以太坊：一种安全去中心化的通用交易账本

EIP-150 版本 (6f97c9f - 2018-03-20)

原文作者: DR. GAVIN WOOD, GAVIN@ETHCORE.IO
译者: 崔广斌 (微信号: 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。

以太坊是一个尝试达到通用性的技术项目, 可以构建任何基于交易的状态机。而且以太坊致力于为开发者提供一个紧凑的、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到状态变化或者判决, 但是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.

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

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

Vishnumurthy et al. [2003] 最早使用工作量证明作为强大的经济信号保证货币安全。在这个案例中，代币用作检查点对点（peer-to-peer,p2p）文件交易，同时保证“消费者”能支付给为他们提供服务的“供应商”。这种通过工作量证明的安全模型逐步扩展, 包括使用电子签名和账本技术, 以保证历史记录不被篡改, 怀有恶意的用户不能进行欺诈支付或不公平的抱怨。五年后（2008 年），中本聪 Nakamoto [2008] 介绍了另一种更广泛的工作量证明安全价值代币。这个项目的成果就是比特币, 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, 通过修改比特币的协议创建了大量的其它数字货币。比较知名的有莱特币 (Litecoin) 和素数币 (Primecoin) , 参见 Sprankel [2013] 。一些项目使用比特币的核心机制并重新改造以应用在其它领域, 例如域名币（Namecoin）致力于提供一个去中心化的名字解析系统, 参见 Aron [2012] 。

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

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

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

2. 区块链

以太坊在整体上可以看作一个基于交易的状态机：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, 挖矿获得激励后, 会执行状态转移函数, 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。这个机制称为工作量证明, 会在 11.5 详细讨论。

公式如下:

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

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

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

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

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

2.1. 面值**.** 为了激励网络中的计算，需要定义一种转账方法。以太坊设计了一种内置货币以太币 (Ether)，ETH 是大家所熟知的符号，有时用 Ð 表示。以太币最小的面额是 Wei(伟)，所有货币值都以 Wei 的整数倍记录。一个以太币等于 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权威的交易，其它节点则相信另外一些区块包含权的交易，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。

我们使用了一个简易 GHOST 协议版本来达成共识, 参见 Sompolinsky and Zohar [2013] 。我们会在小结 10 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

我们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)。

作用在高度结构化数据上的函数，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, 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 B 给出了正式的定义。用下角标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 4.3。

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，且含首尾限制，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，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，我们遵循在给定的范围内去定义的原则，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。详细定义见 4.3。

整个过程中，我们定义了大量的函数。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 B) 的映射。虽然世界状态没有直接储存在区块链上，但会假定在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, 字典树, 详见附录 D)。字典树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，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, 存储根节点：保存账户内容的 Merkle Patricia 树根节点的 256 位哈希编码到字典树中，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字典树树的根哈希，而是所保存的键值对集合，我做了一个更方便的定义：

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

字典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 和字典树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 F。

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。仅当合约账户创建时会执行一次 **init**。

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 F 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 $T_{\mathbf{d}}$。

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

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，使用 $\mathbb{B}_0$ 表示：

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

4.3. 区块. 在以太坊中，区块是相关信息的集合。与区块头 H 想对应的交易信息为 $\mathbf{T}$，其它区块头数据的集合 $\mathbf{U}$ 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。

[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

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。

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。

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:

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

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 2048 位字节序列的前三位。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:

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

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

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

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

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 11）：

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

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 11.4 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预备函数。类似交易收据预备函数 L_R，当转换为 RLP 格式时, 假设对应的类型、顺序及结构如下:

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

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

其中 L^*_T 和 L^*_H 是元素序列转换函数, 因此:

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

元素类型定义如下:

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

其中

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

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 B) 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:

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

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 $D(H)$:

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

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 需满足下面关系:

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

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

其中 $H_{\not H}$ 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 11.5): 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且耗时的, 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:

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

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

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

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

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

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

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

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

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

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

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, 在以太坊中所有的程序执行都需要费用。各种操作费用以 *gas* (详见附录 G) 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数量。

每一个交易都有燃料上限: **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, 交易会被当作无效交易。之所以命名为燃料上限, 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) 用作购买燃料, 未退回的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

交易执行是以太坊协议中最复杂的部分: 它定义了状态转换函数 Υ。所有交易在执行时, 都要先通过内部的有效性测试, 这些包含:

(1) 交易是 RLP 格式数据, 没有多余的后缀字节;
(2) 交易的签名是有效的;
(3) 交易的随机数是有效的 (等于发送者账户的当前随机数) ;
(4) 燃料上限不小于实际交易过程中用的燃料 g_0;
(5) 发送者账户的余额至少大于费用 v_0, 需要提前支付。

T 表示交易, $\boldsymbol{\sigma}$ 表示状态, 状态转移函数 Υ 如下:

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

$\boldsymbol{\sigma}'$ 是交易后的状态。我们定义 Υ^g 为交易执行所消耗的燃料量, $\Upsilon^{\mathbf{l}}$ 为交易过程中产生的日志记录, 这些都会在后文正式定义。

6.1. 子状态. 从交易执行过程来看，伴随交易会产生一些特定的信息, 我们称为交易子状态，用 A 来表示:

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

元组内容包含自毁集合 $A_{\mathbf{s}}$: 一个在交易完成后被删除的账户集合。$A_{\mathbf{l}}$ 是一系列的日志：在代码执行过程中是可归档, 可索引的检查点, 允许以太坊世界的外部旁观者 (例如去中心化应用的前端) 跟踪合约调用。A_r 表示返回的余额，当使用 SSTORE 指令将非 0 的合约存贮空间置为 0 时, 返回余额会增加。虽然不是立即返回余额，但可以部分抵消整个执行费用。

我们定义空的子状态 A^0, 它没有自毁、没有日志、返回余额为 0:

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

6.2. 执行. 执行过程中需要的燃料需要在交易进行之前支付, 定义 g_0 :

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

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

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

$T_{\mathbf{i}}$ 合约初始化 EVM 代码, $T_{\mathbf{d}}$ 是交易附带的关联数据, 取决于交易是合约创建还是消息调用。如果这个交易是合约创建, 那么会增加 G_{txcreate} , 其它情况则不增加。G 的详细定义见附录 G。

预支付的费用 v_0 计算如下:

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

需满足下面关系:

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

注意最后的条件: 交易燃料限制 T_g <= 这个区块的燃料限制 B_{Hl} - 这个区块之前的所有交易已使用的燃料 $\ell(B_{\mathbf{R}})_u$ 。

有效交易的执行起始于一个对状态不能撤回的改变：发送者账户 $S(T)$ 的随机数会加 1, 账户余额扣减部分预付费用 $T_g T_p$。计算过程中有效的燃料 $g = T_g - g_0$。无论是合约创建还是消息调用后的的最终状态 (可能等于当前的状态), 这个改变是确定的且从来没有无效的: 从这个观点看不会存在无效的交易。

我们定义检查点状态 $\boldsymbol{\sigma}_0$:

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

从 $\boldsymbol{\sigma}_0$ 转变为 $\boldsymbol{\sigma}_P$ 和交易类型相关。不管它是创建合约还是消息调用, 我们定义元组包括执行后的临时状态 $\boldsymbol{\sigma}_P$, 剩余的燃料 g' 和子状态 A :

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

其中 g 是燃料上限扣除已有的交易消耗的燃料:

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

T_o 是原始执行者，与和合约创建或消息调用 (直接由交易触发) 不同, 可以通过 EVM 代码内部调用来执行。

注意，我们使用 Θ_3 表示取函数值的前三个元素; 最终结果是消息调用的输出 (一个字节数组) , 最终结果并没有在交易的上下文使用。

在消息调回或者创建合约被处理后，再确定退回的燃料后最终状态就确定了。剩余的燃料 g', 再加上一些补偿, 得到最终需要退回发送者的燃料 g^*:

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

最终要退回发送者的燃料 g^* 等于当前剩余的燃料 g', 再加一个补偿, 这个补偿是 A_r 和总使用燃料量 $T_g - g'$ 的一半中的小者。

燃料对应的以太币给了矿工, 矿工地址是当前区块 B 的受益者地址。我们定义预备最终状态 $\boldsymbol{\sigma}^*$, 临时状态 $\boldsymbol{\sigma}_P$:

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

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

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

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

删除所有出现在自毁集合中的账户后, 达到了最终状态 $\boldsymbol{\sigma}'$:

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

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

最后, 我们定义这次交易中总共使用的燃料 Υ^g, 这次交易中创建的日志 $\Upsilon^{\mathbf{l}}$:

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

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

这些有助于后续讨论的交易收据。

7. 合约创建

当创建一个账户时会有很多参数: 发送者 (s)、原始执行者 (o)、可用的燃料 (g)、燃料价格 (p)、转账额度 (v)、任意长度的字节数组 $\mathbf{i}$、EVM 初始化代码、消息调用/合约创建的当前栈深度 (e) 。

我们定义创建函数为函数 Λ，相关的变量有状态 $\boldsymbol{\sigma}$, 包含新状态、剩余燃料及交易子状态 $(\boldsymbol{\sigma}', g', A)$ (详见第 6 小结):

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

这个新账户的地址被定义为包含发送者和随机数 RLP 编码的 Keccak 哈希的最边 160 位。因此我们定义新帐户 a 的地址:

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

其中 KEC 是 Keccak 256 位哈希函数，RLP 是 RLP 编码函数，$\mathcal{B}_{a..b}(X)$ 表示取二进制数据 X 位数为范围 $[a, b]$ 的值,$\boldsymbol{\sigma}[x]$ 是地址 x 的状态，$\varnothing$ 表示地址不存在时的状态。注意, 我们使用的是一个比发送者随机数要小的值; 我们断言会在在这个合约创建是前会增加发送者账户的随机数，因此在刚开始在交易或 VM 操作中使用的随机数就是发送者的随机数。

账户的随机数开始被定义为 0，余额为传递的转账值，存储空间为空，哈希代码为 Keccak 256 位的空字符串哈希值；发送者的余额会减去转账值。这个变化的状态变成 $\boldsymbol{\sigma}^*$：

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

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

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

其中 v' 是账户在交易之前就有的余额：

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

最后，这个账户是通过执行初始化账户的 EVM 代码 $\mathbf{i}$ (执行模型详见小结 9) 来初始化的。代码执行可以影响一些事件：可以改变当前账户的存储，能创建更多的账户，执行更多的消息调用。代码执行函数 Ξ 可以得到一个元组，包括结果状态 $\boldsymbol{\sigma}^{**}$，可用的剩余燃料 g^{**}，子状态 A，以及账户 $\mathbf{o}$ 的代码。赋值到最后的状态的数组中，有效的剩余燃料，当前产生的子状态 A 和账号代码本身 $\mathbf{o}$。

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

其中 I 包含执行环境的相关参数，这些参数在小结 9 中有详细定义：

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

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

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

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

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

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

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

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

如果没有输入数据，使用 $I_{\mathbf{d}}$ 表示空的元组。I_H 没有什么特别的，它由区块链决定。

代码执行会消耗燃料，且燃料不能低于 0，因此程序执行可能会在自然终止之前退出。在这个（以及其它几个）异常情况，我们称发生了燃料不足（out-of-gas, OOG）异常：演进的状态变成空集合 $\varnothing$，整个创建操作对状态没有影响，状态就像尝试创建合约之前一样。

如果这个初始化代码成功地执行完，那么对应的合约创建也会被支付。代码保存费用 c 和创建的合约代码大小成正比：

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

如果没有足够的燃料支付这些，例如 $g^{**} < c$，就会抛出燃料不足异常。

发生这样的异常后，剩余燃料将变为 0。如果合约创建是用于接受一个交易，它不会影响合约创建内部消耗的燃料，无论如何都必须支付。然而，当燃料不足时，并不会给这个合约地址转账。

如果没有出现这样的异常，那么剩余的燃料会退会给最原始的发送者，改变后的新状态也会永久保存。最终状态、燃料和子状态 $(\boldsymbol{\sigma}', g', A)$ 的关系如下：

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

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

where

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

上述 $\boldsymbol{\sigma}'$ 的公式，表明了执行代码初始化得到的最终字节序列 $\mathbf{o}$，就是新创建账户的代码体。

当合约成功创建时，如果有转账，对应的转账也会转到合约中；如果没有转账，则仅是合约创建。

7.1. 细微之处. 有一种情况需要注意，当初始化代码正在执行时，新创建的地址出现了，但还没有内部的代码时。在这个时间内，任意消息调用不会引起代码执行。如果这个初始化执行结束于一个 SELFDESTRUCT 指令，这个账号会在交易执行完前将被删除，这个行为是无意义的。对于一个正常的 STOP 指令代码，或者返回的代码是空的，这时候会出现一个僵尸账户，而且账户中剩余的余额将被永远被锁定在这个僵尸账户中。

8. 消息调用

当执行消息调用时需要多个参数：发送者（s）、交易发起人（o）、接受者（r）、执行代码的账户（c，通常就是接受者）、可用的燃料（g）、转账额度（v）、燃料价格（p）、函数调用的一个任意长度字节的数组 $\mathbf{d}$ 的输入数据以及消息调用/合约创建的当前栈（e）深度。

除了转变到新的状态和交易子状态外，消息调用还有一个额外的元素 — 用字节数组 $\mathbf{o}$ 表示的输出数据。执行交易时输出数据是被忽略的，但在消息调用时，输出的数据可以由虚拟机代码执行时进行初始化，在这种情况下也使用了这些信息。

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

注意我们的是，当执行 DELEGATECALL 指令时，我们需要区别转账额度 v 和执行上下文中的 $\tilde{v}$ 。

我们定义在原始状态基础上进行了发送者向接收者转账后的状态为第一个转变状态 $\boldsymbol{\sigma}_1$：

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

unless $s = r$.

我们假设如果 $\boldsymbol{\sigma}_1[r]$ 还未定义，则会创建一个没有代码或没有状态且余额和随机数都为 0 的账户。我们改进上一个公式：

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

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

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

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

账户关联的代码（整个代码碎片，Keccak 哈希为 $\boldsymbol{\sigma}[c]_c$）依靠执行模式（见小结 9）执行。合约创建时，如果执行因为一个异常（例如：耗尽了燃料、堆栈溢出、无效的跳转目的地或者无效的指令）而被终止，燃料不会返回给调用者，且状态也会立即恢复到转账之前的状态 (例如 $\boldsymbol{\sigma}$)。

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

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

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

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

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

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

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

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

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

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

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

我们假设客户端会先保存数据对 $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$，以便更方便地根据 $I_{\mathbf{b}}$ 做出决定。

如我们所见，消息调用执行框架 Ξ 中有 4 个特例: 这四个是‘预编译’合约，作为最初架构中的一部分后续可能会变成本地扩展。这四个合约地址分别为 1、2、3 和 4，分别是用来执行椭圆曲线公钥恢复函数、SHA2 256 位哈希方案、RIPEMD 160 位哈希方案和身份函数。

这 4 个合约的详细定义见附录 E。

9. 执行模型

执行模型具体说明怎么使用一系列字节代码指令和一个小的环境数据元组去改变这个系统状态。这些是通过以太坊虚拟机 (Ethereum Virtual Machine - EVM), 这个虚拟状态机来实现的。它是一个准图灵机, 说“准”是因为计算会被燃料所限制。

9.1. 基础. EVM 基于栈结构, 机器的字大小（以及栈中数据的大小）是 256 位。主要是便于执行 Keccak-256 位哈希及椭圆曲线计算。内存模型基于字寻址的字节数据。栈的最大深度为 1024。EVM 也有一个独立的存储模型；类似内存但更像一个字节数组, 一个基于字寻址的字数组。不像易变的内存，存储是非易变的且是作为系统状态的一部分被维护。所有内存和存储中的数据会初始化为 0。

EVM 不是标准的诺依曼结构。它通过一个特别的指令把程序代码保存在一个虚拟的可以交互的 ROM 中, 而不是保存在一般性可访问的内存或存储中。

EVM 在某些情况下会有异常发生，包括堆栈溢出和非法指令。就像燃料缺乏异常那样, 不会改变状态。有异常时, EVM 立即停止并告知执行代理 (交易的处理者，或执行环境), 代理会单独处理异常。

9.2. 费用概述. 在三个不同的情况下使用费用（命名为燃料），在这三种情况下，费用都是执行任何操作的必备条件。第一种情况也是最普通的情况就是计算操作费用（详见附录 G）。第二种情况，可能因为一个子消息调用或者合约创建而消耗燃料, 这是执行 CREATE, CALL and CALLCODE 费用中的一部分。第三种情况，可能因为内存使用的增多而为燃料付费。

对于一个账户的执行，内存总费用和需要的 32 字节最小倍数的内存量成正比, 所有的内存是按在一定范围中被包含的所有记忆索引（无论是读还是写）下要求的 32 位字节的最小倍数成比例的。这个为了 just-in-time (JIT) 的基础而去支付的；访问一个大于索引量 32 字节的地址, 就会需要额外的内存使用费。地址很容易超过 32 字节的限制, 但 EVM 不同。综上所述，必须能够去管理这些可能发生的事件。

存储费用有一个微妙的行为——为最小化存储费用（直接与一个在所有结点上的大型状态数据通信），清除存储的执行费用指令不仅仅免除, 而且会返回一些费用；因为初始化的存储费用往往比实际使用的多, 在清除存储空间后会有返回费用, 这个返回的费用是预先支付的费用中的一部分。

详细的 EVM 燃料消耗定义见附录 H。

9.3. 执行环境. 除了系统状态 $\boldsymbol{\sigma}$ 和计算过程中剩余的燃料 g 外, 还有一些在计算过程中需要提供的信息, 这些信息组成了元组 I:

- I_a, 拥有正在执行代码的账户地址。
- I_o, 发起这次交易的发送者地址。
- I_p, 发起这次交易中的燃料价格。
- $I_{\mathbf{d}}$, 执行过程中的输入字节数组；如果这次执行是一个交易，这个就是交易数据。
- I_s, 触发代码执行的账户地址；如果这次执行是一个交易，则为交易发送者地址。
- I_v, 以 Wei 为单位的值, 作为计算中的一部分传递给这个账户; 如果这次执行是一个交易, 这个值为转账额度。
- $I_{\mathbf{b}}$, 机器代码字节数组, 会用来执行。
- I_H, 当前区块的区块头。
- I_e, 当前消息调用或合约创建的深度 (例如: 当前 CALLs 或 CREATEs 被执行的次数)。

执行模型定义了函数 Ξ, 用来计算结果状态 $\boldsymbol{\sigma}'$, 剩余的燃料 g', 产生的子状态 A 和结果输出 $\mathbf{o}$, 根据当前的上下文, 我们定义:

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

应该还记得 A, 这个产生的子状态定义为自毁集合 $\mathbf{s}$、日志系列 $\mathbf{l}$ 和回推金额 r 的元组:

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

9.4. 执行概述. 我们现在必须去定义函数 Ξ。在大多实际执行过程中, 将整个系统状态 $\boldsymbol{\sigma}$ 和机器状态 $\boldsymbol{\mu}$ 的迭代过程建模。我们定义一个递归函数 X, 它使用这个迭代函数 O(定义状态机中单循环的结果), 定义函数 Z 用来表示当前状态是一个异常终止状态, 定义函数 H 表示当前状态是正常终止时得到的结果数据。

空序列使用 () 表示, 它不等于空的集合 $\varnothing$；这对理解 H 的输出结果非常重要, 当 H 的输出结果是 $\varnothing$ 时需要继续执行, 但当 H 的输出结果是时一个空的序列时执行应该终止。

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

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

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

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

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

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

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

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

where

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

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

需要注意的是, 在推导 Ξ 时, 我们去掉了第 4 个元素 I' 并从结果状态 $\boldsymbol{\mu}'$ 中提取了剩余的燃料 $\boldsymbol{\mu}'_g$

X 被循环调用 (这里是递归，但是通常是去执行一个简单的迭代循环) 直到 Z 变成 true, 表示当前状态有异常并且需要终止, 并且所有的改动都会被舍弃; 或者直到 H 变成了一个序列 (不是空集合)，表示达到了正常的终止状态。

9.4.1. 机器状态. 机器状态 $\boldsymbol{\mu}$ 定义为一个元组 $(g, pc, \mathbf{m}, i, \mathbf{s})$, 包括可用的燃料 g, 程序计数器 $pc \in \mathbb{P}_{256}$, 内存内容 $\mathbf{m}$, 内存中激活的字数 (从位置 0 开始连续计数) i, 以及栈内容 $\mathbf{s}$。内存内容 $\boldsymbol{\mu}_{\mathbf{m}}$ 表示空间大小为 2^{256} 但内容都是 0 的空间。

为了提高可读性, 使用大写字母 (例如 ADD) 的指令助记符都应该看作对应的数字方程式, 详细的指标表及定义见附录 H 。

为了更好的定义 Z, H 和 O, 我们定义 w 为当前待执行的指令:

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

我们使用 δ 表示栈操作移除的元素数, 使用 α 表示栈操作添加的元素数, 可以使用指令作为 δ 、 α 和指令费用函数 C 的下标, C 表示执行指定指令消耗的燃料。

9.4.2. 异常终止. 异常终止函数 Z 定义如下:

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

以下情况都会发生异常而进入终止状态: 燃料不足、无效指令 (无效指令的 δ 下角标不会进行定义)、缺少栈数据、指令 JUMP/JUMPI 的目标地址无效、新的栈大小会大于 1024。聪明的读者将很容易发现, 在整个执行过程中没有一个指令能显式触发异常而终止。

9.4.3. 跳转地址验证. 我们之前使用 D 作为一个函数去定义正在运行的指定代码的有效跳转地址的集合。我们使用 JUMPDEST 指令表示代码可以跳转的地址。

这些跳转地址都必须在有效指令边界内，并且需要在显式定义在代码中, 而不是基于 PUSH 指令操作的数据。(也不是使用隐含定义的 STOP 指令去跟踪它)。

正式定义如下:

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

其中:

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

N 是代码中下一个有效指令位置, 如果有 PUSH 指令, 会跳过相关数据。

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

9.4.4. 正常停止. 正常停止的函数 H 定义如下:

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

数据返回停止操作 RETURN, 在附录 H 中有详细的定义, 函数名字是 H_{RETURN} 。

9.5. 执行周期. 序列的最左侧的、低端的数据被添加到栈或从栈中移除; 所有其它数据都被保留不变:

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

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

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

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

对与大多数指令, 燃料随着指令的执行而减少, 每个周期程序计数器都会增加; 有三个特例，我们定义函数 J，下角标是二个指令中的一个，如下:

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

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

在一般情况下，我们认为内存、自毁集合和系统状态不改变:

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

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

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

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

然而，指令通常会改变一个到几个元素的值。附录 H 列出了指令修改的元素, 以及入栈数 α 、出栈数 δ 和燃料消耗。

10. 区块树到区块链

权威的区块链是覆盖整个区块树的从根节点到叶子节点的路径。为了形成路径共识，我们通过最大的计算量，或者说是最重的路径来识别它。识别最重路径的一个有效的因素是叶子节点的数量，它也是路径中区块的数量，不包含没有进行挖矿的创世块 (译者注: 创始块通过硬编码写在了代码中及区块链中)。路径越长，到达最后一个叶子节点的代价越大。这和已有的比特币及其衍生协议类似。

因为一个区块头包含难度信息，所以仅通过区块头就能验证已经做了挖矿计算。区块链中的任意一个区块都对总的计算或区块链难度有贡献。

我们递归地定义区块 B 的难度:

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

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

给定区块 B, B_t 是它的总难度, B' 是它的父块难度, B_d 是它的难度。

11. 区块定稿

区块定稿涉及到 4 个步骤:

(1) 验证叔链 (如果是挖矿, 确定叔链);
(2) 验证交易 (如果是挖矿, 确定交易);
(3) 应用奖励;

(4) 验证状态和随机数 (如果是挖矿, 计算有效的状态和随机数)。

11.1. 叔链验证. 验证叔链头意味着需要验证每个叔链头的有效性并且是当前区块父块的第 N 代, $N \leq 6$, 最多只能有 2 个叔链头, 公式如下:

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

k 表示直系关系:

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

s 表示兄弟关系:

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

$B(H)$ 表示对应的区块头 H.

11.2. 交易验证. 使用的燃料数量一定要和交易列表中列出的一致：区块燃料总使用量 B_{Hg} ，应该和这个区块中所有交易使用的燃料量之和相等:

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

11.3. 奖励. 对一个区块, 会奖励当前区块的受益账户及叔链账户。以 R_b 表示当前区块受益者的固定奖励额度; 对每个叔链, 奖励当前区块受益者固定奖励的 $\frac{1}{32}$, 叔链受益者的奖励额度取决于区块编号。我们定义函数 Ω :

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

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

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

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

如果叔链和区块的受益者账号冲突（比如：两个相同的叔链受益者账户或者一个叔链受益者账户和当前区块的受益者账户相同），额外的奖励会累积。

我们定义区块的奖励为 5 以太币:

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

11.4. 状态和随机数验证. 我们定义函数 Γ, 映射到区块 B 的初始状态:

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

其中, $\texttt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ 表示状态 $\boldsymbol{\sigma}_i$ 对应字典树根节点的哈希; 执行时会将其保存在状态数据库中, 字典树结构天生不易变, 所以它是高效的。

最后定义区块转换函数 Φ, 映射了一个不完整区块 B 到一个完整的区块 B':

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

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

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

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

其中 $\mathbf{d}$ 是一个数据集, 详见附录 J。

就像在本文开始定义的那样,Π 是状态转换函数，通过区块定稿函数 Ω 和交易演变函数 Υ 去定义 Π。

之前的定义中，$\mathbf{R}[n]_{\boldsymbol{\sigma}}$, $\mathbf{R}[n]_{\mathbf{l}}$ 和 $\mathbf{R}[n]_u$，表示第 n 个对应的状态、日志和累计使用的燃料（在元组中的第四个元素 $\mathbf{R}[n]_b$，之前以日志条目去定义)。前者简单定义为通过对之前的交易状态执行交易得到新的结果状态（如果是第一个交易，之前的交易状态为创始块的状态）：

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

对于 $B_{\mathbf{R}}[n]_u$, 我们定义从之前的交易转变对应交易的累计使用燃料（如果是第一个, 则为 0 ）:

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

对于 $\mathbf{R}[n]_{\mathbf{l}}$, 我们使用之前定义的交易执行函数 $\Upsilon^{\mathbf{l}}$。

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

最后, 我们定义函数 Π 是在交易结果状态 $\ell(B_{\mathbf{R}})_{\boldsymbol{\sigma}}$ 之上执行了回报函数 Ω 后的新状态::

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

到此, 我们完成了区块转换机制以及工作量证明 PoW 的定义。

11.5. 挖矿和工作量证明. 工作量证明（PoW）通过加密安全随机数, 来保证要获得要确定满足一定条件的 n, 要花费一定量的计算资源。通过计算给定意义的数值和难度系数来增加区块链安全性。然而因为挖出一个新的区块会附带一些奖励，工作量证明不仅是保证区块链权威的安全函数，而且也是一个健康的分配机制。

出于以上原因，工作量证明函数有两个重要的目标：首先，它应该尽可能的被更多人去接受。对定制特别硬件的需求或者回报，应该被减到最小。这使得分配模型尽可能开放，理想情况是，通过挖矿消耗电力获得以太币, 在全世界各个地方都是一样的比例。

第二，应该不允许获得超线性的收益，尤其是在有一个非常高的初始障碍条件下。如果允许这样, 一个资金充足的恶意者可以获得引起麻烦的网络挖矿算力, 并允许给他们获得超线性的汇报（按他们的意愿改变收益分布），并且会弱化网络的安全性。

比特币世界中一个灾难是 ASICs。有一些计算硬件仅仅是为了做一个简单的任务而存在。在比特币的案例中，这个任务就是 SHA256 哈希函数。当 ASICs 为了工作量证明函数而存在时，两个的目标都会变得危险。因此，一个可抵抗 ASIC 的工作量证明函数（比如难以在专用硬件上执行, 或者在专用硬件执行时并不划算）可以作为众所周知的银弹。

防止 ASIC 漏洞的两个方向：第一是去让它变成有序列的内存困难，比如：设计一个函数, 确定随机数需要大量的内存和带宽, 以至于这些内存不能被并行地去计算随机数。第二个方向是让计算变得更普遍化; 对于这个普遍化的计算, 使得特殊硬件和普通的桌面计算机计算起来都差不多。在以太坊 1.0 中，我们选择了第一个方向。

以太坊的工作量证明函数如下:

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

其中 $H_{\not\mathcal{H}}$ 是没有包含随机数和混合哈希元素的新区块头; H_n 是区块头的随机数; $\mathbf{d}$ 是一个需要计算混合哈希值的数据集合;H_d 是新区块的难度值（区块难度详见小结 10 ）。PoW 是工作量证明函数，这个函数根据数组做计算, 这个数

组中的第一个元素是混合哈希, 第二个元素是依赖于 H 和 $\mathbf{d}$ 的加密伪随机数。这个算法称为 Ethash , 下文会详细介绍。

11.5.1. *Ethash.* Ethash 是以太坊 1.0 的 PoW 算法。它是 Dagger-Hashimoto 的最后版本，Buterin [2013b] 和 Dryja[2014] 介绍过。再称它为 Dagger-Hashimoto 已不大合适, 因为很多算法的最初特性已经被大量地修改。这个算法的概况如下:

通过扫描区块头得到一个种子。根据种子可以计算一个初始大小为 $J_{cacheinit}$ 字节位随机数缓存。轻节点客户端可以保存这个缓存。根据缓存，我们能生成一个初始化为 $J_{datasetinit}$ 字节的数据集合，数据集合中每个元素依赖于缓存中的一小部分数据。全节点客户和矿工保存着数据集合。数据集合随时间线性增长。

挖矿涉及到组合数据集合的随机碎片并将他们哈希在一起。使用少量内存就可以验证, 可以借助缓存重新产生需要的数据片段, 所以仅需要保存缓存即可。大的数据集合每 J_{epoch} (译者注: 30000) 个区块更新一次，所以大多数矿工的工作都是读取这些数据集合，而不是尝试改变它。附录 J 有 Ethash 的详细解释。

12. 执行合约

有一些特别有用的合约模式; 我们会讨论其中两个, 分别是数据订阅和随机数。

12.1. 数据订阅**.** 一个数据订阅合约提供简单的服务：它允许外部的信息进入以太坊系统内。以太坊系统不会保证这个信息的精确度和及时性，这是第二方合约作者的任务, 第二方合约作者使用数据订阅并决定能否信任单个数据订阅服务。

通常的模式是, 通过消息调用触发单个以太坊中合约，得到一个关于外部现象的信息回应。一个例子可能是纽约当地温度。这个会作为合约执行并返回存储中的一些值。当然存储中的这些值比如温度需要维护以保证准确, 这样模式中的第二方才可以将其作为外部服务来运行以太坊节点, 当发现一个新区块时立即创建一个新的有效交易发送到合约并更新存储中的值。合约代码将仅仅接受包含指定服务的数据更新。

12.2. 随机数**.** 在确定系统中提供随机数字, 显然是一个不可能实现的任务。然而我们可以利用根据交易时还不可知的数据来生成伪随机数。比如区块哈希值、区块时间戳、区块受益人的地址。为了避免被矿工难以控制这些随机数, 建议使用 BLOCKHASH 操作获得最近 256 个区块的某个区块哈希作为伪随机数。对于获得这样一系列这样的数字，一个方法就是去增加一个常量并对结果做哈希。

13. 未来方向

未来状态数据库将不被强行维护所有之前的字典树状态。它应该对每个节点维护一定时间, 并抛弃较久远的且不是检查点的节点；检查点或允许特别的区块状态字典树穿越的数据中的系列节点, 可以用来最大限度的去替代一些计算, 来获得区块链中的任意一个状态。

区块链合并可以用来减少作为全节点或挖矿节点客户端需要下载的区块数量。及时将给定节点（也许每 10000 个区块）对应的的字典树压缩, 并在点对点网络中维护，且有效地重塑创世块。下载单个归档时, 可以减少下载的区块数量。

最后，或许会引导压缩区块链：可以抛弃在一定量的区块没有发送或接受交易的字典树状态节点, 以便减少状态数据库的增长。

13.1. 可扩展性**.** 可扩展性仍然是一个永恒的顾虑。因为一个一般化的状态交易函数，使得切分及并行化交易难以使用分而治之的策略。仍未解决的是，系统的平均交易值在增加, 但系统的能力范围却是固定的, 其中低价值的交易保存在主账本中已无经济意义, 因而会被忽略。尽管如此，一些策略存有可能发展为可扩展性的协议来解决此问题。

一些形式的层次结构，通过合并小的轻量的链到主区块或通过组合小的交易集合构建主区块, 或许可能实现交易组合和区块构建的并行化。并行化机制也可以这样实现: 针对有优先顺序的并列的区块链, 通过合并或抛弃无效交易来合并区块。

最后，可验证的计算，如果可用并且足够有效，可能提供一个方法允许工作量证明成为最后状态的验证。

14. 结论

我们已经介绍讨论并正式定义了以太坊协议。通过这个协议，读者可以太坊网络上部署一个节点并在一个去中心化的安全的开放系统中连接其它节点。合约可能会被授权, 以便计算性地确定和自动化执行交互规则。

15. 致谢

非常感谢 Aeron Buchanan 编写家园版本, Christoph Jentzsch 编写 Ethash 算法,Yoichi Hirai 做了 EIP-150 大多数更新。以太坊开发组织和社区提供了重要的维护和大量的矫正及建议, 其中包括 Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari, 当然, 还包含 Vitalik Buterin。

16. 可用性

本文的英文源文件维护在 `https://github.com/ethereum/yellowpaper`。一个自动生成的英文 PDF 格式文件见 `https://ethereum.github.io/yellowpaper/paper.pdf`。

本文的中文源文件维护在 `https://github.com/yuange1024/ethereum_yellowpaper`。一个自动生成的中文 PDF 格式文件见 `https://github.com/yuange1024/ethereum_yellowpaper/blob/master/ethereum_yellow_paper_cn.pdf`。(本段由译者增加)

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

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

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

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

Formally, we define R_b:

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

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

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

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

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

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

Thus we finish by formally defining R_l:

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

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

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:

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

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

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

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

Thus the high nibble of the first byte contains two flags; the lowest bit encoding the oddness of the length and the second-lowest encoding the flag t. The low nibble of the first byte is zero in the case of an even number of nibbles and the first nibble in the case of an odd number. All remaining nibbles (now an even number) fit properly into the remaining bytes.

Appendix D. Modified Merkle Patricia Tree

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(183) $\Xi_{\texttt{ECREC}} \equiv \Xi_{\texttt{PRE}}$ where:

(184) $g_r = 3000$

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

(186) if $|\mathbf{o}| = 32$:

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

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

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

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

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

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

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

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

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.

(195) $\Xi_{\texttt{SHA256}} \equiv \Xi_{\texttt{PRE}}$ where:

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

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

(198) $\Xi_{\texttt{RIP160}} \equiv \Xi_{\texttt{PRE}}$ where:

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

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

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

(202)

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

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

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

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

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

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

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

Appendix F. Signing Transactions

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

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

We assert the functions `ECDSASIGN`, `ECDSARESTORE` and `ECDSAPUBKEY`. These are formally defined in the literature.

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

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

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

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

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

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

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

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

where:

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix G. Fee Schedule

The fee schedule G is a tuple of 31 scalar values corresponding to the relative costs, in gas, of a number of abstract operations that a transaction may effect.

Name	Value	Description*
G_{zero}	0	Nothing paid for operations of the set W_{zero}.
G_{base}	2	Amount of gas to pay for operations of the set W_{base}.
$G_{verylow}$	3	Amount of gas to pay for operations of the set $W_{verylow}$.
G_{low}	5	Amount of gas to pay for operations of the set W_{low}.
G_{mid}	8	Amount of gas to pay for operations of the set W_{mid}.
G_{high}	10	Amount of gas to pay for operations of the set W_{high}.
$G_{extcode}$	700	Amount of gas to pay for operations of the set $W_{extcode}$.
$G_{balance}$	400	Amount of gas to pay for a BALANCE operation.
G_{sload}	200	Paid for a SLOAD operation.
$G_{jumpdest}$	1	Paid for a JUMPDEST operation.
G_{sset}	20000	Paid for an SSTORE operation when the storage value is set to non-zero from zero.
G_{sreset}	5000	Paid for an SSTORE operation when the storage value's zeroness remains unchanged or is set to zero.
R_{sclear}	15000	Refund given (added into refund counter) when the storage value is set to zero from non-zero.
$R_{selfdestruct}$	24000	Refund given (added into refund counter) for self-destructing an account.
$G_{selfdestruct}$	5000	Amount of gas to pay for a SELFDESTRUCT operation.
G_{create}	32000	Paid for a CREATE operation.
$G_{codedeposit}$	200	Paid per byte for a CREATE operation to succeed in placing code into state.
G_{call}	700	Paid for a CALL operation.
$G_{callvalue}$	9000	Paid for a non-zero value transfer as part of the CALL operation.
$G_{callstipend}$	2300	A stipend for the called contract subtracted from $G_{callvalue}$ for a non-zero value transfer.
$G_{newaccount}$	25000	Paid for a CALL or SELFDESTRUCT operation which creates an account.
G_{exp}	10	Partial payment for an EXP operation.
$G_{expbyte}$	50	Partial payment when multiplied by $\lceil \log_{256}(exponent) \rceil$ for the EXP operation.
G_{memory}	3	Paid for every additional word when expanding memory.
G_{txcreate}	32000	Paid by all contract-creating transactions after the *Homestead transition*.
$G_{txdatazero}$	4	Paid for every zero byte of data or code for a transaction.
$G_{txdatanonzero}$	68	Paid for every non-zero byte of data or code for a transaction.
$G_{transaction}$	21000	Paid for every transaction.
G_{log}	375	Partial payment for a LOG operation.
$G_{logdata}$	8	Paid for each byte in a LOG operation's data.
$G_{logtopic}$	375	Paid for each topic of a LOG operation.
G_{sha3}	30	Paid for each SHA3 operation.
$G_{sha3word}$	6	Paid for each word (rounded up) for input data to a SHA3 operation.
G_{copy}	3	Partial payment for *COPY operations, multiplied by words copied, rounded up.
$G_{blockhash}$	20	Payment for BLOCKHASH operation.

Appendix H. Virtual Machine Specification

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

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

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

(222)

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

(223)

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

where:

(224)

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

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

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

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

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

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

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

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

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

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

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

Note also that C_{mem} is the memory cost function (the expansion function being the difference between the cost before and after). It is a polynomial, with the higher-order coefficient divided and floored, and thus linear up to 724B of memory used, after which it costs substantially more.

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

(225)

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

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

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

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

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

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

0s: Stop and Arithmetic Operations

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

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

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

10s: Comparison & Bitwise Logic Operations

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

20s: SHA3

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

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

40s: Block Information

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

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

60s & 70s: Push Operations

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

80s: Duplication Operations

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

90s: Exchange Operations

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

a0s: Logging Operations

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

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

f0s: System operations

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

0xf4	DELEGATECALL	6	1	Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_\mathbf{s}[2]$. As a result, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. Otherwise exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_s, I_o, I_a, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_p, 0, I_v, \mathbf{i}, I_e + 1) & \text{if} \quad I_v \leqslant \boldsymbol{\sigma}[I_a]_b \ \wedge \ I_e < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ Note the changes (in addition to that of the fourth parameter) to the second and ninth parameters to the call Θ. This means that the recipient is in fact the same account as at present, simply that the code is overwritten *and* the context is almost entirely identical.
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_a\}$ $\boldsymbol{\sigma}'[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}]_b + \boldsymbol{\sigma}[I_a]_b$ $\boldsymbol{\sigma}'[I_a]_b \equiv 0$ $A'_r \equiv A_r + \begin{cases} R_{selfdestruct} & \text{if} \quad I_a \notin A_\mathbf{s} \\ 0 & \text{otherwise} \end{cases}$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{selfdestruct} + \begin{cases} G_{newaccount} & \text{if} \quad \boldsymbol{\sigma}[\boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}] = \varnothing \\ 0 & \text{otherwise} \end{cases}$

Appendix I. Genesis Block

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

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

Where 0_{256} refers to the parent hash, a 256-bit hash which is all zeroes; 0_{160} refers to the beneficiary address, a 160-bit hash which is all zeroes; 0_{2048} refers to the log bloom, 2048-bit of all zeros; 2^{17} refers to the difficulty; the transaction trie root, receipt trie root, gas used, block number and extradata are both 0, being equivalent to the empty byte array. The sequences of both ommers and transactions are empty and represented by (). $\texttt{KEC}\big((42)\big)$ refers to the Keccak hash of a byte array of length one whose first and only byte is of value 42, used for the nonce. $\texttt{KEC}\big(\texttt{RLP}\big(()\big)\big)$ value refers to the hash of the ommer lists in RLP, both empty lists.

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

Appendix J. Ethash

J.1. **Definitions.** We employ the following definitions:

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

J.2. **Size of dataset and cache.** The size for Ethash's cache $\mathbf{c} \in \mathbb{B}$ and dataset $\mathbf{d} \in \mathbb{B}$ depend on the epoch, which in turn depends on the block number.

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

The size of the dataset growth by $J_{datasetgrowth}$ bytes, and the size of the cache by $J_{cachegrowth}$ bytes, every epoch. In order to avoid regularity leading to cyclic behavior, the size must be a prime number. Therefore the size is reduced by

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

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

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

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

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

J.3. **Dataset generation.** In order the generate the dataset we need the cache $\mathbf{c}$, which is an array of bytes. It depends on the cache size c_{size} and the seed hash $\mathbf{s} \in \mathbb{B}_{32}$.

J.3.1. *Seed hash.* The seed hash is different for every epoch. For the first epoch it is the Keccak-256 hash of a series of 32 bytes of zeros. For every other epoch it is always the Keccak-256 hash of the previous seed hash:

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

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

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

J.3.2. *Cache.* The cache production process involves using the seed hash to first sequentially filling up c_{size} bytes of memory, then performing $J_{cacherounds}$ passes of the RandMemoHash algorithm created by Lerner [2014]. The initial cache $\mathbf{c}'$, being an array of arrays of single bytes, will be constructed as follows.

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

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

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

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

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

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

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

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

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

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

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

J.3.3. *Full dataset calculation.* Essentially, we combine data from $J_{parents}$ pseudorandomly selected cache nodes, and hash that to compute the dataset. The entire dataset is then generated by a number of items, each $J_{hashbytes}$ bytes in size:

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

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

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

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

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

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

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

J.4. **Proof-of-work function.** Essentially, we maintain a "mix" $J_{mixbytes}$ bytes wide, and repeatedly sequentially fetch $J_{mixbytes}$ bytes from the full dataset and use the E_{FNV} function to combine it with the mix. $J_{mixbytes}$ bytes of sequential access are used so that each round of the algorithm always fetches a full page from RAM, minimizing translation lookaside buffer misses which ASICs would theoretically be able to avoid.

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

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

(247)

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

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

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

The seed hash being:

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

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

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

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

The dataset $\mathbf{d}$ is obtained as described in section J.3.3.

The number of replicated sequences in the mix is:

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

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

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

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

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

(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)

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