ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER

SHANGHAI VERSION f3553dd – 2024-11-04

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.

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

Overall, 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 Sprankel [2013]. Other projects sought to take the core value content mechanism of the protocol and repurpose it; Aron [2012] discusses, for example,

the Namecoin project which aims to provide a decentralised name-resolution system.

Other projects still aim to build upon the Bitcoin network itself, leveraging the large amount of value placed in the system and the vast amount of computation that goes into the consensus mechanism. The Mastercoin project, first proposed by Willett [2013], aims to build a richer protocol involving many additional high-level features on top of the Bitcoin protocol through utilisation of a number of auxiliary parts to the core protocol. The Coloured Coins project, proposed by Rosenfeld et al. [2012], takes a similar but more simplified strategy, embellishing the rules of a transaction in order to break the fungibility of Bitcoin's base currency and allow the creation and tracking of tokens through a special "chroma-wallet"-protocol-aware piece of software.

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

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

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 balances, reputations, trust arrangements, data pertaining to information of the physical world; in short, anything that can currently be represented by a computer is admissible. Transactions thus represent a valid arc between two states; the 'valid' part is important—there exist far more invalid state changes than valid state changes. Invalid state changes might, e.g., be things such as reducing an account balance without an equal and opposite increase elsewhere. A valid state transition is one which comes about through a transaction. Formally:

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

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

Transactions are collated into blocks; blocks are chained together using a cryptographic hash as a means of reference. Blocks function as a journal, recording a series of transactions together with the previous block and an identifier for the final state (though do not store the final state itself—that would be far too big).

Formally, we expand to:

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

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

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

Where B is this block, which includes a series of transactions amongst some other components and Π is the block-level state-transition function for transactions[1].

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

2.1. Value.

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

Multiplier	Name
10^0	Wei
10^9	Gwei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

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

2.2. Which History?

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

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

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

[1]Note that since the Shanghai fork, blocks also needs to process *withdrawal operations* in order to reach their final state. Withdrawal operations are defined in sub-section 4.3, and block final state is discussed in greater detail in section 12.

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 *Shanghai* 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:[2]

Name	First Block Number
$F_{\text{Homestead}}$	1150000
$F_{\text{TangerineWhistle}}$	2463000
$F_{\text{SpuriousDragon}}$	2675000
$F_{\text{Byzantium}}$	4370000
$F_{\text{Constantinople}}$	7280000
$F_{\text{Petersburg}}$	7280000
F_{Istanbul}	9069000
$F_{\text{MuirGlacier}}$	9200000
F_{Berlin}	12244000
F_{London}	12965000
$F_{\text{ArrowGlacier}}$	13773000
$F_{\text{GrayGlacier}}$	15050000
F_{Paris}	15537394
F_{Shanghai}	17034870

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 β. For the Ethereum main network

$$\beta = 1 \tag{5}$$

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

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

For most functions, an uppercase letter is used, e.g. C, the general cost function. These may be subscripted to denote specialised variants, e.g. C_{SSTORE}, the cost function for the SSTORE operation. For specialised and possibly externally defined functions, 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 KEC (and generally referred to as plain Keccak). Also, 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}, denotes the nonce of said transaction. The form of the subscript is used to denote its type; e.g. uppercase subscripts refer to tuples with subscriptable components.

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

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

Throughout, we assume scalars are 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 4.4.

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

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

When considering variants of existing values, 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 4.4.

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

$$\ell(\mathbf{x}) \equiv \mathbf{x}[\|\mathbf{x}\| - 1] \tag{6}$$

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.

[2]Note that while the Paris, Shanghai, and every upcoming forks activated at a given block number (e.g. 15,537,394 for Paris), the trigger was not the block number, but rather reaching a specified *timestamp* (or *total difficulty* for Paris). The trigger for the *Paris* and subsequent hard fork are discussed in greater detail in section 10.

4.1. **World State.** The world state (*state*), is a mapping between addresses (160-bit identifiers) and account states (a data structure serialised as RLP, see Appendix B). Though not stored on the blockchain, it is assumed that the implementation will maintain this mapping in a modified Merkle Patricia tree (*trie*, see Appendix D). The trie requires a simple database backend that maintains a mapping of 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, $\boldsymbol{\sigma}[a]$, 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 $\boldsymbol{\sigma}$, this would be formally denoted $\boldsymbol{\sigma}[a]_{\mathrm{n}}$.

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

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

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

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:

$$\mathtt{TRIE}\big(L_{\mathrm{I}}^{*}(\boldsymbol{\sigma}[a]_{\mathbf{s}})\big) \equiv \boldsymbol{\sigma}[a]_{\mathrm{s}} \tag{7}$$

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

$$L_{\mathrm{I}}\big((k, v)\big) \equiv \big(\mathtt{KEC}(k), \mathtt{RLP}(v)\big) \tag{8}$$

where:

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{N} \tag{9}$$

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

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

Thus we may define a world-state collapse function L_{S}:

$$L_{\mathrm{S}}(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{10}$$

where

$$p(a) \equiv \big(\mathtt{KEC}(a), \mathtt{RLP}\big((\boldsymbol{\sigma}[a]_{\mathrm{n}}, \boldsymbol{\sigma}[a]_{\mathrm{b}}, \boldsymbol{\sigma}[a]_{\mathrm{s}}, \boldsymbol{\sigma}[a]_{\mathrm{c}})\big)\big) \tag{11}$$

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

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \ \vee \ (a \in \mathbb{B}_{20} \ \wedge \ v(\boldsymbol{\sigma}[a])) \tag{12}$$

where v is the account validity function:

$$v(x) \equiv x_{\mathrm{n}} \in \mathbb{N}_{256} \wedge x_{\mathrm{b}} \in \mathbb{N}_{256} \wedge x_{\mathrm{s}} \in \mathbb{B}_{32} \wedge x_{\mathrm{c}} \in \mathbb{B}_{32} \tag{13}$$

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

$$\mathtt{EMPTY}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_{\mathrm{c}} = \mathtt{KEC}\big(()\big) \wedge \boldsymbol{\sigma}[a]_{\mathrm{n}} = 0 \wedge \boldsymbol{\sigma}[a]_{\mathrm{b}} = 0 \tag{14}$$

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:

$$\mathtt{DEAD}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a] = \varnothing \vee \mathtt{EMPTY}(\boldsymbol{\sigma}, a) \tag{15}$$

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[3]. 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 [2020b]), 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_{x}.

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}.

[3]Notably, such 'tools' could ultimately become so causally removed from their human-based initiation—or humans may become so causally-neutral—that there could be a point at which they rightly be considered autonomous agents. e.g. contracts may offer bounties to humans for being sent transactions to initiate their execution.

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-2930 (type 1) and EIP-1559 (type 2) transactions also have:

accessList: List of access entries to warm up; formally $T_{\mathbf{A}}$. Each access list entry E is a tuple of an account address and a list of storage keys: $E \equiv (E_{\mathrm{a}}, E_{\mathbf{s}})$.

chainId: Chain ID; formally T_{c}. Must be equal to the network chain ID β.

yParity: Signature Y parity; formally T_{y}.

Legacy transactions do not have an **accessList** ($T_{\mathbf{A}} = ()$), 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}. $T_{\mathrm{w}} = 27 + T_{\mathrm{y}}$ or $T_{\mathrm{w}} = 2\beta + 35 + T_{\mathrm{y}}$ (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*[4] 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_{f}.

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}.[5]

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 $T_{\mathbf{i}}$.

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

In contrast, a message call transaction contains:

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

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

(16)
$$L_{\mathrm{T}}(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 0 \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}, T_{\mathrm{y}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 1 \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{f}}, T_{\mathrm{m}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}, T_{\mathrm{y}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{x}} = 2 \end{cases}$$

where

(17)
$$\mathbf{p} \equiv \begin{cases} T_{\mathbf{i}} & \text{if } T_{\mathrm{t}} = \varnothing \\ T_{\mathbf{d}} & \text{otherwise} \end{cases}$$

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

(18)
$$\begin{array}{lllllll} T_{\mathrm{x}} \in \{0,1,2\} & \wedge & T_{\mathrm{c}} = \beta & \wedge & T_{\mathrm{n}} \in \mathbb{N}_{256} & \wedge \\ T_{\mathrm{p}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{g}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{v}} \in \mathbb{N}_{256} & \wedge \\ T_{\mathrm{w}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{r}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{s}} \in \mathbb{N}_{256} & \wedge \\ T_{\mathrm{y}} \in \mathbb{N}_{1} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} & \wedge \\ T_{\mathrm{m}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{f}} \in \mathbb{N}_{256} & & & \end{array}$$

where

(19)
$$\mathbb{N}_{\mathrm{n}} = \{P : P \in \mathbb{N} \wedge P < 2^{n}\}$$

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

(20)
$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if } \quad T_{\mathrm{t}} \neq \varnothing \\ \mathbb{B}_{0} & \text{otherwise} \end{cases}$$

4.3. **The Withdrawal.** A withdrawal (formally, W) is a tuple of data describing a consensus layer validator's withdrawal of some amount of its staked Ether. A withdrawal is created and validated in the consensus layer of the blockchain and then pushed to the execution layer. A withdrawal is composed of the following fields:

globalIndex: zero based incrementing withdrawal index that acts as a unique identifier for this withdrawal; formally W_{g}.

validatorIndex: index of consensus layer's validator this withdrawal corresponds to; formally W_{v}.

recipient: the 20-byte address that will receives Ether form this withdrawal; formally W_{r}.

amount: a nonzero amount of Ether denominated in Gwei (10^9 Wei); formally W_{a}.

Withdrawal serialisation is defined as:

(21)
$$L_{\mathrm{W}}(W) \equiv (W_{\mathrm{g}}, W_{\mathrm{v}}, W_{\mathrm{r}}, T_{\mathrm{a}})$$

Here, we assume all components are interpreted by the RLP as integer values except for $W_{\mathbf{r}}$ which is a 20-byte address:

(22)
$$\begin{array}{lllll} W_{\mathrm{g}} \in \mathbb{N}_{64} & \wedge & W_{\mathrm{v}} \in \mathbb{N}_{64} & \wedge \\ W_{\mathrm{r}} \in \mathbb{B}_{20} & \wedge & W_{\mathrm{a}} \in \mathbb{N}_{64} & \end{array}$$

[4] The *priority fee* is discussed in greater detail in sections 5 and 6.

[5] Type 0 and type 1 transactions will get the same gas price behavior as a type 2 transaction with T_{m} and T_{f} set to the value of T_{p}.

[6] *ommer* is a gender-neutral term to mean "sibling of parent"; see `https://nonbinary.miraheze.org/wiki/Gender_neutral_language_in_English#Aunt/Uncle`

4.4. **The Block.** The block in Ethereum is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$, a now deprecated property $\mathbf{U}$ that prior to the *Paris* hard fork contained headers of blocks whose parents were equal to the present block's parent's parent (such blocks were known as *ommers*[6]), and, since the *Shanghai* hard fork, $\mathbf{W}$, a collection of validator's withdrawal pushed by the consensus layer. The block header contains several pieces of information:

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

ommersHash: A 256-bit hash field that is now deprecated due to the replacement of proof of work consensus. It is now to a constant, KEC(RLP(())); formally H_{o}.

beneficiary: The 160-bit address to which priority fees from this block are transferred; formally H_{c}.

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

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

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

logsBloom: The Bloom filter composed from indexable information (logger address and log topics) contained in each log entry from the receipt of each transaction in the transactions list; formally H_{b}.

difficulty: A scalar field that is now deprecated due to the replacement of proof of work consensus. It is set to 0; formally H_{d}.

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

gasLimit: A scalar value equal to the current limit of gas expenditure per block; formally H_{l}.

gasUsed: A scalar value equal to the total gas used in transactions in this block; formally H_{g}.

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

extraData: An arbitrary byte array containing data relevant to this block. This must be 32 bytes or fewer; formally H_{x}.

prevRandao: the latest RANDAO mix[7] of the post beacon state of the previous block; formally H_{a}.

nonce: A 64-bit value that is now deprecated due to the replacement of proof of work consensus. It is set to 0x0000000000000000; formally H_{n}.

baseFeePerGas: A scalar value equal to the amount of wei that is burned for each unit of gas consumed; formally H_{f}.

withdrawalsRoot: The Keccak 256-bit hash of the root node of the trie structure populated with each withdrawal operations pushed by the consensus layer for this block; formally H_{w}.

The other three components in the block are a series of transactions, $B_{\mathbf{T}}$, an empty array which was previously reserved for ommer block headers, $B_{\mathbf{U}}$, and a series of withdrawals, $B_{\mathbf{W}}$. Formally, we can refer to a block B:

$$B \equiv (B_{\mathrm{H}}, B_{\mathbf{T}}, B_{\mathbf{U}}, B_{\mathbf{W}}) \tag{23}$$

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

The transaction receipt, R, is a tuple of five items comprising: the type of the transaction, R_{x}, the status code of the transaction, R_{z}, the cumulative gas used in the block containing the transaction receipt as of immediately after the transaction has happened, R_{u}, the set of logs created through execution of the transaction, $R_{\mathbf{l}}$ and the Bloom filter composed from information in those logs, R_{b}:

$$R \equiv (R_{\mathrm{x}}, R_{\mathrm{z}}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}) \tag{24}$$

R_{x} is equal to the type of the corresponding transaction.

The function L_{R} prepares a transaction receipt for being transformed into an RLP-serialised byte array:

$$L_{\mathrm{R}}(R) \equiv (R_{\mathrm{z}}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}) \tag{25}$$

We assert that the status code R_{z} is a non-negative integer:

$$R_{\mathrm{z}} \in \mathbb{N} \tag{26}$$

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):

$$R_{\mathrm{u}} \in \mathbb{N} \quad \wedge \quad R_{\mathrm{b}} \in \mathbb{B}_{256} \tag{27}$$

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

$$O \equiv (O_{\mathrm{a}}, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{28}$$

$$O_{\mathrm{a}} \in \mathbb{B}_{20} \quad \wedge \quad \forall x \in O_{\mathbf{t}} : x \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{29}$$

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

$$M(O) \equiv \bigvee_{x \in \{O_{\mathrm{a}}\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(x)\right) \tag{30}$$

where $M_{3:2048}$ 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

[7] RANDAO is a pseudorandom value generated by validators on the Ethereum consensus layer. Refer to the consensus layer specs (https://github.com/ethereum/consensus-specs) for more detail on RANDAO.

[8] $2048 = 2^{11}$ (11 bits), and the low-order 11 bits is the modulo 2048 of the operand, which is in this case is "each of the first three pairs of bytes in a Keccak-256 hash of the byte sequence."

sequence.[8] Formally:

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \tag{31}$$
$$\mathbf{y} = (0, 0, ..., 0) \quad \text{except:} \tag{32}$$
$$\forall i \in \{0, 2, 4\} : \mathcal{B}_{2047-m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{33}$$
$$m(\mathbf{x}, i) \equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{34}$$

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

4.4.2. *Holistic Validity.* We can assert a block's validity if and only if it satisfies several conditions: the block's ommers field $B_{\mathbf{U}}$ must be an empty array and the block's header must be consistent with the given transactions $B_{\mathbf{T}}$ and withdrawals $B_{\mathbf{W}}$. For the header to be consistent with the transactions $B_{\mathbf{T}}$ and withdrawals $B_{\mathbf{W}}$, **stateRoot** (H_{r}) must match the resultant state after executing all transactions, then all withdrawals, in order on the base state $\boldsymbol{\sigma}$ (as specified in section 12), and **transactionsRoot** (H_{t}), **receiptsRoot** (H_{e}), **logsBloom** (H_{b}) and **withdrawalsRoot** (H_{w}) must be correctly derived from the transactions themselves, the transaction receipts resulting from execution, the resulting logs, and the withdrawals, respectively.

$$\begin{aligned}
B_{\mathbf{U}} &\equiv () &\wedge\\
H_{\mathrm{r}} &\equiv \texttt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) &\wedge\\
H_{\mathrm{t}} &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{N} :\\
&\qquad p_{\mathrm{T}}(i, B_{\mathbf{T}}[i])\}) &\wedge\\
H_{\mathrm{e}} &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{N} :\\
&\qquad p_{\mathrm{R}}(i, B_{\mathbf{R}}[i])\}) &\wedge\\
H_{\mathrm{w}} &\equiv \texttt{TRIE}(\{\forall i < \|B_{\mathbf{W}}\|, i \in \mathbb{N} :\\
&\qquad p_{\mathrm{W}}(i, B_{\mathbf{W}}[i])\}) &\wedge\\
H_{\mathrm{b}} &\equiv \bigvee_{\mathbf{r} \in B_{\mathbf{R}}} (\mathbf{r}_{\mathrm{b}})
\end{aligned} \tag{35}$$

where $p_{\mathrm{W}}(k, v)$ is a pairwise RLP transformation for withdrawals:

$$p_{\mathrm{W}}(k, W) \equiv (\texttt{RLP}(k), \texttt{RLP}(L_{\mathrm{W}}(W))) \tag{36}$$

similarly, $p_{\mathrm{T}}(k, v)$ and $p_{\mathrm{R}}(k, v)$ are pairwise RLP transformations, but with a special treatment for EIP-2718 transactions:

$$p_{\mathrm{T}}(k, T) \equiv \left(\texttt{RLP}(k), \begin{cases} \texttt{RLP}(L_{\mathrm{T}}(T)) & \text{if} \quad T_{\mathrm{x}} = 0 \\ (T_{\mathrm{x}}) \cdot \texttt{RLP}(L_{\mathrm{T}}(T)) & \text{otherwise} \end{cases}\right) \tag{37}$$

and

$$p_{\mathrm{R}}(k, R) \equiv \left(\texttt{RLP}(k), \begin{cases} \texttt{RLP}(L_{\mathrm{R}}(R)) & \text{if} \quad R_{\mathrm{x}} = 0 \\ (R_{\mathrm{x}}) \cdot \texttt{RLP}(L_{\mathrm{R}}(R)) & \text{otherwise} \end{cases}\right) \tag{38}$$

($\cdot$ is the concatenation of byte arrays).

Furthermore:

$$\texttt{TRIE}(L_{\mathrm{S}}(\boldsymbol{\sigma})) = P(B_H)_{H_{\mathrm{r}}} \tag{39}$$

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

The values stemming from the computation of transactions, specifically the transaction receipts, $B_{\mathbf{R}}$, and that defined through the transaction's state-accumulation function, Π, are formalised later in section 12.3.

4.4.3. *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:

$$\begin{aligned}
L_{\mathrm{H}}(H) \equiv (\ & H_{\mathrm{p}}, H_{\mathrm{o}}, H_{\mathrm{c}}, H_{\mathrm{r}}, H_{\mathrm{t}}, H_{\mathrm{e}}, H_{\mathrm{b}}, H_{\mathrm{d}},\\
& H_{\mathrm{i}}, H_{\mathrm{l}}, H_{\mathrm{g}}, H_{\mathrm{s}}, H_{\mathrm{x}}, H_{\mathrm{a}}, H_{\mathrm{n}}, H_{\mathrm{f}}, H_{\mathrm{w}}\ )
\end{aligned} \tag{40}$$
$$L_{\mathrm{B}}(B) \equiv \big(L_{\mathrm{H}}(B_{\mathrm{H}}), \widetilde{L}^*_{\mathrm{T}}(B_{\mathbf{T}}), L^*_{\mathrm{H}}(B_{\mathbf{U}}), L^*_{\mathrm{W}}(B_{\mathbf{W}})\big) \tag{41}$$

where $\widetilde{L}_{\mathrm{T}}$ takes a special care of EIP-2718 transactions:

$$\widetilde{L}_{\mathrm{T}}(T) = \begin{cases} L_{\mathrm{T}}(T) & \text{if} \quad T_{\mathrm{x}} = 0 \\ (T_{\mathrm{x}}) \cdot \texttt{RLP}(L_{\mathrm{T}}(T)) & \text{otherwise} \end{cases} \tag{42}$$

with $\widetilde{L}^*_{\mathrm{T}}$, L^*_{H}, and L^*_{W}, being element-wise sequence transformations, thus:

$$f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{for any function } f \tag{43}$$

The component types are defined thus:

$$\begin{aligned}
&H_{\mathrm{p}} \in \mathbb{B}_{32} \wedge H_{\mathrm{o}} \in \mathbb{B}_{32} \wedge H_{\mathrm{c}} \in \mathbb{B}_{20} \wedge\\
&H_{\mathrm{r}} \in \mathbb{B}_{32} \wedge H_{\mathrm{t}} \in \mathbb{B}_{32} \wedge H_{\mathrm{e}} \in \mathbb{B}_{32} \wedge\\
&H_{\mathrm{b}} \in \mathbb{B}_{256} \wedge H_{\mathrm{d}} \in \mathbb{N} \wedge H_{\mathrm{i}} \in \mathbb{N} \wedge\\
&H_{\mathrm{l}} \in \mathbb{N} \wedge H_{\mathrm{g}} \in \mathbb{N} \wedge H_{\mathrm{s}} \in \mathbb{N}_{256} \wedge\\
&H_{\mathrm{x}} \in \mathbb{B} \wedge H_{\mathrm{a}} \in \mathbb{B}_{32} \wedge H_{\mathrm{n}} \in \mathbb{B}_{8} \wedge\\
&H_{\mathrm{f}} \in \mathbb{N} \wedge H_{\mathrm{w}} \in \mathbb{B}_{32}
\end{aligned} \tag{44}$$

where

$$\mathbb{B}_{\mathrm{n}} = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{45}$$

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

4.4.4. *Block Header Validity.* We define $P(B_{\mathrm{H}})$ to be the parent block of B, formally:

$$P(H) \equiv B' : \texttt{KEC}(\texttt{RLP}(B'_{\mathrm{H}})) = H_{\mathrm{p}} \tag{46}$$

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

$$H_{\mathrm{i}} \equiv P(H)_{\mathrm{H_i}} + 1 \tag{47}$$

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 $F(H)$:

$$F(H) \equiv \begin{cases} 1000000000 & \text{if} \quad H_{\mathrm{i}} = F_{\mathrm{London}} \\ P(H)_{\mathrm{H_f}} & \text{if} \quad P(H)_{\mathrm{H_g}} = \tau \\ P(H)_{\mathrm{H_f}} - \nu & \text{if} \quad P(H)_{\mathrm{H_g}} < \tau \\ P(H)_{\mathrm{H_f}} + \nu & \text{if} \quad P(H)_{\mathrm{H_g}} > \tau \end{cases} \tag{48}$$

where:

$$\tau \equiv \lfloor \frac{P(H)_{\mathrm{H_l}}}{\rho} \rfloor \tag{49}$$

$$\rho \equiv 2 \tag{50}$$

$$\nu^* \equiv \begin{cases} \lfloor \frac{P(H)_{\mathrm{H_f}} \times (\tau - P(H)_{\mathrm{H_g}})}{\tau} \rfloor & \text{if} \quad P(H)_{\mathrm{H_g}} < \tau \\ \lfloor \frac{P(H)_{\mathrm{H_f}} \times (P(H)_{\mathrm{H_g}} - \tau)}{\tau} \rfloor & \text{if} \quad P(H)_{\mathrm{H_g}} > \tau \end{cases} \tag{51}$$

$$\nu \equiv \begin{cases} \lfloor \frac{\nu^*}{\xi} \rfloor & \text{if} \quad P(H)_{\mathrm{H_g}} < \tau \\ max(\lfloor \frac{\nu^*}{\xi} \rfloor, 1) & \text{if} \quad P(H)_{\mathrm{H_g}} > \tau \end{cases} \tag{52}$$

$$\xi \equiv 8 \tag{53}$$

The *gas target*, τ, is defined as the gas limit H_l divided by the *elasticity multiplier*, ρ, 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 ν, 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 ξ, 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_l of a block of header H must fulfil the relation:

$$\begin{aligned} & H_\mathrm{l} < P(H)_{\mathrm{H_{l'}}} + \left\lfloor \frac{P(H)_{\mathrm{H_{l'}}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} > P(H)_{\mathrm{H_{l'}}} - \left\lfloor \frac{P(H)_{\mathrm{H_{l'}}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} \geqslant 5000 \end{aligned} \tag{54}$$

where:

$$P(H)_{\mathrm{H_{l'}}} \equiv \begin{cases} P(H)_{\mathrm{H_l}} \times \rho & \text{if} \quad H_\mathrm{i} = F_\mathrm{London} \\ P(H)_{\mathrm{H_l}} & \text{if} \quad H_\mathrm{i} > F_\mathrm{London} \end{cases} \tag{55}$$

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*, ρ. We call this modified value $P(H)_{\mathrm{H_{l'}}}$. 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 fulfil the relation:

$$H_\mathrm{s} > P(H)_{\mathrm{H_s}} \tag{56}$$

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** ($H_\mathbf{n}$), **ommersHash** ($H_\mathbf{o}$), **difficulty** ($H_\mathbf{d}$), and **mixHash** ($H_\mathbf{m}$).

mixHash has been replaced with a new field **prevRandao** ($H_\mathbf{a}$). The other header fields related to proof of work have been replaced with constants:

$$H_\mathrm{o} \equiv \texttt{KEC}(\texttt{RLP}(())) \tag{57}$$

$$H_\mathrm{d} \equiv 0 \tag{58}$$

$$H_\mathrm{n} \equiv \texttt{0x0000000000000000} \tag{59}$$

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 PREVRANDAO().

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

$$\begin{aligned} V(H) \equiv \; & H_\mathrm{g} \leq H_\mathrm{l} \quad \wedge \\ & H_\mathrm{l} < P(H)_{\mathrm{H_{l'}}} + \left\lfloor \frac{P(H)_{\mathrm{H_{l'}}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} > P(H)_{\mathrm{H_{l'}}} - \left\lfloor \frac{P(H)_{\mathrm{H_{l'}}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} \geqslant 5000 \quad \wedge \\ & H_\mathrm{s} > P(H)_{\mathrm{H_s}} \quad \wedge \\ & H_\mathrm{i} = P(H)_{\mathrm{H_i}} + 1 \quad \wedge \\ & \|H_\mathrm{x}\| \leq 32 \quad \wedge \\ & H_\mathrm{f} = F(H) \wedge \\ & H_\mathrm{o} = \texttt{KEC}(\texttt{RLP}(())) \quad \wedge \\ & H_\mathrm{d} = 0 \quad \wedge \\ & H_\mathrm{n} = \texttt{0x0000000000000000} \quad \wedge \\ & H_\mathrm{a} = \texttt{PREVRANDAO}() \end{aligned} \tag{60}$$

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 6. 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 4.4 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 transactor 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.

Transactors 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 be 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 specifiying 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.

Transactors 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, transactors 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 Υ. It is assumed that any transactions executed first pass the initial tests of intrinsic validity. These include:

(1) The transaction is well-formed RLP, with no additional trailing bytes;
(2) the transaction signature is valid;
(3) the transaction nonce is valid (equivalent to the sender account's current nonce);
(4) the 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 **maxFeePerGas**, T_m, in the case of type 2 transactions, or **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_f; and
(8) for type 2 transactions, **maxPriorityFeePerGas**, T_f, must be no larger than **maxFeePerGas**, T_m.

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

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{61}$$

Thus $\boldsymbol{\sigma}'$ is the post-transactional state. We also define Υ^g to evaluate to the amount of gas used in the execution of a transaction, $\Upsilon^\mathbf{l}$ to evaluate to the transaction's accrued log items and Υ^z 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 *accrued transaction substate*, or *accrued substate* for short, and represent it as A, which is a tuple:

$$A \equiv (A_\mathbf{s}, A_\mathbf{l}, A_\mathbf{t}, A_\mathrm{r}, A_\mathbf{a}, A_\mathbf{K}) \tag{62}$$

The tuple contents include $A_\mathbf{s}$, the self-destruct set: a set of accounts that will be discarded following the transaction's completion. $A_\mathbf{l}$ is the log series: this is a series of archived and indexable 'checkpoints' in VM code execution that allow for contract-calls to be easily tracked by onlookers external to the Ethereum world (such as decentralised application front-ends). $A_\mathbf{t}$ 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 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 $A_\mathbf{a}$, the set of accessed account addresses, and $A_\mathbf{K}$, the set of accessed storage keys (more accurately, each element of $A_\mathbf{K}$ 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:

$$A^0 \equiv (\varnothing, (), \varnothing, 0, \pi, \varnothing) \tag{63}$$

where π 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:

$$\begin{aligned} g_0 \equiv & \sum_{i \in T_\mathbf{i}, T_\mathbf{d}} \begin{cases} G_\mathrm{txdatazero} & \text{if} \quad i = 0 \\ G_\mathrm{txdatanonzero} & \text{otherwise} \end{cases} \\ & + \begin{cases} G_\mathrm{txcreate} + R(\|T_\mathbf{i}\|) & \text{if} \quad T_\mathrm{t} = \varnothing \\ 0 & \text{otherwise} \end{cases} \\ & + G_\mathrm{transaction} \\ & + \sum_{j=0}^{\|T_\mathbf{A}\|-1} \left(G_\mathrm{accesslistaddress} + \|T_\mathbf{A}[j]_\mathbf{s}\| G_\mathrm{accessliststorage}\right) \end{aligned} \tag{64}$$

where $T_\mathbf{i}, T_\mathbf{d}$ means the series of bytes of the transaction's associated data and initialisation EVM-code, depending on

whether the transaction is for contract-creation or message-call. G_{txcreate} is added if the transaction is contract-creating, but not if it is a message call. $G_{\text{accesslistaddress}}$ and $G_{\text{accessliststorage}}$ are the costs of warming up account and storage access, respectively. G is fully defined in Appendix G.

We define the *initcode cost function*, formally R, as the amount of gas that needs to be paid for each word of the initcode prior to executing the creation code of a new contract:

$$R(x) \equiv G_{\text{initcodeword}} \times \lceil \frac{x}{32} \rceil \tag{65}$$

We define the *effective gas price*, formally p, as the amount of wei the transaction signer will pay per unit of gas consumed during the transaction's execution. It is calculated as follows:

$$p \equiv \begin{cases} T_{\text{p}} & \text{if } T_{\text{x}} = 0 \vee T_{\text{x}} = 1 \\ f + H_{\text{f}} & \text{if } T_{\text{x}} = 2 \end{cases} \tag{66}$$

where f is the *priority fee*—the amount of wei the block's beneficiary address will receive per unit of gas consumed during the transaction's execution. It is calculated as:

$$f \equiv \begin{cases} T_{\text{p}} - H_{\text{f}} & \text{if } T_{\text{x}} = 0 \vee T_{\text{x}} = 1 \\ min(T_{\text{f}}, T_{\text{m}} - H_{\text{f}}) & \text{if } T_{\text{x}} = 2 \end{cases} \tag{67}$$

The up-front cost v_0 is calculated as:

$$v_0 \equiv \begin{cases} T_{\text{g}} T_{\text{p}} + T_{\text{v}} & \text{if } T_{\text{x}} = 0 \vee T_{\text{x}} = 1 \\ T_{\text{g}} T_{\text{m}} + T_{\text{v}} & \text{if } T_{\text{x}} = 2 \end{cases} \tag{68}$$

The validity is determined as:

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)]_{\text{c}} &= \texttt{KEC}(()) \quad \wedge \\ T_{\text{n}} &= \boldsymbol{\sigma}[S(T)]_{\text{n}} \quad \wedge \\ g_0 &\leqslant T_{\text{g}} \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_{\text{b}} \quad \wedge \\ m &\geqslant H_{\text{f}} \quad \wedge \\ n &\leqslant 49152 \quad \wedge \\ T_{\text{g}} &\leqslant B_{\text{Hl}} - \ell(B_{\mathbf{R}})_{\text{u}} \end{aligned} \tag{69}$$

where

$$m \equiv \begin{cases} T_{\text{p}} & \text{if } T_{\text{x}} = 0 \vee T_{\text{x}} = 1 \\ T_{\text{m}} & \text{if } T_{\text{x}} = 2 \end{cases} \tag{70}$$

and

$$n \equiv \begin{cases} \|T_{\text{i}}\| & \text{if } T_{\text{t}} \neq \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{71}$$

The penultimate condition ensures that, for create transactions, the length of the initcode is no greater than 49152 bytes. Note the final condition; the sum of the transaction's gas limit, T_{g}, and the gas utilised in this block prior, given by $\ell(B_{\mathbf{R}})_{\text{u}}$, must be no greater than the block's **gasLimit**, B_{Hl}. Also, with a slight abuse of notation, we assume that $\boldsymbol{\sigma}[S(T)]_{\text{c}} = \texttt{KEC}(())$, $\boldsymbol{\sigma}[S(T)]_{\text{n}} = 0$, and $\boldsymbol{\sigma}[S(T)]_{\text{b}} = 0$ if $\boldsymbol{\sigma}[S(T)] = \varnothing$.

For type 2 transactions, we add an additional check that **maxPriorityFeePerGas** is no larger than **maxFeePerGas**:

$$T_{\text{m}} \geqslant T_{\text{f}} \tag{72}$$

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

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

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{73}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\text{b}} \equiv \boldsymbol{\sigma}[S(T)]_{\text{b}} - T_{\text{g}}p \tag{74}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\text{n}} \equiv \boldsymbol{\sigma}[S(T)]_{\text{n}} + 1 \tag{75}$$

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

$$(\boldsymbol{\sigma}_{\text{P}}, g', A, z) \equiv \begin{cases} \Lambda_4(\boldsymbol{\sigma}_0, A^*, S(T), S(T), g, & \\ \quad p, T_{\text{v}}, T_{\text{i}}, 0, \varnothing, \top) & \text{if} \quad T_{\text{t}} = \varnothing \\ \Theta_4(\boldsymbol{\sigma}_0, A^*, S(T), S(T), T_{\text{t}}, & \\ \quad T_{\text{t}}, g, p, T_{\text{v}}, T_{\text{v}}, T_{\mathbf{d}}, 0, \top) & \text{otherwise} \end{cases} \tag{76}$$

where

$$A^* \equiv A^0 \quad \text{except} \tag{77}$$

$$A^*_{\mathbf{K}} \equiv \bigcup_{E \in T_{\mathbf{A}}} \left\{ \forall i < \|E_{\mathbf{s}}\|, i \in \mathbb{N} : \ (E_{\text{a}}, E_{\mathbf{s}}[i]) \right\} \tag{78}$$

$$A^*_{\mathbf{a}} \equiv \begin{cases} a \cup T_{\text{t}} & \text{if} \quad T_{\text{t}} \neq \varnothing \\ a & \text{otherwise} \end{cases} \tag{79}$$

$$a \equiv A^0_{\mathbf{a}} \cup S(T) \cup H_c \cup_{E \in T_{\mathbf{A}}} \{E_{\text{a}}\} \tag{80}$$

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

$$g \equiv T_{\text{g}} - g_0 \tag{81}$$

Note we use Θ_4 and Λ_4 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.

$$g^* \equiv g' + \min \left\{ \left\lfloor \frac{T_{\text{g}} - g'}{5} \right\rfloor, A_{\text{r}} \right\} \tag{82}$$

The total refundable amount is the legitimately remaining gas g', added to A_{r}, with the latter component being capped up to a maximum of one fifth[9] (rounded down) of the total amount used $T_{\text{g}} - g'$. Therefore, g^* is the total gas that remains after the transaction has been executed.

[9]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

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 transactor that goes toward the base fee is debited from the transactor's account but credited to no other account, so it is burned.

We define the pre-final state $\boldsymbol{\sigma}^*$ in terms of the provisional state $\boldsymbol{\sigma}_{\mathrm{P}}$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_{\mathrm{P}} \quad \text{except} \tag{83}$$

$$\boldsymbol{\sigma}^*[S(T)]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_{\mathrm{P}}[S(T)]_{\mathrm{b}} + g^* p \tag{84}$$

$$\boldsymbol{\sigma}^*[B_{\mathrm{Hc}}]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_{\mathrm{P}}[B_{\mathrm{Hc}}]_{\mathrm{b}} + (T_{\mathrm{g}} - g^*) f \tag{85}$$

The final state, $\boldsymbol{\sigma}'$, is reached after deleting all accounts that either appear in the self-destruct set or are touched and empty:

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{86}$$

$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] = \varnothing \tag{87}$$

$$\forall i \in A_{\mathbf{t}} : \boldsymbol{\sigma}'[i] = \varnothing \quad \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^*, i) \tag{88}$$

And finally, we specify Υ^{g}, the total gas used in this transaction Υ^{l}, the logs created by this transaction and Υ^{z}, the status code of this transaction:

$$\Upsilon^{\mathrm{g}}(\boldsymbol{\sigma}, T) \equiv T_{\mathrm{g}} - g^* \tag{89}$$

$$\Upsilon^{\mathrm{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{90}$$

$$\Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}, T) \equiv z \tag{91}$$

These are used to help define the transaction receipt and are also used later for state validation.

7. Contract Creation

There are a number of intrinsic parameters used when creating an account: sender (s), original transactor[10] (o), available gas (g), effective gas price (p), endowment (v) together with an arbitrary length byte array, $\mathbf{i}$, the initialisation EVM code, the present depth of the message-call/contract-creation stack (e), the salt for new account's address (ζ) and finally the permission to make modifications to the state (w). The salt ζ might be missing ($\zeta = \varnothing$); formally,

$$\zeta \in \mathbb{B}_{32} \cup \mathbb{B}_0 \tag{92}$$

If the creation was caused by CREATE2, then $\zeta \neq \varnothing$.

We define the creation function formally as the function Λ, which evaluates from these values, together with the state $\boldsymbol{\sigma}$ and the accrued substate A, to the tuple containing the new state, remaining gas, new accrued substate, status code and output $(\boldsymbol{\sigma}', g', A', z, \mathbf{o})$:

$$(\boldsymbol{\sigma}', g', A', z, \mathbf{o}) \equiv \Lambda(\boldsymbol{\sigma}, A, s, o, g, p, v, \mathbf{i}, e, \zeta, w) \tag{93}$$

The address of the new account is defined as being the rightmost 160 bits of the Keccak-256 hash of the RLP encoding of the structure containing only the sender and the account nonce. For CREATE2 the rule is different and is described in EIP-1014 by Buterin [2018]. Combining the two cases, we define the resultant address for the new account a:

$$a \equiv \texttt{ADDR}(s, \boldsymbol{\sigma}[s]_{\mathrm{n}} - 1, \zeta, \mathbf{i}) \tag{94}$$

$$\texttt{ADDR}(s, n, \zeta, \mathbf{i}) \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\big(L_{\mathrm{A}}(s, n, \zeta, \mathbf{i})\big)\Big) \tag{95}$$

$$L_{\mathrm{A}}(s, n, \zeta, \mathbf{i}) \equiv \begin{cases} \texttt{RLP}\big(\ (s, n)\ \big) & \text{if } \zeta = \varnothing \\ (255) \cdot s \cdot \zeta \cdot \texttt{KEC}(\mathbf{i}) & \text{otherwise} \end{cases} \tag{96}$$

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

The address of the new account is added to the set of accessed accounts:

$$A^* \equiv A \quad \text{except} \quad A^*_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{a\} \tag{97}$$

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

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{98}$$

$$\boldsymbol{\sigma}^*[a] = \big(1, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\big) \tag{99}$$

$$\boldsymbol{\sigma}^*[s] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[s] = \varnothing \ \wedge \ v = 0 \\ \mathbf{a}^* & \text{otherwise} \end{cases} \tag{100}$$

$$\mathbf{a}^* \equiv (\boldsymbol{\sigma}[s]_{\mathrm{n}}, \boldsymbol{\sigma}[s]_{\mathrm{b}} - v, \boldsymbol{\sigma}[s]_{\mathbf{s}}, \boldsymbol{\sigma}[s]_{\mathrm{c}}) \tag{101}$$

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

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_{\mathrm{b}} & \text{otherwise} \end{cases} \tag{102}$$

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

$$(\boldsymbol{\sigma}^{**}, g^{**}, A^{**}, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, A^*, I) \tag{103}$$

where I contains the parameters of the execution environment, that is:

[10] 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

$$I_{\mathrm{a}} \equiv a \tag{104}$$
$$I_{\mathrm{o}} \equiv o \tag{105}$$
$$I_{\mathrm{p}} \equiv p \tag{106}$$
$$I_{\mathbf{d}} \equiv () \tag{107}$$
$$I_{\mathrm{s}} \equiv s \tag{108}$$
$$I_{\mathrm{v}} \equiv v \tag{109}$$
$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{110}$$
$$I_{\mathrm{e}} \equiv e \tag{111}$$
$$I_{\mathrm{w}} \equiv w \tag{112}$$

$I_{\mathbf{d}}$ evaluates to the empty tuple as there is no input data to this call. I_{H} has no special treatment and is determined from the blockchain.

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, $\varnothing$, and the entire create operation should have no effect on the state, effectively leaving it as it was immediately prior to attempting the creation.

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:

$$c \equiv G_{\mathrm{codedeposit}} \times \|\mathbf{o}\| \tag{113}$$

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 $(\boldsymbol{\sigma}', g', A', z)$ where:

$$g' \equiv \begin{cases} 0 & \text{if} \quad F \\ g^{**} - c & \text{otherwise} \end{cases} \tag{114}$$

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad F \ \vee \ \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a] = \varnothing & \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^{**}, a) \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_{\mathrm{c}} = \texttt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases} \tag{115}$$

$$A' \equiv \begin{cases} A^{*} & \text{if} \quad F \ \vee \ \boldsymbol{\sigma}^{**} = \varnothing \\ A^{**} & \text{otherwise} \end{cases} \tag{116}$$

$$z \equiv \begin{cases} 0 & \text{if} \quad F \ \vee \ \boldsymbol{\sigma}^{**} = \varnothing \\ 1 & \text{otherwise} \end{cases} \tag{117}$$

where

$$\begin{aligned} F \equiv & \big(\boldsymbol{\sigma}[a] \neq \varnothing \ \wedge \ \big(\boldsymbol{\sigma}[a]_c \neq \texttt{KEC}(()) \vee \boldsymbol{\sigma}[a]_n \neq 0\big)\big) \quad \vee \\ & (\boldsymbol{\sigma}^{**} = \varnothing \ \wedge \ \mathbf{o} = \varnothing) \quad \vee \\ & g^{**} < c \quad \vee \\ & \|\mathbf{o}\| > 24576 \quad \vee \\ & \mathbf{o}[0] = \texttt{0xef} \end{aligned} \tag{118}$$

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 $\boldsymbol{\sigma}'$ dictates that $\mathbf{o}$, 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 ($\boldsymbol{\sigma}^{**} = \varnothing \ \wedge \ \mathbf{o} \neq \varnothing$), 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[11]. 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

[11] During initialization code execution, `EXTCODESIZE` on the address should return zero, which is the length of the code of the account while `CODESIZE` should return the length of the initialization code (as defined in H.2).

(c, usually the same as recipient), available gas (g), value (v) and effective gas price (p) together with an arbitrary length byte array, $\mathbf{d}$, 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 $\mathbf{o}$. This is ignored when executing transactions, however message calls can be initiated due to VM-code execution and in this case this information is used.

(119)
$$(\boldsymbol{\sigma}', g', A', z, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, A, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e, w)$$

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

$$\boldsymbol{\sigma}_1[r]_\mathrm{b} \equiv \boldsymbol{\sigma}[r]_\mathrm{b} + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_\mathrm{b} \equiv \boldsymbol{\sigma}[s]_\mathrm{b} - v \tag{120}$$

unless $s = r$.

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

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{121}$$

$$\boldsymbol{\sigma}_1[s] \equiv \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}'_1[s] = \varnothing \ \wedge \ v = 0 \\ \mathbf{a}_1 & \text{otherwise} \end{cases} \tag{122}$$

$$\mathbf{a}_1 \equiv \left(\boldsymbol{\sigma}'_1[s]_\mathrm{n}, \boldsymbol{\sigma}'_1[s]_\mathrm{b} - v, \boldsymbol{\sigma}'_1[s]_\mathbf{s}, \boldsymbol{\sigma}'_1[s]_\mathrm{c}\right) \tag{123}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{124}$$

(125)
$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (0, v, \texttt{TRIE}(\varnothing), \texttt{KEC}(())) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \wedge v \neq 0 \\ \boldsymbol{\sigma}'_1[r] \equiv \varnothing & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \wedge v = 0 \\ \boldsymbol{\sigma}'_1[r] \equiv \mathbf{a}'_1 & \text{otherwise} \end{cases}$$

$$\mathbf{a}'_1 \equiv (\boldsymbol{\sigma}[r]_\mathrm{n}, \boldsymbol{\sigma}[r]_\mathrm{b} + v, \boldsymbol{\sigma}[r]_\mathbf{s}, \boldsymbol{\sigma}[r]_\mathrm{c}) \tag{126}$$

The account's associated code (identified as the fragment whose Keccak-256 hash is $\boldsymbol{\sigma}[c]_\mathrm{c}$) is executed according to the execution model (see section 9). Just as with contract creation, if the execution halts in an exceptional fashion (i.e. due to an exhausted gas supply, stack underflow, invalid jump destination or invalid instruction), then no gas is refunded to the caller and the state is reverted to the point immediately prior to balance transfer (i.e. $\boldsymbol{\sigma}$).

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases} \tag{127}$$

$$g' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \ \wedge \\ & \quad \mathbf{o} = \varnothing \\ g^{**} & \text{otherwise} \end{cases} \tag{128}$$

$$A' \equiv \begin{cases} A & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ A^{**} & \text{otherwise} \end{cases} \tag{129}$$

$$z \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ 1 & \text{otherwise} \end{cases} \tag{130}$$

$$(\boldsymbol{\sigma}^{**}, g^{**}, A^{**}, \mathbf{o}) \equiv \Xi \tag{131}$$

$$I_\mathrm{a} \equiv r \tag{132}$$

$$I_\mathrm{o} \equiv o \tag{133}$$

$$I_\mathrm{p} \equiv p \tag{134}$$

$$I_\mathbf{d} \equiv \mathbf{d} \tag{135}$$

$$I_\mathbf{s} \equiv s \tag{136}$$

$$I_\mathrm{v} \equiv \tilde{v} \tag{137}$$

$$I_\mathrm{e} \equiv e \tag{138}$$

$$I_\mathrm{w} \equiv w \tag{139}$$

where

$$\Xi \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 4 \\ \Xi_{\texttt{EXPMOD}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 5 \\ \Xi_{\texttt{BN\_ADD}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 6 \\ \Xi_{\texttt{BN\_MUL}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 7 \\ \Xi_{\texttt{SNARKV}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 8 \\ \Xi_{\texttt{BLAKE2\_F}}(\boldsymbol{\sigma}_1, g, A, I) & \text{if} \quad c = 9 \\ \Xi(\boldsymbol{\sigma}_1, g, A, I) & \text{otherwise} \end{cases} \tag{140}$$

and

$$\texttt{KEC}(I_\mathbf{b}) = \boldsymbol{\sigma}[c]_\mathrm{c} \tag{141}$$

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

As can be seen, there are nine exceptions to the usage of the general execution framework Ξ 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 RIPEMD 160-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 `F` respectively. Their full formal definition is in Appendix E. We denote the set of the addresses of the precompiled contracts by π:

$$\pi \equiv \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{142}$$

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 G). Secondly, gas may be deducted in order to form the payment for a subordinate message call or contract creation; this forms part of the payment for CREATE, 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 $\boldsymbol{\sigma}$, 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_{o}, 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 p in section 6.
- $I_{\mathbf{d}}$, the byte array that is the input data to this execution; if the execution agent is a transaction, this would be the transaction data.
- I_{s}, the address of the account which caused the code to be executing; if the execution agent is a transaction, this would be the transaction sender.
- I_{v}, the value, in Wei, passed to this account as part of the same procedure as execution; if the execution agent is a transaction, this would be the transaction value.
- $I_{\mathbf{b}}$, the byte array that is the machine code to be executed.
- I_{H}, the block header of the present block.
- I_{e}, the depth of the present message-call or contract-creation (i.e. the number of CALLs or CREATE(2)s being executed at present).
- I_{w}, the permission to make modifications to the state.

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

$$(\boldsymbol{\sigma}', g', A', \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, A, I) \tag{143}$$

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

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

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

should halt.

$$\Xi(\boldsymbol{\sigma}, g, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_{\mathrm{g}}, A', \mathbf{o}) \tag{144}$$

$$(\boldsymbol{\sigma}', \boldsymbol{\mu}', A', ..., \mathbf{o}) \equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \tag{145}$$

$$\boldsymbol{\mu}_{\mathrm{g}} \equiv g \tag{146}$$

$$\boldsymbol{\mu}_{\mathrm{pc}} \equiv 0 \tag{147}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{148}$$

$$\boldsymbol{\mu}_{\mathrm{i}} \equiv 0 \tag{149}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{150}$$

$$\boldsymbol{\mu}_{\mathbf{o}} \equiv () \tag{151}$$

(152)

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

where

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{153}$$

$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{154}$$

$$\boldsymbol{\mu}' \equiv \boldsymbol{\mu} \text{ except:} \tag{155}$$

$$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \tag{156}$$

Note that, when we evaluate Ξ, we drop the fourth element I' and extract the remaining gas $\boldsymbol{\mu}'_{\mathrm{g}}$ from the resultant machine state $\boldsymbol{\mu}'$.

X is thus cycled (recursively here, but implementations are generally expected to use a simple iterative loop) until either Z becomes true indicating that the present state is exceptional and that the machine must be halted and any changes discarded or until H becomes a series (rather than the empty set) indicating that the machine has reached a controlled halt.

9.4.1. *Machine State.* The machine state $\boldsymbol{\mu}$ is defined as the tuple $(g, pc, \mathbf{m}, i, \mathbf{s}, \mathbf{o})$ which are the gas available, the program counter $pc \in \mathbb{N}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), the stack contents, and the returndata buffer. The memory contents $\boldsymbol{\mu}_{\mathbf{m}}$ are a series of zeroes of size 2^{256}.

For the ease of reading, the instruction mnemonics, written in small-caps (e.g. ADD), should be interpreted as their numeric equivalents; the full table of instructions and their specifics is given in Appendix H.

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

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{157}$$

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

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

(158)

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv \; & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w = \text{JUMP} \; \wedge \; \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{JUMPI} \; \wedge \; \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \; \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \text{RETURNDATACOPY} \; \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \quad \vee \\ & (\neg I_{\mathrm{w}} \; \wedge \; W(w, \boldsymbol{\mu})) \quad \vee \\ & (w = \text{SSTORE} \; \wedge \; \boldsymbol{\mu}_g \leqslant G_{\text{callstipend}}) \end{aligned}$$

where

$$\begin{aligned} W(w, \boldsymbol{\mu}) \equiv \; & w \in \{\text{CREATE}, \text{CREATE2}, \text{SSTORE}, \\ & \text{SELFDESTRUCT}\} \vee \\ & \text{LOG0} \leq w \; \wedge \; w \leq \text{LOG4} \quad \vee \\ & w = \text{CALL} \; \wedge \; \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \end{aligned} \tag{159}$$

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid, 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 JUMPDEST instruction.

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

Formally:

$$D(\mathbf{c}) \equiv D_{\mathrm{J}}(\mathbf{c}, 0) \tag{160}$$

where:

(161)

$$D_{\mathrm{J}}(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant \|\mathbf{c}\| \\ \{i\} \cup D_{\mathrm{J}}(\mathbf{c}, N(i, \mathbf{c}[i])) & \\ \qquad \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} & \\ D_{\mathrm{J}}(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases}$$

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

(162)

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

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

(163)
$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) \text{ if } & w \in \{\text{RETURN}, \text{REVERT}\} \\ () & \text{if } \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases}$$

The data-returning halt operations, RETURN and REVERT, have a special function H_{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:

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

(165) $$\Delta \equiv \alpha_w - \delta_w$$

(166) $$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta$$

(167) $$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x - \Delta]$$

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:

(168) $$\boldsymbol{\mu}'_{\text{g}} \equiv \boldsymbol{\mu}_{\text{g}} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)$$

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

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

(170) $$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}}$$

(171) $$\boldsymbol{\mu}'_{\text{i}} \equiv \boldsymbol{\mu}_{\text{i}}$$

(172) $$A' \equiv A$$

(173) $$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}$$

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 α and δ 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:

(174) $$B_{\text{t}} \geqslant 58750000000000000000000$$

(175) $$P(B_{\text{H}})_{\text{t}} < 58750000000000000000000$$

where B_{t} is the total difficulty of block B and $P(B_{\text{H}})_{\text{t}}$ is the total difficulty of its parent.

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

(176) $$B_{\text{t}} \equiv P(B_{\text{H}})_{\text{t}} + H_{\text{d}}$$

where H_{d} is the difficulty of the current block B.

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

10.1. **Post-Paris Updates.** Because the Beacon Chain generate a new slot every 12 seconds, post-*Paris* updates can be scheduled to occur at a specific timestamp. At the execution layer, the update will then happen in the first produced block after the scheduled timestamp. For example the *Shanghai* hard fork was scheduled to occur at 2023-04-12 10:27:35 UTC, on Epoch 194,048. Validators failed to propose a block during the two first slots of this Epoch, but at slot 6,209,538 a validator finally proposed the block 17,034,870, marking the transition to *Shanghai* on the execution layer.

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*.[12]

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_FORKCHOICE_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_FORKCHOICE_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_FORKCHOICE_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 three stages:

(1) executing withdrawals;
(2) validate transactions;
(3) verify state.

[12] *LMD GHOST* comprises two acronyms, "Latest Message Driven", and "Greedy Heaviest-Observed Sub-Tree".

12.1. **Executing Withdrawals.** After processing the block's transactions, the withdrawals are executed. A withdrawal is simply an increase of the recipient account's balance of the specified Gwei amount. No other balances are decreased, a withdrawal is not a transfer but a creation of funds. A withdrawal operation cannot fail and has no gas cost. We define the function E as the withdrawal state transition function:

$$E(\boldsymbol{\sigma}_w, W) \equiv \boldsymbol{\sigma}_{w+1} \tag{177}$$

$$\boldsymbol{\sigma}_{w+1} \equiv \boldsymbol{\sigma}_w \quad \text{except:} \tag{178}$$

$$\boldsymbol{\sigma}_{w+1}[W_r]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_w[W_r]_{\mathrm{b}} + (W_a \times 10^9) \tag{179}$$

Finally, we define K, the block-level state transition function for withdrawals:

$$K(\boldsymbol{\sigma}, B) \equiv E(E(\boldsymbol{\sigma}, W_0), W_1)... \tag{180}$$

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

$$B_{\mathrm{Hg}} = \ell(\mathbf{R})_{\mathrm{u}} \tag{181}$$

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

(182)
$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{if} \quad P(B_{\mathrm{H}}) = \varnothing \\ \boldsymbol{\sigma}_{\mathrm{i}} : \texttt{TRIE}(L_{\mathrm{S}}(\boldsymbol{\sigma}_{\mathrm{i}})) = P(B_{\mathrm{H}})_{\mathrm{H_r}} & \text{otherwise} \end{cases}$$

Here, $\texttt{TRIE}(L_{\mathrm{S}}(\boldsymbol{\sigma}_{\mathrm{i}}))$ means the hash of the root node of a trie of state $\boldsymbol{\sigma}_{\mathrm{i}}$; 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 Φ, the block transition function, which maps an incomplete block B to a complete block B':

$$\Phi(B) \equiv B' : \quad B' = B \quad \text{except:} \tag{183}$$

$$B'_{\mathrm{H_r}} = \texttt{TRIE}(L_{\mathrm{S}}(K(\Pi(\Gamma(B), B), B))) \tag{184}$$

As specified at the beginning of the present work, Π is the state-transition function, which is defined in terms of Υ, the transaction-evaluation function.

As previously detailed, $\mathbf{R}[n]_{\mathrm{z}}$, $\mathbf{R}[n]_{\mathbf{l}}$ and $\mathbf{R}[n]_{\mathrm{u}}$ are the nth corresponding status code, logs and cumulative gas used after each transaction ($\mathbf{R}[n]_{\mathrm{b}}$, the fourth component in the tuple, has already been defined in terms of the logs). We also define the nth state $\boldsymbol{\sigma}[n]$, 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):

$$\boldsymbol{\sigma}[n] = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases} \tag{185}$$

In the case of $B_{\mathbf{R}}[n]_{\mathrm{u}}$, we take a similar approach defining each item as the gas used in evaluating the corresponding transaction summed with the previous item (or zero, if it is the first), giving us a running total:

$$\mathbf{R}[n]_{\mathrm{u}} = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_{\mathrm{u}} & \text{otherwise} \end{cases} \tag{186}$$

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

$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) \tag{187}$$

We define $\mathbf{R}[n]_{\mathrm{z}}$ in a similar manner.

$$\mathbf{R}[n]_{\mathrm{z}} = \Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) \tag{188}$$

Finally, we define Π as the final transaction's resultant state, $\ell(\boldsymbol{\sigma})$:

$$\Pi(\boldsymbol{\sigma}, B) \equiv \ell(\boldsymbol{\sigma}) \tag{189}$$

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 single data feed.

The general pattern involves a single contract within Ethereum which, when given a message call, replies with some timely information concerning an external phenomenon. An example might be the local temperature of New York City. This would be implemented as a contract that returned that value of some known point in storage. Of course this point in storage must be maintained with the correct such temperature, and thus the second part of the pattern would be for an external server to run an Ethereum node, and immediately on discovery of a new block, creates a new valid transaction, sent to the contract, updating said value in storage. The contract's code would accept such updates only from the identity contained on said server.

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 state trie structures into the future. It should maintain an age for each node and eventually discard nodes that are neither recent enough nor checkpoints. Checkpoints, or a set of nodes in the database that allow a particular block's state trie to be traversed, could be used to place a maximum limit on the amount of computation needed in order to retrieve any state throughout the blockchain.

Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download

to act as a full 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 `https://github.com/ethereum/yellowpaper/`. An auto-generated PDF is located at `https://ethereum.github.io/yellowpaper/paper.pdf`.

References

Jacob Aron. BitCoin software finds new life. *New Scientist*, 213(2847):20, 2012. URL `http://www.sciencedirect.com/science/article/pii/S0262407912601055`.

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

Alex Beregszaszi, Paweł Bylica, Andrei Maiboroda, Alexey Akhunov, Christian Reitwiessner, and Martin Swende. EIP-3541: Reject new contract code starting with the 0xef byte, March 2021. URL `https://eips.ethereum.org/EIPS/eip-3541`.

Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. The KECCAK SHA-3 submission, 2011. URL `https://keccak.team/files/Keccak-submission-3.pdf`.

Roman Boutellier and Mareike Heinzen. Pirates, Pioneers, Innovators and Imitators. In *Growth Through Innovation*, pages 85–96. Springer, 2014. URL `https://www.springer.com/gb/book/9783319040158`.

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 2013. URL `https://github.com/ethereum/wiki/wiki/White-Paper`.

Vitalik Buterin. EIP-2: Homestead hard-fork changes, 2015. URL `https://eips.ethereum.org/EIPS/eip-2`.

Vitalik Buterin. EIP-155: Simple replay attack protection, October 2016. URL `https://eips.ethereum.org/EIPS/eip-155`.

Vitalik Buterin. EIP-1014: Skinny CREATE2, April 2018. URL `https://eips.ethereum.org/EIPS/eip-1014`.

Vitalik Buterin and Martin Swende. EIP-2929: Gas cost increases for state access opcodes, September 2020a. URL `https://eips.ethereum.org/EIPS/eip-2929`.

Vitalik Buterin and Martin Swende. EIP-2930: Optional access lists, August 2020b. URL `https://eips.ethereum.org/EIPS/eip-2930`.

Vitalik Buterin and Martin Swende. EIP-3529: Reduction in refunds, April 2021. URL `https://eips.ethereum.org/EIPS/eip-3529`.

Vitalik Buterin, Eric Conner, Rick Dudley, Matthew Slipper, Ian Norden, and Abdelhamid Bakhta. EIP-1559: Fee market change for eth 1.0 chain, 2019. URL `https://eips.ethereum.org/EIPS/eip-1559`.

Nicolas T. Courtois, Marek Grajek, and Rahul Naik. *Optimizing SHA256 in Bitcoin Mining*, pages 131–144. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-44893-9. doi: 10.1007/978-3-662-44893-9_12. URL `https://doi.org/10.1007/978-3-662-44893-9_12`.

B.A. Davey and H.A. Priestley. *Introduction to lattices and order. 2nd ed.* Cambridge: Cambridge University Press, 2nd ed. edition, 2002. ISBN 0-521-78451-4/pbk.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In 12th Annual International Cryptology Conference*, pages 139–147, 1992. URL `http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf`.

Dankrad Feist, Dmitry Khovratovich, and Marius van der Wijden. EIP-3607: Reject transactions from senders with deployed code, June 2021. URL `https://eips.ethereum.org/EIPS/eip-3607`.

Phong Vo Glenn Fowler, Landon Curt Noll. FowlerNollVo hash function, 1991. URL `http://www.isthe.com/chongo/tech/comp/fnv/index.html`.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and RSA on 8-bit CPUs. In *Cryptographic Hardware and Embedded Systems-CHES 2004*, pages 119–132. Springer, 2004. URL `https://www.iacr.org/archive/ches2004/31560117/31560117.pdf`.

Tjaden Hess, Matt Luongo, Piotr Dyraga, and James Hancock. EIP-152: Add BLAKE2 compression function 'F' precompile, October 2016. URL `https://eips.ethereum.org/EIPS/eip-152`.

Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA), 2001. URL `https://web.archive.org/web/20170921160141/http://cs.ucsb.edu/~koc/ccs130h/notes/ecdsa-cert.pdf`. Accessed 21 September 2017, but the original link was inaccessible on 19 October 2017. Refer to section 6.2 for ECDSAPUBKEY, and section 7 for ECDSASIGN and ECDSARECOVER.

Sergio Demian Lerner. Strict Memory Hard Hashing Functions, 2014. URL `http://www.hashcash.org/papers/memohash.pdf`.

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997. URL https://drive.google.com/file/d/0Bw0VXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing.

Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL http://www.bitcoin.org/bitcoin.pdf.

Meni Rosenfeld, Yoni Assia, Vitalik Buterin, m liorhakiLior, Oded Leiba, Assaf Shomer, and Eliran Zach. Colored Coins Protocol Specification, 2012. URL https://github.com/Colored-Coins/Colored-Coins-Protocol-Specification.

Markku-Juhani Saarinen and Jean-Philippe Aumasson. RFC 7693: The BLAKE2 cryptographic hash and message authentication code (MAC), November 2015. URL https://tools.ietf.org/html/rfc7693.

Simon Sprankel. Technical Basis of Digital Currencies, 2013. URL http://www.coderblog.de/wp-content/uploads/technical-basis-of-digital-currencies.pdf.

Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. URL http://firstmonday.org/ojs/index.php/fm/article/view/548.

Wei Tang. EIP-2200: Structured definitions for net gas metering, 2019. URL https://eips.ethereum.org/EIPS/eip-2200.

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gn Sirer. KARMA: A secure economic framework for peer-to-peer resource sharing, 2003. URL https://www.cs.cornell.edu/people/egs/papers/karma.pdf.

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

Micah Zoltu. EIP-2718: Typed transaction envelope, June 2020. URL https://eips.ethereum.org/EIPS/eip-2718.

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 $\mathbb{T}$:

$$\mathbb{T} \equiv \mathbb{L} \uplus \mathbb{B} \tag{190}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \ \wedge \ \forall n < \|\mathbf{t}\| : \mathbf{t}[n] \in \mathbb{T}\} \tag{191}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \ \wedge \ \forall n < \|\mathbf{b}\| : \mathbf{b}[n] \in \mathbb{O}\} \tag{192}$$

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 $\uplus$ is needed only to distinguish the empty byte array $() \in \mathbb{B}$ from the empty list $() \in \mathbb{L}$, 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 $\texttt{RLP}$ through two sub-functions, the first handling the instance when the value is a byte array, the second when it is a sequence of further values:

$$\texttt{RLP}(\mathbf{x}) \equiv \begin{cases} R_{\text{b}}(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathbb{B} \\ R_{\text{l}}(\mathbf{x}) & \text{otherwise} \end{cases} \tag{193}$$

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 $\mathbb{L}$.

Formally, we define R_{b}:

$$R_{\text{b}}(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\texttt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \texttt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 2^{64} \\ \varnothing & \text{otherwise} \end{cases} \tag{194}$$

$$\texttt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{\|\mathbf{b}\|-1} b_n \cdot 256^{\|\mathbf{b}\|-1-n} \tag{195}$$

$$(x_1, ..., x_n) \cdot (y_1, ..., y_m) = (x_1, ..., x_n, y_1, ..., y_m) \tag{196}$$

Thus $\texttt{BE}$ 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_{l}:

$$R_{\text{l}}(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad s(\mathbf{x}) \neq \varnothing \wedge \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\texttt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \texttt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{else if} \quad s(\mathbf{x}) \neq \varnothing \wedge \|s(\mathbf{x})\| < 2^{64} \\ \varnothing & \text{otherwise} \end{cases} \tag{197}$$

$$s(\mathbf{x}) \equiv \begin{cases} \texttt{RLP}(\mathbf{x}[0]) \cdot \texttt{RLP}(\mathbf{x}[1]) \cdot ... & \text{if} \quad \forall i : \texttt{RLP}(\mathbf{x}[i]) \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{198}$$

If RLP is used to encode a scalar, defined only as a non-negative integer (in $\mathbb{N}$, or in $\mathbb{N}_x$ 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:

$$\mathtt{RLP}(i : i \in \mathbb{N}) \equiv \mathtt{RLP}(\mathtt{BE}(i)) \tag{199}$$

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

$$\mathtt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if } \|\mathbf{x}\| \text{ is even} \\ (16(f(t)+1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{200}$$

$$f(t) \equiv \begin{cases} 2 & \text{if } t \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{201}$$

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 a 32-byte sequence or the empty byte sequence. It is left as an implementation consideration to store and maintain the structure of the trie in a manner that allows effective and efficient realisation of the protocol.

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

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{202}$$

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

$$\forall I \in \mathfrak{I} : I \equiv (I_0, I_1) \tag{203}$$

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

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{204}$$

$$\forall n : \quad \forall i < 2\|\mathbf{k}_n\| : \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases} \tag{205}$$

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

$$\mathtt{TRIE}(\mathfrak{I}) \equiv \mathtt{KEC}\big(\mathtt{RLP}(c(\mathfrak{I}, 0))\big) \tag{206}$$

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:

$$n(\mathfrak{I}, i) \equiv \begin{cases} () \in \mathbb{B} & \text{if } \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if } \|\mathtt{RLP}\big(c(\mathfrak{I}, i)\big)\| < 32 \\ \mathtt{KEC}\big(\mathtt{RLP}(c(\mathfrak{I}, i))\big) & \text{otherwise} \end{cases} \tag{207}$$

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:

$$c(\mathfrak{I}, i) \equiv \begin{cases} \big(\texttt{HP}(I_0[i..(\|I_0\|-1)], 1), I_1\big) & \text{if} \quad \|\mathfrak{I}\| = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \big(\texttt{HP}(I_0[i..(j-1)], 0), n(\mathfrak{I}, j)\big) & \text{if} \quad i \neq j \quad \text{where } j = \max\{x : \exists \mathbf{l} : \|\mathbf{l}\| = x \wedge \forall I \in \mathfrak{I} : I_0[0..(x-1)] = \mathbf{l}\} \\ (u(0), u(1), ..., u(15), v) & \text{otherwise} \quad \text{where } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad\qquad\qquad\quad v = \begin{cases} I_1 & \text{if} \quad \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () \in \mathbb{B} & \text{otherwise} \end{cases} \end{cases} \tag{208}$$

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

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

Appendix E. Precompiled Contracts

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

$$\Xi_{\text{PRE}}(\boldsymbol{\sigma}, g, A, I) \equiv \begin{cases} (\varnothing, 0, A, ()) & \text{if} \quad g < g_{\text{r}} \\ (\boldsymbol{\sigma}, g - g_{\text{r}}, A, \mathbf{o}) & \text{otherwise} \end{cases} \tag{209}$$

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

We define Ξ_{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 `ECDSARECOVER` and the constant `secp256k1n`. We also define $\mathbf{d}$ 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.

$$\Xi_{\text{ECREC}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{210}$$

$$g_{\text{r}} = 3000 \tag{211}$$

$$\|\mathbf{o}\| = \begin{cases} 0 & \text{if} \quad v \notin \{27, 28\} \vee r = 0 \vee r \geq \texttt{secp256k1n} \vee s = 0 \vee s \geq \texttt{secp256k1n} \\ 0 & \text{if} \quad \texttt{ECDSARECOVER}(h, v-27, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases} \tag{212}$$

$$\text{if} \quad \|\mathbf{o}\| = 32 : \tag{213}$$

$$\mathbf{o}[0..11] = 0 \tag{214}$$

$$\mathbf{o}[12..31] = \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v-27, r, s)\big)[12..31] \quad \text{where:} \tag{215}$$

$$\mathbf{d}[0..(\|I_{\mathbf{d}}\|-1)] = I_{\mathbf{d}} \tag{216}$$

$$\mathbf{d}[\|I_{\mathbf{d}}\|..] = (0, 0, ...) \tag{217}$$

$$h = \mathbf{d}[0..31] \tag{218}$$

$$v = \mathbf{d}[32..63] \tag{219}$$

$$r = \mathbf{d}[64..95] \tag{220}$$

$$s = \mathbf{d}[96..127] \tag{221}$$

We define $\Xi_{\texttt{SHA256}}$ and $\Xi_{\texttt{RIP160}}$ 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.

$$\Xi_{\texttt{SHA256}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{222}$$
$$g_{\text{r}} = 60 + 12\left\lceil \frac{\|I_{\mathbf{d}}\|}{32} \right\rceil \tag{223}$$
$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{224}$$
$$\Xi_{\texttt{RIP160}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{225}$$
$$g_{\text{r}} = 600 + 120\left\lceil \frac{\|I_{\mathbf{d}}\|}{32} \right\rceil \tag{226}$$
$$\mathbf{o}[0..11] = 0 \tag{227}$$
$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{228}$$

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

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{229}$$
$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{230}$$

The fourth contract, the identity function $\Xi_{\texttt{ID}}$ simply defines the output as the input:

$$\Xi_{\texttt{ID}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{231}$$
$$g_{\text{r}} = 15 + 3\left\lceil \frac{\|I_{\mathbf{d}}\|}{32} \right\rceil \tag{232}$$
$$\mathbf{o} = I_{\mathbf{d}} \tag{233}$$

The fifth contract performs arbitrary-precision exponentiation under modulo. Here, 0^0 is taken to be one, and $x \bmod 0$ 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.

$$\Xi_{\texttt{EXPMOD}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{234}$$
$$g_{\text{r}} = \max\left(200, \left\lfloor \frac{f\big(\max(\ell_{\text{M}}, \ell_{\text{B}})\big)\max(\ell'_{\text{E}}, 1)}{G_{\text{quaddivisor}}} \right\rfloor\right) \tag{235}$$
$$G_{\text{quaddivisor}} \equiv 3 \tag{236}$$
$$f(x) \equiv \left\lceil \frac{x}{8} \right\rceil^2 \tag{237}$$
$$\ell'_{\text{E}} = \begin{cases} 0 & \text{if } \ell_{\text{E}} \le 32 \wedge E = 0 \\ \lfloor \log_2(E) \rfloor & \text{if } \ell_{\text{E}} \le 32 \wedge E \neq 0 \\ 8(\ell_{\text{E}} - 32) + \lfloor \log_2(i[(96+\ell_{\text{B}})..(127+\ell_{\text{B}})]) \rfloor & \text{if } 32 < \ell_{\text{E}} \wedge i[(96+\ell_{\text{B}})..(127+\ell_{\text{B}})] \neq 0 \\ 8(\ell_{\text{E}} - 32) & \text{otherwise} \end{cases} \tag{238}$$
$$\mathbf{o} = \left(B^E \bmod M\right) \in \mathbb{N}_{8\ell_{\text{M}}} \tag{239}$$
$$\ell_{\text{B}} \equiv i[0..31] \tag{240}$$
$$\ell_{\text{E}} \equiv i[32..63] \tag{241}$$
$$\ell_{\text{M}} \equiv i[64..95] \tag{242}$$
$$B \equiv i[96..(95+\ell_{\text{B}})] \tag{243}$$
$$E \equiv i[(96+\ell_{\text{B}})..(95+\ell_{\text{B}}+\ell_{\text{E}})] \tag{244}$$
$$M \equiv i[(96+\ell_{\text{B}}+\ell_{\text{E}})..(95+\ell_{\text{B}}+\ell_{\text{E}}+\ell_{\text{M}})] \tag{245}$$
$$i[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases} \tag{246}$$

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

$$p \equiv 21888242871839275222246405745257275088696311157297823662689037894645226208583 \tag{247}$$
$$q \equiv 21888242871839275222246405745257275088548364400416034343698204186575808495617 \tag{248}$$

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

$$C_1 \equiv \{(X, Y) \in F_\mathrm{p} \times F_\mathrm{p} \mid Y^2 = X^3 + 3\} \cup \{(0, 0)\} \tag{249}$$

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

$$\begin{aligned}(X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases}(X, Y) & \text{if } X_1 \neq X_2\\ (0, 0) & \text{otherwise}\end{cases}\\ \lambda &\equiv \frac{Y_2 - Y_1}{X_2 - X_1}\\ X &\equiv \lambda^2 - X_1 - X_2\\ Y &\equiv \lambda(X_1 - X) - Y_1\end{aligned} \tag{250}$$

In the case where $(X_1, Y_1) = (X_2, Y_2)$, we define $+$ on C_1 with

$$\begin{aligned}(X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases}(X, Y) & \text{if } Y_1 \neq 0\\ (0, 0) & \text{otherwise}\end{cases}\\ \lambda &\equiv \frac{3X_1^2}{2Y_1}\\ X &\equiv \lambda^2 - 2X_1\\ Y &\equiv \lambda(X_1 - X) - Y_1\end{aligned} \tag{251}$$

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

$$n \cdot P \equiv (0, 0) + \underbrace{P + \cdots + P}_{n} \tag{252}$$

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 $\log_{P_1}(P)$ to be the smallest natural number n satisfying $n \cdot P_1 = P$. $\log_{P_1}(P)$ 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

$$C_2 \equiv \{(X, Y) \in F_{p^2} \times F_{p^2} \mid Y^2 = X^3 + 3(i + 9)^{-1}\} \cup \{(0, 0)\} \tag{253}$$

We define a binary operation $+$ and scalar multiplication $\cdot$ with the same equations (250), (251) and (252). $(C_2, +)$ is also known to be a group. We define P_2 in C_2 with

$$\begin{aligned}P_2 \equiv\ &(11559732032986387107991004021392285783925812861821192530917403151452391805634 \times i\\ &+10857046999023057135944570762232829481370756359578518086990519993285655852781,\\ &4082367875863433681332203403145435568316851327593401208105741076214120093531 \times i\\ &+8495653923123431417604973247489272438418190587263600148770280649306958101930)\end{aligned} \tag{254}$$

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 $\log_{P_2}(P)$ be the smallest natural number n satisfying $n \cdot P_2 = P$. With this definition, $\log_{P_2}(P)$ is at most $q - 1$.

Let G_T be the multiplicative abelian group underlying $F_{q^{12}}$. It is known that a non-degenerate bilinear map $e : G_1 \times G_2 \to G_T$ 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

$$\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q \tag{255}$$

$$\prod_{i=0}^{k} e(a_i, b_i) = P_T \tag{256}$$

Thus the pairing operation provides a method to verify (255).

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

$$\delta_\mathrm{p}(\mathbf{x}) \equiv \begin{cases}\mathbf{x} & \text{if } \mathbf{x} < p\\ \varnothing & \text{otherwise}\end{cases} \tag{257}$$

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

$$\delta_1(\mathbf{x}) \equiv \begin{cases} g_1 & \text{if } g_1 \in G_1 \\ \varnothing & \text{otherwise} \end{cases} \tag{258}$$

$$g_1 \equiv \begin{cases} (x, y) & \text{if } x \neq \varnothing \wedge y \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{259}$$

$$x \equiv \delta_{\mathrm{p}}(\mathbf{x}[0..31]) \tag{260}$$

$$y \equiv \delta_{\mathrm{p}}(\mathbf{x}[32..63]) \tag{261}$$

A 128 byte data $\mathbf{x} \in \mathbf{B}_{1024}$ might and might not represent an element of G_2.

$$\delta_2(\mathbf{x}) \equiv \begin{cases} g_2 & \text{if } g_2 \in G_2 \\ \varnothing & \text{otherwise} \end{cases} \tag{262}$$

$$g_2 \equiv \begin{cases} ((x_0 i + y_0), (x_1 i + y_1)) & \text{if } x_0 \neq \varnothing \wedge y_0 \neq \varnothing \wedge x_1 \neq \varnothing \wedge y_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{263}$$

$$x_0 \equiv \delta_{\mathrm{p}}(\mathbf{x}[0..31]) \tag{264}$$

$$y_0 \equiv \delta_{\mathrm{p}}(\mathbf{x}[32..63]) \tag{265}$$

$$x_1 \equiv \delta_{\mathrm{p}}(\mathbf{x}[64..95]) \tag{266}$$

$$y_1 \equiv \delta_{\mathrm{p}}(\mathbf{x}[96..127]) \tag{267}$$

We define $\Xi_{\texttt{SNARKV}}$ as a precompiled contract which checks if (255) holds, for intended use in zkSNARK verification.

$$\Xi_{\texttt{SNARKV}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{268}$$

$$\Xi_{\texttt{SNARKV}}(\boldsymbol{\sigma}, g, A, I) = (\varnothing, 0, A, ()) \quad \text{if } F \tag{269}$$

$$F \equiv (\|I_{\mathbf{d}}\| \bmod 192 \neq 0 \vee (\exists j.\ a_{\mathrm{j}} = \varnothing \vee b_{\mathrm{j}} = \varnothing)) \tag{270}$$

$$k = \frac{\|I_{\mathbf{d}}\|}{192} \tag{271}$$

$$g_{\mathrm{r}} = 34000k + 45000 \tag{272}$$

$$\mathbf{o}[0..31] \equiv \begin{cases} 0x0000000000000000000000000000000000000000000000000000000000000001 & \text{if } v \wedge \neg F \\ 0x0000000000000000000000000000000000000000000000000000000000000000 & \text{if } \neg v \wedge \neg F \end{cases} \tag{273}$$

$$v \equiv (\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q) \tag{274}$$

$$a_1 \equiv \delta_1(I_{\mathbf{d}}[0..63]) \tag{275}$$

$$b_1 \equiv \delta_2(I_{\mathbf{d}}[64..191]) \tag{276}$$

$$\vdots$$

$$a_k \equiv \delta_1(I_{\mathbf{d}}[(\|I_{\mathbf{d}}\| - 192)..(\|I_{\mathbf{d}}\| - 129)]) \tag{277}$$

$$b_k \equiv \delta_2(I_{\mathbf{d}}[(\|I_{\mathbf{d}}\| - 128)..(\|I_{\mathbf{d}}\| - 1)]) \tag{278}$$

We define a precompiled contract for addition on G_1.

$$\Xi_{\texttt{BN\_ADD}} \equiv \Xi_{\texttt{BN\_PRE}} \quad \text{except:} \tag{279}$$

$$\Xi_{\texttt{BN\_ADD}}(\boldsymbol{\sigma}, g, A, I) = (\varnothing, 0, A, ()) \quad \text{if } x = \varnothing \vee y = \varnothing \tag{280}$$

$$g_{\mathrm{r}} = 150 \tag{281}$$

$$\mathbf{o} \equiv \delta_1^{-1}(x + y) \quad \text{where } + \text{ is the group operation in } G_1 \tag{282}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[0..63]\right) \tag{283}$$

$$y \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[64..127]\right) \tag{284}$$

$$\bar{I}_{\mathbf{d}}[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases} \tag{285}$$

We define a precompiled contract for scalar multiplication on G_1, where $\bar{I}_{\mathbf{d}}$ is defined in (285).

$$\Xi_{\texttt{BN\_MUL}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{286}$$

$$\Xi_{\texttt{BN\_MUL}}(\boldsymbol{\sigma}, g, A, I) = (\varnothing, 0, A, ()) \quad \text{if } x = \varnothing \tag{287}$$

$$g_{\mathrm{r}} = 6000 \tag{288}$$

$$\mathbf{o} \equiv \delta_1^{-1}(n \cdot x) \quad \text{where } \cdot \text{ is the scalar multiplication in } G_1 \tag{289}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[0..63]\right) \tag{290}$$

$$n \equiv \bar{I}_{\mathbf{d}}[64..95] \tag{291}$$

E.2. **BLAKE2 Precompiled Contract.** EIP-152 by Hess et al. [2016] defines $\Xi_{\texttt{BLAKE2\_F}}$ as a precompiled contract implementing the compression function `F` used in the BLAKE2 cryptographic hashing algorithm. The `F` compression function is specified in RFC 7693 by Saarinen and Aumasson [2015].

$$\Xi_{\texttt{BLAKE2\_F}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{292}$$

$$\Xi_{\texttt{BLAKE2\_F}}(\boldsymbol{\sigma}, g, A, I) = (\varnothing, 0, A, ()) \quad \text{if } \|I_{\mathbf{d}}\| \neq 213 \vee f \notin \{0, 1\} \tag{293}$$

$$g_{\text{r}} = r \tag{294}$$

$$\mathbf{o} \equiv \texttt{LE}_8(h'_0) \cdot ... \cdot \texttt{LE}_8(h'_7) \tag{295}$$

$$(h'_0, \ldots, h'_7) \equiv \texttt{F}(h, m, t_{\text{low}}, t_{\text{high}}, f) \quad \text{with } r \text{ rounds and } w = 64 \tag{296}$$

$$\texttt{BE}_4(r) \equiv I_{\mathbf{d}}[0..4] \tag{297}$$

$$\texttt{LE}_8(h_0) \equiv I_{\mathbf{d}}[4..12] \tag{298}$$

$$\ldots \tag{299}$$

$$\texttt{LE}_8(h_7) \equiv I_{\mathbf{d}}[60..68] \tag{300}$$

$$\texttt{LE}_8(m_0) \equiv I_{\mathbf{d}}[68..76] \tag{301}$$

$$\ldots \tag{302}$$

$$\texttt{LE}_8(m_{15}) \equiv I_{\mathbf{d}}[188..196] \tag{303}$$

$$\texttt{LE}_8(t_{\text{low}}) \equiv I_{\mathbf{d}}[196..204] \tag{304}$$

$$\texttt{LE}_8(t_{\text{high}}) \equiv I_{\mathbf{d}}[204..212] \tag{305}$$

$$f \equiv I_{\mathbf{d}}[212] \tag{306}$$

where $r \in \mathbb{B}_{32}$, $\forall i \in 0..7 : h_i \in \mathbb{B}_{64}$, $\forall i \in 0..15 : m_i \in \mathbb{B}_{64}$, $t_{\text{low}} \in \mathbb{B}_{64}$, $t_{\text{high}} \in \mathbb{B}_{64}$, $f \in \mathbb{B}_8$, $\texttt{BE}_k$ is the k-byte big-endian representation—compare with(195):

$$\texttt{BE}_k(x) \equiv (b_0, b_1, ..., b_{k-1}) : x = \sum_{n=0}^{k-1} b_n \cdot 256^{k-1-n} \tag{307}$$

and $\texttt{LE}_k$ is the k-byte little-endian representation:

$$\texttt{LE}_k(x) \equiv (b_0, b_1, ..., b_{k-1}) : x = \sum_{n=0}^{k-1} b_n \cdot 256^{n} \tag{308}$$

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_{r}, which is a randomly selected positive integer (represented as a byte array of length 32 in big-endian form) in the range $[1, \texttt{secp256k1n} - 1]$.

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

$$\texttt{ECDSAPUBKEY}(p_{\text{r}} \in \mathbb{B}_{32}) \equiv p_{\text{u}} \in \mathbb{B}_{64} \tag{309}$$

$$\texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_{\text{r}} \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{310}$$

$$\texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_{\text{u}} \in \mathbb{B}_{64} \tag{311}$$

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_{r} 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:

$$0 < r < \texttt{secp256k1n} \tag{312}$$

$$0 < s < \texttt{secp256k1n} \div 2 + 1 \tag{313}$$

$$v \in \{0, 1\} \tag{314}$$

where:

$$\texttt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{315}$$

Note that this restriction on s is more stringent than restriction 212 in the $\Xi_{\texttt{ECREC}}$ precompile; see EIP-2 by Buterin [2015] for more detail.

For a given private key, p_{r}, the Ethereum address $A(p_{\text{r}})$ (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:

$$A(p_{\text{r}}) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_{\text{r}})\big)\big) \tag{316}$$

The message hash, $h(T)$, to be signed is the Keccak-256 hash of the transaction. Four different flavours of signing schemes are available:

$$L_{\mathrm{X}}(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}) & \text{if } T_{\mathrm{x}} = 0 \wedge T_{\mathrm{w}} \in \{27, 28\} \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, \beta, (), ()) & \text{if } T_{\mathrm{x}} = 0 \wedge T_{\mathrm{w}} \in \{2\beta + 35, 2\beta + 36\} \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}) & \text{if } T_{\mathrm{x}} = 1 \\ (T_{\mathrm{c}}, T_{\mathrm{n}}, T_{\mathrm{f}}, T_{\mathrm{m}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, T_{\mathbf{A}}) & \text{if } T_{\mathrm{x}} = 2 \end{cases} \tag{317}$$

where

$$\mathbf{p} \equiv \begin{cases} T_{\mathbf{i}} & \text{if } T_{\mathrm{t}} = \varnothing \\ T_{\mathbf{d}} & \text{otherwise} \end{cases}$$

$$h(T) \equiv \begin{cases} \mathtt{KEC}(\mathtt{RLP}(L_{\mathrm{X}}(T))) & \text{if } T_{\mathrm{x}} = 0 \\ \mathtt{KEC}(T_{\mathrm{x}} \cdot \mathtt{RLP}(L_{\mathrm{X}}(T))) & \text{otherwise} \end{cases} \tag{318}$$

The signed transaction $G(T, p_{\mathrm{r}})$ is defined as:

$$G(T, p_{\mathrm{r}}) \equiv T \quad \text{except:} \tag{319}$$

$$(T_{\mathrm{y}}, T_{\mathrm{r}}, T_{\mathrm{s}}) = \mathtt{ECDSASIGN}(h(T), p_{\mathrm{r}}) \tag{320}$$

Reiterating from previously:

$$T_{\mathrm{r}} = r \tag{321}$$

$$T_{\mathrm{s}} = s \tag{322}$$

and T_{w} of legacy transactions is either $27 + T_{\mathrm{y}}$ or $2\beta + 35 + T_{\mathrm{y}}$.

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

$$S(T) \equiv \mathcal{B}_{96..255}\big(\mathtt{KEC}\big(\mathtt{ECDSARECOVER}(h(T), v, T_{\mathrm{r}}, T_{\mathrm{s}})\big)\big) \tag{323}$$

$$v \equiv \begin{cases} T_{\mathrm{w}} - 27 & \text{if } T_{\mathrm{x}} = 0 \wedge T_{\mathrm{w}} \in \{27, 28\} \\ (T_{\mathrm{w}} - 35) \bmod 2 & \text{if } T_{\mathrm{x}} = 0 \wedge T_{\mathrm{w}} \in \{2\beta + 35, 2\beta + 36\} \\ T_{\mathrm{y}} & \text{if } T_{\mathrm{x}} = 1 \vee T_{\mathrm{x}} = 2 \end{cases} \tag{324}$$

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

$$\forall T : \forall p_{\mathrm{r}} : S(G(T, p_{\mathrm{r}})) \equiv A(p_{\mathrm{r}}) \tag{325}$$

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.

Name	Value	Description
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{jumpdest}	1	Amount of gas to pay for a JUMPDEST operation.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
G_{verylow}	3	Amount of gas to pay for operations of the set W_{verylow}.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{\text{warmaccess}}$	100	Cost of a warm account or storage access.
$G_{\text{accesslistaddress}}$	2400	Cost of warming up an account with the access list.
$G_{\text{accessliststorage}}$	1900	Cost of warming up a storage with the access list.
$G_{\text{coldaccountaccess}}$	2600	Cost of a cold account access.
$G_{\text{coldsload}}$	2100	Cost of a cold storage access.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	2900	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	4800	Refund given (added into refund counter) when the storage value is set to zero from non-zero. The refund amount is defined as $G_{\text{sreset}} + G_{\text{accessliststorage}}$.
$G_{\text{selfdestruct}}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{\text{codedeposit}}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
$G_{\text{initcodeword}}$	2	Paid per word of the initcode at the beginning of a deploy transaction, a CREATE, or a CREATE2 operation.
$G_{\text{callvalue}}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{\text{callstipend}}$	2300	A stipend for the called contract subtracted from $G_{\text{callvalue}}$ for a non-zero value transfer.
$G_{\text{newaccount}}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
G_{expbyte}	50	Partial payment when multiplied by the number of bytes in the exponent for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead* transition.
$G_{\text{txdatazero}}$	4	Paid for every zero byte of data or code for a transaction.
$G_{\text{txdatanonzero}}$	16	Paid for every non-zero byte of data or code for a transaction.
$G_{\text{transaction}}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
G_{logdata}	8	Paid for each byte in a LOG operation's data.
G_{logtopic}	375	Paid for each topic of a LOG operation.
$G_{\text{keccak256}}$	30	Paid for each KECCAK256 operation.
$G_{\text{keccak256word}}$	6	Paid for each word (rounded up) for input data to a KECCAK256 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{\text{blockhash}}$	20	Payment for each BLOCKHASH operation.

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:

(326)
$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv C_{\text{mem}}(\boldsymbol{\mu}'_{\text{i}}) - C_{\text{mem}}(\boldsymbol{\mu}_{\text{i}}) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SSTORE} \\ G_{\text{exp}} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{\text{exp}} + G_{\text{expbyte}} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{\text{verylow}} + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w \in W_{\text{copy}} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) + G_{\text{copy}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ C_{\text{aaccess}}(\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}, A) & \text{if} \quad w \in W_{\text{extaccount}} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{\text{logtopic}} & \text{if} \quad w = \text{LOG1} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{\text{logtopic}} & \text{if} \quad w = \text{LOG2} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{\text{logtopic}} & \text{if} \quad w = \text{LOG3} \\ G_{\text{log}} + G_{\text{logdata}} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{\text{logtopic}} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{if} \quad w \in W_{\text{call}} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if} \quad w = \text{SELFDESTRUCT} \\ G_{\text{create}} + R(\boldsymbol{\mu}_{\mathbf{s}}[2]) & \text{if} \quad w = \text{CREATE} \\ G_{\text{create}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil + R(\boldsymbol{\mu}_{\mathbf{s}}[2]) & \text{if} \quad w = \text{CREATE2} \\ G_{\text{keccak256}} + G_{\text{keccak256word}} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[1] \div 32 \rceil & \text{if} \quad w = \text{KECCAK256} \\ G_{\text{jumpdest}} & \text{if} \quad w = \text{JUMPDEST} \\ C_{\text{SLOAD}}(\boldsymbol{\mu}, A, I) & \text{if} \quad w = \text{SLOAD} \\ G_{\text{zero}} & \text{if} \quad w \in W_{\text{zero}} \\ G_{\text{base}} & \text{if} \quad w \in W_{\text{base}} \\ G_{\text{verylow}} & \text{if} \quad w \in W_{\text{verylow}} \\ G_{\text{low}} & \text{if} \quad w \in W_{\text{low}} \\ G_{\text{mid}} & \text{if} \quad w \in W_{\text{mid}} \\ G_{\text{high}} & \text{if} \quad w \in W_{\text{high}} \\ G_{\text{blockhash}} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

(327)
$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\text{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\text{pc}} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

where:

(328)
$$C_{\text{mem}}(a) \equiv G_{\text{memory}} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

(329)
$$C_{\text{aaccess}}(x, A) \equiv \begin{cases} G_{\text{warmaccess}} & \text{if} \quad x \in A_{\mathbf{a}} \\ G_{\text{coldaccountaccess}} & \text{otherwise} \end{cases}$$

with C_{CALL}, $C_{\text{SELFDESTRUCT}}$, C_{SLOAD} and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:

$W_{\text{zero}} = \{\text{STOP, RETURN, REVERT}\}$
$W_{\text{base}} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, PREVRANDAO, GASLIMIT, CHAINID, RETURNDATASIZE, POP, PC, MSIZE, GAS, BASEFEE, PUSH0}\}$
$W_{\text{verylow}} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, SHL, SHR, SAR, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH1, ..., PUSH32, DUP*, SWAP*}\}$
$W_{\text{low}} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND, SELFBALANCE}\}$
$W_{\text{mid}} = \{\text{ADDMOD, MULMOD, JUMP}\}$
$W_{\text{high}} = \{\text{JUMPI}\}$
$W_{\text{copy}} = \{\text{CALLDATACOPY, CODECOPY, RETURNDATACOPY}\}$
$W_{\text{call}} = \{\text{CALL, CALLCODE, DELEGATECALL, STATICCALL}\}$
$W_{\text{extaccount}} = \{\text{BALANCE, EXTCODESIZE, EXTCODEHASH}\}$

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, $\boldsymbol{\mu}_{\text{i}}$ in order that all accesses reference valid memory whether for read or write. Such accesses must be for non-zero number of bytes.

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

Note also that C_{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:

$$M(s, f, l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f + l) \div 32 \rceil) & \text{otherwise} \end{cases} \tag{330}$$

Another useful function is "all but one 64th" function L defined as:

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{331}$$

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

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted} \tag{332}$$

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

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.

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

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

10s: Comparison & Bitwise Logic Operations

Value	Mnemonic	δ	α	Description
0x10	LT	2	1	Less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x11	GT	2	1	Greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x12	SLT	2	1	Signed less-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] < \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x13	SGT	2	1	Signed greater-than comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] > \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$ Where all values are treated as two's complement signed 256-bit integers.
0x14	EQ	2	1	Equality comparison. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = \boldsymbol{\mu}_{\mathbf{s}}[1] \\ 0 & \text{otherwise} \end{cases}$
0x15	ISZERO	1	1	Simple not operator. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = 0 \\ 0 & \text{otherwise} \end{cases}$
0x16	AND	2	1	Bitwise AND operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\text{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\text{i}} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1]_{\text{i}}$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\text{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\text{i}} \vee \boldsymbol{\mu}_{\mathbf{s}}[1]_{\text{i}}$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\text{i}} \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_{\text{i}} \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_{\text{i}}$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\text{i}} \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_{\text{i}} = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_{\text{i}} \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i-248+8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i \geq 248 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).
0x1b	SHL	2	1	Left shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \times 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]}) \bmod 2^{256}$
0x1c	SHR	2	1	Logical right shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lfloor \boldsymbol{\mu}_{\mathbf{s}}[1] \div 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]} \rfloor$
0x1d	SAR	2	1	Arithmetic (signed) right shift operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lfloor \boldsymbol{\mu}_{\mathbf{s}}[1] \div 2^{\boldsymbol{\mu}_{\mathbf{s}}[0]} \rfloor$ Where $\boldsymbol{\mu}'_{\mathbf{s}}[0]$ and $\boldsymbol{\mu}_{\mathbf{s}}[1]$ are treated as two's complement signed 256-bit integers, while $\boldsymbol{\mu}_{\mathbf{s}}[0]$ is treated as unsigned.

20s: KECCAK256

Value	Mnemonic	δ	α	Description
0x20	KECCAK256	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{KEC}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_{\text{i}} \equiv M(\boldsymbol{\mu}_{\text{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

Value	Mnemonic	δ	α	Description
				30s: Environmental Information
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{a}}$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{b}} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{o}}$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{s}}$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{v}}$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \|I_{\mathbf{d}}\|$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{d}}\|$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{b}}\|$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo.
0x3a	GASPRICE	0	1	Get price of gas in current environment. This is the *effective gas price* defined in section 6. 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{p}}$
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \|\mathbf{b}\| & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$ where $\texttt{KEC}(\mathbf{b}) \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{b}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \|\mathbf{b}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\texttt{KEC}(\mathbf{b}) \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}}$ We assume $\mathbf{b} \equiv ()$ if $\boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}] = \varnothing$. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo. $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$

0x3d	RETURNDATASIZE	0	1	Get size of output data from the previous call from the current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\mu}_{\mathbf{o}}\|$
0x3e	RETURNDATACOPY	3	0	Copy output data from the previous call to memory. $\forall i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2] - 1\} : \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{o}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|\boldsymbol{\mu}_{\mathbf{o}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$
0x3f	EXTCODEHASH	1	1	Get hash of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}) \\ \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}]_{\mathrm{c}} & \text{otherwise} \end{cases}$ $A'_{\mathbf{a}} \equiv A_{\mathbf{a}} \cup \{\boldsymbol{\mu}_{\mathbf{s}}[0] \bmod 2^{160}\}$

40s: Block Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{\mathrm{H_p}}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ where P is the hash of a block of a particular number, up to a maximum age. 0 is left on the stack if the looked for block number is greater than or equal to the current block number or more than 256 blocks behind the current block. $P(h, n, a) \equiv \begin{cases} 0 & \text{if} \quad n > H_{\mathrm{i}} \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_{\mathrm{i}} \\ P(H_{\mathrm{p}}, n, a + 1) & \text{otherwise} \end{cases}$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hc}}$
0x42	TIMESTAMP	0	1	Get the current block's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hs}}$
0x43	NUMBER	0	1	Get the current block's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hi}}$
0x44	PREVRANDAO	0	1	Get the latest RANDAO mix of the post beacon state of the previous block. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Ha}}$
0x45	GASLIMIT	0	1	Get the current block's gas limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hl}}$
0x46	CHAINID	0	1	Get the chain ID. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \beta$
0x47	SELFBALANCE	0	1	Get balance of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathrm{b}}$
0x48	BASEFEE	0	1	Get the current block's base fee. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{Hf}}$

50s: Stack, Memory, Storage and Flow Operations

Value	Mnemonic	δ	α	Description
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ $A'_{\mathbf{K}} \equiv A_{\mathbf{K}} \cup \{(I_{\mathrm{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0])\}$ $C_{\mathrm{SLOAD}}(\boldsymbol{\mu}, A, I) \equiv \begin{cases} G_{\mathrm{warmaccess}} & \text{if } (I_{\mathrm{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0]) \in A_{\mathbf{K}} \\ G_{\mathrm{coldsload}} & \text{otherwise} \end{cases}$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $A'_{\mathbf{K}} \equiv A_{\mathbf{K}} \cup \{(I_{\mathrm{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0])\}$ $C_{\mathrm{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ and A'_{r} are specified by EIP-2200 as follows. We remind the reader that the checkpoint ("original") state $\boldsymbol{\sigma}_0$ is the state if the current transaction were to revert. Let $v_0 = \boldsymbol{\sigma}_0[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ be the original value of the storage slot. Let $v = \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$ be the current value. Let $v' = \boldsymbol{\mu}_{\mathbf{s}}[1]$ be the new value. Then: $C_{\mathrm{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv \begin{cases} 0 & \text{if } (I_{\mathrm{a}}, \boldsymbol{\mu}_{\mathbf{s}}[0]) \in A_{\mathbf{K}} \\ G_{\mathrm{coldsload}} & \text{otherwise} \end{cases}$ $+ \begin{cases} G_{\mathrm{warmaccess}} & \text{if } v = v' \ \vee \ v_0 \neq v \\ G_{\mathrm{sset}} & \text{if } v \neq v' \ \wedge \ v_0 = v \ \wedge \ v_0 = 0 \\ G_{\mathrm{sreset}} & \text{if } v \neq v' \ \wedge \ v_0 = v \ \wedge \ v_0 \neq 0 \end{cases}$ $A'_{\mathrm{r}} \equiv A_{\mathrm{r}} + \begin{cases} R_{\mathrm{sclear}} & \text{if } v \neq v' \ \wedge \ v_0 = v \ \wedge \ v' = 0 \\ r_{\mathrm{dirtyclear}} + r_{\mathrm{dirtyreset}} & \text{if } v \neq v' \ \wedge \ v_0 \neq v \\ 0 & \text{otherwise} \end{cases}$ where $r_{\mathrm{dirtyclear}} \equiv \begin{cases} -R_{\mathrm{sclear}} & \text{if } v_0 \neq 0 \ \wedge \ v = 0 \\ R_{\mathrm{sclear}} & \text{if } v_0 \neq 0 \ \wedge \ v' = 0 \\ 0 & \text{otherwise} \end{cases}$ $r_{\mathrm{dirtyreset}} \equiv \begin{cases} G_{\mathrm{sset}} - G_{\mathrm{warmaccess}} & \text{if } v_0 = v' \ \wedge \ v_0 = 0 \\ G_{\mathrm{sreset}} - G_{\mathrm{warmaccess}} & \text{if } v_0 = v' \ \wedge \ v_0 \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\mathrm{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\mathrm{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{\mathrm{pc}} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathrm{pc}}$

0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_{\mathrm{i}}$
0x5a	GAS	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathrm{g}}$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

5f, 60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x5f	PUSH0	0	1	Place 0 on the stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 0$
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{\mathrm{pc}} + 1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if } x < \lVert I_{\mathbf{b}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \dots (\boldsymbol{\mu}_{\mathrm{pc}} + 2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\lVert x \rVert - 1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \dots (\boldsymbol{\mu}_{\mathrm{pc}} + 32)\big)$ where $\boldsymbol{c}$ is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

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

90s: Exchange Operations

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

a0s: Logging Operations

For all logging operations, the state change is to append an additional log entry on to the substate's log series:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_{\mathrm{a}}, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
and to update the memory consumption counter:
$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

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

f0s: System operations

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[1] \dots (\boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] - 1)]$ $\zeta \equiv \varnothing$ $(\boldsymbol{\sigma}', g', A', z, \mathbf{o}) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, A, I_\mathrm{a}, I_\mathrm{o}, L(\boldsymbol{\mu}_\mathrm{g}), I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[0], \mathbf{i}, I_\mathrm{e}+1, \zeta, I_\mathrm{w}) & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \wedge \|\mathbf{i}\| \leqslant 49152 \\ (\boldsymbol{\sigma}, L(\boldsymbol{\mu}_\mathrm{g}), A, 0, ()) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}$ except $\boldsymbol{\sigma}^*[I_\mathrm{a}]_\mathrm{n} = \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{n} + 1$ $\boldsymbol{\mu}'_\mathrm{g} \equiv \boldsymbol{\mu}_\mathrm{g} - L(\boldsymbol{\mu}_\mathrm{g}) + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ where $x = 0$ if $z = 0$, i.e., the contract creation process failed, or $\|\mathbf{i}\| > 49152$ (initcode is longer than 49152 bytes), or $I_\mathrm{e} = 1024$ (the maximum call depth limit is reached), or $\boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = \texttt{ADDR}(I_\mathrm{a}, \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{n}, \zeta, \mathbf{i})$, the address of the newly created account (95). $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[1], \boldsymbol{\mu}_\mathbf{s}[2])$ $\boldsymbol{\mu}'_\mathbf{o} \equiv \begin{cases} () & \text{if } z = 1 \\ \mathbf{o} & \text{otherwise} \end{cases}$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_\mathrm{a}, I_\mathrm{o}, t, t, C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A^*, 0, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], \|\mathbf{o}\|\})$ $\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\boldsymbol{\mu}'_\mathbf{o} = \mathbf{o}$ $\boldsymbol{\mu}'_\mathrm{g} \equiv \boldsymbol{\mu}_\mathrm{g} - C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A^* \equiv A$ except $A^*_\mathbf{a} \equiv A_\mathbf{a} \cup \{t\}$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \bmod 2^{160}$ $\boldsymbol{\mu}'_\mathrm{i} \equiv M(M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ where $x = 0$ if the code execution for this operation failed, or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (not enough funds) or $I_\mathrm{e} = 1024$ (call depth limit reached); $x = 1$ otherwise. Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_\mathrm{CALL}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)$ $C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) + G_\mathrm{callstipend} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) & \text{otherwise} \end{cases}$ $C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_\mathrm{g} - C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A)), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if } \boldsymbol{\mu}_\mathrm{g} \geq C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A) \equiv C_\mathrm{aaccess}(t, A) + C_\mathrm{XFER}(\boldsymbol{\mu}) + C_\mathrm{NEW}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_\mathrm{XFER}(\boldsymbol{\mu}) \equiv \begin{cases} G_\mathrm{callvalue} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_\mathrm{NEW}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_\mathrm{newaccount} & \text{if } \texttt{DEAD}(\boldsymbol{\sigma}, t) \wedge \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_\mathrm{a}, I_\mathrm{o}, I_\mathrm{a}, t, C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A^*, 0, ()) & \text{otherwise} \end{cases}$ Note the change in the fourth parameter to the call Θ from the 2nd stack value $\boldsymbol{\mu}_\mathbf{s}[1]$ (as in CALL) to the present address I_a. This means that the recipient is in fact the same account as at present, simply that the code is overwritten.
0xf3	RETURN	2	0	Halt execution returning output data. $H_\mathrm{RETURN}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 9. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$

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 $\boldsymbol{\mu}_\mathbf{s}[2]$. As a result, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. Otherwise it is equivalent to CALL except: $(\boldsymbol{\sigma}', g', A', x, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, A^*, I_\mathrm{s}, I_\mathrm{o}, I_\mathrm{a}, t, C_{\mathrm{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), & \text{if} \quad I_\mathrm{e} < 1024 \\ \quad I_\mathrm{p}, 0, I_\mathrm{v}, \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \\ (\boldsymbol{\sigma}, C_{\mathrm{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}, A), A^*, 0, ()) & \text{otherwise} \end{cases}$ Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. 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 $\zeta \equiv \boldsymbol{\mu}_\mathbf{s}[3]$.
0xfa	STATICCALL	6	1	Static message-call into an account. Exactly equivalent to CALL except: The argument $\boldsymbol{\mu}_\mathbf{s}[2]$ is replaced with 0. The deeper argument $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ are respectively replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. The last argument of Θ is $\bot$.
0xfd	REVERT	2	0	Halt execution reverting state changes but returning data and remaining gas. $H_{\mathrm{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ The effect of this operation is described in (152). For the gas calculation, we use the memory expansion function, $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$
0xfe	INVALID	∅	∅	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. Readers should be warned, since the *Shanghai* update, this opcode has been marked as deprecated. Its behavior might be subject to breaking change in an upcoming update. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_\mathrm{a}\}$ $A'_\mathbf{a} \equiv A_\mathbf{a} \cup \{r\}$ $\boldsymbol{\sigma}'[r] \equiv \begin{cases} \varnothing & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \ \wedge \ \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} = 0 \\ (\boldsymbol{\sigma}[r]_\mathrm{n}, \boldsymbol{\sigma}[r]_\mathrm{b} + \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}, \boldsymbol{\sigma}[r]_\mathbf{s}, \boldsymbol{\sigma}[r]_\mathrm{c}) & \text{if} \quad r \neq I_\mathrm{a} \\ (\boldsymbol{\sigma}[r]_\mathrm{n}, 0, \boldsymbol{\sigma}[r]_\mathbf{s}, \boldsymbol{\sigma}[r]_\mathrm{c}) & \text{otherwise} \end{cases}$ where $r = \boldsymbol{\mu}_\mathbf{s}[0] \bmod 2^{160}$ $\boldsymbol{\sigma}'[I_\mathrm{a}]_\mathrm{b} = 0$ $C_{\mathrm{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{\mathrm{selfdestruct}} + \begin{cases} 0 & \text{if} \quad r \in A_\mathbf{a} \\ G_{\mathrm{coldaccountaccess}} & \text{otherwise} \end{cases} + \begin{cases} G_{\mathrm{newaccount}} & \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}, r) \wedge \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \neq 0 \\ 0 & \text{otherwise} \end{cases}$

Appendix I. Genesis Block

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

$$(333) \qquad \big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 0, 0_{2048}, 2^{34}, 0, 0, 3141592, \mathit{time}, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), ()\big)$$

Where 0_{256} refers to the parent hash, a 256-bit hash which is all zeroes; 0_{160} refers to the beneficiary address, a 160-bit hash which is all zeroes; 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 ommers and transactions are empty and represented by (). $\texttt{KEC}\big((42)\big)$ 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. $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ 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:

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

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

$$E_{\text{epoch}}(H_{\text{i}}) = \left\lfloor \frac{H_{\text{i}}}{J_{\text{epoch}}} \right\rfloor \tag{334}$$

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_{mixbytes}, for the dataset, and $J_{\text{hashbytes}}$ for the cache. Let $d_{\text{size}} = \|\mathbf{d}\|$ be the size of the dataset. Which is calculated using

$$d_{\text{size}} = E_{\text{prime}}(J_{\text{datasetinit}} + J_{\text{datasetgrowth}} \cdot E_{\text{epoch}} - J_{\text{mixbytes}}, J_{\text{mixbytes}}) \tag{335}$$

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

$$c_{\text{size}} = E_{\text{prime}}(J_{\text{cacheinit}} + J_{\text{cachegrowth}} \cdot E_{\text{epoch}} - J_{\text{hashbytes}}, J_{\text{hashbytes}}) \tag{336}$$

$$E_{\text{prime}}(x, y) = \begin{cases} x & \text{if} \quad x/y \in \mathbb{N} \\ E_{\text{prime}}(x - 2 \cdot y, y) & \text{otherwise} \end{cases} \tag{337}$$

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

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:

$$\mathbf{s} = C_{\text{seedhash}}(H_{\text{i}}) \tag{338}$$

$$C_{\text{seedhash}}(H_{\text{i}}) = \begin{cases} \mathbf{0}_{32} & \text{if} \quad E_{\text{epoch}}(H_{\text{i}}) = 0 \\ \texttt{KEC}(C_{\text{seedhash}}(H_{\text{i}} - J_{\text{epoch}})) & \text{otherwise} \end{cases} \tag{339}$$

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

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

We define the array $\mathbf{c}_i$, consisting of 64 single bytes, as the ith element of the initial cache:

$$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{if} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{otherwise} \end{cases} \tag{340}$$

Therefore $\mathbf{c}'$ can be defined as

$$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n \tag{341}$$

$$n = \left\lfloor \frac{c_{\text{size}}}{J_{\text{hashbytes}}} \right\rfloor \tag{342}$$

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

$$\mathbf{c} = E_{\text{cacherounds}}(\mathbf{c}', J_{\text{cacherounds}}) \tag{343}$$

$$E_{\text{cacherounds}}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{if} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{if} \quad y = 1 \\ E_{\text{cacherounds}}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{otherwise} \end{cases} \tag{344}$$

Where a single round modifies each subset of the cache as follows:

$$E_{\text{RMH}}(\mathbf{x}) = \big(E_{\text{rmh}}(\mathbf{x}, 0), E_{\text{rmh}}(\mathbf{x}, 1), ..., E_{\text{rmh}}(\mathbf{x}, n - 1)\big) \tag{345}$$

$$E_{\text{rmh}}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i - 1 + n) \bmod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \bmod n]) \quad \text{with} \quad \mathbf{x}' = \mathbf{x} \quad \text{except} \quad \mathbf{x}'[j] = E_{\text{rmh}}(\mathbf{x}, j) \quad \forall \quad j < i \tag{346}$$

J.4.3. *Full dataset calculation.* Essentially, we combine data from J_{parents} pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset is then generated by a number of items, each $J_{\text{hashbytes}}$ bytes in size:

$$\mathbf{d}[i] = E_{\text{datasetitem}}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{\text{size}}}{J_{\text{hashbytes}}} \right\rfloor \tag{347}$$

In order to calculate the single item we use an algorithm inspired by the FNV hash (Glenn Fowler [1991]) in some cases as a non-associative substitute for XOR.

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (\text{0x01000193} \oplus \mathbf{y})) \bmod 2^{32} \tag{348}$$

The single item of the dataset can now be calculated as:

$$E_{\text{datasetitem}}(\mathbf{c}, i) = E_{\text{parents}}(\mathbf{c}, i, -1, \varnothing) \tag{349}$$

$$E_{\text{parents}}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{\text{parents}}(\mathbf{c}, i, p + 1, E_{\text{mix}}(\mathbf{m}, \mathbf{c}, i, p + 1)) & \text{if} \quad p < J_{\text{parents}} - 2 \\ E_{\text{mix}}(\mathbf{m}, \mathbf{c}, i, p + 1) & \text{otherwise} \end{cases} \tag{350}$$

$$E_{\text{mix}}(\mathbf{m}, \mathbf{c}, i, p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \bmod c_{\text{size}}] \oplus i) & \text{if} \quad p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \bmod \lfloor J_{\text{hashbytes}} / J_{\text{wordbytes}} \rfloor]) \bmod c_{\text{size}}]\big) & \text{otherwise} \end{cases} \tag{351}$$

J.5. **Proof-of-work function.** Essentially, we maintain a "mix" J_{mixbytes} bytes wide, and repeatedly sequentially fetch J_{mixbytes} bytes from the full dataset and use the E_{FNV} function to combine it with the mix. J_{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.

If the output of this algorithm is below the desired target, then the nonce is valid. Note that the extra application of KEC 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:

$$\texttt{PoW}(H_{\not H}, H_{\text{n}}, \mathbf{d}) = \{\mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_{\text{H}}(H_{\not H}))), H_{\text{n}}, \mathbf{d}), \texttt{KEC}(\mathbf{s}_{\text{h}}(\texttt{KEC}(\texttt{RLP}(L_{\text{H}}(H_{\not H}))), H_{\text{n}}) + \mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_{\text{H}}(H_{\not H}))), H_{\text{n}}, \mathbf{d}))\} \tag{352}$$

With $H_{\not H}$ being the hash of the header without the nonce. The compressed mix $\mathbf{m}_{\text{c}}$ is obtained as follows:

$$\mathbf{m}_{\text{c}}(\mathbf{h}, \mathbf{n}, \mathbf{d}) = E_{\text{compress}}(E_{\text{accesses}}(\mathbf{d}, \sum_{i=0}^{n_{\text{mix}}} \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), -1), -4) \tag{353}$$

The seed hash being:

$$\mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}) = \texttt{KEC512}(\mathbf{h} + E_{\text{revert}}(\mathbf{n})) \tag{354}$$

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

$$E_{\text{revert}}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{355}$$

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

The dataset $\mathbf{d}$ is obtained as described in section J.4.3.

The number of replicated sequences in the mix is:

$$n_{\text{mix}} = \left\lfloor \frac{J_{\text{mixbytes}}}{J_{\text{hashbytes}}} \right\rfloor \tag{356}$$

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

$$E_{\text{accesses}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = \begin{cases} E_{\text{mixdataset}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) & \text{if} \quad i = J_{\text{accesses}} - 2 \\ E_{\text{accesses}}(E_{\text{mixdataset}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i), \mathbf{s}, i + 1) & \text{otherwise} \end{cases} \tag{357}$$

$$E_{\text{mixdataset}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = E_{\text{FNV}}(\mathbf{m}, E_{\text{newdata}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)) \tag{358}$$

E_{newdata} returns an array with n_{mix} elements:

$$E_{\text{newdata}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)[j] = \mathbf{d}[E_{\text{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \bmod \left\lfloor \frac{J_{\text{mixbytes}}}{J_{\text{wordbytes}}} \right\rfloor]) \bmod \left\lfloor \frac{d_{\text{size}}/J_{\text{hashbytes}}}{n_{\text{mix}}} \right\rfloor \cdot n_{\text{mix}} + j] \quad \forall \quad j < n_{\text{mix}} \tag{359}$$

The mix is compressed as follows:

$$E_{\text{compress}}(\mathbf{m}, i) = \begin{cases} \mathbf{m} & \text{if} \quad i \geqslant \|\mathbf{m}\| - 8 \\ E_{\text{compress}}(E_{\text{FNV}}(E_{\text{FNV}}(E_{\text{FNV}}(\mathbf{m}[i+4], \mathbf{m}[i+5]), \mathbf{m}[i+6]), \mathbf{m}[i+7]), i+8) & \text{otherwise} \end{cases} \tag{360}$$

Appendix K. Anomalies on the Main Network

K.1. **Deletion of an Account Despite Out-of-gas.** At block 2675119, in the transaction 0xcf416c536ec1a19ed1fb89e4ec7ffb3cf73aa413b3aa9b77d60e4fd81a4296ba, an account at address 0x03 was called and an out-of-gas occurred during the call. Against the equation (209), this added 0x03 in the set of touched addresses, and this transaction turned $\boldsymbol{\sigma}[0x03]$ into $\varnothing$.

Appendix L. List of mathematical symbols

Symbol	Latex Command	Description
$\bigvee$	`\bigvee`	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]).