>1</td>
<td>1</td>
<td>Load word from memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'[0] \equiv \mu_m[\mu_s[0] \ldots (\mu_s[0] + 31)]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_i \equiv \max(\mu_i, (\mu_s[0] + 32) \div 32)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The addition in the calculation of $\mu'_i$ is not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x52</td>
<td>MSTORE</td>
<td>2</td>
<td>0</td>
<td>Save word to memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'[m][\mu_s[0] \ldots (\mu_s[0] + 31)] \equiv \mu_s[1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_i \equiv \max(\mu_i, (\mu_s[0] + 32) \div 32)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The addition in the calculation of $\mu'_i$ is not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x53</td>
<td>MSTOREs</td>
<td>2</td>
<td>0</td>
<td>Save byte to memory.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'[m][\mu_s[0]] \equiv (\mu_s[1] \mod 256)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\mu'_i \equiv \max(\mu_i, (\mu_s[0] + 1) \div 32)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The addition in the calculation of $\mu'_i$ is not subject to the $2^{256}$ modulo.</td>
</tr>
<tr>
<td>0x54</td>
<td>SLOAD</td>
<td>1</td>
<td>1</td>
<td>Load word from storage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\sigma'[I_s][\mu_s[0]] \equiv \mu_s[1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$A'_K \equiv A_K \cup { (I_s, \mu_s[0]) }$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_{\text{SLOAD}}(\sigma, \mu, A, I) \equiv$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\begin{cases} G_{\text{warmaccess}} &amp; \text{if } (I_s, \mu_s[0]) \in A_K \ G_{\text{coldload}} &amp; \text{otherwise} \end{cases}$</td>
</tr>
<tr>
<td>0x55</td>
<td>SSTORE</td>
<td>2</td>
<td>0</td>
<td>Save word to storage.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\sigma'[I_s][\mu_s[0]] \equiv \mu_s[1]$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$A'_K \equiv A_K \cup { (I_s, \mu_s[0]) }$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$C_{\text{SSTORE}}(\sigma, \mu, A, I) \equiv$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$\begin{cases} 0 &amp; \text{if } (I_s, \mu_s[0]) \in A_K \ G_{\text{coldload}} &amp; \text{otherwise} \end{cases}$</td>
</tr>
</tbody>
</table>
|       |          |   |   | $+ \left\{ \begin{array}{ll} G_{\text{warmaccess}} & \text{if } v = v' \land v_0 = v \\ G_{\text{reset}} & \text{if } v \neq v' \land v_0 = v \\
|       |          |   |   | & \quad \land v_0 = v \\ G_{\text{reset}} & \text{if } v \neq v' \land v_0 = v \\
|       |          |   |   | & \quad \land v_0 = v \\ R_{\text{reset}} & \text{if } v \neq v' \land v_0 = v \\
|       |          |   |   | & \quad \land v_0 = v \end{array} \right.$ |
|       |          |   |   | $A'_s \equiv A_s + \left\{ \begin{array}{ll} R_{\text{reset}} & \text{if } v_0 \neq 0 \\
|       |          |   |   | & \quad \land v = 0 \\ R_{\text{reset}} & \text{if } v_0 \neq 0 \\
|       |          |   |   | & \quad \land v' = 0 \\ 0 & \text{otherwise} \end{array} \right.$ |
|       |          |   |   | $R_{\text{reset}} \equiv \left\{ \begin{array}{ll} -R_{\text{reset}} & \text{if } v_0 \neq 0 \\
|       |          |   |   | & \quad \land v = 0 \\ 0 & \text{otherwise} \end{array} \right.$ |
|       |          |   |   | $R_{\text{reset}} \equiv \left\{ \begin{array}{ll} -R_{\text{reset}} & \text{if } v_0 \neq 0 \\
|       |          |   |   | & \quad \land v' = 0 \\ 0 & \text{otherwise} \end{array} \right.$ |
| 0x56  | JUMP     | 1 | 0 | Alter the program counter. |
|       |          |   |   | $J_{\text{JUMP}}(\mu) \equiv \mu'_s[0]$ |
|       |          |   |   | This has the effect of writing said value to $\mu_{pc}$. See section 34. |
| 0x57  | JUMPI    | 2 | 0 | Conditionally alter the program counter. |
|       |          |   |   | $J_{\text{JUMPI}}(\mu) \equiv$ |
|       |          |   |   | $\begin{cases} \mu'_s[0] & \text{if } \mu_s[1] \neq 0 \\ \mu_{pc} + 1 & \text{otherwise} \end{cases}$ |
|       |          |   |   | This has the effect of writing said value to $\mu_{pc}$. See section 34. |
| 0x58  | PC       | 0 | 1 | Get the value of the program counter prior to the increment corresponding to this instruction. |
|       |          |   |   | $\mu'_s[0] \equiv \mu_{pc}$ |
### 60s & 70s: Push Operations

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x60  | PUSH1    | 0 | 1 | Place 1 byte item on stack.  
\[
\mu_s[0] = c(\mu_{pc} + 1)
\]
\[
where \quad c(x) = \begin{cases} I_b[x] & x < \|I_b\| \\
0 & \text{otherwise}
\end{cases}
\]

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).

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x61  | PUSH2    | 0 | 1 | Place 2-byte item on stack.  
\[
\mu_s[0] = c((\mu_{pc} + 1) \ldots (\mu_{pc} + 2))
\]

with \(c(x) = (c(x_0), \ldots, c(x_{\|x\| - 1}))\) with \(c\) as defined as above. 
The bytes are right-aligned (takes the lowest significant place in big endian).

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x7f  | PUSH32   | 0 | 1 | Place 32-byte (full word) item on stack.  
\[
\mu_s[0] = c((\mu_{pc} + 1) \ldots (\mu_{pc} + 32))
\]

where \(c\) is defined as above. 
The bytes are right-aligned (takes the lowest significant place in big endian).

### 80s: Duplication Operations

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x80  | DUP1     | 1 | 2 | Duplicate 1st stack item.  
\[
\mu_s[0] = \mu_s[0]
\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x81  | DUP2     | 2 | 3 | Duplicate 2nd stack item.  
\[
\mu_s[0] = \mu_s[1]
\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x8f  | DUP16    | 16 | 17 | Duplicate 16th stack item.  
\[
\mu_s[0] = \mu_s[15]
\]

### 90s: Exchange Operations

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x90  | SWAP1    | 2 | 2 | Exchange 1st and 2nd stack items.  
\[
\mu_s[0] = \mu_s[1] \\
\mu_s[1] = \mu_s[0]
\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x91  | SWAP2    | 3 | 3 | Exchange 1st and 3rd stack items.  
\[
\mu_s[0] = \mu_s[2] \\
\mu_s[2] = \mu_s[0]
\]

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0x9f  | SWAP16   | 17 | 17 | Exchange 1st and 17th stack items.  
\[
\mu_s[0] = \mu_s[16] \\
\mu_s[16] = \mu_s[0]
\]
**a0s: Logging Operations**

For all logging operations, the state change is to append an additional log entry on to the substate’s log series:

\[ A'_1 \equiv A_1 \cdot (I, t, \mu_m[\mu_s[0] \ldots (\mu_s[0] + \mu_s[1] - 1)]) \]

and to update the memory consumption counter:

\[ \mu'_i \equiv M(\mu_i, \mu_s[0], \mu_s[1]) \]

The entry’s topic series, t, differs accordingly:

<table>
<thead>
<tr>
<th>Value</th>
<th>Mnemonic</th>
<th>δ</th>
<th>α</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xa0</td>
<td>LOG0</td>
<td>2</td>
<td>0</td>
<td>Append log record with no topics.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( t \equiv () )</td>
</tr>
<tr>
<td>0xa1</td>
<td>LOG1</td>
<td>3</td>
<td>0</td>
<td>Append log record with one topic.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( t \equiv (\mu_s[2]) )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>0xa4</td>
<td>LOG4</td>
<td>6</td>
<td>0</td>
<td>Append log record with four topics.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( t \equiv (\mu_s[2], \mu_s[3], \mu_s[4], \mu_s[5]) )</td>
</tr>
</tbody>
</table>
Value Mnemonic δ α Description
0xf0 CREATE 3 1 Create a new account with associated code.
(σ’, g’, A’, z, o) ≡
(σ, L(μs), A, 0, (i)) if μs[0] ≤ σ[Is]n ∧ Is < 1024
otherwise
σ∗ ≡ σ except σ[Is]n = σ[Is]n + 1
μs[0] ≡ μs − L[μs] + g’
μs[0] ≡ 0 if z = 1
otherwise

0xf1 CALL 7 1 Message-call into an account.
(σ’, g’, A’, x, o) ≡
(σ, CCALLGAS(σ, μ, A), A, 0, (i)) if μs[2] ≤ σ[Is]n ∧ Is < 1024
otherwise
n ≡ min({μs[0], |o|})
μs[0] = o
μs[6] ≡ μs − CCALLGAS(σ, μ, A) + g’
μs[0] ≡ x
A’ ≡ A except A’ ≡ A ∪ {t}
t ≡ μs[1] mod 2160
μs[2] ≡ M(μs, μs[3], μs[4], μs[5], μs[6]),
where x = 0 if the code execution for this operation failed; or if
μs[2] > σ[Is]n (not enough funds) or Is = 1024 (call depth limit reached); x = 1
otherwise.
Thus the operand order is: gas, to, value, in offset, in size, out offset, out size.
CCALL(σ, μ, A) ≡ CGASCAP(σ, μ, A) + CEXTRA(σ, μ, A)
CCALLGAS(σ, μ, A) ≡ CGASCAP(σ, μ, A) + Ccallaspend if μs[2] ≠ 0
CGASCAP(σ, μ, A) ≡ min{L[μs] − CEXTRA(σ, μ, A), μs[0]} if μs ≥ CEXTRA(σ, μ, A)
otherwise
CEXTRA(σ, μ, A) ≡ Caccess(t, A) + CFEXPER(μ) + CNEW(σ, μ)
CFEXPER(μ) ≡ Gcalvalue if μs[2] ≠ 0
0 otherwise
CNEW(σ, μ) ≡ Gnewaccount if DEAD(σ, t) ∧ μs[2] ≠ 0
0 otherwise

0xf2 CALLCODE 7 1 Message-call into this account with an alternative account’s code.
Exactly equivalent to CALL except:
(σ’, g’, A’, x, o) ≡
(σ, L(μs), A, 0, (i)) if μs[2] ≤ σ[Is]n ∧ Is < 1024
otherwise
Note the change in the fourth parameter to the call from the 2nd stack value μs[1] (as in CALL) to the present address Is. This means that the recipient is in
the same account as at present, simply that the code is overwritten.

0xf3 RETURN 2 0 Halt execution returning output data.
Hreturns(μ) ≡ μm[μs[0]⋯(μs[0] + μs[1] − 1)]
This has the effect of halting the execution at this point with output defined.
See section 28:
μs[1] ≡ M(μs[1], μs[0], μs[1])
0xf4  DELEGATECALL  6  1 Message-call into this account with an alternative account’s code, but persisting the current values for sender and value. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is \( \mu_2 \). As a result, \( \mu_3, \mu_4, \mu_5 \) and \( \mu_6 \) in the definition of CALL should respectively be replaced with \( \mu_3, \mu_5, \mu_4 \) and \( \mu_5 \). Otherwise it is equivalent to CALL except:

\[
\begin{cases}
\Theta(\sigma, A^*, I_s, I_o, I_t, t, \mathcal{C}_{\text{CALLGAS}}(\sigma, \mu, A)), & \text{if } I_s < 1024 \\
(\sigma, \mathcal{C}_{\text{CALLGAS}}(\sigma, \mu, A), 0, 0, ()) & \text{otherwise}
\end{cases}
\]

Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call \( \Theta \).

This means that the recipient is in fact the same account as at present, simply that the code is overwritten and the context is almost entirely identical.

0xf5  CREATE2  4  1 Create a new account with associated code. Exactly equivalent to CREATE except:

The salt \( \zeta \equiv \mu_3 \).

0xfa  STATICCALL  6  1 Static message-call into an account. Exactly equivalent to CALL except:

The argument \( \mu_2 \) is replaced with 0. The deeper argument \( \mu_3, \mu_4, \mu_5 \) and \( \mu_6 \) are respectively replaced with \( \mu_3, \mu_5, \mu_4 \) and \( \mu_5 \). The last argument of \( \Theta \) is \( \perp \).

0xfd  REVERT  2  0 Halt execution reverting state changes but returning data and remaining gas.

\[ H_{\text{RETURN}}(\mu) \equiv \mu_m[\mu_0] \ldots (\mu_0[0] + \mu_1[1] - 1) \]

The effect of this operation is described in (147).

For the gas calculation, we use the memory expansion function, \( \mu_i \equiv M(\mu_0, \mu_1[1]) \).

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

\[
\begin{align*}
A'_s & \equiv A_s \cup \{L_s\} \\
A'_n & \equiv A_n \cup \{r\}
\end{align*}
\]

\[
\begin{cases}
\sigma & \text{if } \sigma[r] = \emptyset \land \sigma[L_s] = 0 \\
(\sigma[r], 0, \sigma[r]) & \text{if } r \neq I_s
\end{cases}
\]

where \( r = \mu_0[0] \mod 2^{160} \)

\[
\sigma'[L_s] = 0
\]

\[
\begin{cases}
G_{\text{SELFDESTRUCT}}(\sigma, \mu) \equiv G_{\text{selfdestruct}} + G_{\text{coldaccountaccess}} & \text{if } r \in A_s \setminus A'_s \\
G_{\text{coldaccountaccess}} & \text{otherwise}
\end{cases} + \begin{cases}
G_{\text{newaccount}} & \text{if } \text{DEAD}(\sigma, r) \land \sigma[L_s] \neq 0 \\
0 & \text{otherwise}
\end{cases}
\]

### Appendix I. Genesis Block

The genesis block is 15 items, and is specified thus:

\[
\{ (0_{256}, \text{KEC}(\text{RLP}(())), 0_{160}, \text{stateRoot}, 0, 0, 0_{2048}, 2^{34}, 0, 0, 3141592, \text{time}, 0, 0_{256}, \text{KEC}((42))), ((), ()) \}
\]

Where \( 0_{256} \) refers to the parent hash, a 256-bit hash which is all zeros; \( 0_{160} \) refers to the beneficiary address, a 160-bit hash which is all zeros; \( 0_{2048} \) refers to the log bloom, 2048-bit of all zeros; \( 2^{34} \) refers to the difficulty; the transaction trie root, receipt trie root, gas used, block number and extradata are both 0, being equivalent to the empty byte array. The sequences of both ommer and transactions are empty and represented by (). \( \text{KEC}((42)) \) refers to the Keccak-256 hash of a byte array of length one whose first and only byte is of value 42, used for the nonce. \( \text{KEC}(\text{RLP}(())) \) value refers to the hash of the ommer list in RLP, both empty lists.

The proof-of-concept series include a development premine, making the state root hash some value stateRoot. Also time will be set to the initial timestamp of the genesis block. The latest documentation should be consulted for those values.
Appendix J. Ethash

J.1. Deprecation. This section is kept for historical purposes, but the Ethash algorithm is no longer used for consensus as of the Paris hard fork.

J.2. Definitions. We employ the following definitions:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_{\text{wordbytes}}$</td>
<td>4</td>
<td>Bytes in word.</td>
</tr>
<tr>
<td>$J_{\text{datasetinit}}$</td>
<td>$2^{30}$</td>
<td>Bytes in dataset at genesis.</td>
</tr>
<tr>
<td>$J_{\text{datasetgrowth}}$</td>
<td>$2^{23}$</td>
<td>Dataset growth per epoch.</td>
</tr>
<tr>
<td>$J_{\text{cacheinit}}$</td>
<td>$2^{24}$</td>
<td>Bytes in cache at genesis.</td>
</tr>
<tr>
<td>$J_{\text{cachegrowth}}$</td>
<td>$2^{17}$</td>
<td>Cache growth per epoch.</td>
</tr>
<tr>
<td>$J_{\text{epoch}}$</td>
<td>30000</td>
<td>Blocks per epoch.</td>
</tr>
<tr>
<td>$J_{\text{mixbytes}}$</td>
<td>128</td>
<td>Mix length in bytes.</td>
</tr>
<tr>
<td>$J_{\text{hashbytes}}$</td>
<td>64</td>
<td>Hash length in bytes.</td>
</tr>
<tr>
<td>$J_{\text{parents}}$</td>
<td>256</td>
<td>Number of parents of each dataset element.</td>
</tr>
<tr>
<td>$J_{\text{cacherounds}}$</td>
<td>3</td>
<td>Number of rounds in cache production.</td>
</tr>
<tr>
<td>$J_{\text{accesses}}$</td>
<td>64</td>
<td>Number of accesses in hashimoto loop.</td>
</tr>
</tbody>
</table>

J.3. Size of dataset and cache. The size for Ethash’s cache $c \in \mathbb{B}$ and dataset $d \in \mathbb{B}$ depend on the epoch, which in turn depends on the block number.

\[
E_{\text{epoch}}(H_i) = \left\lfloor\frac{H_i}{J_{\text{epoch}}}\right\rfloor
\]

The size of the dataset growth by $J_{\text{datasetgrowth}}$ bytes, and the size of the cache by $J_{\text{cachegrowth}}$ bytes, every epoch. In order to avoid regularity leading to cyclic behavior, the size must be a prime number. Therefore the size is reduced by a multiple of $J_{\text{mixbytes}}$, for the dataset, and $J_{\text{hashbytes}}$ for the cache. Let $d_{\text{size}} = \|d\|$ be the size of the dataset. Which is calculated using

\[
d_{\text{size}} = E_{\text{prime}}(J_{\text{datasetinit}} + J_{\text{datasetgrowth}} \cdot E_{\text{epoch}} - J_{\text{mixbytes}} \cdot J_{\text{mixbytes}})
\]

The size of the cache, $c_{\text{size}}$, is calculated using

\[
c_{\text{size}} = E_{\text{prime}}(J_{\text{cacheinit}} + J_{\text{cachegrowth}} \cdot E_{\text{epoch}} - J_{\text{hashbytes}} \cdot J_{\text{hashbytes}})
\]

\[
E_{\text{prime}}(x, y) = \begin{cases} x & \text{if } x/y \in \mathbb{N} \\ E_{\text{prime}}(x - 2 \cdot y, y) & \text{otherwise} \end{cases}
\]

J.4. Dataset generation. In order to generate the dataset we need the cache $c$, which is an array of bytes. It depends on the cache size $c_{\text{size}}$ and the seed hash $s \in \mathbb{B}^{32}$.

J.4.1. Seed hash. The seed hash is different for every epoch. For the first epoch it is the Keccak-256 hash of a series of 32 bytes of zeros. For every other epoch it is always the Keccak-256 hash of the previous seed hash:

\[
s = C_{\text{seedhash}}(H_i)
\]

\[
C_{\text{seedhash}}(H_i) = \begin{cases} 0_{32} & \text{if } E_{\text{epoch}}(H_i) = 0 \\ \text{KEC}(C_{\text{seedhash}}(H_i - J_{\text{epoch}})) & \text{otherwise} \end{cases}
\]

With $0_{32}$ being 32 bytes of zeros.

J.4.2. Cache. The cache production process involves using the seed hash to first sequentially filling up $c_{\text{size}}$ bytes of memory, then performing $J_{\text{cacherounds}}$ passes of the RandMemoHash algorithm created by Lerner [2014]. The initial cache $c'$, being an array of arrays of single bytes, will be constructed as follows.

We define the array $c_i$, consisting of 64 single bytes, as the $i$th element of the initial cache:

\[
c_i = \begin{cases} \text{KEC512}(s) & \text{if } i = 0 \\ \text{KEC512}(c_{i-1}) & \text{otherwise} \end{cases}
\]

Therefore $c'$ can be defined as

\[
c'[i] = c_i \quad \forall \ i < n
\]

\[
n = \left\lfloor\frac{c_{\text{size}}}{J_{\text{hashbytes}}}\right\rfloor
\]

The cache is calculated by performing $J_{\text{cacherounds}}$ rounds of the RandMemoHash algorithm to the initial cache $c'$:

\[
c = E_{\text{cacherounds}}(c', J_{\text{cacherounds}})
\]
Where a single round modifies each subset of the cache as follows:

\[
E_{\text{mixdataset}}(d, m, s, i) = \begin{cases} 
E_{\text{datasetitem}}(c, i) & \text{if } i = 0 \\
E_{\text{mix}}(m, c, i, p + 1) & \text{if } p < J_{\text{parents}} - 2 \\
E_{\text{mix}}(m, c, i, p + 1) & \text{otherwise}
\end{cases}
\]

\[
E_{\text{mix}}(m, c, i, p) = \begin{cases}
\text{KEC12}(c[i \mod \text{Cache}] \oplus i) & \text{if } p = 0 \\
E_{\text{datasetitem}}(c, i) = E_{\text{parents}}(c, i, -1, \emptyset) & \text{otherwise}
\end{cases}
\]

J.5. Proof-of-work function. Essentially, we maintain a “mix” \( J_{\text{mixbytes}} \) bytes wide, and repeatedly sequentially fetch \( J_{\text{mixbytes}} \) bytes from the full dataset and use the \( E_{\text{mix}} \) function to combine it with the mix. \( J_{\text{mixbytes}} \) bytes of sequential access are used so that each round of the algorithm always fetches a full page from RAM, minimizing translation lookaside buffer misses which ASICs would theoretically be able to avoid.

The output of this algorithm is below the desired target, then the nonce is valid. Note that the extra application of KECC at the end ensures that there exists an intermediate nonce which can be provided to prove that at least a small amount of work was done; this quick outer PoW verification can be used for anti-DDoS purposes. It also serves to provide statistical assurance that the result is an unbiased, 256 bit number.

The PoW-function returns an array with the compressed mix as its first item and the Keccak-256 hash of the concatenation of the compressed mix with the seed hash as the second item:

\[
\text{PoW}(H_{\text{mix}}, H_{\text{n}}, d) = \{ m_{\text{c}}(\text{KEC(RLP}(L_{\text{HT}}(H_{\text{mix}}))), H_{\text{n}}, d), \text{KEC}(s_{\text{h}}(\text{KEC(RLP}(L_{\text{HT}}(H_{\text{mix}}))), H_{\text{n}}) + m_{\text{c}}(\text{KEC(RLP}(L_{\text{HT}}(H_{\text{mix}}))), H_{\text{n}}, d)) \}
\]

With \( H_{\text{mix}} \) being the hash of the header without the nonce. The compressed mix \( m_{\text{c}} \) is obtained as follows:

\[
m_{\text{c}}(h, n, d) = E_{\text{compress}}(E_{\text{accesses}}(d, \sum_{i=0}^{n_{\text{mix}}} s_{\text{h}}(h, n), s_{\text{h}}(h, n), -1, -4))
\]

The seed hash being:

\[
s_{\text{h}}(h, n) = \text{KEC12}(h + E_{\text{revert}}(n))
\]

\[
E_{\text{revert}}(n) \text{ returns the reverted bytes sequence of the nonce } n:
\]

\[
E_{\text{revert}}(n)[i] = n[|n| - i]
\]

We note that the “+”-operator between two byte sequences results in the concatenation of both sequences.

The dataset \( d \) is obtained as described in section J.4.3.

The number of replicated sequences in the mix is:

\[
n_{\text{mix}} = \left\lfloor \frac{J_{\text{mixbytes}}}{J_{\text{hashbytes}}} \right\rfloor
\]

In order to add random dataset nodes to the mix, the \( E_{\text{accesses}} \) function is used:

\[
E_{\text{accesses}}(d, m, s, i) = \begin{cases} 
E_{\text{datasetitem}}(d, m, s, i) & \text{if } i = J_{\text{accesses}} - 2 \\
E_{\text{accesses}}(E_{\text{datasetitem}}(d, m, s, i), s, i + 1) & \text{otherwise}
\end{cases}
\]

\[
E_{\text{datasetitem}}(d, m, s, i) = E_{\text{datasetitem}}(E_{\text{mixdataset}}(d, m, s, i))
\]

\[
E_{\text{datasetitem}}(d, m, s, i) = E_{\text{mixdataset}}(d, m, s, i)
\]

\[
E_{\text{mixdataset}}(d, m, s, i) = E_{\text{mix}}(m, d, s, i)
\]
$E_{\text{newdata}}$ returns an array with $n_{\text{mix}}$ elements:

$E_{\text{newdata}}(d, m, s, i)[j] = d[EFNV(i \oplus s[0], m[i \mod \left\lfloor \frac{J_{\text{mixbytes}}}{J_{\text{wordbytes}}} \right\rfloor) \mod \left\lfloor \frac{d_{\text{size}}/J_{\text{hashbytes}}}{n_{\text{mix}}} \right\rfloor \cdot n_{\text{mix}} + j] \quad \forall \ j < n_{\text{mix}}$

The mix is compressed as follows:

$E_{\text{compress}}(m, i) = \begin{cases} m & \text{if } i \geq \|m\| - 8 \\ E_{\text{compress}}(EFNV(E_{\text{compress}}(EFNV(m[i + 4], m[i + 5]), m[i + 6]), m[i + 7]), i + 8) & \text{otherwise} \end{cases}$

**Appendix K. Anomalies on the Main Network**

K.1. **Deletion of an Account Despite Out-of-gas.** At block 2675119, in the transaction 0xcf416c536ec1a19ed1fb89e4ec7ffb3cf73aa413b3a9b77d60e4fd81a4296ba, an account at address 0x03 was called and an out-of-gas occurred during the call. Against the equation (200), this added 0x03 in the set of touched addresses, and this transaction turned $\sigma[0x03]$ into $\emptyset$.

**Appendix L. List of Mathematical Symbols**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Latex Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\bigvee</td>
<td>\bigvee</td>
<td>This is the least upper bound, supremum, or join of all elements operated on. Thus it is the greatest element of such elements (Davey and Priestley [2002]).</td>
</tr>
</tbody>
</table>