以太坊：一种安全去中心化的通用交易账本

EIP-150 版本 (fce8263 - 2017-08-09)

原文作者: DR. GAVIN WOOD, GAVIN@ETHCORE.IO
译者: 猿哥, ROOT@CXYYM.COM, 微信 YUANGE1024; 高天露, TIANLU.JORDEN.GAO@GMAIL.COM

Abstract. 区块链对交易数据加密以保证安全, 已经通过一系列项目展示了它的实用性, 尤其是比特币。每一个这个的项目都可以看作是一个基于去中心化的单实例且拥有计算资源的应用。我们称这种模式为可以共享状态的单例状态机。

以太坊以更广义的方式实现了这种模式。它提供了大量的资源, 每一个资源都拥有独立的状态和操作码, 并且可以通过消息传递方式和其它资源交互。我们讨论了它的设计、实现难题、它提供的机会以及以后可能有的一些问题。

1. 简介

随着互联网连接了世界上绝大多数地方，全球信息共享的成本越来越低。比特币网络通过共识机制、自愿遵守的社会合约, 实现一个去中心化的价值转移系统且可以在全球范围内自由使用, 这样的技术改革展示了它的巨大力量。这样的系统可以说是加密安全、基于交易的状态机的一种具体应用。后续类似这样的系统, 如域名币（Namecoin）, 从最原先的" 货币应用" 发展到了其它应用, 虽然它只是其中很简单的一种应用。

以太坊是一个尝试实现通用性技术的项目, 所有基于交易的状态机都可以被构建。而且以太坊致力于为开发者提供主流一个紧凑的、整合的端到端系统, 这个系统提供了一种可信的消息传递计算框架让开发者以一种前所未有的范式来构建软件。

1.1. 驱动因素.

这个项目有很多目标, 其中最重要的目标是为了促成不信任对方的个体之间的交易。这些不信任可能是因为地理位置分离、接口对接难度, 或者是不兼容、不称职、不情愿、支出、不确定、不方便, 或现有法律系统的腐败。于是我们想用一个丰富且清晰的语言去实现一个状态变化的系统，期望协议可以自动被执行, 我们可以为此提供一种实现方式。

这个提议系统中的交易, 有一些在现实世界中并不常见的属性。审判廉洁，在现实世界往往很难找到，但对公正的算法解释器是天然的; 透明，或者说通过交易日志和规则或代码指令能够清晰的看见到状态变化或者判决, 但因为人类语言的模糊性、信息的缺乏以及老的偏见难以撼动, 导致基于人的系统中从来没有完美实现透明。

总的来说，我希望能提供一个系统，能够保证用户无论是和其他个体、系统还是组织交互, 都能对可能的结果及产生结果的过程完全信任。

1.2. 前人工作.

Buterin [2013a] 在 2013 年 9 月下旬第一次提出了这种系统的核心机制。虽然现在发展出了多种方案, 但最关键的部分, 具备图灵完备语言且不受限制的内部交易存贮容量的区块链, 仍未变化。

Dwork and Naor [1992] 提出了一种使用计算支出的密码学证明方式 (proof-of-work, 工作量证明) 在互联网上传递信号值。信号值用作阻挡垃圾邮件，而不是任何一种货币, 但展示了一个基本的数据通道可以承载强大经济信号的可能性, 允许接受者无需依赖信任而做出物理断言。Back [2002] 后来设计了一个类似的系统。

Vishnumurthy et al. [2003] 最早使用工作量证明作为强大的经济信号保证货币安全。在这个案例中，代币用作检查点对点（peer-to-peer,p2p）文件交易，同时保证“消费者”能支付给为他们提供服务的“供应商”。这种通过工作量证明的安全模型逐步扩展, 包括使用电子签名和账本技术, 以保证历史记录不被篡改, 怀有恶意的用户不能进行欺诈支付或不公平的抱怨服务。五年后（2008 年），中本聪 Nakamoto [2008] 介绍了另一种更广泛的工作量证明安全价值代币。这个项目的成果比特币, 成为了第一个被全球广泛认可的去中心化交易账本。

由于比特币的成功, 竞争币 (alt-coins) 开始兴起, 通过改变比特币的协议去创建了大量的其他数字货币。比较知名的有莱特币 (Litecoin) 和素数币 (Primecoin) , 参见 Sprankel [2013] 。一些项目使用比特币的核心机制并重新改造以应用在其它领域, 例如域名币（Namecoin）, 致力于提供一个去中心化的名字解析系统, 参见 Aron [2012] 。

其它在比特币网络之上构建的项目, 也是依赖巨大的系统价值和巨大的算力来保证共识机制。万事达币（Mastercoin）项目, 在比特币协议之上, 通过一系列基于核心协议的的辅助插件, 构建一个包含许多高级功能的富协议, 参见 Willett [2013]。彩色币（Coloured Coins, 参见 Rosenfeld [2012], 采用了类似的但更简化的协议, 以实现比特币基础货币的可替代性, 并允许通过色度钱包（“chroma-wallet”）来创建和跟踪代币。

其它一些工作通过放弃中心化来进行。瑞波币（Ripple）, 参见 Boutellier 和 Heinzen [2014], 试图去创建一个货币兑换的联邦系统（“federated ” system）和一个新的金融清算系统。这个系统展示了放弃去中心化特性可以获得性能上的提升。

Szabo [1997] 和 Miller [1997] 进行了智能合约（smart contract）的早期工作。大约在上世纪 90 年代，人们逐渐认识到协议算法的执行可以成为人类合作的重要力量。虽然当时没有这样的系统, 但可以预见未来的法律将会受到这种系统的影响。基于此, 以太坊或许可以成为这种密码学-法律系统的通用实现。

2. 区块链

以太坊在整体上可以看成是一个基于交易的状态机：我们起始于一个创世块（Genesis）状态，然后随着交易的执行状态逐步改变一直到最终状态, 这个最终状态是以太坊世界的权威版本。状态中包含的信息有: 账户余额、名誉度、信誉度、现实世界的附属数据等; 简而言之，能包含电脑可以描绘的任何信息。因此，交易是连接两个状态的有效桥梁；“有效”非常重要—因为无效的状态改变远超过有效的状态改变。例如：无效的状态改变可能是减少账号余额，但是没有在其它账户上加上同等的额度。一个有效的状态转换是通过交易进行的, 表达式如下:

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

Υ 是以太坊状态转换函数。在以太坊中, Υ 和 $\boldsymbol{\sigma}$ 比已有的任何比较系统都强; Υ 可以执行任意计算, 而 $\boldsymbol{\sigma}$ 可以存贮交易中的任意状态。

区块中记录着交易信息; 区块之间通过密码学哈希 (hash) 链接起来。区块链就像一个分类账, 将一系列交易记录在一起, 并且连接上一个区块及最终状态 (并没有直接保存最终状态本身—否则整个区块链就太大了)。系统激励节点去挖矿, 挖矿激励时, 有执行状态转移函数, 增加挖矿者的账户余额。

挖矿是和其它潜在区块竞争一系列交易 (一个区块) 的记账权。它是通过密码安全证明的方式来实现的。这个机制称为工作量证明, 会在 11.5 详细讨论。

公式如下:

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

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

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

其中 Ω 是区块定稿状态转换函数 (这个函数奖励一个特定的账户); B 表示包含一系列交易的区块; Π 是区块级的状态转换函数。

上述是区块链的基本内容, 这个模型不仅是以太坊的基础, 还是迄今为止所有基于共识的去中心化交易系统的基础。

2.1. 面值. 为了激励网络中的计算, 需要定义一种转账方法。以太坊设计了一个内置货币以太币 (Ether), ETH 是大家所熟知的符号, 有时用 Ð 表示。以太币最小的面额是 Wei(伟), 所有货币值都以 Wei 的整数倍记录。一个以太币被定义成位。一个以太币等于 10^{18} Wei 。不同的面值如下表:

倍数	面值
10^0	Wei(伟))
10^{12}	Szabo(萨博)
10^{15}	Finney(芬尼)
10^{18}	Ether(以太)

在整个工作中, 任何涉及到价值、以太币相关的、货币、余额或者支付, 都以 Wei 作为单位来计算。

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

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

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

Sometimes, a path follows a new protocol from a particular height. This document describes one version of the protocol. In order to follow back the history of a path, one might need to reference multiple versions of this document.

3. Conventions

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

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

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

For most functions, an uppercase letter is used, e.g. C, the general cost function. These may be subscripted to denote specialised variants, e.g. C_{SSTORE}, the cost function for the SSTORE operation. For specialised and possibly externally defined functions, I may format as typewriter text, e.g. the Keccak-256 hash function (as per the winning entry to the SHA-3 contest) is denoted `KEC` (and generally referred to as plain Keccak). Also `KEC512` is referring 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 positive integers and thus belong to the set $\mathbb{P}$. The set of all byte sequences is $\mathbb{B}$, formally defined in Appendix B. If such a set of sequences is restricted to those of a particular length, it is denoted with a subscript, thus the set of all byte sequences of length 32 is named $\mathbb{B}_{32}$ and the set of all positive integers smaller than 2^{256} is named $\mathbb{P}_{256}$. This is formally defined in section 4.3.

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

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

When considering variants of existing values, I follow the rule that within a given scope for definition, if we assume that the unmodified 'input' value be denoted by the placeholder $\square$ then the modified and utilisable value is denoted as $\square'$, and intermediate values would be $\square^*$,

$\square^{**}$ &c. On very particular occasions, in order to maximise readability and only if unambiguous in meaning, I may use alpha-numeric subscripts to denote intermediate values, especially those of particular note.

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

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

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

4. Blocks, State and Transactions

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

4.1. World State.

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

The account state comprises the following four fields:

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

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

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

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

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

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

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

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

where:

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

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

If the **codeHash** field is the Keccak-256 hash of the empty string, i.e. $\boldsymbol{\sigma}[a]_c = \texttt{KEC}\big(()\big)$, 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:

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

where

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

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

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

where v is the account validity function:

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

4.2. The Transaction.

A transaction (formally, T) is a single cryptographically-signed instruction constructed by an actor externally to the scope of Ethereum. While it is assumed that the ultimate external actor will be human in nature, software tools will be used in its construction and dissemination[1]. There are two types of transactions: those which result in message calls and those which result in the creation of new accounts with associated code (known informally as 'contract creation'). Both types specify a number of common fields:

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

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

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

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

[1] 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.

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

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

Additionally, a contract creation transaction contains:

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

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

In contrast, a message call transaction contains:

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

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

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

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

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

where

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

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 $\mathbb{B}_0$:

(16)
$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if } T_t \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases}$$

4.3. **The Block.** The block in Ethereum is the collection of relevant pieces of information (known as the block *header*), H, together with information corresponding to the comprised transactions, $\mathbf{T}$, and a set of other block headers $\mathbf{U}$ that are known to have a parent equal to the present block's parent's parent (such blocks are known as *ommers*[2]). 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: The Keccak 256-bit hash of the ommers list portion of this block; formally H_o.

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

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

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

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

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

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

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

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

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

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

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

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

nonce: A 64-bit hash which proves combined with the mix-hash that a sufficient amount of computation has been carried out on this block; formally H_n.

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

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

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

[2] *ommer* is the most prevalent (not saying much) gender-neutral term to mean "sibling of parent"; see `http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms`

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

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

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

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

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

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

$$R_u \in \mathbb{P} \quad \wedge \quad R_b \in \mathbb{B}_{256} \tag{20}$$

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

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

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

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

$$M(O) \equiv \bigvee_{t \in \{O_a\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(t)\right) \tag{23}$$

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 sequence. Formally:

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \tag{24}$$

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

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

$$m(\mathbf{x}, i) \equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{27}$$

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

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

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

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

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

Furthermore:

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

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

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

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

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

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

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

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

The component types are defined thus:

$$\begin{aligned}
&H_p \in \mathbb{B}_{32} \quad \wedge \quad H_o \in \mathbb{B}_{32} \quad \wedge \quad H_c \in \mathbb{B}_{20} \quad \wedge\\
&H_r \in \mathbb{B}_{32} \quad \wedge \quad H_t \in \mathbb{B}_{32} \quad \wedge \quad H_e \in \mathbb{B}_{32} \quad \wedge\\
&H_b \in \mathbb{B}_{256} \quad \wedge \quad H_d \in \mathbb{P} \quad \wedge \quad H_i \in \mathbb{P} \quad \wedge\\
&H_l \in \mathbb{P} \quad \wedge \quad H_g \in \mathbb{P} \quad \wedge \quad H_s \in \mathbb{P}_{256} \quad \wedge\\
&H_x \in \mathbb{B} \quad \wedge \quad H_m \in \mathbb{B}_{32} \quad \wedge \quad H_n \in \mathbb{B}_8
\end{aligned} \tag{34}$$

where

$$\mathbb{B}_n = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{35}$$

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

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

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

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

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

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

$$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_i = 0 \\ \max\big(D_0, P(H)_{H_d} + x \times \varsigma_2 + \epsilon\big) & \text{otherwise} \end{cases} \tag{38}$$

where:

$$D_0 \equiv 131072 \tag{39}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_d}}{2048} \right\rfloor \tag{40}$$

$$\varsigma_2 \equiv \max\left(1 - \left\lfloor \frac{H_s - P(H)_{H_s}}{10} \right\rfloor, -99\right) \tag{41}$$

$$\epsilon \equiv \left\lfloor 2^{\lfloor H_i \div 100000 \rfloor - 2} \right\rfloor \tag{42}$$

The canonical gas limit H_l of a block of header H must fulfil the relation:

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{43}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{44}$$

$$H_l \geqslant 125000 \tag{45}$$

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

$$H_s > P(H)_{H_s} \tag{46}$$

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

The nonce, H_n, must satisfy the relations:

$$n \leqslant \frac{2^{256}}{H_d} \quad \wedge \quad m = H_m \tag{47}$$

with $(n, m) = \texttt{PoW}(H_{\not n}, H_n, \mathbf{d})$.

Where $H_{\not n}$ is the new block's header H, but *without* the nonce and mix-hash components, $\mathbf{d}$ being the current DAG, a large data set needed to compute the mix-hash, and PoW is the proof-of-work function (see section 11.5): this evaluates to an array with the first item being the mix-hash, to proof that a correct DAG has been used, and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. Given an approximately uniform distribution in the range $[0, 2^{64})$, the expected time to find a solution is proportional to the difficulty, H_d.

This is the foundation of the security of the blockchain and is the fundamental reason why a malicious node cannot propagate newly created blocks that would otherwise overwrite ("rewrite") history. Because the nonce must satisfy this requirement, and because its satisfaction depends on the contents of the block and in turn its composed transactions, creating new, valid, blocks is difficult and, over time, requires approximately the total compute power of the trustworthy portion of the mining peers.

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

$$V(H) \equiv n \leqslant \frac{2^{256}}{H_d} \wedge m = H_m \quad \wedge \tag{48}$$

$$H_d = D(H) \quad \wedge \tag{49}$$

$$H_g \leq H_l \quad \wedge \tag{50}$$

$$H_l < P(H)_{H_l} + \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{51}$$

$$H_l > P(H)_{H_l} - \left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor \quad \wedge \tag{52}$$

$$H_l \geqslant 125000 \quad \wedge \tag{53}$$

$$H_s > P(H)_{H_s} \quad \wedge \tag{54}$$

$$H_i = P(H)_{H_i} + 1 \quad \wedge \tag{55}$$

$$\|H_x\| \leq 32 \tag{56}$$

where $(n, m) = \texttt{PoW}(H_{\not n}, H_n, \mathbf{d})$

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

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

6. Transaction Execution

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

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

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

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{57}$$

Thus $\boldsymbol{\sigma}'$ is the post-transactional state. We also define Υ^g to evaluate to the amount of gas used in the execution

of a transaction and $\Upsilon^{\mathbf{l}}$ to evaluate to the transaction's accrued log items, both to be formally defined later.

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

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_r) \tag{58}$$

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). Finally there is A_r, the refund balance, increased through using the SSTORE instruction in order to reset contract storage to zero from some non-zero value. Though not immediately refunded, it is allowed to partially offset the total execution costs.

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

$$A^0 \equiv (\varnothing, (), 0) \tag{59}$$

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

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{txdatazero} & \text{if} \quad i = 0 \\ G_{txdatanonzero} & \text{otherwise} \end{cases} \tag{60}$$

$$+ \begin{cases} G_{\text{txcreate}} & \text{if} \quad T_t = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{61}$$

$$+ G_{transaction} \tag{62}$$

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 a result of EVM-code. G is fully defined in Appendix G.

The up-front cost v_0 is calculated as:

$$v_0 \equiv T_g T_p + T_v \tag{63}$$

The validity is determined as:

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_n &= \boldsymbol{\sigma}[S(T)]_n \quad \wedge \\ g_0 &\leqslant T_g \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_b \quad \wedge \\ T_g &\leqslant B_{Hl} - \ell(B_{\mathbf{R}})_u \end{aligned} \tag{64}$$

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

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

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

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{65}$$

$$\boldsymbol{\sigma}_0[S(T)]_b \equiv \boldsymbol{\sigma}[S(T)]_b - T_g T_p \tag{66}$$

$$\boldsymbol{\sigma}_0[S(T)]_n \equiv \boldsymbol{\sigma}[S(T)]_n + 1 \tag{67}$$

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

$$(\boldsymbol{\sigma}_P, g', A) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad g, T_p, T_v, T_{\mathbf{i}}, 0) & \text{if} \quad T_t = \varnothing \\ \Theta_3(\boldsymbol{\sigma}_0, S(T), T_o, & \\ \quad T_t, T_t, g, T_p, T_v, T_v, T_{\mathbf{d}}, 0) & \text{otherwise} \end{cases} \tag{68}$$

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

$$g \equiv T_g - g_0 \tag{69}$$

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

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

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

$$g^* \equiv g' + \min\{\left\lfloor \frac{T_g - g'}{2} \right\rfloor, A_r\} \tag{70}$$

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

The Ether for the gas is given to the miner, whose address is specified as the beneficiary of the present block B. So we define the pre-final state $\boldsymbol{\sigma}^*$ in terms of the provisional state $\boldsymbol{\sigma}_P$:

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{71}$$

$$\boldsymbol{\sigma}^*[S(T)]_b \equiv \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \tag{72}$$

$$\boldsymbol{\sigma}^*[m]_b \equiv \boldsymbol{\sigma}_P[m]_b + (T_g - g^*) T_p \tag{73}$$

$$m \equiv B_{Hc} \tag{74}$$

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

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{75}$$

$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] \equiv \varnothing \tag{76}$$

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

$$\Upsilon^g(\boldsymbol{\sigma}, T) \equiv T_g - g' \tag{77}$$

$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{78}$$

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

7. Contract Creation

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

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

$$(\boldsymbol{\sigma}', g', A) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e) \tag{79}$$

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

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_n - 1)\ \big)\Big)\Big) \tag{80}$$

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

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

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{81}$$

$$\boldsymbol{\sigma}^*[a] \equiv \big(0, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\big) \tag{82}$$

$$\boldsymbol{\sigma}^*[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{83}$$

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

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_b & \text{otherwise} \end{cases} \tag{84}$$

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

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I) \tag{85}$$

where I contains the parameters of the execution environment as defined in section 9, that is:

$$I_a \equiv a \tag{86}$$

$$I_o \equiv o \tag{87}$$

$$I_p \equiv p \tag{88}$$

$$I_{\mathbf{d}} \equiv () \tag{89}$$

$$I_s \equiv s \tag{90}$$

$$I_v \equiv v \tag{91}$$

$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{92}$$

$$I_e \equiv e \tag{93}$$

$I_{\mathbf{d}}$ 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:

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{94}$$

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.

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 and substate as $(\boldsymbol{\sigma}', g', A)$ where:

$$g' \equiv \begin{cases} 0 & \text{if} \quad F \\ g^{**} - c & \text{otherwise} \end{cases} \tag{95}$$

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad F \\ \boldsymbol{\sigma}^{**} \quad \text{except:} & \\ \quad \boldsymbol{\sigma}'[a]_c = \texttt{KEC}(\mathbf{o}) & \text{otherwise} \end{cases} \tag{96}$$

where

$$F \equiv \big(\boldsymbol{\sigma}^{**} = \varnothing \ \vee \ g^{**} < c \ \vee \ |\mathbf{o}| > 24576\big) \tag{97}$$

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.

7.1. Subtleties.

Note that while the initialisation code is executing, the newly created address exists but with no intrinsic body code. Thus any message call received by it during this time causes no code to be executed. If the initialisation execution ends with a SELFDESTRUCT

instruction, the matter is moot since the account will be deleted before the transaction is completed. For a normal STOP code, or if the code returned is otherwise empty, then the state is left with a zombie account, and any remaining balance will be locked into the account forever.

8. Message Call

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

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

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Theta(\boldsymbol{\sigma}, s, o, r, c, g, p, v, \tilde{v}, \mathbf{d}, e) \tag{98}$$

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:

$$\boldsymbol{\sigma}_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v \quad \wedge \quad \boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}[s]_b - v \tag{99}$$

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:

$$\boldsymbol{\sigma}_1 \equiv \boldsymbol{\sigma}'_1 \quad \text{except:} \tag{100}$$

$$\boldsymbol{\sigma}_1[s]_b \equiv \boldsymbol{\sigma}'_1[s]_b - v \tag{101}$$

$$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{102}$$

$$\begin{cases} \boldsymbol{\sigma}'_1[r] \equiv (v, 0, \texttt{KEC}(()), \texttt{TRIE}(\varnothing)) & \text{if} \quad \boldsymbol{\sigma}[r] = \varnothing \\ \boldsymbol{\sigma}'_1[r]_b \equiv \boldsymbol{\sigma}[r]_b + v & \text{otherwise} \end{cases} \tag{103}$$

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

$$\boldsymbol{\sigma}' \equiv \begin{cases} \boldsymbol{\sigma} & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ \boldsymbol{\sigma}^{**} & \text{otherwise} \end{cases} \tag{104}$$

$$g' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \\ g^{**} & \text{otherwise} \end{cases} \tag{105}$$

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, I) & \text{if} \quad r = 4 \\ \Xi(\boldsymbol{\sigma}_1, g, I) & \text{otherwise} \end{cases} \tag{106}$$

$$I_a \equiv r \tag{107}$$

$$I_o \equiv o \tag{108}$$

$$I_p \equiv p \tag{109}$$

$$I_{\mathbf{d}} \equiv \mathbf{d} \tag{110}$$

$$I_s \equiv s \tag{111}$$

$$I_v \equiv \tilde{v} \tag{112}$$

$$I_e \equiv e \tag{113}$$

$$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_c \tag{114}$$

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

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

Their full formal definition is in Appendix E.

9. Execution Model

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

9.1. **Basics.** The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to facilitate the Keccak-256 hash scheme and elliptic-curve computations. The memory model is a simple word-addressed byte array. The stack has 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, CALL and CALLCODE. Finally, gas may be paid due to an increase in the usage of the memory.

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

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

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

9.3. **Execution Environment.** In addition to the system state $\boldsymbol{\sigma}$, and the remaining gas for computation g, there are several pieces of important information used in the execution environment that the execution agent must provide; these are contained in the tuple I:

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

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

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{115}$$

where we will remember that A, the accrued substate is defined as the tuple of the suicides set $\mathbf{s}$, the log series $\mathbf{l}$ and the refunds r:

$$A \equiv (\mathbf{s}, \mathbf{l}, r) \tag{116}$$

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

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

$$\Xi(\boldsymbol{\sigma}, g, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A, \mathbf{o}) \tag{117}$$

$$(\boldsymbol{\sigma}, \boldsymbol{\mu}', A, ..., \mathbf{o}) \equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{118}$$

$$\boldsymbol{\mu}_g \equiv g \tag{119}$$

$$\boldsymbol{\mu}_{pc} \equiv 0 \tag{120}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{121}$$

$$\boldsymbol{\mu}_i \equiv 0 \tag{122}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{123}$$

(124)

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

where

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{125}$$

$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{126}$$

Note that, when we evaluate Ξ, we drop the fourth element I' and extract the remaining gas $\boldsymbol{\mu}'_g$ 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 $(g, pc, \mathbf{m}, i, \mathbf{s})$ which are the gas available, the program counter $pc \in \mathbb{P}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents $\boldsymbol{\mu}_{\mathbf{m}}$ are a series of zeroes of size 2^{256}.

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

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

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases} \tag{127}$$

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

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

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w \in \{\text{JUMP}, \text{JUMPI}\} \quad \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \end{aligned} \tag{128}$$

This states that the execution is in an exceptional halting state if there is insufficient gas, if the instruction is invalid (and therefore its δ subscript is undefined), if there are insufficient stack items, if a JUMP/JUMPI destination is invalid or the new stack size would be larger then 1024. The astute reader will realise that this implies that no instruction can, through its execution, cause an exceptional halt.

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

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

Formally:

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{129}$$

where:

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{if} \quad \mathbf{c}[i] = \text{JUMPDEST} \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{130}$$

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

$$N(i, w) \equiv \begin{cases} i + w - \text{PUSH1} + 2 & \text{if} \quad w \in [\text{PUSH1}, \text{PUSH32}] \\ i + 1 & \text{otherwise} \end{cases} \tag{131}$$

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

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\text{RETURN}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{RETURN} \\ () & \text{if} \quad w \in \{\text{STOP}, \text{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases} \tag{132}$$

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

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

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{133}$$

$$\Delta \equiv \alpha_w - \delta_w \tag{134}$$

$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{135}$$

$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x + \Delta] \tag{136}$$

The gas is reduced by the instruction's gas cost and for most instructions, the program counter increments on each cycle, for the three exceptions, we assume a function J, subscripted by one of two instructions, which evaluates to the according value:

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{137}$$

$$\boldsymbol{\mu}'_{pc} \equiv \begin{cases} J_{\text{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMP} \\ J_{\text{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = \text{JUMPI} \\ N(\boldsymbol{\mu}_{pc}, w) & \text{otherwise} \end{cases} \tag{138}$$

In general, we assume the memory, self-destruct set and system state don't change:

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{139}$$

$$\boldsymbol{\mu}'_i \equiv \boldsymbol{\mu}_i \tag{140}$$

$$A' \equiv A \tag{141}$$

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{142}$$

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

10. Blocktree to Blockchain

The canonical blockchain is a path from root to leaf through the entire block tree. In order to have consensus over which path it is, conceptually we identify the path that has had the most computation done upon it, or, the *heaviest* path. Clearly one factor that helps determine the heaviest path is the block number of the leaf, equivalent to the number of blocks, not counting the unmined genesis block, in the path. The longer the path, the greater the total mining effort that must have been done in order to arrive at the leaf. This is akin to existing schemes, such as that employed in Bitcoin-derived protocols.

Since a block header includes the difficulty, the header alone is enough to validate the computation done. Any block contributes toward the total computation or *total difficulty* of a chain.

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

$$B_t \equiv B'_t + B_d \tag{143}$$

$$B' \equiv P(B_H) \tag{144}$$

As such given a block B, B_t is its total difficulty, B' is its parent block and B_d is its difficulty.

11. Block Finalisation

The process of finalising a block involves four stages:

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

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

$$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{U \in B_{\mathbf{U}}} V(U) \ \wedge \ k(U, P(B_H)_H, 6) \tag{145}$$

where k denotes the "is-kin" property:

$$k(U,H,n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U,H) & \\ \quad \vee\, k(U, P(H)_H, n-1) & \text{otherwise} \end{cases} \tag{146}$$

and s denotes the "is-sibling" property:

$$s(U,H) \equiv (P(H) = P(U) \ \wedge \ H \neq U \ \wedge \ U \notin B(H)_{\mathbf{U}}) \tag{147}$$

where $B(H)$ is the block of the corresponding header H.

11.2. **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:

$$B_{Hg} = \ell(\mathbf{R})_u \tag{148}$$

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

$$\Omega(B, \boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:} \tag{149}$$

$$\boldsymbol{\sigma}'[B_{Hc}]_b = \boldsymbol{\sigma}[B_{Hc}]_b + (1 + \frac{\|B_{\mathbf{U}}\|}{32})R_b \tag{150}$$

$$\forall_{U \in B_{\mathbf{U}}} : \tag{151}$$

$$\boldsymbol{\sigma}'[U_c]_b = \boldsymbol{\sigma}[U_c]_b + (1 + \frac{1}{8}(U_i - B_{Hi}))R_b$$

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

We define the block reward as 5 Ether:

$$\text{Let} \quad R_b = 5 \times 10^{18} \tag{152}$$

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

$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 & \text{if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_i : \texttt{TRIE}(L_S(\boldsymbol{\sigma}_i)) = P(B_H)_{H_r} & \text{otherwise} \end{cases} \tag{153}$$

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

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

$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{except:} \tag{154}$$

$$B'_n = n : \quad x \leqslant \frac{2^{256}}{H_d} \tag{155}$$

$$B'_m = m \quad \text{with } (x, m) = \texttt{PoW}(B^*_{\mathbf{H}}, n, \mathbf{d}) \tag{156}$$

$$B^* \equiv B \quad \text{except:} \quad B^*_r = r(\Pi(\Gamma(B), B)) \tag{157}$$

With $\mathbf{d}$ being a dataset as specified in appendix J.

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

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

$$\mathbf{R}[n]_{\boldsymbol{\sigma}} = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases} \tag{158}$$

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

$$\mathbf{R}[n]_u = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_u & \text{otherwise} \end{cases} \tag{159}$$

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

$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) \tag{160}$$

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

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\mathbf{R})_{\boldsymbol{\sigma}}) \tag{161}$$

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

11.5. **Mining Proof-of-Work.** The mining proof-of-work (PoW) exists as a cryptographically secure nonce that proves beyond reasonable doubt that a particular amount of computation has been expended in the determination of some token value n. It is utilised to enforce the blockchain security by giving meaning and credence to the notion of difficulty (and, by extension, total difficulty). However, since mining new blocks comes with

an attached reward, the proof-of-work not only functions as a method of securing confidence that the blockchain will remain canonical into the future, but also as a wealth distribution mechanism.

For both reasons, there are two important goals of the proof-of-work function; firstly, it should be as accessible as possible to as many people as possible. The requirement of, or reward from, specialised and uncommon hardware should be minimised. This makes the distribution model as open as possible, and, ideally, makes the act of mining a simple swap from electricity to Ether at roughly the same rate for anyone around the world.

Secondly, it should not be possible to make super-linear profits, and especially not so with a high initial barrier. Such a mechanism allows a well-funded adversary to gain a troublesome amount of the network's total mining power and as such gives them a super-linear reward (thus skewing distribution in their favour) as well as reducing the network security.

One plague of the Bitcoin world is ASICs. These are specialised pieces of compute hardware that exist only to do a single task. In Bitcoin's case the task is the SHA256 hash function. While ASICs exist for a proof-of-work function, both goals are placed in jeopardy. Because of this, a proof-of-work function that is ASIC-resistant (i.e. difficult or economically inefficient to implement in specialised compute hardware) has been identified as the proverbial silver bullet.

Two directions exist for ASIC resistance; firstly make it sequential memory-hard, i.e. engineer the function such that the determination of the nonce requires a lot of memory and bandwidth such that the memory cannot be used in parallel to discover multiple nonces simultaneously. The second is to make the type of computation it would need to do general-purpose; the meaning of "specialised hardware" for a general-purpose task set is, naturally, general purpose hardware and as such commodity desktop computers are likely to be pretty close to "specialised hardware" for the task. For Ethereum 1.0 we have chosen the first path.

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

(162)
$$m = H_m \quad \wedge \quad n \leqslant \frac{2^{256}}{H_d} \quad \text{with} \quad (m, n) = \texttt{PoW}(H_{\not n}, H_n, \mathbf{d})$$

Where $H_{\not n}$ is the new block's header but *without* the nonce and mix-hash components; H_n is the nonce of the header; $\mathbf{d}$ is a large data set needed to compute the mixHash and H_d is the new block's difficulty value (i.e. the block difficulty from section 10). PoW is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudo-random number cryptographically dependent on H and $\mathbf{d}$. The underlying algorithm is called Ethash and is described below.

11.5.1. *Ethash.* Ethash is the PoW algorithm for Ethereum 1.0. It is the latest version of Dagger-Hashimoto, introduced by Buterin [2013b] and Dryja [2014], although it can no longer appropriately be called that since many of the original features of both algorithms have been drastically changed in the last month of research and development. The general route that the algorithm takes is as follows:

There exists a seed which can be computed for each block by scanning through the block headers up until that point. From the seed, one can compute a pseudorandom cache, $J_{cacheinit}$ bytes in initial size. Light clients store the cache. From the cache, we can generate a dataset, $J_{datasetinit}$ bytes in initial size, with the property that each item in the dataset depends on only a small number of items from the cache. Full clients and miners store the dataset. The dataset grows linearly with time.

Mining involves grabbing random slices of the dataset and hashing them together. Verification can be done with low memory by using the cache to regenerate the specific pieces of the dataset that you need, so you only need to store the cache. The large dataset is updated once every J_{epoch} blocks, so the vast majority of a miner's effort will be reading the dataset, not making changes to it. The mentioned parameters as well as the algorithm is explained in detail in appendix J.

12. Implementing Contracts

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

12.1. **Data Feeds.** A data feed contract is one which provides a single service: it gives access to information from the external world within Ethereum. The accuracy and timeliness of this information is not guaranteed and it is the task of a secondary contract author—the contract that utilises the data feed—to determine how much trust can be placed in any single data feed.

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

12.2. **Random Numbers.** Providing random numbers within a deterministic system is, naturally, an impossible task. However, we can approximate with pseudo-random numbers by utilising data which is generally unknowable at the time of transacting. Such data might include the block's hash, the block's timestamp and the block's beneficiary address. In order to make it hard for malicious miner to control those values, one should use the BLOCKHASH operation in order to use hashes of the previous 256 blocks as pseudo-random numbers. For a series of such numbers, a trivial solution would be to add some constant amount and hashing the result.

13. Future Directions

The state database won't be forced to maintain all past state trie structures into the future. It should maintain

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

Blockchain consolidation could be used in order to reduce the amount of blocks a client would need to download to act as a full, mining, node. A compressed archive of the trie structure at given points in time (perhaps one in every 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.

13.1. **Scalability.** Scalability remains an eternal concern. With a generalised state transition function, it becomes difficult to partition and parallelise transactions to apply the divide-and-conquer strategy. Unaddressed, the dynamic value-range of the system remains essentially fixed and as the average transaction value increases, the less valuable of them become ignored, being economically pointless to include in the main ledger. However, several strategies exist that may potentially be exploited to provide a considerably more scalable protocol.

Some form of hierarchical structure, achieved by either consolidating smaller lighter-weight chains into the main block or building the main block through the incremental combination and adhesion (through proof-of-work) of smaller transaction sets may allow parallelisation of transaction combination and block-building. Parallelism could also come from a prioritised set of parallel blockchains, consolidated each block and with duplicate or invalid transactions thrown out accordingly.

Finally, verifiable computation, if made generally available and efficient enough, may provide a route to allow the proof-of-work to be the verification of final state.

14. Conclusion

I have introduced, discussed and formally defined the protocol of Ethereum. Through this protocol the reader may implement a node on the Ethereum network and join others in a decentralised secure social operating system. Contracts may be authored in order to algorithmically specify and autonomously enforce rules of interaction.

15. Acknowledgements

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.

16. 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.

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

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

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

Vitalik Buterin. Dagger: A Memory-Hard to Compute, Memory-Easy to Verify Scrypt Alternative. 2013b. URL {http://vitalik.ca/ethereum/dagger.html}.

Thaddeus Dryja. Hashimoto: I/O bound proof of work. 2014. URL {https://mirrorx.com/files/hashimoto.pdf}.

Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *In 12th Annual International Cryptology Conference*, pages 139–147, 1992.

Phong Vo Glenn Fowler, Landon Curt Noll. Fowler – Noll – Vo hash function. 1991. URL {https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function#cite_note-2}.

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.

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.

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

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

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

Simon Sprankel. Technical Basis of Digital Currencies, 2013.

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

Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gün Sirer. Karma: A secure economic framework for peer-to-peer resource sharing, 2003.

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

Appendix A. Terminology

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

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

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

Transaction: A piece of data, signed by an External Actor. It represents either a Message or a new Autonomous Object. Transactions are recorded into each block of the blockchain.

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

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

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

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

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

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

Object: Synonym for Autonomous Object.

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

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

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

Ethereum Runtime Environment: (aka ERE) The environment which is provided to an Autonomous Object executing in the EVM. Includes the EVM but also the structure of the world state on which the EVM relies for certain I/O instructions including CALL & CREATE.

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

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

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

Appendix B. Recursive Length Prefix

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

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

$$\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B} \tag{163}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \wedge \forall_{n<\|\mathbf{t}\|}\ \mathbf{t}[n] \in \mathbb{T}\} \tag{164}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \wedge \forall_{n<\|\mathbf{b}\|}\ \mathbf{b}[n] \in \mathbb{O}\} \tag{165}$$

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

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

$$\texttt{RLP}(\mathbf{x}) \equiv \begin{cases} R_b(\mathbf{x}) & \text{if } \mathbf{x} \in \mathbb{B} \\ R_l(\mathbf{x}) & \text{otherwise} \end{cases} \tag{166}$$

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

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

Formally, we define R_b:

$$R_b(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{otherwise} \end{cases} \tag{167}$$

$$\mathtt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_n \cdot 256^{\|\mathbf{b}\|-1-n} \tag{168}$$

$$(a) \cdot (b, c) \cdot (d, e) = (a, b, c, d, e) \tag{169}$$

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

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

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

Thus we finish by formally defining R_l:

$$R_l(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\mathtt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \tag{170}$$

$$s(\mathbf{x}) \equiv \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1)... \tag{171}$$

If RLP is used to encode a scalar, defined only as a positive integer ($\mathbb{P}$ or any x for $\mathbb{P}_x$), it must be specified as the shortest byte array such that the big-endian interpretation of it is equal. Thus the RLP of some positive integer i is defined as:

$$\mathtt{RLP}(i : i \in \mathbb{P}) \equiv \mathtt{RLP}(\mathtt{BE}(i)) \tag{172}$$

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 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 $\mathbb{B}$):

$$\mathtt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if} \quad \|\mathbf{x}\| \text{ is even} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{173}$$

$$f(t) \equiv \begin{cases} 2 & \text{if} \quad t \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{174}$$

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:

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{175}$$

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

$$\forall_{I \in \mathfrak{I}} I \equiv (I_0, I_1) \tag{176}$$

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:

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{177}$$

$$\forall_n \quad \forall_{i:i<2\|\mathbf{k}_n\|} \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases} \tag{178}$$

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

$$\text{TRIE}(\mathfrak{I}) \equiv \text{KEC}(c(\mathfrak{I}, 0)) \tag{179}$$

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

$$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{if } \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if } \|c(\mathfrak{I}, i)\| < 32 \\ \text{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases} \tag{180}$$

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 $true$.

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

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

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

$$c(\mathfrak{I}, i) \equiv \begin{cases} \text{RLP}\Big(\big(\text{HP}(I_0[i..(\|I_0\|-1)], true), I_1\big)\Big) & \text{if } \|\mathfrak{I}\| = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \text{RLP}\Big(\big(\text{HP}(I_0[i..(j-1)], false), n(\mathfrak{I}, j)\big)\Big) & \text{if } i \neq j \quad \text{where } j = \arg\max_x : \exists \mathbf{l} : \|\mathbf{l}\| = x : \forall_{I \in \mathfrak{I}} : I_0[0..(x-1)] = \mathbf{l} \\ \text{RLP}\Big((u(0), u(1), ..., u(15), v)\Big) & \text{otherwise} \quad \text{where } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad\qquad v = \begin{cases} I_1 & \text{if } \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () & \text{otherwise} \end{cases} \end{cases} \tag{181}$$

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

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

Appendix E. Precompiled Contracts

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

$$\Xi_{\text{PRE}}(\boldsymbol{\sigma}, g, I) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{if} \quad g < g_r \\ (\boldsymbol{\sigma}, g - g_r, A^0, \mathbf{o}) & \text{otherwise} \end{cases} \tag{182}$$

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

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

$$\Xi_{\text{ECREC}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{183}$$

$$g_r = 3000 \tag{184}$$

$$|\mathbf{o}| = \begin{cases} 0 & \text{if} \quad \texttt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases} \tag{185}$$

$$\text{if} \quad |\mathbf{o}| = 32: \tag{186}$$

$$\mathbf{o}[0..11] = 0 \tag{187}$$

$$\mathbf{o}[12..31] = \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{where:} \tag{188}$$

$$\mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] = I_{\mathbf{d}} \tag{189}$$

$$\mathbf{d}[|I_{\mathbf{d}}|..] = (0, 0, ...) \tag{190}$$

$$h = \mathbf{d}[0..31] \tag{191}$$

$$v = \mathbf{d}[32..63] \tag{192}$$

$$r = \mathbf{d}[64..95] \tag{193}$$

$$s = \mathbf{d}[96..127] \tag{194}$$

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

$$\Xi_{\text{SHA256}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{195}$$

$$g_r = 60 + 12\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{196}$$

$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{197}$$

$$\Xi_{\text{RIP160}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{198}$$

$$g_r = 600 + 120\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{199}$$

$$\mathbf{o}[0..11] = 0 \tag{200}$$

$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{201}$$

(202)

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

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{203}$$

$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{204}$$

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

$$\Xi_{\text{ID}} \equiv \Xi_{\text{PRE}} \quad \text{where:} \tag{205}$$

$$g_r = 15 + 3\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{206}$$

$$\mathbf{o} = I_{\mathbf{d}} \tag{207}$$

Appendix F. Signing Transactions

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

It is assumed that the sender has a valid private key p_r, 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 assert the functions `ECDSASIGN`, `ECDSARESTORE` and `ECDSAPUBKEY`. These are formally defined in the literature.

$$\texttt{ECDSAPUBKEY}(p_r \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{208}$$

$$\texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_r \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{209}$$

$$\texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_u \in \mathbb{B}_{64} \tag{210}$$

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

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

$$0 < r < \texttt{secp256k1n} \tag{211}$$

$$0 < s < \texttt{secp256k1n} \div 2 + 1 \tag{212}$$

$$v \in \{27, 28\} \tag{213}$$

where:

$$\texttt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{214}$$

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

$$A(p_r) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_r)\big)\big) \tag{215}$$

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

$$L_S(T) \equiv \begin{cases} (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{i}}) & \text{if } T_t = 0 \\ (T_n, T_p, T_g, T_t, T_v, T_{\mathbf{d}}) & \text{otherwise} \end{cases} \tag{216}$$

$$h(T) \equiv \texttt{KEC}(L_S(T)) \tag{217}$$

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

$$G(T, p_r) \equiv T \quad \text{except:} \tag{218}$$

$$(T_w, T_r, T_s) = \texttt{ECDSASIGN}(h(T), p_r) \tag{219}$$

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

$$S(T) \equiv \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSARECOVER}(h(T), T_w, T_r, T_s)\big)\big) \tag{220}$$

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

$$\forall T : \forall p_r : S(G(T, p_r)) \equiv A(p_r) \tag{221}$$

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 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_{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}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	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 $\lceil \log_{256}(exponent) \rceil$ 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*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	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_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for 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 left most 12 are discarded or filled with zeroes, thus the integer values (when the bytes are interpreted as big-endian) are equivalent.

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

(222)

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv C_{mem}(\boldsymbol{\mu}'_i) - C_{mem}(\boldsymbol{\mu}_i) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if } w = \text{SSTORE} \\ G_{exp} & \text{if } w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if } w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{verylow} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if } w = \text{CALLDATACOPY} \vee \text{CODECOPY} \\ G_{extcode} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if } w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if } w = \text{LOG0} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{logtopic} & \text{if } w = \text{LOG1} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{logtopic} & \text{if } w = \text{LOG2} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{logtopic} & \text{if } w = \text{LOG3} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{logtopic} & \text{if } w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if } w = \text{CALL} \vee \text{CALLCODE} \vee \text{DELEGATECALL} \\ C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if } w = \text{SELFDESTRUCT} \\ G_{create} & \text{if } w = \text{CREATE} \\ G_{sha3} + G_{sha3word} \lceil \mathbf{s}[1] \div 32 \rceil & \text{if } w = \text{SHA3} \\ G_{jumpdest} & \text{if } w = \text{JUMPDEST} \\ G_{sload} & \text{if } w = \text{SLOAD} \\ G_{zero} & \text{if } w \in W_{zero} \\ G_{base} & \text{if } w \in W_{base} \\ G_{verylow} & \text{if } w \in W_{verylow} \\ G_{low} & \text{if } w \in W_{low} \\ G_{mid} & \text{if } w \in W_{mid} \\ G_{high} & \text{if } w \in W_{high} \\ G_{extcode} & \text{if } w \in W_{extcode} \\ G_{balance} & \text{if } w = \text{BALANCE} \\ G_{blockhash} & \text{if } w = \text{BLOCKHASH} \end{cases}$$

(223)

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

where:

(224)

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

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

$W_{zero} = \{\text{STOP, RETURN}\}$

$W_{base} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, POP, PC, MSIZE, GAS}\}$

$W_{verylow} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}\}$

$W_{low} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}\}$

$W_{mid} = \{\text{ADDMOD, MULMOD, JUMP}\}$

$W_{high} = \{\text{JUMPI}\}$

$W_{extcode} = \{\text{EXTCODESIZE}\}$

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

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

Note also that C_{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 724B of memory used, after which it costs substantially more.

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

(225)

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

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

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{226}$$

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:

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted} \tag{227}$$

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). $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ -2^{255} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] = -2^{255} \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = -1 \\ \mathbf{sgn}(\boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1]) \lfloor \lvert \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rvert \rfloor & \text{otherwise} \end{cases}$ 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. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_t & \text{if} \quad i \leqslant t \quad \text{where } t = 256 - 8(\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \\ \boldsymbol{\mu}_{\mathbf{s}}[1]_i & \text{otherwise} \end{cases}$

$\boldsymbol{\mu}_{\mathbf{s}}[x]_i$ 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. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \wedge \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \vee \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x18	XOR	2	1	Bitwise XOR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \oplus \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x19	NOT	1	1	Bitwise NOT operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} 1 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0]_i = 0 \\ 0 & \text{otherwise} \end{cases}$
0x1a	BYTE	2	1	Retrieve single byte from word. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i+8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i < 8 \wedge \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ For Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

20s: SHA3

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

30s: Environmental Information

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_a$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_b & \text{if } \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_o$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_s$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_v$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \|I_{\mathbf{d}}\|$ 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. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ 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. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_p$ This is gas price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_c\|$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \|\mathbf{c}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ where $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_c$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo.

40s: Block Information

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

50s: Stack, Memory, Storage and Flow Operations				
Value	**Mnemonic**	δ	α	**Description**
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_i \equiv \max(\boldsymbol{\mu}_i, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_i$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \wedge \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \wedge \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\text{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 9.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{pc} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section 9.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{pc}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
0x5a	GAS	0	1	Get the amount of available gas, including the corresponding reduction for the cost of this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_g$
0x5b	JUMPDEST	0	0	Mark a valid destination for jumps. This operation has no effect on machine state during execution.

60s & 70s: Push Operations

Value	Mnemonic	δ	α	Description
0x60	PUSH1	0	1	Place 1 byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{pc}+1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if } x < \|I_{\mathbf{b}}\| \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\|x\|-1}))$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{pc}+1)\dots(\boldsymbol{\mu}_{pc}+32)\big)$ 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:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_a, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
and to update the memory consumption counter:
$\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
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. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[1] \dots (\boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A^+) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_a, I_o, L(\boldsymbol{\mu}_g), I_p, \boldsymbol{\mu}_\mathbf{s}[0], \mathbf{i}, I_e + 1) & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_g, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}$ except $\boldsymbol{\sigma}^*[I_a]_n = \boldsymbol{\sigma}[I_a]_n + 1$ $A' \equiv A \Cup A^+$ which implies: $A'_\mathbf{s} \equiv A_\mathbf{s} \cup A^+_\mathbf{s} \quad \wedge \quad A'_\mathbf{l} \equiv A_\mathbf{l} \cdot A^+_\mathbf{l} \quad \wedge \quad A'_\mathbf{r} \equiv A_\mathbf{r} + A^+_\mathbf{r}$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}^*, \boldsymbol{\mu}, I) = \top$ or $I_e = 1024$ (the maximum call depth limit is reached) or $\boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\sigma}[I_a]_b$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_a, \boldsymbol{\sigma}[I_a]_n)$, the address of the newly created account, otherwise. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[1], \boldsymbol{\mu}_\mathbf{s}[2])$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], |\mathbf{o}|\})$ $\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting $Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) = \top$ or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_a]_b$ (not enough funds) or $I_e = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_i \equiv M(M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if } \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if } \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_a, I_o, I_a, t, & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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. $H_{\text{RETURN}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 9. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$

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 exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \text{if} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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.
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_a\}$ $\boldsymbol{\sigma}'[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b + \boldsymbol{\sigma}[I_a]_b$ $\boldsymbol{\sigma}'[I_a]_b \equiv 0$ $A'_r \equiv A_r + \begin{cases} R_{selfdestruct} & \text{if} \quad I_a \notin A_\mathbf{s} \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{selfdestruct} + \begin{cases} G_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$

Appendix I. Genesis Block

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

$$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 0, 0_{2048}, 2^{17}, 0, 0, 3141592, \mathit{time}, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), ()\big) \tag{228}$$

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^{17} 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 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 lists 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. **Definitions.** We employ the following definitions:

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

J.2. **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.

$$E_{epoch}(H_i) = \left\lfloor \frac{H_i}{J_{epoch}} \right\rfloor \tag{229}$$

The size of the dataset growth by $J_{datasetgrowth}$ bytes, and the size of the cache by $J_{cachegrowth}$ 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 $J_{hashbytes}$ for the cache. Let $d_{size} = \|\mathbf{d}\|$ be the size of the dataset. Which is calculated using

$$d_{size} = E_{prime}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{epoch} - J_{mixbytes}, J_{mixbytes}) \tag{230}$$

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

$$c_{size} = E_{prime}(J_{cacheinit} + J_{cachegrowth} \cdot E_{epoch} - J_{hashbytes}, J_{hashbytes}) \tag{231}$$

$$E_{prime}(x, y) = \begin{cases} x & \text{if} \quad x/y \in \mathbb{P} \\ E_{prime}(x - 1 \cdot y, y) & \text{otherwise} \end{cases} \tag{232}$$

J.3. **Dataset generation.** In order the 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.3.1. *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:

$$\mathbf{s} = C_{seedhash}(H_i) \tag{233}$$

$$C_{seedhash}(H_i) = \begin{cases} \texttt{KEC}(\mathbf{0}_{32}) & \text{if} \quad E_{epoch}(H_i) = 0 \\ \texttt{KEC}(C_{seedhash}(H_i - J_{epoch})) & \text{otherwise} \end{cases} \tag{234}$$

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

J.3.2. *Cache.* The cache production process involves using the seed hash to first sequentially filling up c_{size} bytes of memory, then performing $J_{cacherounds}$ 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:

$$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{if} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{otherwise} \end{cases} \tag{235}$$

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

$$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n \tag{236}$$

$$n = \left\lfloor \frac{c_{size}}{J_{hashbytes}} \right\rfloor \tag{237}$$

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

$$\mathbf{c} = E_{cacherounds}(\mathbf{c}', J_{cacherounds}) \tag{238}$$

$$E_{cacherounds}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{if} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{if} \quad y = 1 \\ E_{cacherounds}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{otherwise} \end{cases} \tag{239}$$

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

$$E_{\text{RMH}}(\mathbf{x}) = \big(E_{rmh}(\mathbf{x}, 0), E_{rmh}(\mathbf{x}, 1), ..., E_{rmh}(\mathbf{x}, n - 1)\big) \tag{240}$$

$$E_{rmh}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i - 1 + n) \mod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \mod n]) \tag{241}$$
$$\text{with} \quad \mathbf{x}' = \mathbf{x} \quad \text{except} \quad \mathbf{x}'[j] = E_{rmh}(\mathbf{x}, j) \quad \forall \quad j < i$$

J.3.3. *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 $J_{hashbytes}$ bytes in size:

$$\mathbf{d}[i] = E_{datasetitem}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{size}}{J_{hashbytes}} \right\rfloor \tag{242}$$

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.

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (0\text{x}01000193 \oplus \mathbf{y})) \mod 2^{32} \tag{243}$$

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

$$E_{datasetitem}(\mathbf{c}, i) = E_{parents}(\mathbf{c}, i, -1, \varnothing) \tag{244}$$

$$E_{parents}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{parents}(\mathbf{c}, i, p + 1, E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1)) & \text{if} \quad p < J_{parents} - 2 \\ E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1) & \text{otherwise} \end{cases} \tag{245}$$

$$E_{mix}(\mathbf{m}, \mathbf{c}, i, p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \mod c_{size}] \oplus i) & \text{if } p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \mod \lfloor J_{hashbytes}/J_{wordbytes} \rfloor]) \mod c_{size}]\big) & \text{otherwise} \end{cases} \tag{246}$$

J.4. **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:

$$\texttt{PoW}(H_{\not H}, H_n, \mathbf{d}) = \{\mathbf{m}_c(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_n, \mathbf{d}), \texttt{KEC}(\mathbf{s}_h(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_n) + \mathbf{m}_c(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_n, \mathbf{d}))\} \tag{247}$$

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

$$\mathbf{m}_c(\mathbf{h}, \mathbf{n}, \mathbf{d}) = E_{compress}(E_{accesses}(\mathbf{d}, \sum_{i=0}^{n_{mix}} \mathbf{s}_h(\mathbf{h}, \mathbf{n}), \mathbf{s}_h(\mathbf{h}, \mathbf{n}), -1), -4) \tag{248}$$

The seed hash being:

$$\mathbf{s}_h(\mathbf{h}, \mathbf{n}) = \texttt{KEC512}(\mathbf{h} + E_{revert}(\mathbf{n})) \tag{249}$$

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

$$E_{revert}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{250}$$

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.3.3.

The number of replicated sequences in the mix is:

$$n_{mix} = \left\lfloor \frac{J_{mixbytes}}{J_{hashbytes}} \right\rfloor \tag{251}$$

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

$$E_{accesses}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = \begin{cases} E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) & \text{if } i = J_{accesses} - 2 \\ E_{accesses}(E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i), \mathbf{s}, i + 1) & \text{otherwise} \end{cases} \tag{252}$$

$$E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = E_{\text{FNV}}(\mathbf{m}, E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) \tag{253}$$

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

$$E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)[j] = \mathbf{d}[E_{\text{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \mod \left\lfloor \frac{J_{mixbytes}}{J_{wordbytes}} \right\rfloor]) \mod \left\lfloor \frac{d_{size}/J_{hashbytes}}{n_{mix}} \right\rfloor \cdot n_{mix} + j] \quad \forall \quad j < n_{mix} \tag{254}$$

The mix is compressed as follows:

$$E_{compress}(\mathbf{m}, i) = \begin{cases} \mathbf{m} & \text{if } i \geqslant \|\mathbf{m}\| - 8 \\ E_{compress}(E_{\text{FNV}}(E_{\text{FNV}}(E_{\text{FNV}}(\mathbf{m}[i+4], \mathbf{m}[i+5]), \mathbf{m}[i+6]), \mathbf{m}[i+7]), i+8) & \text{otherwise} \end{cases} \tag{255}$$