以太坊：一种安全去中心化的通用交易账本

EIP-150 版本 (759dccd - 2017-08-07)

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，我们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. **?** 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。

? 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。**?** 后来设计了一个类似的系统。

? 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**?** 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 **?** 。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 **?** 。

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 **?**。彩色币（Coloured Coins, 参见 **?**, 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 **?**, 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。

? 和 **?** 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 **??** 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 Đ 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. 历史? 因为这是一个去中心化的系统, 所有人都有机会在之前的某一个区块创建新的区块并连接在其后, 这会行成一个树状的区块。为了能在这个树状结构上从根节点 (创世块) 到叶子节点 (包含最新交易的区块) 能形成一个一致的区块链, 必须有一个共识方案。如果有人认为从根节点到叶子节点的路径不是" 最佳" 的区块链, 那这时候就会发生分叉。

这个就意味着在一个给定的时间点, 系统中会有多个状态共存: 一些节点相信一个区块是包含标准的交易, 其他的节点则相信另外一些区块包含标准的交易，其中就包含彻底不同或者不兼容的交易。这一点必须要避免, 因为它会破坏整个系统信用。

我们使用了一个简单 GHOST 协议版本来达成共识, 参见 **?** 。我们会在**??** 详细说明。

有时会从一个特定的区块链高度启用新的协议。本文描述了协议的一个版本, 如果要跟踪历史区块链路径, 可能需要查看这份文档的历史版本。(译者注: 可以到 https://github.com/ethereum/yellowpaper 查看英文版历史版本)

3. 约定

我用了大量的印刷约定表示公式中的符号，其中一些需要特别说明:

有两个高度结构化的顶层状态值, 使用粗体小写希腊字母: $\boldsymbol{\sigma}$ 表示世界状态 (world-state)；$\boldsymbol{\mu}$ 表示机器状态 (machine-state)。

作用在高度结构化值上的函数，使用大写的希腊字母，例如：Υ , 是以太坊中的状态转换函数。

对于大部分函数来说，通常用一个大写的字母表示，例如：C, 表示费用函数。可能会使用下角标表示为一些特别的变量，例如：C_{SSTORE}，表示执行 SSTORE 操作的费用函数。对于一些可能是外部定义的函数，可能会使用打印机文字字体，例如：KEC512 哈希函数 (为赢得进入 SHA-3 竞赛)，使用 KEC 表示。KEC512 表示 Keccack-512 哈希函数。

元组通常使用一个大写字母，例如：T，表示一个以太坊交易。使用下标可能会表示一个独立的变量, 例如：T_n ，表示交易中随机数。角标的形式用于表示它们的类型；例如：大写的下角标表示元组包含的下角标变量。

标量和固定大小的字节序列 (或数组) 都使用小写字母来表示，例如：n 在本文中表示交易随机数。小写的希腊字母一般表示一些特别的含义，例如：δ 表示在栈上一个给定操作需要的条目数量。

任意长度的序列通常用加粗的小写字母表示，例如 $\mathbf{o}$ 表示消息调用中输出的数据字节序列。有时候，会对特别重要的值使用粗体。

我们认为标量都是正整数且属于集合 $\mathbb{P}$。所有的字节序列属于集合 $\mathbb{B}$，附录 **??** 给出了正式的定义。用下角标符号表示这样的序列集合限制在一定长度以内, 长度为 32 的字节序列使用 $\mathbb{B}_{32}$ 表示, 所有比 2^{256} 小的正整数使用 $\mathbb{P}_{256}$ 表示。详细定义见 **??**。

使用方括号表示序列中的一个元素或子序列, 例如: $\boldsymbol{\mu}_{\mathbf{s}}[0]$ 表示计算机堆栈中的第一个条目。对于子序列来说，使用省略号表示一定的范围，且含头尾的限制，例如：$\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ 表示计算机内存中的前 32 个条目。

在全局状态的情况下， 是一个含多个账号的序列，本身的数组，正方形括号被用作去表示一个单独的账号。

以全局状态 $\boldsymbol{\sigma}$ 为例, 它表示一系列的账户, 它们自身的元组, 方括号用于表示一个独立的账户。

当去考虑现有的变量时，我遵循在给定的范围内去定义的原则，我们使用占位符 $\square$ 表示未修改的输入变量, 使用 $\square'$ 表示修改的和可用的变量, $\square^*$, $\square^{**}$ &c 表示中间变量。在特殊情况下，为了提高可读性和清晰性，我可能会使用字母-数字下角标表示中间值。

当使用去已有的函数时，给定一个函数 f, 那么函数 f^* 表示一个相似的、替换序列的函数映射。详细的定义见 **??**。

整个过程中，我定义了大量的函数。一个常见的函数是 ℓ，表示给定序列的最后一个条目:

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

4. 块、状态和交易

介绍了以太坊的基本概念后, 我们将更详细地讨论交易、区块和状态的含义

4.1. 世界状态. 世界状态是在地址 (160 位的标志符) 和账户状态 (序列化为 RLP 的数据结构，详见附录 **??**) 的映射。虽然世界状态没有直接储存在区块链上，但会假定实施过程中会将这个映射维护在一个修改过的 Merkle Patricia 树 (简称 trie，详见附录 **??**)。trie 需要一个简单的后端数据库去维护字节数组到字节数组的映射；我们称这个后端数据库为状态数据库。它有一系列的好处: 第一这个结构的根节点是加密的且依赖于所有的内部数据，且它的哈希可以作为整个系统状态的一个安全标志; 第二，作为一个不变的数据结构，因此它允许任何一个之前状态 (根部哈希已知的条件下) 通过简单地改变根部哈希值而被召回。因为我们在区块链中储存了所以这样的根部哈希值，所以我们能恢复到指定的历史状态。

账户状态包含以下四个字段:

nonce, 随机数: 这个值等于账户发出的交易数及这个账户创建的合约数量之和。$\boldsymbol{\sigma}[a]_n$ 表示状态 $\boldsymbol{\sigma}$ 中的地址 a 的 nonce 值。

balance, 余额: $\boldsymbol{\sigma}[a]_b$, 表示这个账户拥有多少 Wei。

storageRoot, 存储跟节点: 保存账户内容的 Merkle Patricia 树根节点的 256 位哈希编码到 trie 中, 作

为从 256 位整数键值哈希的 Keccak 256 位哈希到 256 位整数的 RLP-编码映射。这个哈希定义为 $\boldsymbol{\sigma}[a]_s$。

codeHash, 代码哈希: 这个账户的 EVM(Ethereum Virtual Machine, 以太坊虚拟机) 代码的哈希值—代码执行时, 这个地址会接收一个消息调用; 它和其它字段不同, 创建后不可更改。状态数据库中包含所有像这样的代码片段的哈希, 以便后续使用。这个哈希定义为 $\boldsymbol{\sigma}[a]_c$,$\mathbf{b}$ 表示代码, $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$.

因为我通常希望所指的并不是 trie 树的根哈希, 而是所保存的键值对集合，我做了一个更方便的定义:

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

trie 树中的键值对集合函数, L_I^*, 定义为基于基础函数 L_I 的元素转换:

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

其中:

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

需要说明的是,$\boldsymbol{\sigma}[a]_\mathbf{s}$ 不应算作这个账户的" 物理" 成员, 它不参与序列化。

如果 **codeHash** 字段是一个空字符串的 Keccak-256 哈希, 例如 $\boldsymbol{\sigma}[a]_c = \texttt{KEC}\big(()\big)$, 则表示对应的节点表示一个简单账户, 有时简称" 非合约" 账户。

因此我们可能定义一个世界状态的函数 L_S:

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

where

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

函数 L_S 和 trie 函数是为了提供一个世界状态的简短身份（哈希）。我们假定:

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

v 是账户合法性验证函数:

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

4.2. **交易.** 交易（符号, T）是个单一的加密指令，通过以太坊中系统之外的操作者创建。我们假设外部的操作者是人, 软件工具用于创建和传播[1]。这里的交易类型有两种: 一种是消息调用, 另一种通过代码创建新的账户（称为“合约创建”）。两种类型的交易都有的共同字段如下:

nonce, 随机数: T_n, 账户发出的交易数量。

gasPrice, 燃料价格: T_p, 为执行交易所需要的计算资源付的 *gas* 价格, 以 Wei 为单位。

gasLimit, 燃料上限: T_g, 用于执行交易的最大 gas 数量。这个值须在交易前设置, 且设定后不能再修改。

to, 接收者地址: 消息调用接收者的 160 位的地址。对与合约创建交易, 无需接收者地址, 使用 $\varnothing$ 表示, $\varnothing$ 是 $\mathbb{B}_0$ 的唯一成员。

value, 转账额度: T_v, 转到接收者账户的额度, 以 Wei 为单位。对于合约创建, 表示捐赠到合约地址的额度。

v, r, s: T_w, T_r and T_s, 和交易签名相关的变量, 用于确定交易的发送者。详见附录 **??**。

此外, 合约创建还包含以下字段:

init, 初始化: $T_\mathbf{i}$, 一个不限制大小的字节数组, 表示账户初始化程序的 EVM 代码。

init 是 EVM 代码片段; 执行 init 后会返回另外一个代码片段, 每次合约接受消息调用 (通过交易或内部调用) 后都会执行这个代码片段。**init** 仅当合约账户创建的时候执行一次。

相比之下, 一个消息调用的交易包括:

data, 数据: $T_\mathbf{d}$, 一个不限制大小的字节数组, 表示消息调用的输入数据。

附录 **??** 详细描述了映射发送者交易的函数 S, 通过 SECP-256k1 的 ECDSA 曲线, 使用交易 (除了最后的 3 个签名字段) 作为数据来签名。目前我们先简单使用 $S(T)$ 表示发送者的指定交易 T 。

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

在这里, 我们假设所有变量都是 RLP 编码的整数, 除了 2 个任意长度的字节数组 $T_\mathbf{i}$ 和 $T_\mathbf{d}$。

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

其中

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

地址哈希 $T_\mathbf{t}$ 稍微有些不同: 它是一个 20 字节的地址哈希值, 但创建合约时它是 RLP 空字节系列, 表示为 $\mathbb{B}_0$:

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

4.3. **区块.** 在以太坊中, 区块是相关信息的集合。区块头 H, 与之想对应的交易信息 $\mathbf{T}$, 其它区块头数据的集合 $\mathbf{U}$, $\mathbf{U}$ 表示它的父级区块中有和当前区块的爷爷辈区块是相同的。(这样的区块称为 *ommers*[2], 译者注: 不妨称之为叔链)。区块头包含的的信息如下:

parentHash, 父块哈希: H_p, 父区块头的 Keccak 256 位哈希。

ommersHash, 叔链哈希: H_o, 当前区块的叔链列表 Keccak 256 位哈希。

beneficiary, 受益者地址: H_c, 成功挖到这个区块的 160 位地址, 这个区块中的所有交易费用都会转到这个地址。

stateRoot, 状态字典树根节点哈希: 状态字典树根节点的 Keccak 256 位哈希, 交易打包到当前区块且区块定稿后可以生成这个值。

transactionsRoot, 交易字典树根节点哈希: 交易字典树根节点的 Keccak 256 位哈希, 在交易字典树含有区块中的所有交易列表。

receiptsRoot, 接受者字典树根节点哈希: 接受者字典树根节点的 Keccak 256 位哈希, 在接受者字典树含有区块中的所有交易信息中的接受者。

logsBloom, 日志 Bloom: H_b, 日记 Bloom 过滤器由可索引信息（日志地址和日志主题）组成，这个信息包含在每个日志入口, 来自交易列表中的每个交易的接受者。

[1] 显著地, 这样的“工具”可以从基于人类行为的初始化中移除—或者人类可能变成有原因的中立—可能有一点他们被视为自治的代理人。例如: 合约可能会给人类好处, 让人发送交易从而触发合约的执行。

[2] *ommer* 的意思和自然界中的" 父母的兄弟姐妹" 最相近, 详见 `http://nonbinary.org/wiki/Gender_neutral_language#Family_Terms`

difficulty, 难度**:** H_d, 表示当前区块的难度水平, 这个值根据前一个区块的难度水平和时间戳计算得到。

number, 区块编号**:** H_i, 等于当前区块的直系前辈区块数量。创始区块的区块编号为 0。

gasLimit, 燃料限制**:** H_l, 目前每个区块的燃料消耗上限。

gasUsed, 燃料使用量**:** H_g, 当前区块的所有交易使用燃料之和。

timestamp, 时间戳**:** H_s, 当前区块初始化时的 Unix 时间戳。

extraData, 附加数据**:** H_x, 32 字节以内的字节数组。

mixHash, 混合哈希**:** H_m, 与一个与随机数 (nonce) 相关的 256 位哈希计算, 用于证明针对当前区块已经完成了足够的计算。

nonce, 随机数**:** H_n, 一个 64 位哈希, 和计算混合哈希相关, 用于证明针对当前区块已经完成了足够的计算。

此外, 当前区块还记录着这个区块的交易列表, 以及 2 个叔链 (ommer) 的区块头列表。我们以 B 表示一个区块:

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

4.3.1. 交易收据. 为了让交易信息编码能有利于零知识证明、索引、搜索, 我们将每个包含一定信息的交易收据进行编码。以 $B_{\mathbf{R}}[i]$ 表示第 i 个交易, 保存在一个索引字典树中,H_e 是这个字典树的根节点。为了去编译信息关于每一个有关系的交易而且可能是一个有用的工具去形成一个零知识证明（zero-knowledge proof)，或者索引和搜索。我们编译每一个包含当前交易执行的某些信息为交易收据。每个收据 (在第个交易中表示为) 是被放置于一个带关键字索引的 trie 中和这个区块头中被记录下的根值，表示为。

交易收据是一个包含四个条目的元组：交易后的状态,$R_{\boldsymbol{\sigma}}$；当前区块中交易累计燃料使用量, R_u, 交易发生后会立即更新这个值; 交易执行过程中创建的日志集合,$R_{\mathbf{l}}$; 和日志 Bloom 过滤器,R_b ：

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

函数 L_R 是一个将交易收据转换为 RLP 编码的预处理函数:

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

交易后状态 $R_{\boldsymbol{\sigma}}$ 会编码到一个字典树中, 字典树的根节点组成了第一个条目。

我们假定累计的燃料使用量 R_u 是一个正整数, 日志 Bloom R_b 是 2048 位 (256 字节) 的哈希:

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

$R_{\mathbf{l}}$ 是一系列的日志入口, 例如 $(O_0, O_1, ...)$。一个日志入口 O 是一个日志记录器的地址 O_a 的元组, $O_{\mathbf{t}}$ 是一系列 32 字节的日志主题, $O_{\mathbf{d}}$ 是一些字节数据:

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

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

我们定义 Bloom 过滤器函数 M 将一个日志入口转换为一个 256 字节哈希:

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

其中 $M_{3:2048}$ 是一个特别的 Bloom 过滤器, 针对任意一个字节序列, 它舍弃这个字节序列 2048 位的前三位。它通过对一个字节序列的 Keccak-256 哈希的每一个前三对字节取其的低 11 位来实现:

$$\begin{aligned}
&(24) & M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) &\equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \\
&(25) & \mathbf{y} &= (0, 0, ..., 0) \quad \text{except:} \\
&(26) & \forall_{i \in \{0,2,4\}} &: \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \\
&(27) & m(\mathbf{x}, i) &\equiv \texttt{KEC}(\mathbf{x})[i, i+1] \bmod 2048
\end{aligned}$$

其中 $\mathcal{B}$ 是位引用函数, $\mathcal{B}_j(\mathbf{x})$ 等于字节数组 $\mathbf{x}$ 中的索引 j(从 0 开始索引) 的位。

4.3.2. 整体有效性. 如果一个区块同时满足以下几个条件, 我们才能认为这个区块是有效的：当从起始状态 $\boldsymbol{\sigma}$ (父块的最终状态) 按顺序执行完生成新的状态 H_r 后, 在内部上要保持一致, 包括叔链、交易区块哈希、给定的交易 $B_{\mathbf{T}}$（详细描述见 **??**）：

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

其中 $p(k, v)$ 是 RLP 的简单对转换, 在这个例子中, k 为这个区块中的交易索引,v 为交易收据:

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

此外:

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

$\texttt{TRIE}(L_S(\boldsymbol{\sigma}))$ 是包含以 RLP 编码的状态 $\boldsymbol{\sigma}$ 键值对的 Merkle Patricia 树根节点哈希, $P(B_H)$ 是父节点。

这些值根据交易计算产生，特别是交易收据 $B_{\mathbf{R}}$, 这个通过交易状态累积函数 Π 定义，在 **??** 会详细说明。

4.3.3. 序列化. 函数 L_B 和 L_H 分别是区块和区块头的准备函数。类似交易收据准备函数 L_R，当转换为 RLP 格式时, 假设对应的类型、顺序及结构如下:

$$\begin{aligned}
&(31) & L_H(H) &\equiv \big(\; H_p, H_o, H_c, H_r, H_t, H_e, H_b, H_d, \\
& & & \qquad H_i, H_l, H_g, H_s, H_x, H_m, H_n \;\big) \\
&(32) & L_B(B) &\equiv \big(L_H(B_H), L_T^*(B_{\mathbf{T}}), L_H^*(B_{\mathbf{U}})\big)
\end{aligned}$$

其中 L_T^* 和 L_H^* 是元素序列转换函数, 因此:

$$\begin{array}{l}
(33) \\
f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{对于任何函数 } f
\end{array}$$

元素类型定义如下:

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

其中

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

我们现在有了一个严密正式的区块结构结构说明。RLP 函数（见附录 **??**）提供了一个标准方法来把这个结构转换为一个字节序列。

4.3.4. 区块头验证. 我们定义 $P(B_H)$ 为 B 的父区块::

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

当前区块编号等于它的父块编号加 1:

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

区块难度定义为 $D(H)$:

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

其中:

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

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

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

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

区块的燃料限制 H_l 需要满足下面条件:

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

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

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

H_s 是区块 H 的时间戳, 需满足下面条件:

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

这个机制保证了区块间时间平衡；如果最近的两个区块时间间隔短, 则会导致难度系数增加, 因此需要额外的计算量，大概率会延长下个区块的出块时间。相反，如果最近的 2 个区块时间间隔时间过长，难度系数和下一个区块的出块预期时间也会减少。

随机数 H_n，必须满足下面关系:

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

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

其中 $H_{\not{n}}$ 是新区块的区块头，但不包含随机数和混合哈希值，$\mathbf{d}$ 是当前的大数据集合 DAG(有向无环图)，需要去计算混合哈希, PoW 是工作量证明函数（见 **??**）: 第一个元素用于计算混合哈希值，以证明使用了一个正确的 DAG, 第 2 个元素是伪随机数, 依赖于 H 及 $\mathbf{d}$ 。给定一个范围在 $[0, 2^{64})$ 的均匀分布，则求解时间和难度 H_d 成比例。

这就是区块链安全基础，这也是一个恶意节点不能用其新创建的区块中重写历史数据的重要原因。因为这个随机数必须满足这些条件，且因为条件依赖于这个区块的内容和相关交易, 创建新的合法的区块是困难的、耗时的, 需要超过所有诚实矿工的算力总和。

因此, 我们定义这个区块头的验证函数 $V(H)$ 为:

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

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

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

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

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

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

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

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

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

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

此外, **extraData** 最多 32 字节。

5. 燃料和支付

为了避免网络滥用及回避由于图灵完整性而带来的一些不可避免的问题，在以太坊中所有的编程计算都需要费用。各种操作费用以 *gas* (详见附录 **??**) 为单位计算。任意的程序片段（包括合约创建、信息调回、利用及访问账户存储、在虚拟机上执行操作等）都可以根据规则计算出消耗的燃料。

每一个交易都有一个燃料上限：**gasLimit** (燃料上限)。这些燃料从发送者的账户中扣除。具体从账户上扣除的额度和 **gasPrice**(燃料价格) 有关 (译者注：扣除额度 = **gasLimit** * **gasPrice**), 在执行交易前会指定燃料价格。如果这个账户不能支付起燃料费用，这个交易会被当作无效交易。之所以它被命名为燃料上限，是因为剩余的燃料在交易完成之后会被退回（以购买时的同样价格）到发送者账户。燃料不会被用在交易执行之外。因此对于可信任账户，应该设置一个相对较高的燃料上限。

通常来说，以太币（Ether）用作去购买燃料, 未退回的那部分转到了区块受益人的地址，通常这个账户的地址是由矿工设定。交易者可以任意设定燃料价格，然而矿工也可以任意地忽略某个交易。在一个交易中，高价格的燃料将消费这个发送者更多的以太币, 并转给矿工更多的以太币，因此这个交易会被更多的矿工选择。通常来说，矿工将会选择去通知这是他们执行交易最低燃料价格, 交易者们一般也会些选择一个高过燃料价格下限的价格。因此, 会有一个（加权的）最低燃料可接受价格分布，交易者们需要权衡降低燃料价格和交易快速被矿工打包。

6. 交易执行

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 Υ^1 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:

$$\begin{aligned}
(60)\qquad g_0 &\equiv \sum_{i \in T_\mathbf{i}, T_\mathbf{d}} \begin{cases} G_{txdatazero} & \text{if} \quad i = 0 \\ G_{txdatanonzero} & \text{otherwise} \end{cases} \\
(61)\qquad &+ \begin{cases} G_{\text{txcreate}} & \text{if} \quad T_t = \varnothing \\ 0 & \text{otherwise} \end{cases} \\
(62)\qquad &+ G_{transaction}
\end{aligned}$$

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 **??**.

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:

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

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 $\boldsymbol{\sigma}_0$:

$$\begin{aligned}
(65)\qquad \boldsymbol{\sigma}_0 &\equiv \boldsymbol{\sigma} \quad \text{except:} \\
(66)\qquad \boldsymbol{\sigma}_0[S(T)]_b &\equiv \boldsymbol{\sigma}[S(T)]_b - T_g T_p \\
(67)\qquad \boldsymbol{\sigma}_0[S(T)]_n &\equiv \boldsymbol{\sigma}[S(T)]_n + 1
\end{aligned}$$

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:

(68)

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

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.

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

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 $\boldsymbol{\sigma}_P$:

$$\begin{aligned}
(71)\qquad \boldsymbol{\sigma}^* &\equiv \boldsymbol{\sigma}_P \quad \text{except} \\
(72)\qquad \boldsymbol{\sigma}^*[S(T)]_b &\equiv \boldsymbol{\sigma}_P[S(T)]_b + g^* T_p \\
(73)\qquad \boldsymbol{\sigma}^*[m]_b &\equiv \boldsymbol{\sigma}_P[m]_b + (T_g - g^*) T_p \\
(74)\qquad m &\equiv B_{Hc}
\end{aligned}$$

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

$$\begin{aligned}
(75)\qquad \boldsymbol{\sigma}' &\equiv \boldsymbol{\sigma}^* \quad \text{except} \\
(76)\qquad \forall i \in A_\mathbf{s} : \boldsymbol{\sigma}'[i] &\equiv \varnothing
\end{aligned}$$

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 **??**:

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

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 $\texttt{KEC}$ is the Keccak 256-bit hash function, $\texttt{RLP}$ 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 $\boldsymbol{\sigma}^*$:

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

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

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

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 **??**). 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 **??**, 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:

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

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

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

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

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 **??**). 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. $\boldsymbol{\sigma}$).

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

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

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

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

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

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

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

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

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

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

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

It is assumed that the client will have stored the pair $(\mathtt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$ 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 **??**.

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 **??**). 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 **??** 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.

$$\begin{aligned}
\Xi(\boldsymbol{\sigma}, g, I) &\equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A, \mathbf{o}) && (117)\\
(\boldsymbol{\sigma}, \boldsymbol{\mu}', A, ..., \mathbf{o}) &\equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) && (118)\\
\boldsymbol{\mu}_g &\equiv g && (119)\\
\boldsymbol{\mu}_{pc} &\equiv 0 && (120)\\
\boldsymbol{\mu}_{\mathbf{m}} &\equiv (0, 0, ...) && (121)\\
\boldsymbol{\mu}_i &\equiv 0 && (122)\\
\boldsymbol{\mu}_{\mathbf{s}} &\equiv () && (123)
\end{aligned}$$

$$X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv \begin{cases} \big(\varnothing, \boldsymbol{\mu}, A^0, I, ()\big) & \text{if} \quad Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \\ O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \cdot \mathbf{o} & \text{if} \quad \mathbf{o} \neq \varnothing \\ X\big(O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) & \text{otherwise} \end{cases} \tag{124}$$

where

$$\begin{aligned}
\mathbf{o} &\equiv H(\boldsymbol{\mu}, I) && (125)\\
(a, b, c, d) \cdot e &\equiv (a, b, c, d, e) && (126)
\end{aligned}$$

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 **??**.

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

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

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:

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

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:

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

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

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

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

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

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

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:

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

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

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

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

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 **??**, 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:

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

and s denotes the “is-sibling” property:

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

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_{H_g}, the total gas used in the block, must be equal to the accumulated gas used according to the final transaction:

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

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:

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

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

(151) $\forall_{U \in B_{\mathbf{U}}}$:

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

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:

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

Here, $\mathtt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ 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':

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

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

$$B'_m = m \quad \text{with } (x, m) = \mathtt{PoW}(B^*_{\not{H}}, n, \mathbf{d}) \tag{156}$$

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

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

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

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

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 $\mathtt{PoW}$:

(162)

$$m = H_m \quad \wedge \quad n \leqslant \frac{2^{256}}{H_d} \quad \text{with} \quad (m, n) = \mathtt{PoW}(H_{\cancel{n}}, H_n, \mathbf{d})$$

Where $H_{\cancel{n}}$ 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 **??**). $\mathtt{PoW}$ is the proof-of-work function which evaluates to an array with the first item being the mixHash and the second item being a pseudorandom 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 **?** and **?**, 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 **??**.

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 autogenerated 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 $\mathtt{RLP}$ 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:

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

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:

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

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

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

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:

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

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

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:

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

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

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

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

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:

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

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:

(181)

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

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, $\Xi_{\mathtt{PRE}}$, which implements the out-of-gas checking.

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

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 ($\mathtt{ECDSARECOVER}(h, v, r, s) = \varnothing$), then we have no output.

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

$$\tag{202}$$

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

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

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

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

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

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

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

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 **?**.

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 $[1, \mathtt{secp256k1n} - 1]$.

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

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

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

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

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

(223)

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

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} = {STOP, RETURN}

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

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

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

W_{mid} = {ADDMOD, MULMOD, JUMP}

W_{high} = {JUMPI}

$W_{extcode}$ = {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)

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

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 **??**, 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 **??** 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 **??**.

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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.
0x09	MULMOD	3	1	Modulo multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ 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. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \land \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
0x17	OR	2	1	Bitwise OR operation. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]_i \lor \boldsymbol{\mu}_{\mathbf{s}}[1]_i$
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. $\forall i \in [0..255] : \boldsymbol{\mu}'_{\mathbf{s}}[0]_i \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[1]_{(i + 8\boldsymbol{\mu}_{\mathbf{s}}[0])} & \text{if} \quad i < 8 \land \boldsymbol{\mu}_{\mathbf{s}}[0] < 32 \\ 0 & \text{otherwise} \end{cases}$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

30s: Environmental Information

Value	Mnemonic	δ	α	Description
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_a$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_b & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_o$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_s$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_v$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \lVert I_{\mathbf{d}} \rVert$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{d}} \rVert$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{d}} \rVert \\ 0 & \text{otherwise} \end{cases}$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert I_{\mathbf{b}} \rVert$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \lVert I_{\mathbf{b}} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ 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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \lVert \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_c \rVert$
0x3c	EXTCODECOPY	4	0	Copy an account's code to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[3]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] \equiv \begin{cases} \mathbf{c}[\boldsymbol{\mu}_{\mathbf{s}}[2] + i] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] + i < \lVert \mathbf{c} \rVert \\ \text{STOP} & \text{otherwise} \end{cases}$ 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. $\boldsymbol{\sigma}'[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\text{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \ \wedge \ \boldsymbol{\sigma}[I_a]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
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 **??**.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\text{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{pc} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{pc}$. See section **??**.
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 $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if} \quad x < \Vert I_{\mathbf{b}} \Vert \\ 0 & \text{otherwise} \end{cases}$ 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 $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\Vert x \Vert - 1}))$ 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:
$A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_a, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$
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. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[1] \dots (\boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_g, A^+) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_a, I_o, L(\boldsymbol{\mu}_g), I_p, \boldsymbol{\mu}_{\mathbf{s}}[0], \mathbf{i}, I_e + 1) & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \leqslant \boldsymbol{\sigma}[I_a]_b \;\wedge\; I_e < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_g, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except} \quad \boldsymbol{\sigma}^*[I_a]_n = \boldsymbol{\sigma}[I_a]_n + 1$ $A' \equiv A \Cup A^+$ 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. $\mathbf{i} \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[3] \dots (\boldsymbol{\mu}_{\mathbf{s}}[3] + \boldsymbol{\mu}_{\mathbf{s}}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_a, I_o, t, t, & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \;\wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_{\mathbf{s}}[6], \lvert\mathbf{o}\rvert\})$ $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[5] \dots (\boldsymbol{\mu}_{\mathbf{s}}[5] + n - 1)] = \mathbf{o}[0 \dots (n - 1)]$ $\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g + g'$ $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}$ 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. $C_{\text{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{CALLGAS}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_{\text{GASCAP}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_g - C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_{\mathbf{s}}[0]\} & \text{if} \quad \boldsymbol{\mu}_g \geq C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{otherwise} \end{cases}$ $C_{\text{EXTRA}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_{\text{XFER}}(\boldsymbol{\mu}) + C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_{\text{XFER}}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{NEW}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[1] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_a, I_o, I_a, t, & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] \leqslant \boldsymbol{\sigma}[I_a]_b \;\wedge \\ \quad C_{\text{CALLGAS}}(\boldsymbol{\mu}), I_p, \boldsymbol{\mu}_{\mathbf{s}}[2], \boldsymbol{\mu}_{\mathbf{s}}[2], \mathbf{i}, I_e + 1) & \quad I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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 **??**. $\boldsymbol{\mu}'_i \equiv M(\boldsymbol{\mu}_i, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

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: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, & \text{if} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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	∅	∅	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:

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

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 **?**. 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:

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

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

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 (**?**) in some cases as a non-associative substitute for XOR.

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (0x01000193 \oplus \mathbf{y})) \mod 2^{32} \tag{243}$$

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

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

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

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

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:

(247)

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

With $H_{\not n}$ 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 **??**.

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:

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

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

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

(254)

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

The mix is compressed as follows:

(255)

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