以太坊：一种安全去中心化的通用交易账本

拜占庭版本 **f72032b - 2018-05-04**

原文作者：DR. GAVIN WOOD, GAVIN@ETHCORE.IO
正文翻译：崔广斌（微信号：YUANGE1024）；高天露（TIANLU.JORDEN.GAO@GMAIL.COM）
拜占庭版本增补、原译文校订：杨镇（RIVERS.YANG@ICLOUD.COM）

Abstract. 由密码学安全交易（cryptographically-secured transactions）所构成的区块链范式，已经通过一系列项目展示了它的实用性，不止是比特币。每一个这类项目都可以看作是一个基于去中心化的单实例计算资源所构建的简单应用程序。我们可以把这种范式称为具有共享状态（shared-state）的交易化单例状态机（transactional singleton machine）。

以太坊以更通用的方式实现了这种范式。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。我们将讨论它的设计、实现上的问题、它所提供的机遇以及我们所设想的未来障碍。

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，实现了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。尽管还很简陋，但类似域名币（Namecoin）这样的后续系统，把这种技术从最初的“货币应用”发展到了其它领域。

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密整合的端到端系统，这个系统提供了一种可信对象消息传递计算框架（a trustful object messaging compute framework），使开发者可以用一种前所未有的计算范式来构建软件。

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那些本来无法信任对方的个体之间的交易。这种不信任可能是因为地理上的分隔、对接的困难，或者是不兼容、不称职、不情愿、费用、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。

这里所提议的系统中的交易，有一些在现实世界中并不常见的属性。公允的裁判通常很难找到，但无私的算法解释器却可以天然地提供这种特性。自然语言必然是模糊的，会导致信息的缺失，同时那些平白的、旧有的成见很难被动摇，所以透明性，或者说通过交易日志和规则或代码指令来清晰地查看状态变化或者裁判结果的能力，在基于人来构建的系统中从来无法完美实现。

总的来说，我希望能提供一个系统，来给用户提供一些保证：无论是与其他个体、系统还是组织进行交互，他们都能完全信任可能的结果以及产生结果的过程。

1.2. 前人工作.

Buterin [2013a] 在 2013 年 11 月下旬第一次提出了这种系统的核心机制。虽然现在已经在很多方面都有了进化，但其核心特性没有变化。那就是：这是一个具有图灵完备的语言和实质上具有无限的内部交易数据存储容量的区块链。

Dwork and Naor [1992] 第一次提出了使用计算开销的密码学证明（“proof-of-work”，即工作量证明）来作为在互联网上传递价值信号的一种手段。这里所使用的价值信号，是作为对抗垃圾邮件的震慑机制的，并不是任何形式的货币；但这却极大地论证了一种承载强大的经济信号的基本数据通道的潜在可能，这种通道允许数据接受者不依赖任何信任就可以做出物理断言。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，代币被用来在点到点（peer-to-peer）文件交易中检查并确保“消费者”可以向服务“提供商”进行小额支付。通过在工作量证明中增加数字签名和账本，一种新的安全模型产生了。这种模型是为了防止历史记录被篡改，并使恶意用户无法伪造交易或者不公平的抗议服务的交付。五年后（2008 年），中本聪 Nakamoto [2008] 引入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，它也成为了第一个被广泛认可的全球化去中心化交易账本。

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，人们创建了许多其他的数字货币。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称解析系统，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 et al. [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去中心化来进行。瑞波币（Ripple），参见 Boutellier and 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，但可以表明法律的未来发展将会受到这种系统的影响。就此而言，以太坊或许可以视为这种加密-法律（cypto-law）系统的一种通用实现。

本文中所使用的术语列表，请参考附录 A。

2. 区块链范式

以太坊在整体上可以看作一个基于交易的状态机：起始于一个创世区块（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。而交易则成为了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户余额，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。可以正式地描述为：

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

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 $\boldsymbol{\sigma}$ 可以储存交易中的任意状态。

区块中记录着交易信息；区块之间通过密码学哈希（hash）以引用的方式链接起来。区块以流水账的方式组织起来，每个区块保留若干交易数据和前一个区块的引用，加上一个最终状态的标识符（最终状态本身不会保存到区块中——否则区块就太大了）。系统激励节点去挖矿（mine）。这种激励以状态转换函数的形式产生，会给指定的账户增加价值（即给指定的账户分配挖矿奖励，校订注）。

挖矿就是通过付出一定的努力（工作量）来与其它潜在的区块竞争一系列交易（一个区块）的记账权。它是通过密码学安全证明来实现的。这个机制就是工作量证明（proof-of-work），会在 11.5 详细讨论。

我们可以正式地描述为：

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

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

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

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 表示一个区块，包含一系列交易和一些其他组成部分；Π 是区块级的状态转换函数。

这就是区块链范式的基础，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. 单位. 为了激励网络中的计算，需要一种一致同意的转移价值方法。为了解决这个问题，以太坊设计了一种内置的货币——以太币（Ether），也就是我们所知的 ETH，有时也用古英语中的 Ð 表示。以太币最小的单位是 Wei（伟），所有货币值都以 Wei 的整数倍来记录。一个以太币等于 10^{18} Wei。以太币还有以下单位：

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

在当前状况下，任何在以太币的上下文中所使用的价值标示，比如货币、余额或者支付，都应以 Wei 为单位来计算。

2.2. 如何选择历史? 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之后创建新的区块，这必然会形成一个区块树。为了能在这个区块树上从根节点（the genesis block）到叶节点（包含最新交易的区块）形成一个一致的区块链，必须有一个共识方案。如果各个节点所认为的从根节点到叶节点的路径（最佳区块链）不同，那么就会发生分叉。

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包含权威的交易，其它节点则相信另外的区块包含权威的交易，其中就包含彻底不同或者不兼容的交易。必须要不惜一切代价避免这点，因为它会破坏整个系统信用。

我们使用了一个简易的 GHOST 协议版本来达成共识，参见 Sompolinsky and Zohar [2013]。我们会在第 10 章中详细说明。

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。

3. 约定

我使用了大量的排印约定来表示公式中的符号，其中一些需要特别说明：

有两个高度结构化的顶层状态值，用粗体小写希腊字母 $\boldsymbol{\sigma}$ 表示世界状态（world-state）；用 $\boldsymbol{\mu}$ 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 是总体费用函数。此外，可能会使用下角标表示一些特别的变量，例如：C_{SSTORE} 是执行 SSTORE 操作的费用函数。对于一些可能是外部定义的函数，我可能会使用打印机文字字体，例如使用 KEC （一般指单纯的 Keccak）来表示 Keccak-256 哈希函数（这是由 Bertoni et al. [2017] 提出的，在 SHA-3 竞争中获胜的那种算法，而不是最新的版本）。也会用 KEC512 来表示 Keccack-512 哈希函数。

元组通常使用一个大写字母来表示，例如用 T 表示一个以太坊交易。也可以通过使用下标来表示其中的一个独立组成部分，例如用 T_{n} 来表示交易的 nonce。下角标形式则用于表示它们的类型；例如：大写的下角标表示下角标所对应的组成部分所构成的元组。

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 交易的 nonce。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所需要的条目数量。

任意长度的序列通常用粗体小写字母表示，例如 $\mathbf{o}$ 表示消息调用中输出的字节序列数据。对于特别重要的值，可能会使用粗体大写字母。

我们认为标量都是正整数且属于集合 $\mathbb{N}$。所有的字节序列属于集合 $\mathbb{B}$，附录 B 给出了正式的定义。可以用下角标表示这样的序列集合的限定长度，因此，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 $\mathbb{N}_{256}$ 表示。详细定义参见 4.3。

使用方括号表示序列中的一个元素或子序列，例如 $\boldsymbol{\mu}_{\mathbf{s}}[0]$ 表示虚拟机栈中的第一个条目。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 $\boldsymbol{\mu}_{\mathbf{m}}[0..31]$ 表示虚拟机内存中的前 32 个条目。

以全局状态 $\boldsymbol{\sigma}$ 为例，它表示一系列的账户（账户自身是元组），方括号被用来表示一个独立的账户。

当去考虑现有的变量时，我遵循在给定的范围内去定义的原则，如果我们使用占位符 $\square$ 表示未修改的“输入”值，那么就可以使用 $\square'$ 来表示修改过的可用值，$\square^*$，$\square^{**}$ &c 表示中间值。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，例如一个给定函数 f，函数 f^* 表示一个相似的、对集合中所有元素都生效的函数映射。正式定义参见 4.3。

以上，我定义了大量的函数。一个最常见的函数是 ℓ，它表示给定序列中的最后一个条目：

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

4. 区块、状态和交易

在介绍了以太坊的基本概念之后，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（*state*）是在地址（160 位的标识符）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 树上（即 *trie*，详见附录 D）。这个 trie 需要一个简单的后端数据库去维护字节数组到字节数组的映射；我们称这个后端数据库为状态数据库。它有一系列的好处: 第一，这个结构的根节点是基于密码学依赖于所有内部数据的，它的哈希可以作为整个系统状态的一个安全标识；第二，作为一个不变的数据结构，我们可以通过简单地改变根节点哈希来召回任何一个先前的状态（在根节点哈希已知的条件下）。因为我们在区块链中储存了所以这样的根节点哈希值，所以我们可以很容易地恢复到特定的历史状态。

账户状态 $\boldsymbol{\sigma}[a]$ 包含以下四个字段：

nonce: 这个值等于由此账户地址发出的交易数量，或者由这个账户所创建的合约数量（当这个账户有关联代码时）。$\boldsymbol{\sigma}[a]_n$ 即表示状态 $\boldsymbol{\sigma}$ 中的地址 a 的 nonce 值。

balance: $\boldsymbol{\sigma}[a]_\mathrm{b}$，表示这个账户地址拥有多少 Wei（即账户余额，译者注）。

storageRoot: 保存了账户的存储内容的 Merkle Patricia 树的根节点的 256 位哈希值，这个树中保存的是 256 位整数键值的 Keccak 256 位哈希值到 256 位整数值的 RLP 编码的映射。这个哈希定义为 $\boldsymbol{\sigma}[a]_s$。

codeHash: 这个账户的 EVM 代码哈希值——当这个地址接收到一个消息调用时，这些代码会被执行；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这样的代码片段哈希，以便后续使用。这个哈希可以定义为 $\boldsymbol{\sigma}[a]_c$，然后我们用 $\mathbf{b}$ 来表示代码，则有 $\texttt{KEC}(\mathbf{b}) = \boldsymbol{\sigma}[a]_c$。

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

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

Trie 中的键值对集合函数 L_I^* 被定义为适用于基础函数 L_I 中所有元素的变换：

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

其中：

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

需要说明的是，$\boldsymbol{\sigma}[a]_\mathbf{s}$ 不应算作这个账户的“物理”成员，它不会参与之后的序列化。

如果 **codeHash** 字段是一个空字符串的 Keccak-256 哈希，也就是说 $\boldsymbol{\sigma}[a]_c = \texttt{KEC}\big(()\big)$，那么这个节点则表示一个简单账户，有时简称为“非合约”账户。

因此我们可以定义一个世界状态函数 L_S：

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

其中

$$p(a) \equiv \big(\texttt{KEC}(a), \texttt{RLP}\big((\boldsymbol{\sigma}[a]_\mathrm{n}, \boldsymbol{\sigma}[a]_\mathrm{b}, \boldsymbol{\sigma}[a]_\mathrm{s}, \boldsymbol{\sigma}[a]_\mathrm{c})\big)\big) \tag{10}$$

函数 L_S 和 Trie 函数一起用来提供一个世界状态的简短标识（哈希）。我们假定：

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

其中，v 是账户有效性验证函数：

$$v(x) \equiv x_\mathrm{n} \in \mathbb{N}_{256} \wedge x_\mathrm{b} \in \mathbb{N}_{256} \wedge x_\mathrm{s} \in \mathbb{B}_{32} \wedge x_\mathrm{c} \in \mathbb{B}_{32} \tag{12}$$

如果一个账户没有代码，它将是 *empty*（空的），且 nonce 和 balance 均为 0：

$$\texttt{EMPTY}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_\mathrm{c} = \texttt{KEC}\big(()\big) \wedge \boldsymbol{\sigma}[a]_\mathrm{n} = 0 \wedge \boldsymbol{\sigma}[a]_\mathrm{b} = 0 \tag{13}$$

即使是所谓的预编译合约也可能处于 empty 状态。这是因为它们的账户状态并不总是包含可以表述它们的行为的代码。

当一个账户的状态不存在或为 empty 时，就表示它 *dead*（死了）：

$$\texttt{DEAD}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a] = \varnothing \vee \texttt{EMPTY}(\boldsymbol{\sigma}, a) \tag{14}$$

4.2. 交易. 交易（符号，T）是个单一的加密学签名的指令，通常由以太坊系统之外的操作者创建。我们假设外部的操作者是人，软件工具则用于构建和散播 [1]。这里的交易有两种类型：一种表现为消息调用，另一种则通过代码创建新的账户（称为“合约创建”）。这两种类型的交易都有一些共同的字段：

nonce: 由交易发送者发出的的交易的数量，由 T_n 表示。

gasPrice: 为执行这个交易所需要进行的计算步骤消耗的每单位 *gas* 的价格，以 Wei 为单位，由 T_p 表示。

gasLimit: 用于执行这个交易的最大 gas 数量。这个值须在交易开始前设置，且设定后不能再增加，由 T_g 表示。

to: 160 位的消息调用接收者地址；对与合约创建交易，用 $\varnothing$ 表示 $\mathbb{B}_0$ 的唯一成员。此字段由 T_t 表示

value: 转移到接收者账户的 Wei 的数量；对于合约创建，则代表给新建合约地址的初始捐款。由 T_v 表示。

v, r, s: 与交易签名相符的若干数值，用于确定交易的发送者，由 T_w，T_r 和 T_s 表示。详见附录 F。

此外，合约创建还包含以下字段:

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

init 是 EVM 代码片段；它将返回 **body**，这是这个账户每次接收到消息调用时会执行的代码（通过一个交易或者代码的内部执行）。**init** 代码仅会在合约创建时被执行一次，然后就会被丢弃。

与此相对，一个消息调用交易包括:

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

附录 F 详细描述了将交易映射到发送者的函数 S。这种映射通过 SECP-256k1 曲线的 ECDSA 算法实现，使用交易哈希（除去后 3 个签名字段）作为数据来进行签名。目前我们先简单地使用 $S(T)$ 表示指定交易 T 的发送者。

$$L_T(T) \equiv \begin{cases} (T_\mathrm{n}, T_\mathrm{p}, T_\mathrm{g}, T_\mathrm{t}, T_\mathrm{v}, T_\mathbf{i}, T_\mathrm{w}, T_\mathrm{r}, T_\mathrm{s}) & \text{if } T_\mathrm{t} = \varnothing \\ (T_\mathrm{n}, T_\mathrm{p}, T_\mathrm{g}, T_\mathrm{t}, T_\mathrm{v}, T_\mathbf{d}, T_\mathrm{w}, T_\mathrm{r}, T_\mathrm{s}) & \text{otherwise} \end{cases} \tag{15}$$

在这里，我们假设除了任意长度的字节数组 $T_\mathbf{i}$ 和 $T_\mathbf{d}$ 以外，所有变量都是作为整数来进行 RLP 编码的。

$$\begin{aligned} &T_\mathrm{n} \in \mathbb{N}_{256} \quad \wedge \quad T_\mathrm{v} \in \mathbb{N}_{256} \quad \wedge \quad T_\mathrm{p} \in \mathbb{N}_{256} \quad \wedge \\ &T_\mathrm{g} \in \mathbb{N}_{256} \quad \wedge \quad T_\mathrm{w} \in \mathbb{N}_{5} \quad \wedge \quad T_\mathrm{r} \in \mathbb{N}_{256} \quad \wedge \\ &T_\mathrm{s} \in \mathbb{N}_{256} \quad \wedge \quad T_\mathbf{d} \in \mathbb{B} \quad \wedge \quad T_\mathbf{i} \in \mathbb{B} \end{aligned} \tag{16}$$

其中

$$\mathbb{N}_\mathrm{n} = \{P : P \in \mathbb{N} \wedge P < 2^\mathrm{n}\} \tag{17}$$

[1] 特别是，这样的“工具”最终将会从基于人类行为的初始化中剥离——或者人类可能变成中立状态——当到达某个临界点时，这种工具可能会被看作一种自治的代理。例如，合约可以向那些发送交易来启动它们的人们提供奖金。

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，或者当创建合约时（将等于 $\varnothing$）是 RLP 空字节序列，所以是 $\mathbb{B}_0$ 的成员：

$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_{\mathrm{t}} \neq \varnothing \\ \mathbb{B}_0 & \text{otherwise} \end{cases} \tag{18}$$

4.3. 区块. 在以太坊中，区块是由以下部分组成的：一些相关信息片段组成的集合（称为 block *header*，即区块头）；组成区块的交易 $\mathbf{T}$ 和 其它一些区块头 $\mathbf{U}$ （这是一些父区块与当前区块的爷爷辈区块相同的区块，这样的区块称为 *ommers*[2]）。区块头包含的的信息如下：

parentHash: 父区块头的 Keccak 256 位哈希，由 H_{p} 表示。

ommersHash: 当前区块的 ommers 列表的 Keccak 256 位哈希，由 H_{o} 表示。

beneficiary: 成功挖到这个区块所得到的所有交易费的 160 位接收地址，由 H_{c} 表示。

stateRoot: 所有交易被执行完且区块定稿后的状态树（state trie）根节点的 Keccak 256 位哈希。

transactionsRoot: 由当前区块中所包含的所有交易所组成的树结构（transaction trie）根节点的 Keccak 256 位哈希，由 H_{t} 表示。

receiptsRoot: 由当前区块中所有交易的收据所组成的树结构（receipt trie）根节点的 Keccak 256 位哈希。由 H_{e} 表示。

logsBloom: 由当前区块中所有交易的收据数据中的可索引信息（产生日志的地址和日志主题）组成的 Bloom 过滤器，由 H_{b} 表示。

difficulty: 当前区块难度水平的纯量值，它可以根据前一个区块的难度水平和时间戳计算得到，由 H_{d} 表示。

number: 当前区块的祖先的数量，由 H_{i} 表示。创世区块的这个数量为 0。

gasLimit: 目前每个区块的 gas 开支上限，由 H_{l} 表示。

gasUsed: 当前区块的所有交易所用掉的 gas 之和，由 H_{g} 表示。

timestamp: 当前区块初始化时的 Unix 时间戳，由 H_{s} 表示。

extraData: 与当前区块相关的任意字节数据，但必须在 32 字节以内，由 H_{x} 表示。

mixHash: 一个 256 位的哈希值，用来与 nonce 一起证明当前区块已经承载了足够的计算量，由 H_{m} 表示。

nonce: 一个 64 位的值，用来与 mixHash 一起于证明当前区块已经承载了足够的计算量，由 H_{n} 表示。

区块的另两个组成部分就是 ommer 区块头（与以上格式相同）列表 $B_{\mathbf{U}}$ 和一系列的交易 $B_{\mathbf{T}}$。我们可以表示一个区块 B：

$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}}) \tag{19}$$

4.3.1. 交易收据. 为了能使交易信息对零知识证明、索引和搜索都是有用的，我们将每个交易执行过程中的一些特定信息编码为交易收据。我们以 $B_{\mathbf{R}}[i]$ 表示第 i 个交易的收据，并把收据信息保存在一个以索引为键的树（index-keyed trie）中，树的根节点用 H_{e} 保存到区块头中。

交易收据 R 是一个包含四个条目的元组：包含交易收据的区块中当交易发生后的累积 gas 使用量 R_{u}；交易过程中创建的日志集合 R_{l}；由这些日志信息所构成的 Bloom 过滤器 R_{b} 和交易的状态代码 R_{z}：

$$R \equiv (R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathrm{l}}, R_{\mathrm{z}}) \tag{20}$$

函数 L_R 是将交易收据转换为 RLP 序列化字节数组的预处理函数：

$$L_R(R) \equiv (0 \in \mathbb{B}_{256}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathrm{l}}) \tag{21}$$

其中，$0 \in \mathbb{B}_{256}$ 代替了先前协议版本中的交易前状态根（the pre-transaction state root）。

我们要求状态代码 R_{z} 是一个整数。

$$R_{\mathrm{z}} \in \mathbb{N} \tag{22}$$

我们要求累积的 gas 使用量 R_{u} 是一个正整数，且日志的 Bloom R_{b} 是一个 2048 位（256 字节）数据的哈希：

$$R_{\mathrm{u}} \in \mathbb{N} \quad \wedge \quad R_{\mathrm{b}} \in \mathbb{B}_{256} \tag{23}$$

R_{l} 是一系列的日志项 $(O_0, O_1, ...)$。一个日志项 O 是一个记录了日志产生者的地址 O_a；一系列 32 字节的日志主题 $O_{\mathbf{t}}$ 和一些字节数据 $O_{\mathbf{d}}$ 所组成的元组：

$$O \equiv (O_{\mathrm{a}}, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, ...), O_{\mathbf{d}}) \tag{24}$$

$$O_{\mathrm{a}} \in \mathbb{B}_{20} \quad \wedge \quad \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \quad \wedge \quad O_{\mathbf{d}} \in \mathbb{B} \tag{25}$$

我们定义 Bloom 过滤器函数 M 将一个日志项精简为一个 256 字节哈希：

$$M(O) \equiv \bigvee_{t \in \{O_{\mathrm{a}}\} \cup O_{\mathbf{t}}} \left(M_{3:2048}(t)\right) \tag{26}$$

其中 $M_{3:2048}$ 是一个特别的 Bloom 过滤器，它通过设置 2048 位中的 3 位数值来给定一个随机的字节序列。这是通过取得对一个字节序列中的前 3 对字节的 Keccak-256 哈希值的低位 11 位数据实现的[3]：

$$M_{3:2048}(\mathbf{x} : \mathbf{x} \in \mathbb{B}) \equiv \mathbf{y} : \mathbf{y} \in \mathbb{B}_{256} \quad \text{where:} \tag{27}$$

$$\mathbf{y} = (0, 0, ..., 0) \quad \text{except:} \tag{28}$$

$$\forall_{i \in \{0,2,4\}} : \mathcal{B}_{m(\mathbf{x},i)}(\mathbf{y}) = 1 \tag{29}$$

$$m(\mathbf{x}, i) \equiv \mathtt{KEC}(\mathbf{x})[i, i+1] \bmod 2048 \tag{30}$$

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

4.3.2. 整体有效性. 当且仅当一个区块同时满足以下几个条件，我们才认为它是有效的：它必须由内部一致的 ommer 和交易区块哈希值所组成，且基于起始状态 $\boldsymbol{\sigma}$（由父区块的最终状态继承而来）按顺序执行完成所有的给定交易 $B_{\mathbf{T}}$（就像在 11 节所介绍的那样）后所达到的标识符 H_{r} 的一个新的状态：

$$\begin{aligned}
H_{\mathrm{r}} &\equiv \mathtt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) && \wedge \\
H_{\mathrm{o}} &\equiv \mathtt{KEC}(\mathtt{RLP}(L^*_H(B_{\mathbf{U}}))) && \wedge \\
H_{\mathrm{t}} &\equiv \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{N} : \\
&\qquad p(i, L_T(B_{\mathbf{T}}[i]))\}) && \wedge \\
H_{\mathrm{e}} &\equiv \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{N} : \\
&\qquad p(i, L_R(B_{\mathbf{R}}[i]))\}) && \wedge \\
H_{\mathrm{b}} &\equiv \textstyle\bigvee_{\mathbf{r} \in B_{\mathbf{R}}} \left(\mathbf{r}_{\mathrm{b}}\right)
\end{aligned} \tag{31}$$

其中 $p(k, v)$ 就是简单的 RLP 变换对，在这里，k 是交易在区块中的索引，v 是交易收据：

$$p(k, v) \equiv \left(\mathtt{RLP}(k), \mathtt{RLP}(v)\right) \tag{32}$$

此外：

$$\mathtt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_{\mathrm{r}}} \tag{33}$$

这样，$\mathtt{TRIE}(L_S(\boldsymbol{\sigma}))$ 是以状态 $\boldsymbol{\sigma}$ 和以 RLP 编码的值作为键值对数据的 Merkle Patricia 树根节点哈希，$P(B_H)$ 就是父区块 B。

[2] *ommer* 的意思和自然界中的“父母的兄弟姐妹”最相近，参见 https://nonbinary.miraheze.org/wiki/Gender_neutral_language#Aunt.2FUncle

[3] 11 位 2 进制数值 $= 2^{2048}$，其低位的 11 位数值也就是对操作数进行模 2048 的运算结果，这就是“取得对一个字节序列中的前 3 对字节的 Keccak-256 哈希值的低位 11 位数据”的情况。

这些源于交易计算所产生的值，具体来讲就是交易收据 $B_{\mathbf{R}}$，以及通过交易的状态累积函数所定义的 Π，会在 11.4 节详细说明。

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。与交易收据预备函数 L_R非常相似，当需要进行 RLP 变换时，我们要求结构的类型和顺序如下：

$$L_H(H) \equiv (\ H_\mathrm{p}, H_\mathrm{o}, H_\mathrm{c}, H_\mathrm{r}, H_\mathrm{t}, H_\mathrm{e}, H_\mathrm{b}, H_\mathrm{d}, H_\mathrm{i}, H_\mathrm{l}, H_\mathrm{g}, H_\mathrm{s}, H_\mathrm{x}, H_\mathrm{m}, H_\mathrm{n}\ ) \tag{34}$$

$$L_B(B) \equiv \big(L_H(B_H), L^*_T(B_\mathbf{T}), L^*_H(B_\mathbf{U})\big) \tag{35}$$

其中 L^*_T 和 L^*_H 是适用于所有元素的序列转换，因此：

$$f^*\big((x_0, x_1, ...)\big) \equiv \big(f(x_0), f(x_1), ...\big) \quad \text{对所有函数 } f \tag{36}$$

其组成要素的类型定义如下：

$$\begin{array}{lllllll} H_\mathrm{p} \in \mathbb{B}_{32} & \wedge & H_\mathrm{o} \in \mathbb{B}_{32} & \wedge & H_\mathrm{c} \in \mathbb{B}_{20} & \wedge \\ H_\mathrm{r} \in \mathbb{B}_{32} & \wedge & H_\mathrm{t} \in \mathbb{B}_{32} & \wedge & H_\mathrm{e} \in \mathbb{B}_{32} & \wedge \\ H_\mathrm{b} \in \mathbb{B}_{256} & \wedge & H_\mathrm{d} \in \mathbb{N} & \wedge & H_\mathrm{i} \in \mathbb{N} & \wedge \\ H_\mathrm{l} \in \mathbb{N} & \wedge & H_\mathrm{g} \in \mathbb{N} & \wedge & H_\mathrm{s} \in \mathbb{N}_{256} & \wedge \\ H_\mathrm{x} \in \mathbb{B} & \wedge & H_\mathrm{m} \in \mathbb{B}_{32} & \wedge & H_\mathrm{n} \in \mathbb{B}_8 & \end{array} \tag{37}$$

其中

$$\mathbb{B}_\mathrm{n} = \{B : B \in \mathbb{B} \wedge \|B\| = n\} \tag{38}$$

我们现在有了一个严格的区块结构构造说明。RLP 函数 RLP（见附录 B）提供了权威的方法来把这个结构转换为一个可以通过网络传输或在本地存储的字节序列。

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

$$P(H) \equiv B' : \mathtt{KEC}(\mathtt{RLP}(B'_H)) = H_\mathrm{p} \tag{39}$$

其区块号等于它的父区块号加 1：

$$H_\mathrm{i} \equiv P(H)_{H_\mathrm{i}} + 1 \tag{40}$$

我们将一个区块头 H 的权威难度值定义为 $D(H)$：

$$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_\mathrm{i} = 0 \\ \max\big(D_0, P(H)_{H_\mathrm{d}} + x \times \varsigma_2 + \epsilon\big) & \text{otherwise} \end{cases} \tag{41}$$

其中：

$$D_0 \equiv 131072 \tag{42}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_\mathrm{d}}}{2048} \right\rfloor \tag{43}$$

$$\varsigma_2 \equiv \max\left(y - \left\lfloor \frac{H_\mathrm{s} - P(H)_{H_\mathrm{s}}}{9} \right\rfloor, -99\right) \tag{44}$$

$$y \equiv \begin{cases} 1 & \text{if} \|P(H)_\mathbf{U}\| = 0 \\ 2 & \text{otherwise} \end{cases}$$

$$\epsilon \equiv \left\lfloor 2^{\lfloor H'_\mathrm{i} \div 100000 \rfloor - 2} \right\rfloor \tag{45}$$

$$H'_\mathrm{i} \equiv \max(H_\mathrm{i} - 3000000, 0) \tag{46}$$

注意，D_0 是创世区块的难度值。就像下面介绍的那样，*Homestead* 难度值参数 ς_2 被用来影响出块时间的动态平衡，它已经通过由 Buterin [2015] 提出的 EIP-2 实现了。在 Homestead 版本中，难度符号 ϵ 会越来越快地使难度值缓慢增长（每 10 万个区块），从而增加区块时间差别，也为向权益证明（proof-of-stake）的切换增加了时间压力。这个效果就是所谓的“难度炸弹”（“difficulty bomb”）或“冰河时期”（“ice age”）在由 Schoedon and Buterin [2017] 提出的 EIP-649 中进行了解释，并用来推迟早先在 EIP-2 中的实现。ς_2 也在 EIP-100 通过使用 x（即上面公式中的矫正参数）和分母 9 进行了修改，用来达到由 Buterin [2016] 提出的对包含叔区块（uncle blocks）在内的平均出块时间的调整效果。最终，在拜占庭版本中，伴随 EIP-649，我们通过伪造一个区块号 H'_i 来延迟冰河时期的来临。这是通过用实际区块号减去 300 万来获得的。换句话说，就是减少 ϵ 和区块间的时间间隔，来为权益证明的开发争取更多的时间并防止网络被“冻结”。

区块头 H 的 gas 限制 H_l 需要满足下列条件：

$$\begin{aligned} & H_\mathrm{l} < P(H)_{H_\mathrm{l}} + \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} > P(H)_{H_\mathrm{l}} - \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} \geqslant 5000 \end{aligned} \tag{47}$$

H_s 是区块 H 的时间戳（以 Unix 的 time() 函数的形式），需要满足下列条件:

$$H_\mathrm{s} > P(H)_{H_\mathrm{s}} \tag{48}$$

这个机制保证了区块时间的动态平衡；如果最近的两个区块间隔较短，则会导致难度值增加，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，如果最近的两个区块间隔过长，难度值和下一个区块的预期出块时间也会减少。

这里的 nonce H_n 需满足下列关系：

$$n \leqslant \frac{2^{256}}{H_\mathrm{d}} \quad \wedge \quad m = H_\mathrm{m} \tag{49}$$

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

其中，$H_{\not\mathbf{n}}$ 是不包含 nonce 和 mixHash 的新区块的区块头 H；$\mathbf{d}$ 是当前的 DAG（即有向无环图，是用来计算 mixHash 的一种大型数据集合）和工作量证明函数 PoW（参见 11.5）：这取决于一个数组的值，数组的第一个元素是用来证明使用了一个正确的 DAG 的混合哈希（mix-hash）；数组的第二个元素是伪随机数，密码学依赖于 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 是成比例变化的。

这就是区块链的安全基础，也是一个恶意节点不能用其新创建的区块来覆盖（“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。

这样，我们定义这个区块头的验证函数 $V(H)$ 为：

$$\begin{aligned} V(H) \equiv\ & n \leqslant \frac{2^{256}}{H_\mathrm{d}} \wedge m = H_\mathrm{m} \quad \wedge \\ & H_\mathrm{d} = D(H) \quad \wedge \\ & H_\mathrm{g} \leq H_\mathrm{l} \quad \wedge \\ & H_\mathrm{l} < P(H)_{H_\mathrm{l}} + \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} > P(H)_{H_\mathrm{l}} - \left\lfloor \frac{P(H)_{H_\mathrm{l}}}{1024} \right\rfloor \quad \wedge \\ & H_\mathrm{l} \geqslant 5000 \quad \wedge \\ & H_\mathrm{s} > P(H)_{H_\mathrm{s}} \quad \wedge \\ & H_\mathrm{i} = P(H)_{H_\mathrm{i}} + 1 \quad \wedge \\ & \|H_\mathrm{x}\| \leq 32 \end{aligned} \tag{50}$$

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

此外，**extraData** 最多 32 字节。

5. Gas 及其支付

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 中与不同计算相关的费用说明）为单位计算。任意的程序片段（包括合约创建、消息调用、分配资源以及访问账户 storage、在虚拟机上执行操作等）都会有一个普遍认同的 gas 消耗。

每一个交易都要指定一个 gas 上限：**gasLimit**。这些 gas 会从发送者的账户的 balance 中扣除。这种购买是通过同样在交易中指定的 **gasPrice** 来完成的。如果这个账户的 balance 不能支持这样的购买，交易会被视为无效交易。之所以将其命名为 **gasLimit**，是因为剩余的 gas 会在交易完成后被返还（与购买时同样价格）到发送者账户。Gas 不会在交易执行之外存在，因此对于可信任的账户，应该设置一个相对较高的 gas 上限。

通常来说，以太币（Ether）是用来购买 gas 的，未返还的部分就会移交到 *beneficiary* 的地址（即一般由矿工所控制的一个账户地址）。交易者可以随意指定 **gasPrice**，然而矿工也可以任意地忽略某个交易。一个高 gas 价格的交易将花费发送者更多的以太币，也就将移交给矿工更多的以太币，因此这个交易自然会被更多的矿工选择打包进区块。通常来说，矿工会选择公告他们执行交易的最低 gas 价格，交易者们也就可以据此来提供一个具体的价格。因此，会有一个（加权的）最低的可接受 gas 价格分布，交易者们则需要在降低 gas 价格和使交易能最快地被矿工打包间进行权衡。

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。我们假定任意交易在执行时前都要先通过初始的基础有效性测试。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 nonce 是有效的（等于发送者账户当前的 nonce）；
(4) gas 上限不小于交易所要使用的 gas g_0;
(5) 发送者账户的 balance 应该不少于实际费用 v_0，且需要提前支付。

这样，我们可以定义状态转移函数 Υ，其中 T 表示交易，$\boldsymbol{\sigma}$ 表示状态：

$$\boldsymbol{\sigma}' = \Upsilon(\boldsymbol{\sigma}, T) \tag{51}$$

因此 $\boldsymbol{\sigma}'$ 是交易后的状态。我们也定义 Υ^{g} 为交易执行所消耗的 gas 数量；$\Upsilon^{\mathbf{l}}$ 为交易过程中累积产生的日志项；以及 Υ^{z} 为交易结果的状态代码。这些都会在后文正式定义。

6.1. 子状态.

交易的执行过程中会累积产生一些特定的信息，我们称为交易子状态，用 A 来表示，它是个元组：

$$A \equiv (A_{\mathbf{s}}, A_{\mathbf{l}}, A_{\mathbf{t}}, A_{\mathrm{r}}) \tag{52}$$

元组内容包括自毁集合 $A_{\mathbf{s}}$：一组应该在交易完成后被删除的账户。 $A_{\mathbf{l}}$ 是一系列的日志：这是一些列针对 VM 代码执行的归档的、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）来简单地跟踪合约调用。 $A_{\mathbf{t}}$ 是交易所接触过的账户集合，其中的空账户可以在交易结束时删除。最后是 A_{r}，也就是应该返还的余额；当使用 SSTORE 指令将非 0 的合约 storage 重置为 0 时，这个余额会增加。虽然不是立即返还的余额，但可以部分抵消整体执行费用。

我们定义空的子状态 A^0，它没有自毁、没有日志、没有接触过的账户且返还余额为 0：

$$A^0 \equiv (\varnothing, (), \varnothing, 0) \tag{53}$$

6.2. 执行.

我们定义固有的 gas 消耗 g_0，它是需要在交易执行前支付的 gas 数量：

$$g_0 \equiv \sum_{i \in T_{\mathbf{i}}, T_{\mathbf{d}}} \begin{cases} G_{\mathrm{txdatazero}} & \text{if} \quad i = 0 \\ G_{\mathrm{txdatanonzero}} & \text{otherwise} \end{cases} \tag{54}$$

$$+ \begin{cases} G_{\mathrm{txcreate}} & \text{if} \quad T_{\mathrm{t}} = \varnothing \\ 0 & \text{otherwise} \end{cases} \tag{55}$$

$$+ G_{\mathrm{transaction}} \tag{56}$$

其中，$T_{\mathbf{i}}$，$T_{\mathbf{d}}$ 是交易附带的关联数据的字节序列和 EVM 初始化代码的字节序列，具体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，则会增加 G_{txcreate}，否则不增加。G 的完整定义参见附录 G。

预支付的费用 v_0 计算如下:

$$v_0 \equiv T_{\mathrm{g}} T_{\mathrm{p}} + T_{\mathrm{v}} \tag{57}$$

我们通过下列条件检查有效性：

$$\begin{aligned} S(T) &\neq \varnothing \quad \wedge \\ \boldsymbol{\sigma}[S(T)] &\neq \varnothing \quad \wedge \\ T_{\mathrm{n}} &= \boldsymbol{\sigma}[S(T)]_{\mathrm{n}} \quad \wedge \\ g_0 &\leqslant T_{\mathrm{g}} \quad \wedge \\ v_0 &\leqslant \boldsymbol{\sigma}[S(T)]_{\mathrm{b}} \quad \wedge \\ T_{\mathrm{g}} &\leqslant B_{H1} - \ell(B_{\mathbf{R}})_{\mathrm{u}} \end{aligned} \tag{58}$$

注意最后的条件: 交易的 gas 上限 T_{g} 和由 $\ell(B_{\mathbf{R}})_{\mathrm{u}}$ 所给定的这个区块先前已经使用的 gas 数量之和，不可以超过当前区块的 **gasLimit** B_{H1}。

有效交易的执行起始于一个对状态不能撤回的改变：发送者账户的 nonce，即 $S(T)$ 的 nonce 会加 1，并且从账户余额扣减预付费用 $T_{\mathrm{g}} T_{\mathrm{p}}$。进行实际计算的可用 gas g 被定义为 $T_{\mathrm{g}} - g_0$。无论是合约创建还是消息调用，计算都会产生一个最终状态（可能等于当前的状态），这种改变是确定的且从来不会无效：这样来看，其实并不存在无效的交易。

我们定义检查点状态 $\boldsymbol{\sigma}_0$：

$$\boldsymbol{\sigma}_0 \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{59}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\mathrm{b}} \equiv \boldsymbol{\sigma}[S(T)]_{\mathrm{b}} - T_{\mathrm{g}} T_{\mathrm{p}} \tag{60}$$

$$\boldsymbol{\sigma}_0[S(T)]_{\mathrm{n}} \equiv \boldsymbol{\sigma}[S(T)]_{\mathrm{n}} + 1 \tag{61}$$

从 $\boldsymbol{\sigma}_0$ 到 $\boldsymbol{\sigma}_P$ 的计算依赖于交易的类型。不管它是合约创建还是消息调用，我们都可以定义这样的元组，包含执行后的临时状态 $\boldsymbol{\sigma}_P$，剩余的 gas g'，子状态 A 和状态代码 z：

$$(\boldsymbol{\sigma}_P, g', A, z) \equiv \begin{cases} \Lambda_4(\boldsymbol{\sigma}_0, S(T), T_{\mathrm{o}}, g, \\ \quad T_{\mathrm{p}}, T_{\mathrm{v}}, T_{\mathbf{i}}, 0, \top) & \text{if} \quad T_{\mathrm{t}} = \varnothing \\ \Theta_4(\boldsymbol{\sigma}_0, S(T), T_{\mathrm{o}}, T_{\mathrm{t}}, T_{\mathrm{t}}, \\ \quad g, T_{\mathrm{p}}, T_{\mathrm{v}}, T_{\mathbf{d}}, 0, \top) & \text{otherwise} \end{cases} \tag{62}$$

其中 g 是扣除掉合约存在所需要的基本 gas 数量之后的剩余 gas 数量：

$$g \equiv T_{\mathrm{g}} - g_0 \tag{63}$$

T_{o} 是交易的原始发起人，当一个消息调用或合约创建不是由交易所触发，而是来自于 EVM 代码的运行所触发时，这个原始发起人会与发送者不同（即在这种情况下，T_{o} 会与 $S(T)$ 不同，校订注）。

注意，我们使用 Θ_4 和 Λ_4 来表示只取用那组函数中的前 4 个函数的值，这最终表现为消息调用的输出（一个字节数组），且这个最终结果不会在交易计算的上下文中使用。

在消息调用或合约创建被处理之后，在此过程中自毁的账户的返还余额计数也已经增加完成了。

$$A'_r \equiv A_r + \sum_{i \in A_s} R_{selfdestruct} \tag{64}$$

然后，就可以用剩余的 gas g' 加上基于返还计数的补贴来按照原始比率确定返还给发送者的 gas 数量 g^* 了。

$$g^* \equiv g' + \min\left\{\left\lfloor\frac{T_{\mathrm{g}} - g'}{2}\right\rfloor, A'_{\mathrm{r}}\right\} \tag{65}$$

总共可返还的数量，是合法的剩余 gas g' 加上 A_{r} 与总消耗量 $T_{\mathrm{g}} - g'$ 的一半进行四舍五入后的结果中较小的那个值。所以 g^* 就是交易执行后的总剩余 gas。

交易花费的 gas 所对应的以太币会支付给矿工，它们的地址是由当前区块 B 的 beneficiary 所指定的。所以我们基于临时状态 $\boldsymbol{\sigma}_P$ 来定义预备最终状态 $\boldsymbol{\sigma}^*$：

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}_P \quad \text{except} \tag{66}$$

$$\boldsymbol{\sigma}^*[S(T)]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_P[S(T)]_{\mathrm{b}} + g^* T_{\mathrm{p}} \tag{67}$$

$$\boldsymbol{\sigma}^*[m]_{\mathrm{b}} \equiv \boldsymbol{\sigma}_P[m]_{\mathrm{b}} + (T_{\mathrm{g}} - g^*)T_{\mathrm{p}} \tag{68}$$

$$m \equiv B_{H\mathrm{c}} \tag{69}$$

在删除了所有出现在自毁列表中的账户或被接触过的空账户之后，即可达到最终状态 $\boldsymbol{\sigma}'$：

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma}^* \quad \text{except} \tag{70}$$

$$\forall i \in A_{\mathbf{s}} : \boldsymbol{\sigma}'[i] = \varnothing \tag{71}$$

$$\forall i \in A_{\mathbf{t}} : \boldsymbol{\sigma}'[i] = \varnothing \quad \text{if} \quad \texttt{DEAD}(\boldsymbol{\sigma}^*, i) \tag{72}$$

最后，我们定义这个交易中总共使用的 gas Υ^{g}、这个交易中创建的日志 $\Upsilon^{\mathbf{l}}$ 和这个交易的状态代码 Υ^{z}：

$$\Upsilon^{\mathrm{g}}(\boldsymbol{\sigma}, T) \equiv T_{\mathrm{g}} - g^* \tag{73}$$

$$\Upsilon^{\mathbf{l}}(\boldsymbol{\sigma}, T) \equiv A_{\mathbf{l}} \tag{74}$$

$$\Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}, T) \equiv z \tag{75}$$

这些用来帮助我们定义 交易收据，并且也将用在后续的状态和 nonce 校验中。

7. 合约创建

创建一个账户需要很多固有参数：发送者（s）、原始交易人（o）、可用的 gas（g）、gas 价格（p）、endowment（v，即初始捐款）、任意长度的字节数组（即 EVM 初始化代码）$\mathbf{i}$、消息调用/合约创建的当前栈深度（e）以及对状态进行修改的许可（w）。

我们定义创建函数为函数 Λ，它将使用上述参数，和状态 $\boldsymbol{\sigma}$ 一起来计算出一个新的元组。就像第 6 节所介绍的那样，这个新的元组包含新的状态、剩余的 gas、交易子状态以及一个错误消息 $(\boldsymbol{\sigma}', g', A, \mathbf{o})$：

$$(\boldsymbol{\sigma}', g', A, z, \mathbf{o}) \equiv \Lambda(\boldsymbol{\sigma}, s, o, g, p, v, \mathbf{i}, e, w) \tag{76}$$

新账户的地址是一个哈希值的最右边 160 位，这个哈希值是由一个仅包含发送者地址和其账户 nonce 的结构进行 RLP 编码之后再进行 Keccak 哈希计算所得到的。我们可以定义由此得来的新帐户的地址 a：

$$a \equiv \mathcal{B}_{96..255}\Big(\texttt{KEC}\Big(\texttt{RLP}\big(\ (s, \boldsymbol{\sigma}[s]_{\mathrm{n}} - 1)\ \big)\Big)\Big) \tag{77}$$

其中 KEC 是 Keccak 256 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 的位数范围 $[a, b]$ 的值，$\boldsymbol{\sigma}[x]$ 则是地址 x 的状态，或者 $\varnothing$ 表示什么都不存在。注意，我们使用的是一个比发送者 nonce 要小的值（减了 1）；因为我们认定我们已经在这个调用之前对发送者的 nonce 加了 1，因此所用的值是发送者在该交易或 VM 操作开始时的 nonce。

账户的 nonce 被初始定义为 1，balance 为交易传入的值，storage 为空，codeHash 为空字符串的 Keccak 256 位哈希值；发送者的 balance 会减去转账值。于是这个变化的状态就成为 $\boldsymbol{\sigma}^*$：

$$\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma} \quad \text{except:} \tag{78}$$

$$\boldsymbol{\sigma}^*[a] = \big(1, v + v', \texttt{TRIE}(\varnothing), \texttt{KEC}(())\big) \tag{79}$$

$$\boldsymbol{\sigma}^*[s] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[s] = \varnothing \ \wedge \ v = 0 \\ \mathbf{a}^* & \text{otherwise} \end{cases} \tag{80}$$

$$\mathbf{a}^* \equiv (\boldsymbol{\sigma}[s]_{\mathrm{n}}, \boldsymbol{\sigma}[s]_{\mathrm{b}} - v, \boldsymbol{\sigma}[s]_{\mathbf{s}}, \boldsymbol{\sigma}[s]_{\mathrm{c}}) \tag{81}$$

其中 v' 是账户在交易之前就有的余额，对应于它先前就已经存在的情况：

$$v' \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}[a] = \varnothing \\ \boldsymbol{\sigma}[a]_{\mathrm{b}} & \text{otherwise} \end{cases} \tag{82}$$

最终，账户是根据执行模型（参见第 9 章）通过执行账户初始化的 EVM 代码 $\mathbf{i}$ 来初始化的。代码的执行可以产生一些内部执行状态以外的事件，包括：更改账户的 storage，创建更多的账户，以及进行更多的消息调用。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 gas g^{**}，累积的子状态 A 以及账户的代码 $\mathbf{o}$。

$$(\boldsymbol{\sigma}^{**}, g^{**}, A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}^*, g, I, \{s, a\}) \tag{83}$$

其中 I 包含了执行环境的相关参数：

$$I_{\mathrm{a}} \equiv a \tag{84}$$

$$I_{\mathrm{o}} \equiv o \tag{85}$$

$$I_{\mathrm{p}} \equiv p \tag{86}$$

$$I_{\mathbf{d}} \equiv () \tag{87}$$

$$I_{\mathrm{s}} \equiv s \tag{88}$$

$$I_{\mathrm{v}} \equiv v \tag{89}$$

$$I_{\mathbf{b}} \equiv \mathbf{i} \tag{90}$$

$$I_{\mathrm{e}} \equiv e \tag{91}$$

$$I_{\mathrm{w}} \equiv w \tag{92}$$

由于这个调用没有任何输入数据，$I_{\mathbf{d}}$ 为空的元组。I_H 则没有特别之处，由区块链所决定。

代码执行会消耗 gas，且 gas 不能低于 0，因此执行可能会在自然停止之前结束。在这个（以及其它几个）异常情况，我们称之为发生了 gas 不足（out-of-gas，OOG）异常：计算后的状态将由空集合 $\varnothing$ 表示，整个创建操作将不会影响状态，就像刚开始尝试创建时那样。

如果这个初始化代码成功地执行完，那么对应的合约创建费用也会支付。这个代码保存费用 c 与创建出来的合约代码大小成正比：

$$c \equiv G_{codedeposit} \times |\mathbf{o}| \tag{93}$$

如果没有足够的剩余 gas 来支付这个费用，也就是说如果 $g^{**} < c$，就会产生 gas 不足异常。

发生这样的异常后，剩余 gas 将变为 0。也就是说，如果合约创建是作为对交易的接受来处理的，那么它就不会影响合约创建的固有费用的支付，就是肯定会支付。然而，当 gas 不足时，交易中附带的金额并不会转移到被取消的合约地址。

如果没有出现这样的异常，那么剩余的 gas 会返还给最原始的交易发起人，对状态的改变也将永久保存。这样，我们可以指定结果状态、gas、子状态和状态代码为 $(\boldsymbol{\sigma}', g', A, z)$：

(94)

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

(95)

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

(96)

$$z \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\sigma}^{**} = \varnothing \vee g^{**} < c \\ 1 & \text{otherwise} \end{cases}$$

where

(97)

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

对于确定 $\boldsymbol{\sigma}'$ 的例外，由初始化代码的执行结果字节序列 $\mathbf{o}$ 所决定，它就是新创建账户的最终代码。

注意，这是意图达到这样的结果：要么带着初始捐款（endowment）成功创建合约；要么不会创建任何合约且不会进行转账。

7.1. 细微之处. 注意，在初始化代码执行过程中，一个新创建的地址会出现，但还没有内部的代码[4]。因此在这段时间内，任何消息调用都不会引发代码执行。如果这个初始化执行结束于一个 SELFDESTRUCT 指令，那么这个账户会在交易完成前被删除，这种情况是否合理已在讨论中。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地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）、接收者（r）、执行代码的账户（c，通常与接收者相同）、可用的 gas（g）、转账金额（v）、gas 价格（p）、函数调用的输入数据（一个任意长度的字节数组 $\mathbf{d}$）、消息调用/合约创建的当前栈深度（e）以及对状态修改的许可（w）。

除了可以获得新的状态和交易子状态以外，消息调用还有一个额外的元素——由字节数组 $\mathbf{o}$ 表示的输出数据。执行交易时输出数据是被忽略的，但消息调用可以由 VM 代码执行所产生，在这种情况下就将使用这些信息。

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

注意，对于 DELEGATECALL 指令，我们需要区分转账金额 v 和执行上下文中出现的 $\tilde{v}$。

我们定义 $\boldsymbol{\sigma}_1$ 为第一个交易状态，它是在原始状态的基础上加入由发送者向接收者的转账金额后的状态：

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

除非 $s = r$。

按照先前的论述（我们假设如果 $\boldsymbol{\sigma}_1[r]$ 还未定义），则这会创建一个没有代码或状态且 balance 和 nonce 都为 0 的账户。因此上一个公式可以替换为：

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

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

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

(103) $$\text{and} \quad \boldsymbol{\sigma}'_1 \equiv \boldsymbol{\sigma} \quad \text{except:}$$

(104)

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

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

账户关联的代码（由 Keccak 哈希为 $\boldsymbol{\sigma}[c]_{\mathrm{c}}$ 的代码片段所标识）会依照执行模型（参见第 9 章）来执行。就像合约创建一样，如果执行因为一个异常（也就是说：因为 gas 供给不足、堆栈溢出、无效的跳转目标或者无效的指令）而停止，gas 不会被返还给调用者，并且状态也会立即恢复到转账之前的状态（也就是 $\boldsymbol{\sigma}$）。

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

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

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

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

(109) $$I_{\mathrm{a}} \equiv r$$

(110) $$I_{\mathrm{o}} \equiv o$$

(111) $$I_{\mathrm{p}} \equiv p$$

(112) $$I_{\mathbf{d}} \equiv \mathbf{d}$$

(113) $$I_{\mathrm{s}} \equiv s$$

(114) $$I_{\mathrm{v}} \equiv \tilde{v}$$

(115) $$I_{\mathrm{e}} \equiv e$$

(116) $$I_{\mathrm{w}} \equiv w$$

(117) $$\mathbf{t} \equiv \{s, r\}$$

(118)

其中

(119) $$\Xi \equiv \begin{cases} \Xi_{\texttt{ECREC}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 1 \\ \Xi_{\texttt{SHA256}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 2 \\ \Xi_{\texttt{RIP160}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 3 \\ \Xi_{\texttt{ID}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 4 \\ \Xi_{\texttt{EXPMOD}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 5 \\ \Xi_{\texttt{BN\_ADD}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 6 \\ \Xi_{\texttt{BN\_MUL}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 7 \\ \Xi_{\texttt{SNARKV}}(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{if} \quad r = 8 \\ \Xi(\boldsymbol{\sigma}_1, g, I, \mathbf{t}) & \text{otherwise} \end{cases}$$

(120) $$\text{Let } \texttt{KEC}(I_{\mathbf{b}}) = \boldsymbol{\sigma}[c]_{\mathrm{c}}$$

我们假设客户端会保存键值对数据 $(\texttt{KEC}(I_{\mathbf{b}}), I_{\mathbf{b}})$，来在某些情况下使获得 $I_{\mathbf{b}}$ 成为可能。

[4]在初始化代码的执行过程中，当前地址上的 EXTCODESIZE 也就是账户代码的长度应该返回 0。而 CODESIZE 则应该返回初始化代码的长度。（就像H.2中定义的那样。）

如我们所见，在消息调用的通用执行框架 Ξ 中有 8 个特例：这是 8 个所谓的‘预编译’合约，它们作为最初架构中的一部分，后续可能会变成原生扩展。在地址 1 到 8 上存在的这 8 个合约，分别是椭圆曲线公钥恢复函数、SHA2 256 位哈希方案、RIPEMD 160 位哈希方案、标识函数、任意精度的模幂运算、椭圆曲线加法、椭圆曲线纯量乘法和椭圆曲线配对检查。

它们的完整定义参见附录 E。

9. 执行模型

执行模型说明了如何使用一系列的字节代码指令和一个环境数据的元组去更改系统状态。这是通过一个正规的虚拟状态机来实现的，也就是以太坊虚拟机（Ethereum Virtual Machine - EVM）。它是一个准图灵机，这个“准”的限定来源于其中的运算是通过参数 *gas* 来限制的，也就是限定了可以执行的运算总量。

9.1. 基础. EVM 是一个简单的基于栈的架构。其中字（Word）的大小（也就是栈中数据项的大小）是 256 位。这是为了便于执行 Keccak-256 位哈希和椭圆曲线计算。其内存（memory）模型是简单地基于字寻址（word-addressed）的字节数组。栈的最大深度为 1024。EVM 也有一个独立的存储（storage）模型；它类似于内存的概念，但并不是一个字节数组，而是一个基于字寻址（word-addressable）的字数组（word array）。与不稳定的内存不同，存储是相对稳定的且作为系统状态的一部分被维护。所有内存和存储中的数据都会初始化为 0。

EVM 不是标准的冯诺依曼结构。程序代码被保存在一个独立的、仅能通过特定的指令进行交互的虚拟 ROM（即只读存储器，校订注）中，而不是保存在一般的可访问内存或存储中。

EVM 存在异常执行的情况，包括堆栈溢出和非法指令。在发生像 out-of-gas 这样的异常时，EVM 并不会使状态保持原样，而是会立即停止执行，并告知执行代理（交易的处理程序或递归的执行环境），由代理来独立处理这些异常。

9.2. 费用概述. 在三个不同的情况下会收取执行费用（以 gas 来结算），这三种情况都是执行操作的先决条件。第一种情况，也是最普遍的情况就是计算操作费用（详见附录 G）。第二种情况，执行一个低级别的消息调用或者合约创建可能需要扣除 gas，这也就是执行 CREATE，CALL 和 CALLCODE 的费用的一部分。最后，内存使用的增加也会消耗一定的 gas。

对于一个账户的执行，内存的总费用和其所有内存索引（无论是读还是写）的范围成正比；这个内存范围是 32 字节的最小倍数。这是实时（just-in-time）结算的；也就是说，任何对超出先前已索引的内存区域的访问，都会实时地结算为额外的内存使用费。由于这个费用，内存地址大概不会超过 32 位的界限，但 EVM 的实现必须能够管理这种可能性（就是说，虽然如果内存地址超过 32 位，即大于 2^{32}，会产生很高的内存使用费，但 EVM 的实现还是应该支持这种可能性；校订注）。

存储费用则有一个细微差别的行为——激励存储的最小化使用（这直接反映在所有节点中更大的状态数据库里）。清除一个存储中的记录项的执行费用不仅被免除了，而且还会返还；实际上，这种费用返还（退款）是预先发生的，因为首次使用存储位置的费用比正常使用高出很多。

EVM gas 消耗的严格定义参见附录 H。

9.3. 执行环境. 作为系统状态 $\boldsymbol{\sigma}$ 和计算中剩余的 gas g 的补充，还有一些执行代理必须提供的有关执行环境的重要信息，它们包含在元组 I 中：

- I_{a}，拥有正在执行的代码的账户地址。
- I_{o}，触发这次执行的初始交易的发送者地址。
- I_{p}，触发这次执行的初始交易的 gas 价格。
- $I_{\mathbf{d}}$，这次执行的输入数据字节数组；如果执行代理是一个交易，这就是交易数据。
- I_{s}，触发这次执行的账户地址；如果执行代理是一个交易，则为交易发送者地址。
- I_{v}，作为这次执行的一部分传到当前账户的转账金额，以 Wei 为单位；如果执行代理是一个交易，这就是交易的转账金额。
- $I_{\mathbf{b}}$，所要执行的机器代码字节数组。
- $I_{\mathbf{H}}$，当前区块的区块头。
- I_{e}，当前消息调用或合约创建的深度（也就是当前已经被执行的 CALL 或 CREATE 的数量）。
- I_{w}，修改状态的许可。

执行模型定义了函数 Ξ，它可以用来计算结果状态 $\boldsymbol{\sigma}'$，剩余的 gas g'，累积的子状态 A 和结果输出 $\mathbf{o}$。根据当前的上下文，我们可以把它定义为：

$$(\boldsymbol{\sigma}', g', A, \mathbf{o}) \equiv \Xi(\boldsymbol{\sigma}, g, I) \tag{121}$$

这里，我们应该还记得这个累积的子状态 A 是由自毁集合 $\mathbf{s}$、日志集 $\mathbf{l}$、接触过的账户 $\mathbf{t}$ 和返还金额 r 所组成的元组：

$$A \equiv (\mathbf{s}, \mathbf{l}, \mathbf{t}, r) \tag{122}$$

9.4. 执行概述. 我们现在必须来定义 Ξ 函数了。在大多数可行的实现方案中，都应该对完整的系统状态 $\boldsymbol{\sigma}$ 和机器状态 $\boldsymbol{\mu}$ 一起进行迭代建模。我们定义一个递归函数 X，它使用了一个迭代函数 O（定义状态机中单次循环的结果），与一个用来确定当前机器状态是否是一个异常停止状态的函数 Z 和一个当且仅当当前机器状态是正常停止状态时用来指定指令的输出数据的函数 H。

由 () 表示的空序列并不等于由 $\varnothing$ 表示的空集合，这对于解释 H 的输出非常重要；当 H 的输出是 $\varnothing$ 时需要继续执行，而当其为空序列时则应该停止执行。

$$\Xi(\boldsymbol{\sigma}, g, I, T) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}'_{\mathrm{g}}, A, \mathbf{o}) \tag{123}$$

$$(\boldsymbol{\sigma}', \boldsymbol{\mu}', A, ..., \mathbf{o}) \equiv X\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A^0, I)\big) \tag{124}$$

$$\boldsymbol{\mu}_{\mathrm{g}} \equiv g \tag{125}$$

$$\boldsymbol{\mu}_{\mathrm{pc}} \equiv 0 \tag{126}$$

$$\boldsymbol{\mu}_{\mathbf{m}} \equiv (0, 0, ...) \tag{127}$$

$$\boldsymbol{\mu}_{\mathrm{i}} \equiv 0 \tag{128}$$

$$\boldsymbol{\mu}_{\mathbf{s}} \equiv () \tag{129}$$

$$\boldsymbol{\mu}_{\mathbf{o}} \equiv () \tag{130}$$

(131)

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

其中

$$\mathbf{o} \equiv H(\boldsymbol{\mu}, I) \tag{132}$$

$$(a, b, c, d) \cdot e \equiv (a, b, c, d, e) \tag{133}$$

$$\boldsymbol{\mu}' \equiv \boldsymbol{\mu} \text{ except:} \tag{134}$$

$$\boldsymbol{\mu}'_{\mathrm{g}} \equiv \boldsymbol{\mu}_{\mathrm{g}} - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{135}$$

注意，在推导 Ξ 时，我们去掉了第 4 个元素 I' 并从机器状态的结果 $\boldsymbol{\mu}'$ 中提取了剩余的 gas $\boldsymbol{\mu}'_{\mathrm{g}}$。

X 是被循环调用的（这里是递归，但是在实现方案中通常是去执行一个简单的迭代循环）直到 Z 变为 true，表示当前状态有异常，必须停止执行，并且所有的改动都会被舍

弃；或者直到 H 变为一个序列（不是空集合），表示机器达到了正常控制的停止状态。

9.4.1. 机器状态. 机器状态 $\boldsymbol{\mu}$ 是由一个元组 $(g, pc, \mathbf{m}, i, \mathbf{s})$ 所定义的，其中包括可用的 gas g，程序计数器 $pc \in \mathbb{N}_{256}$，内存的内容 $\mathbf{m}$，内存中激活的字数（从 0 开始的连续计数）i，以及栈的内容 $\mathbf{s}$。内存的内容 $\boldsymbol{\mu}_{\mathbf{m}}$ 是大小为 2^{256} 的全 0 序列。

为了提高可读性，应该使用大写字母简写（例如 ADD）的指令助记符来推演数字等式。完整的指令列表和它们的定义参见附录 H。

为了定义 Z、H 和 O，我们将当前要执行的操作定义为 w：

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathrm{pc}}] & \text{if} \quad \boldsymbol{\mu}_{\mathrm{pc}} < \|I_{\mathbf{b}}\| \\ \mathrm{STOP} & \text{otherwise} \end{cases} \tag{136}$$

我们也假定从栈中移除和添加的条目的固定数量为 δ 和 α，它们都可以作为具体指令的下标；并且指令费用函数 C 可以算出给定指令的全部费用（以 gas 为单位）。

9.4.2. 异常停止. 异常停止函数 Z 定义如下:

$$\begin{aligned} Z(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv \quad & \boldsymbol{\mu}_g < C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \quad \vee \\ & \delta_w = \varnothing \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| < \delta_w \quad \vee \\ & (w = \mathrm{JUMP} \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \mathrm{JUMPI} \quad \wedge \quad \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \quad \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[0] \notin D(I_{\mathbf{b}})) \quad \vee \\ & (w = \mathrm{RETURNDATACOPY} \wedge \\ & \quad \boldsymbol{\mu}_{\mathbf{s}}[1] + \boldsymbol{\mu}_{\mathbf{s}}[2] > \|\boldsymbol{\mu}_{\mathbf{o}}\|) \quad \vee \\ & \|\boldsymbol{\mu}_{\mathbf{s}}\| - \delta_w + \alpha_w > 1024 \vee (\neg I_{\mathrm{w}} \wedge W(w, \boldsymbol{\mu})) \end{aligned} \tag{137}$$

其中

$$\begin{aligned} W(w, \boldsymbol{\mu}) \equiv \quad & w \in \{\mathrm{CREATE}, \mathrm{SSTORE}, \\ & \mathrm{SELFDESTRUCT}\} \vee \\ & \mathrm{LOG0} \leq w \wedge w \leq \mathrm{LOG4} \quad \vee \\ & w \in \{\mathrm{CALL}, \mathrm{CALLCODE}\} \wedge \boldsymbol{\mu}_{\mathbf{s}}[2] \neq 0 \end{aligned} \tag{138}$$

以下情况都会使执行进入异常停止状态：gas 不足、无效的指令（指令的下标 δ 未定义）、栈中的条目不足、指令 JUMP/JUMPI 的目标无效、新栈的大小超过 1024 或者在一个静态调用中去尝试修改状态。聪明的读者会意识到，这表明没有任何一个指令可以通过它的执行直接触发异常停止。

9.4.3. 跳转地址验证. 我们先前使用 D 函数来判断正在运行的代码所给定的有效跳转地址集合。我们以 JUMPDEST 指令所使用的位置来定义它。

所有这些跳转位置都必须在有效的指令块内，而不可以是 PUSH 操作的数据中的位置；并且这些跳转位置还必须在明确定义的代码中（而不是使用隐式定义的 STOP 指令所跟踪到的位置）。

这样：

$$D(\mathbf{c}) \equiv D_J(\mathbf{c}, 0) \tag{139}$$

其中：

$$D_J(\mathbf{c}, i) \equiv \begin{cases} \{\} & \text{if} \quad i \geqslant |\mathbf{c}| \\ \{i\} \cup D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \\ \quad \text{if} \quad \mathbf{c}[i] = \mathrm{JUMPDEST} & \\ D_J(\mathbf{c}, N(i, \mathbf{c}[i])) & \text{otherwise} \end{cases} \tag{140}$$

这里的 N 是下一个有效指令在代码中的位置，且忽略了 PUSH 指令的数据（如果有的话）：

$$N(i, w) \equiv \begin{cases} i + w - \mathrm{PUSH1} + 2 & \\ \quad \text{if} \quad w \in [\mathrm{PUSH1}, \mathrm{PUSH32}] & \\ i + 1 & \text{otherwise} \end{cases} \tag{141}$$

9.4.4. 正常停止. 正常停止函数 H 定义如下：

$$H(\boldsymbol{\mu}, I) \equiv \begin{cases} H_{\mathrm{RETURN}}(\boldsymbol{\mu}) \text{ if} & w \in \{\mathrm{RETURN}, \mathrm{REVERT}\} \\ () & \text{if} \quad w \in \{\mathrm{STOP}, \mathrm{SELFDESTRUCT}\} \\ \varnothing & \text{otherwise} \end{cases} \tag{142}$$

会返回数据的停止操作 RETURN 和 REVERT 有一个特殊的函数 H_{RETURN}。同时也请注意在这里讨论过的，关于空序列和空集合的差别。

9.5. 执行循环. 栈中的条目是在序列的最左侧、以低位索引的方式添加或移除的，所有其它条目都不会变动：

$$O\big((\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I)\big) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \tag{143}$$

$$\Delta \equiv \alpha_w - \delta_w \tag{144}$$

$$\|\boldsymbol{\mu}'_{\mathbf{s}}\| \equiv \|\boldsymbol{\mu}_{\mathbf{s}}\| + \Delta \tag{145}$$

$$\forall x \in [\alpha_w, \|\boldsymbol{\mu}'_{\mathbf{s}}\|) : \boldsymbol{\mu}'_{\mathbf{s}}[x] \equiv \boldsymbol{\mu}_{\mathbf{s}}[x - \Delta] \tag{146}$$

其中，gas 会根据具体指令的 gas 消耗相应扣除，且除了下面的三个特例以外，对于大多数指令而言，每次循环，程序计数器都会自动增加；这里我们假定一个函数 J，其下标是下边两个指令之一，可以算出相应的值：

$$\boldsymbol{\mu}'_g \equiv \boldsymbol{\mu}_g - C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \tag{147}$$

$$\boldsymbol{\mu}'_{\mathrm{pc}} \equiv \begin{cases} J_{\mathrm{JUMP}}(\boldsymbol{\mu}) & \text{if} \quad w = \mathrm{JUMP} \\ J_{\mathrm{JUMPI}}(\boldsymbol{\mu}) & \text{if} \quad w = \mathrm{JUMPI} \\ N(\boldsymbol{\mu}_{\mathrm{pc}}, w) & \text{otherwise} \end{cases} \tag{148}$$

在一般情况下，我们假定内存、自毁集合和系统状态不会改变：

$$\boldsymbol{\mu}'_{\mathbf{m}} \equiv \boldsymbol{\mu}_{\mathbf{m}} \tag{149}$$

$$\boldsymbol{\mu}'_{\mathrm{i}} \equiv \boldsymbol{\mu}_{\mathrm{i}} \tag{150}$$

$$A' \equiv A \tag{151}$$

$$\boldsymbol{\sigma}' \equiv \boldsymbol{\sigma} \tag{152}$$

然而，具体的指令通常会改变这些值中的一个或几个。附录 H 中列出了指令会更改的元素以及入栈数 α、出栈数 δ 和对 gas 需求的正式描述。

10. 区块树到区块链

权威的区块链是在整个的区块树中从根节点到叶子节点的路径。为了形成路径共识，概念上我们选择具有最大的计算量的，或者说是最重的路径来识别它。帮助我们识别最重路径的一个明显事实就是叶子节点的区块数量，等于在路径中除了没有挖矿的创世区块以外的区块数量。路径越长，到达叶子节点的总体挖矿工作量就越大。这和已有的比特币及其衍生协议类似。

因为区块头中已经包含了难度值，所以仅通过区块头就能验证已经完成的计算量。区块链中的任意一个区块都对总计算量或一个链的总难度有所贡献。

由此我们递归地定义区块 B 的总难度：

$$B_{\mathrm{t}} \equiv B'_{\mathrm{t}} + B_{\mathrm{d}} \tag{153}$$

$$B' \equiv P(B_H) \tag{154}$$

对于给定区块 B，B_{t} 是它的总难度，B' 是其父区块，B_{d} 则是它自己的难度。

11. 区块定稿

区块定稿的过程包含 4 个步骤：

(1) 验证（或在挖矿中，确定）ommer；
(2) 验证（或在挖矿中，确定）交易；
(3) 发放奖励；
(4) 校验（或在挖矿中，计算有效的）状态和区块 nonce。

11.1. **Ommer** 验证. 验证ommer 头也就是验证每个 ommer 头即是有效的区块头，又满足一个关联条件：它必须是当前区块 N 代以内的 ommer，$N \leq 6$。一个区块最多有两个 ommer 头。这样：

$$\|B_{\mathbf{U}}\| \leqslant 2 \bigwedge_{\mathbf{U} \in B_{\mathbf{U}}} V(\mathbf{U}) \wedge k(\mathbf{U}, P(\mathbf{B_H})_{\mathbf{H}}, 6) \tag{155}$$

其中 k 表示“是亲属”（“is-kin”）：

$$k(U,H,n) \equiv \begin{cases} false & \text{if} \quad n = 0 \\ s(U,H) & \\ \quad \vee\, k(U, P(H)_H, n-1) & \text{otherwise} \end{cases} \tag{156}$$

s 表示“是兄妹”（“is-sibling”）：

$$s(U,H) \equiv (P(H) = P(U) \wedge H \neq U \wedge U \notin B(H)_{\mathbf{U}}) \tag{157}$$

其中，$B(H)$ 和 $P(H)$ 分别表示区块头 H 所对应的区块和其父区块。

11.2. 交易验证. 给定的 **gasUsed** 必须与列入的交易如实地相符：区块中的 gas 总使用量 B_{Hg}，必须与区块中最后一个交易执行后的累积 gas 使用量相等：

$$B_{Hg} = \ell(\mathbf{R})_{\mathrm{u}} \tag{158}$$

11.3. 奖励发放. 区块奖励的发放，包含了对当前区块和每个 ommer 区块的 beneficiary 地址中账户 balance 的增加。我们给当前区块的 beneficiary 账户增加 R_{block}；对于每个 ommer 区块，我们额外提高 $\frac{1}{32}$ 的区块奖励，并且 ommer 的 beneficiary 会根据区块号获得奖励。这样，我们定义函数 Ω：

$$\Omega(B, \boldsymbol{\sigma}) \equiv \boldsymbol{\sigma}' : \boldsymbol{\sigma}' = \boldsymbol{\sigma} \quad \text{except:} \tag{159}$$

$$\boldsymbol{\sigma}'[\mathbf{B}_{Hc}]_{\mathrm{b}} = \boldsymbol{\sigma}[\mathbf{B}_{Hc}]_{\mathrm{b}} + \left(1 + \frac{\|\mathbf{B}_{\mathbf{U}}\|}{32}\right) R_{\mathrm{block}} \tag{160}$$

$$\forall_{\mathbf{U} \in \mathbf{B}_{\mathbf{U}}} : \quad \boldsymbol{\sigma}'[\mathbf{U}_{\mathrm{c}}] = \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[\mathbf{U}_{\mathrm{c}}] = \varnothing \wedge R = 0 \\ \mathbf{a}' & \text{otherwise} \end{cases} \tag{161}$$

$$\mathbf{a}' \equiv (\boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{n}}, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{b}} + R, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathbf{s}}, \boldsymbol{\sigma}[U_{\mathrm{c}}]_{\mathrm{c}}) \tag{162}$$

$$R \equiv \left(1 + \frac{1}{8}(U_{\mathrm{i}} - B_{H\mathrm{i}})\right) R_{\mathrm{block}} \tag{163}$$

如果 ommer 和当前区块的 beneficiary 地址有重合（即：两个 ommer 有相同的 beneficiary 地址或者某一个 ommer 的 beneficiary 地址与当前区块相同），额外奖励将会累积发放。

我们定义区块奖励为 3 以太币:

$$\text{Let} \quad R_{\mathrm{block}} = 3 \times 10^{18} \tag{164}$$

11.4. 状态和 **nonce** 验证. 现在我们可以定义函数 Γ 了，它区块 B 映射到其初始状态：

$$\Gamma(B) \equiv \begin{cases} \boldsymbol{\sigma}_0 kern10pc \text{ if} \quad P(B_H) = \varnothing \\ \boldsymbol{\sigma}_{\mathrm{i}} : \mathtt{TRIE}(L_S(\boldsymbol{\sigma}_{\mathrm{i}})) = P(B_H)_{H_{\mathrm{r}}} \quad \text{otherwise} \end{cases} \tag{165}$$

这里，$\mathtt{TRIE}(L_S(\boldsymbol{\sigma}_i))$ 指由状态 $\boldsymbol{\sigma}_{\mathrm{i}}$ 所构成的 trie 的根节点哈希；我们假定客户端实现会把这个数据保存在状态数据库中，因为这个 trie 是个不可变的数据结构，所以这是简单有效的。

最后我们定义区块变换函数 Φ，它将一个不完整的区块 B 映射到一个完整区块 B'：

$$\Phi(B) \equiv B' : \quad B' = B^* \quad \text{except:} \tag{166}$$

$$B'_{\mathrm{n}} = n : \quad x \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \tag{167}$$

$$B'_{\mathrm{m}} = m \quad \text{with } (x, m) = \mathtt{PoW}(B^*_{\not\mathbf{H}}, n, \mathbf{d}) \tag{168}$$

$$B^* \equiv B \quad \text{except:} \quad B^*_{\mathrm{r}} = r(\Pi(\Gamma(B), B)) \tag{169}$$

其中 $\mathbf{d}$ 是一个在附录 J 中定义的数据集。

就像本文先前定义的那样，Π 是状态转换函数，它概念上是由区块定稿函数 Ω 和交易计算函数 Υ 来定义的。

像先前详细介绍过的那样，$\mathbf{R}[n]_{\mathrm{z}}$、$\mathbf{R}[n]_{\mathbf{l}}$ 和 $\mathbf{R}[n]_{\mathrm{u}}$ 是第 n 个交易的状态代码、日志和累积 gas 消耗（元组中的第 4 个元素 $\mathbf{R}[n]_{\mathrm{b}}$ 已经在日志的概念中被定义了）。我们也定义第 n 个状态 $\boldsymbol{\sigma}[n]$，我们可以简单地把它定义为基于先前的交易结果状态（或者对于区块中的第一个交易来说，就是区块的初始状态）来执行当前交易所产生的结果状态：

$$\boldsymbol{\sigma}[n] = \begin{cases} \Gamma(B) & \text{if} \quad n < 0 \\ \Upsilon(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \text{otherwise} \end{cases} \tag{170}$$

在 $B_{\mathbf{R}}[n]_{\mathrm{u}}$ 中，我们使用类似的方法来定义每个交易中加算了前一个交易的 gas 用量（当它是区块中的第一个交易时，则为 0）后的累积用量：

$$\mathbf{R}[n]_{\mathrm{u}} = \begin{cases} 0 & \text{if} \quad n < 0 \\ \Upsilon^g(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) & \\ \quad +\mathbf{R}[n-1]_{\mathrm{u}} & \text{otherwise} \end{cases} \tag{171}$$

对于 $\mathbf{R}[n]_{\mathbf{l}}$，我们使用先前在交易执行函数中定义的 $\Upsilon^{\mathbf{l}}$。

$$\mathbf{R}[n]_{\mathbf{l}} = \Upsilon^{\mathbf{l}}(\mathbf{R}[n-1]_{\boldsymbol{\sigma}}, B_{\mathbf{T}}[n]) \tag{172}$$

我们用类似的方式定义 $\mathbf{R}[n]_{\mathrm{z}}$。

$$\mathbf{R}[n]_{\mathrm{z}} = \Upsilon^{\mathrm{z}}(\boldsymbol{\sigma}[n-1], B_{\mathbf{T}}[n]) \tag{173}$$

最后，我们定义 Π 为将给定的区块奖励 Ω 发放到最后一个交易的结果状态 $\ell(\boldsymbol{\sigma})$ 之上的新状态：

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \ell(\boldsymbol{\sigma})) \tag{174}$$

至此，完整的区块转换机制就定义好了。除了工作量证明（Proof-of-Work）函数 $\mathtt{PoW}$ 以外。

11.5. 挖矿工作量证明. 挖矿工作量证明（PoW）通过一个密码学安全的 nonce 来证明为了获得一些象征性的值 n，已经付出了一定量的计算。我们使用它，并通过特定意义的难度（且可以扩展为总体难度）以及对难度的可信度认知来确保区块链的安全。然而，因为挖出新的区块会得到其附带的奖励，所以工作量证明不仅是在未来使区块链的权威的获得保障的安全信任方法，同时也是一个利益分配机制。

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。对特别定制的硬件的需求或由这种硬件所提供的回报，应该被减到最小。这使得分配模型尽可能开放，从而使世界各地的人们都可以以大致相同的比例，通过电力消耗获得以太币。

第二，应该不允许获得超线性的收益，尤其是在有一个很高的初始障碍条件下。否则，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，并允许他们获得超线性的回报（按他们的意愿改变收益分布），这也就会弱化网络的安全性。

比特币世界中一个灾难是 ASIC。有一些计算硬件仅仅是为了做一个简单的任务而存在（Smith [1997]）。在比特币中，这个任务就是 SHA256 哈希函数（Courtois et al.

[2014])。当 ASIC 们为了工作量证明函数而存在时，所有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（也就是难以在专用硬件上执行，或者在专用硬件执行时并不划算）就可以作为谚语中的银弹。

ASIC 抗性的设计有两个方向：第一是去让它成为内存困难（memory-hard）的问题所产生结果，即，设计一个需要大量的内存和带宽，来使这些内存不能被用于并行地计算 nonce。第二个方向是让计算变得更有通用目的（general-purpose）；一个为通用目的“定制的硬件”的意思，就是使得类似于普通的桌面计算机这样的硬件可以适合这种计算。在以太坊 1.0 中，我们选择了第一个方向。

更正式地，工作量证明函数可以用 PoW 表示：

(175)
$$m = H_{\mathrm{m}} \quad \wedge \quad n \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \quad \text{with} \quad (m, n) = \mathtt{PoW}(H_{\not n}, H_{\mathrm{n}}, \mathbf{d})$$

其中 $H_{\not n}$ 是新区块的头，但不包含 nonce 和 mixHash；H_{n} 是区块头的 nonce；$\mathbf{d}$ 是一个计算 mixHash 所需要的大数据集合；H_{d} 是新区块的难度值（也就是第 10 章中的区块难度）。PoW 是工作量证明函数，可以得到一个数组，其中第一个元素是 mixHash，第二个元素是密码学依赖于 H 和 $\mathbf{d}$ 的伪随机数。这个基础算法称为 Ethash，将在下文介绍。

11.5.1. *Ethash.* Ethash 是以太坊 1.0 的 PoW 算法。它是 Dagger-Hashimoto 的最新版本，由 Buterin [2013b] 和 Dryja [2014] 提出。因为原始算法中的很多特性都在 2015 年 2 月到 5 月 4 日的研发（Jentzsch [2015]）中被修改了，所以再这么称呼它其实已经不太合适。这个算法的大体路线如下：

我们通过扫描区块头直到某点，来为每个区块计算得到一个种子。根据种子我们可以得到一个初始大小为 $J_{cacheinit}$ 字节的伪随机 cache。轻客户端保存这个 cache。根据 cache，我们可以生成一个初始大小为 $J_{datasetinit}$ 字节的数据集，数据集中的每个条目仅依赖于 cache 中的一小部分条目。全节点客户端和矿工保存整个数据集。数据集会随时间线性增长。

挖矿则是在数据集中选取随机的部分并将他们一起哈希。可以根据 cache 仅生成验证所需的部分，这样就可以使用少量内存完整验证，所以对于验证来讲，仅需要保存 cache 即可。而大数据集则每 J_{epoch} 个区块更新一次，所以大多数矿工的工作都只是读取这个数据集，而不是改变它。以上提及的参数和算法的详细解释参见附录 J。

12. 实践合约

有一些特别有用的合约模式；我们会讨论其中两个，分别是数据供给（data feeds）和随机数（random numbers）。

12.1. 数据供给. 一个数据供给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，这是二级合约作者（使用数据供给的合约）的任务，他们需要决定对于单次数据供给服务给予多少信任。

通常的模式会包含一个以太坊内的合约，当给定一个消息调用时，可以回应一些由外部服务提供的信息。一个例子可以是纽约当地的温度。这将会作为一个合约来实现，并返回保存在存储中的一些值。当然，存储中的这些值比如温度需要正确地维护，然后这个模式的第二部分就是用一个外部的服务器来运行以太坊节点，找到一个新区块，创建一个合法的交易发送到合约，然后更新存储中的值。合约代码应该仅接受带有这个服务器的标识的数据更新。

12.2. 随机数. 在这样一个严格确定性的系统中提供随机的数字，显然是一个不可能实现的任务。然而我们可以利用在交易时还不可知的数据来生成伪随机数。比如区块的哈希值、区块的时间戳、区块的 beneficiary 地址。为了使恶意矿工难以控制这些数值，建议使用BLOCKHASH 操作获得最近 256 个区块的哈希值作为伪随机数。如果要获得多个这样的数值，可以去加上一些固定的数值并对结果做哈希。

13. 未来方向

未来，状态数据库将不再被强制要求维护所有之前的 trie 结构。只需要树中节点的一定时间内的数据，并最终丢弃那些不够新的且不是检查点的节点数据。检查点，或数据库中的一个节点集合可以允许对特定的区块状态树的遍历，它们可以用来为在区块链中取回任意状态数据所需的最大计算步骤提供一个上限。

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。某一个时点（也许每 10000 个区块）的区块树结构的压缩存档可以在节点网络中维护，它们可以高效地重塑创世区块。可以通过下载这样的单个归档文件加上一些强制数量的区块来减少总体下载量。

最终，或许可以实施区块链压缩：在一定数量的区块中没有进行任何发送/接收交易的状态树节点可以被丢弃，以此来降低以太币的泄露（Ether-leakage）并减小状态数据库的增长。

13.1. 可扩展性. 可扩展性仍然是一个永恒的顾虑。对于一个概括性的状态转换函数，切分和并行化交易将难以使用分而治之的策略。仍未解决的是，系统的动态价值范围大体上是固定，且由于平均交易价值的增加，那些低价值的交易变得可以忽略，从经济角度变得没有必要再包含进主账本中。然而，已经有一些潜在的策略可以用来开发更具可扩展性的协议。

一些形式的层次结构或许可以实现并行化地交易组合和区块构建，比如通过合并小的轻量的链到主区块、或者通过增量式地组合或粘连（通过工作量证明）小的交易集合来构建主区块。并行化也可以通过在一组有优先顺序的并行区块链中，将重复或无效的交易从合并的区块中剔除来实现。

最后，如果可验证的计算能达到足够通用且有效，那么这也许可以提供一种允许将工作量证明作为最终状态的验证的方法。

14. 结论

我们已经介绍、讨论并正式定义了以太坊的协议。通过这个协议，读者可以在以太坊网络上实现一个节点并加入大家，成为一个去中心化的安全的社会化操作系统中的一员。合约可以被创作出来，在算法上指定并自主化地执行交互规则。

15. 鸣谢

非常感谢 Aeron Buchanan 编写 *Homestead* 修订版，Christoph Jentzsch 编写 Ethash 算法，以及 Yoichi Hirai 做了 EIP-150 的大多数修改。以太坊开发组织和社区中的很多其他成员也为这份文档提供了重要的维护、有价值的矫正和建议，包括 Gustav Simonsson, Paweł Bylica, Jutta Steiner, Nick Savers, Viktor Trón, Marko Simovic, Giacomo Tazzari，当然，还有 Vitalik Buterin。

16. 文档获取

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 文件链接为 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 文件链接为 https://github.com/yuange1024/

ethereum_yellowpaper/blob/master/ethereum_yellow_paper_cn.pdf 。

References

Pierre Arnaud, Mathieu Schroeter, and Sam Le Barbare. Electrum, 2017. URL https://www.npmjs.com/package/electrum.

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

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

Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. KECCAK, 2017. URL https://keccak.team/keccak.html.

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

Vitalik Buterin. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform, 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://www.hashcash.org/papers/dagger.html.

Vitalik Buterin. EIP-2: Homestead hard-fork changes, 2015. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-2.md.

Vitalik Buterin. EIP-100: Change difficulty adjustment to target mean block time including uncles, April 2016. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-100.md.

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

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

Thaddeus Dryja. Hashimoto: I/O bound proof of work, 2014. URL http://diyhpl.us/~bryan/papers2/bitcoin/meh/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. URL http://www.wisdom.weizmann.ac.il/~naor/PAPERS/pvp.pdf.

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

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

Christoph Jentzsch. Commit date for ethash, 2015. URL https://github.com/ethereum/yellowpaper/commit/77a8cf2428ce245bf6e2c39c5e652ba58a278666#commitcomment-24644869.

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

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

Mark Miller. The Future of Law. In *paper delivered at the Extro 3 Conference (August 9)*, 1997. URL https://drive.google.com/file/d/0BwOVXJKBgYPMS0J2VGIyWWlocms/edit?usp=sharing.

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

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

Afri Schoedon and Vitalik Buterin. EIP-649: Metropolis difficulty bomb delay and block reward reduction, June 2017. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-649.md.

Michael John Sebastian Smith. *Application-Specific Integrated Circuits*. Addison-Wesley, 1997. ISBN 0201500221.

Yonatan Sompolinsky and Aviv Zohar. Accelerating bitcoin's transaction processing. fast money grows on trees, not chains, 2013. URL https://eprint.iacr.org/2013/881.

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

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

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

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

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 $\mathbb{T}$:

$$\mathbb{T} \equiv \mathbb{L} \cup \mathbb{B} \tag{176}$$

$$\mathbb{L} \equiv \{\mathbf{t} : \mathbf{t} = (\mathbf{t}[0], \mathbf{t}[1], ...) \wedge \forall_{n<\|\mathbf{t}\|}\ \mathbf{t}[n] \in \mathbb{T}\} \tag{177}$$

$$\mathbb{B} \equiv \{\mathbf{b} : \mathbf{b} = (\mathbf{b}[0], \mathbf{b}[1], ...) \wedge \forall_{n<\|\mathbf{b}\|}\ \mathbf{b}[n] \in \mathbb{O}\} \tag{178}$$

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:

$$\texttt{RLP}(\mathbf{x}) \equiv \begin{cases} R_{\mathbb{b}}(\mathbf{x}) & \text{if} \quad \mathbf{x} \in \mathbb{B} \\ R_{\mathbb{l}}(\mathbf{x}) & \text{otherwise} \end{cases} \tag{179}$$

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}:

$$R_{\mathrm{b}}(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if} \quad \|\mathbf{x}\| = 1 \wedge \mathbf{x}[0] < 128 \\ (128 + \|\mathbf{x}\|) \cdot \mathbf{x} & \text{else if} \quad \|\mathbf{x}\| < 56 \\ \left(183 + \left\|\mathtt{BE}(\|\mathbf{x}\|)\right\|\right) \cdot \mathtt{BE}(\|\mathbf{x}\|) \cdot \mathbf{x} & \text{otherwise} \end{cases} \tag{180}$$

$$\mathtt{BE}(x) \equiv (b_0, b_1, ...) : b_0 \neq 0 \wedge x = \sum_{n=0}^{n<\|\mathbf{b}\|} b_{\mathrm{n}} \cdot 256^{\|\mathbf{b}\|-1-n} \tag{181}$$

$$(a) \cdot (b, c) \cdot (d, e) = (a, b, c, d, e) \tag{182}$$

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

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

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

Thus we finish by formally defining R_{l}:

$$R_{\mathrm{l}}(\mathbf{x}) \equiv \begin{cases} (192 + \|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{if} \quad \|s(\mathbf{x})\| < 56 \\ \left(247 + \left\|\mathtt{BE}(\|s(\mathbf{x})\|)\right\|\right) \cdot \mathtt{BE}(\|s(\mathbf{x})\|) \cdot s(\mathbf{x}) & \text{otherwise} \end{cases} \tag{183}$$

$$s(\mathbf{x}) \equiv \mathtt{RLP}(\mathbf{x}_0) \cdot \mathtt{RLP}(\mathbf{x}_1)... \tag{184}$$

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

$$\mathtt{RLP}(i : i \in \mathbb{N}) \equiv \mathtt{RLP}(\mathtt{BE}(i)) \tag{185}$$

When interpreting RLP data, if an expected fragment is decoded as a scalar and leading zeroes are found in the byte sequence, clients are required to consider it non-canonical and treat it in the same manner as otherwise invalid RLP data, dismissing it completely.

There is no specific canonical encoding format for signed or floating-point values.

Appendix C. Hex-Prefix Encoding

Hex-prefix encoding is an efficient method of encoding an arbitrary number of nibbles as a byte array. It is able to store an additional flag which, when used in the context of the trie (the only context in which it is used), disambiguates between node types.

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

$$\mathtt{HP}(\mathbf{x}, t) : \mathbf{x} \in \mathbb{Y} \equiv \begin{cases} (16f(t), 16\mathbf{x}[0] + \mathbf{x}[1], 16\mathbf{x}[2] + \mathbf{x}[3], ...) & \text{if} \quad \|\mathbf{x}\| \text{ is even} \\ (16(f(t) + 1) + \mathbf{x}[0], 16\mathbf{x}[1] + \mathbf{x}[2], 16\mathbf{x}[3] + \mathbf{x}[4], ...) & \text{otherwise} \end{cases} \tag{186}$$

$$f(t) \equiv \begin{cases} 2 & \text{if} \quad t \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{187}$$

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:

$$\mathfrak{I} = \{(\mathbf{k}_0 \in \mathbb{B}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}_1 \in \mathbb{B}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{188}$$

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

$$\forall_{\mathbf{I}\in\mathfrak{I}} I \equiv (I_0, I_1) \tag{189}$$

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:

$$y(\mathfrak{I}) = \{(\mathbf{k}'_0 \in \mathbb{Y}, \mathbf{v}_0 \in \mathbb{B}), (\mathbf{k}'_1 \in \mathbb{Y}, \mathbf{v}_1 \in \mathbb{B}), ...\} \tag{190}$$

$$\forall_n \quad \forall_{i:i<2\|\mathbf{k}_n\|} \quad \mathbf{k}'_n[i] \equiv \begin{cases} \lfloor \mathbf{k}_n[i \div 2] \div 16 \rfloor & \text{if } i \text{ is even} \\ \mathbf{k}_n[\lfloor i \div 2 \rfloor] \bmod 16 & \text{otherwise} \end{cases} \tag{191}$$

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

$$\texttt{TRIE}(\mathfrak{I}) \equiv \texttt{KEC}(c(\mathfrak{I}, 0)) \tag{192}$$

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:

$$n(\mathfrak{I}, i) \equiv \begin{cases} () & \text{if } \mathfrak{I} = \varnothing \\ c(\mathfrak{I}, i) & \text{if } \|c(\mathfrak{I}, i)\| < 32 \\ \texttt{KEC}(c(\mathfrak{I}, i)) & \text{otherwise} \end{cases} \tag{193}$$

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

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

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

$$c(\mathfrak{I}, i) \equiv \begin{cases} \texttt{RLP}\Big((\texttt{HP}(I_0[i..(\|I_0\|-1)], true), I_1)\Big) & \text{if } \|\mathfrak{I}\| = 1 \quad \text{where } \exists I : I \in \mathfrak{I} \\ \texttt{RLP}\Big((\texttt{HP}(I_0[i..(j-1)], false), n(\mathfrak{I}, j))\Big) & \text{if } i \neq j \quad \text{where } j = \arg\max_x : \exists \mathbf{l} : \|\mathbf{l}\| = x : \forall_{I\in\mathfrak{I}} : I_0[0..(x-1)] = \mathbf{l} \\ \texttt{RLP}\Big((u(0), u(1), ..., u(15), v)\Big) & \text{otherwise} \quad \text{where } u(j) \equiv n(\{I : I \in \mathfrak{I} \wedge I_0[i] = j\}, i+1) \\ & \qquad\qquad v = \begin{cases} I_1 & \text{if } \exists I : I \in \mathfrak{I} \wedge \|I_0\| = i \\ () & \text{otherwise} \end{cases} \end{cases} \tag{194}$$

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.

$$\Xi_{\texttt{PRE}}(\boldsymbol{\sigma}, g, I, T) \equiv \begin{cases} (\varnothing, 0, A^0, ()) & \text{if} \quad g < g_{\text{r}} \\ (\boldsymbol{\sigma}, g - g_{\text{r}}, A^0, \mathbf{o}) & \text{otherwise} \end{cases} \tag{195}$$

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

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

$$\Xi_{\texttt{ECREC}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{196}$$

$$g_{\text{r}} = 3000 \tag{197}$$

$$|\mathbf{o}| = \begin{cases} 0 & \text{if} \quad \texttt{ECDSARECOVER}(h, v, r, s) = \varnothing \\ 32 & \text{otherwise} \end{cases} \tag{198}$$

$$\text{if} \quad |\mathbf{o}| = 32: \tag{199}$$

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

$$\mathbf{o}[12..31] = \texttt{KEC}\big(\texttt{ECDSARECOVER}(h, v, r, s)\big)[12..31] \quad \text{where:} \tag{201}$$

$$\mathbf{d}[0..(|I_{\mathbf{d}}| - 1)] = I_{\mathbf{d}} \tag{202}$$

$$\mathbf{d}[|I_{\mathbf{d}}|..] = (0, 0, ...) \tag{203}$$

$$h = \mathbf{d}[0..31] \tag{204}$$

$$v = \mathbf{d}[32..63] \tag{205}$$

$$r = \mathbf{d}[64..95] \tag{206}$$

$$s = \mathbf{d}[96..127] \tag{207}$$

We define $\Xi_{\texttt{SHA256}}$ and $\Xi_{\texttt{RIP160}}$ as precompiled contracts implementing the SHA2-256 and RIPEMD-160 hash functions respectively. Their gas usage is dependent on the input data size, a factor rounded up to the nearest number of words.

$$\Xi_{\texttt{SHA256}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{208}$$

$$g_{\text{r}} = 60 + 12\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{209}$$

$$\mathbf{o}[0..31] = \texttt{SHA256}(I_{\mathbf{d}}) \tag{210}$$

$$\Xi_{\texttt{RIP160}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{211}$$

$$g_{\text{r}} = 600 + 120\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{212}$$

$$\mathbf{o}[0..11] = 0 \tag{213}$$

$$\mathbf{o}[12..31] = \texttt{RIPEMD160}(I_{\mathbf{d}}) \tag{214}$$

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

$$\texttt{SHA256}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{32} \tag{215}$$

$$\texttt{RIPEMD160}(\mathbf{i} \in \mathbb{B}) \equiv o \in \mathbb{B}_{20} \tag{216}$$

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

$$\Xi_{\texttt{ID}} \equiv \Xi_{\texttt{PRE}} \quad \text{where:} \tag{217}$$

$$g_{\text{r}} = 15 + 3\left\lceil \frac{|I_{\mathbf{d}}|}{32} \right\rceil \tag{218}$$

$$\mathbf{o} = I_{\mathbf{d}} \tag{219}$$

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

$$\Xi_{\texttt{EXPMOD}} \equiv \Xi_{\texttt{PRE}} \quad \text{except:} \tag{220}$$

$$g_{\mathrm{r}} = \left\lfloor \frac{f\big(\max(\ell_M, \ell_B)\big)\max(\ell'_E, 1)}{G_{quaddivisor}} \right\rfloor \tag{221}$$

$$f(x) \equiv \begin{cases} x^2 & \text{if } x \leq 64 \\ \left\lfloor \frac{x^2}{4} \right\rfloor + 96x - 3072 & \text{if } 64 < x \leq 1024 \\ \left\lfloor \frac{x^2}{16} \right\rfloor + 480x - 199680 & \text{otherwise} \end{cases} \tag{222}$$

$$\ell'_E = \begin{cases} 0 & \text{if } \ell_E \leq 32 \wedge E = 0 \\ \lfloor \log_2(E) \rfloor & \text{if } \ell_E \leq 32 \wedge E \neq 0 \\ 8(\ell_E - 32) + \lfloor \log_2(i[(96+\ell_B)..(127+\ell_B)]) \rfloor & \text{if } 32 < \ell_E \wedge i[(96+\ell_B)..(127+\ell_B)] \neq 0 \\ 8(\ell_E - 32) & \text{otherwise} \end{cases} \tag{223}$$

$$\mathbf{o} = \left(B^E \bmod M\right) \in \mathbb{N}_{8\ell_M} \tag{224}$$

$$\ell_B \equiv i[0..31] \tag{225}$$

$$\ell_E \equiv i[32..63] \tag{226}$$

$$\ell_M \equiv i[64..95] \tag{227}$$

$$B \equiv i[96..(95+\ell_B)] \tag{228}$$

$$E \equiv i[(96+\ell_B)..(95+\ell_B+\ell_E)] \tag{229}$$

$$M \equiv i[(96+\ell_B+\ell_E)..(95+\ell_B+\ell_E+\ell_M)] \tag{230}$$

$$i[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < |I_{\mathbf{d}}| \\ 0 & \text{otherwise} \end{cases} \tag{231}$$

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

$$p \equiv 21888242871839275222246405745257275088696311157297823662689037894645226208583 \tag{232}$$

$$q \equiv 21888242871839275222246405745257275088548364400416034343698204186575808495617 \tag{233}$$

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

We define a set C_1 with

$$C_1 \equiv \{(X, Y) \in F_{\mathrm{p}} \times F_{\mathrm{p}} \mid Y^2 = X^3 + 3\} \cup \{(0, 0)\} \tag{234}$$

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

$$\begin{aligned} (X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases} (X, Y) & \text{if } X_1 \neq X_2 \\ (0, 0) & \text{otherwise} \end{cases} \\ \lambda &\equiv \frac{Y_2 - Y_1}{X_2 - X_1} \\ X &\equiv \lambda^2 - X_1 - X_2 \\ Y &\equiv \lambda(X_1 - X) - Y_1 \end{aligned} \tag{235}$$

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

$$\begin{aligned} (X_1, Y_1) + (X_2, Y_2) &\equiv \begin{cases} (X, Y) & \text{if } Y_1 \neq 0 \\ (0, 0) & \text{otherwise} \end{cases} \\ \lambda &\equiv \frac{3X_1^2}{2Y_1} \\ X &\equiv \lambda^2 - 2X_1 \\ Y &\equiv \lambda(X_1 - X) - Y_1 \end{aligned} \tag{236}$$

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

$$n \cdot P \equiv (0, 0) + \underbrace{P + \cdots + P}_{n} \tag{237}$$

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

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

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

$$C_2 \equiv \{(X,Y) \in F_{p^2} \times F_{p^2} \mid Y^2 = X^3 + 3(i+9)^{-1}\} \cup \{(0,0)\} \tag{238}$$

We define a binary operation $+$ and scalar multiplication $\cdot$ with the same equations (235), (236) and (237). $(C_2, +)$ is also known to be a group. We define P_2 in C_2 with

$$\begin{aligned} P_2 \equiv\ & (11559732032986387107991004021392285783925812861821192530917403151452391805634 \times i \\ & +10857046999023057135944570762232829481370756359578518086990519993285655852781, \\ & 4082367875863433681332203403145435568316851327593401208105741076214120093531 \times i \\ & +8495653923123431417604973247489272438418190587263600148770280649306958101930) \end{aligned} \tag{239}$$

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

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

$$\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q \tag{240}$$

$$\prod_{i=0}^{k} e(a_i, b_i) = P_T \tag{241}$$

Thus the pairing operation provides a method to verify (240).

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

$$\delta_\mathrm{p}(\mathbf{x}) \equiv \begin{cases} \mathbf{x} & \text{if } \mathbf{x} < p \\ \varnothing & \text{otherwise} \end{cases} \tag{242}$$

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

$$\delta_1(\mathbf{x}) \equiv \begin{cases} g_1 & \text{if } g_1 \in G_1 \\ \varnothing & \text{otherwise} \end{cases} \tag{243}$$

$$g_1 \equiv \begin{cases} (x,y) & \text{if } x \neq \varnothing \wedge y \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{244}$$

$$x \equiv \delta_\mathrm{p}(\mathbf{x}[0..31]) \tag{245}$$

$$y \equiv \delta_\mathrm{p}(\mathbf{x}[32..63]) \tag{246}$$

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

$$\delta_2(\mathbf{x}) \equiv \begin{cases} g_2 & \text{if } g_2 \in G_2 \\ \varnothing & \text{otherwise} \end{cases} \tag{247}$$

$$g_2 \equiv \begin{cases} ((x_0 i + y_0), (x_1 i + y_1)) & \text{if } x_0 \neq \varnothing \wedge y_0 \neq \varnothing \wedge x_1 \neq \varnothing \wedge y_1 \neq \varnothing \\ \varnothing & \text{otherwise} \end{cases} \tag{248}$$

$$x_0 \equiv \delta_\mathrm{p}(\mathbf{x}[0..31]) \tag{249}$$

$$y_0 \equiv \delta_\mathrm{p}(\mathbf{x}[32..63]) \tag{250}$$

$$x_1 \equiv \delta_\mathrm{p}(\mathbf{x}[64..95]) \tag{251}$$

$$y_1 \equiv \delta_\mathrm{p}(\mathbf{x}[96..127]) \tag{252}$$

We define $\Xi_{\texttt{SNARKV}}$ as a precompiled contract which checks if (240) holds, for intended use in zkSNARK verification.

$$\Xi_{\text{SNARKV}} \equiv \Xi_{\text{PRE}} \quad \text{except:} \tag{253}$$

$$\Xi_{\text{SNARKV}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } F \tag{254}$$

$$F \equiv (|I_{\mathbf{d}}| \bmod 192 \neq 0 \vee (\exists j.\ a_{\text{j}} = \varnothing \vee b_{\text{j}} = \varnothing)) \tag{255}$$

$$k = \frac{|I_{\mathbf{d}}|}{192} \tag{256}$$

$$g_{\text{r}} = 60000k + 40000 \tag{257}$$

$$\mathbf{o}[0..31] \equiv \begin{cases} 0x0000000000000000000000000000000000000000000000000000000000000001 & \text{if } v \wedge \neg F \\ 0x0000000000000000000000000000000000000000000000000000000000000000 & \text{if } \neg v \wedge \neg F \end{cases} \tag{258}$$

$$v \equiv (\log_{P_1}(a_1) \times \log_{P_2}(b_1) + \cdots + \log_{P_1}(a_k) \times \log_{P_2}(b_k) \equiv 1 \mod q) \tag{259}$$

$$a_1 \equiv \delta_1(I_{\mathbf{d}}[0..63]) \tag{260}$$

$$b_1 \equiv \delta_2(I_{\mathbf{d}}[64..191]) \tag{261}$$

$$\vdots$$

$$a_k \equiv \delta_1(I_{\mathbf{d}}[(|I_{\mathbf{d}}| - 192)..(|I_{\mathbf{d}}| - 129)]) \tag{262}$$

$$b_k \equiv \delta_2(I_{\mathbf{d}}[(|I_{\mathbf{d}}| - 128)..(|I_{\mathbf{d}}| - 1)]) \tag{263}$$

We define a precompiled contract for addition on G_1.

$$\Xi_{\text{BN\_ADD}} \equiv \Xi_{\text{BN\_PRE}} \quad \text{except:} \tag{264}$$

$$\Xi_{\text{BN\_ADD}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } x = \varnothing \vee y = \varnothing \tag{265}$$

$$g_{\text{r}} = 500 \tag{266}$$

$$\mathbf{o} \equiv \delta_1^{-1}(x + y) \quad \text{where } + \text{ is the group operation in } G_1 \tag{267}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[0..63]\right) \tag{268}$$

$$y \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[64..127]\right) \tag{269}$$

$$\bar{I}_{\mathbf{d}}[x] \equiv \begin{cases} I_{\mathbf{d}}[x] & \text{if } x < |I_{\mathbf{d}}| \\ 0 & \text{otherwise} \end{cases} \tag{270}$$

We define a precompiled contract for scalar multiplication on G_1, where $\bar{I}_{\mathbf{d}}$ is defined in (270).

$$\Xi_{\text{BN\_MUL}} \equiv \Xi_{\text{PRE}} \quad \text{except:} \tag{271}$$

$$\Xi_{\text{BN\_MUL}}(\boldsymbol{\sigma}, g, I) = \left(\varnothing, 0, A^0, ()\right) \quad \text{if } x = \varnothing \tag{272}$$

$$g_{\text{r}} = 40000 \tag{273}$$

$$\mathbf{o} \equiv \delta_1^{-1}(n \cdot x) \quad \text{where } \cdot \text{ is the scalar multiplication in } G_1 \tag{274}$$

$$n \equiv \bar{I}_{\mathbf{d}}[0..31] \tag{275}$$

$$x \equiv \delta_1\left(\bar{I}_{\mathbf{d}}[32..95]\right) \tag{276}$$

Appendix F. Signing Transactions

The method of signing transactions is similar to the 'Electrum style signatures' as defined by Arnaud et al. [2017], heading "Managing styles with Radium" in the bullet point list. This method utilises the SECP-256k1 curve as described by Courtois et al. [2014], and is implemented similarly to as described by Gura et al. [2004] on p. 9 of 15, para. 3.

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

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

$$\texttt{ECDSAPUBKEY}(p_{\text{r}} \in \mathbb{B}_{32}) \equiv p_{\text{u}} \in \mathbb{B}_{64} \tag{277}$$

$$\texttt{ECDSASIGN}(e \in \mathbb{B}_{32}, p_{\text{r}} \in \mathbb{B}_{32}) \equiv (v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \tag{278}$$

$$\texttt{ECDSARECOVER}(e \in \mathbb{B}_{32}, v \in \mathbb{B}_1, r \in \mathbb{B}_{32}, s \in \mathbb{B}_{32}) \equiv p_{\text{u}} \in \mathbb{B}_{64} \tag{279}$$

Where p_{u} is the public key, assumed to be a byte array of size 64 (formed from the concatenation of two positive integers each $< 2^{256}$), p_{r} is the private key, a byte array of size 32 (or a single positive integer in the aforementioned range) and e is the hash of the transaction, $h(T)$. It is assumed that v is the 'recovery identifier'. The recovery identifier is a 1 byte value specifying the parity and finiteness of the coordinates of the curve point for which r is the x-value; this

value is in the range of $[27, 30]$, however we declare the upper two possibilities, representing infinite values, invalid. The value 27 represents an even y value and 28 represents an odd y value.

We declare that an ECDSA signature is invalid unless all the following conditions are true[5]:

$$0 < r < \texttt{secp256k1n} \tag{280}$$

$$0 < s < \texttt{secp256k1n} \div 2 + 1 \tag{281}$$

$$v \in \{27, 28\} \tag{282}$$

where:

$$\texttt{secp256k1n} = 115792089237316195423570985008687907852837564279074904382605163141518161494337 \tag{283}$$

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

$$A(p_{\mathrm{r}}) = \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSAPUBKEY}(p_{\mathrm{r}})\big)\big) \tag{284}$$

The message hash, $h(T)$, to be signed is the Keccak hash of the transaction. Two different flavors of signing schemes are available. One operates without the latter three signature components, formally described as T_{r}, T_{s} and T_{w}. The other operates on nine elements:

$$L_S(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}) & \text{if } v \in \{27, 28\} \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, \mathbf{p}, \texttt{chain\_id}, (), ()) & \text{otherwise} \end{cases} \tag{285}$$

where

$$\mathbf{p} \equiv \begin{cases} T_{\mathbf{i}} & \text{if } T_{\mathrm{t}} = 0 \\ T_{\mathbf{d}} & \text{otherwise} \end{cases}$$

$$h(T) \equiv \texttt{KEC}(L_S(T)) \tag{286}$$

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

$$G(T, p_{\mathrm{r}}) \equiv T \quad \text{except:} \tag{287}$$

$$(T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) = \texttt{ECDSASIGN}(h(T), p_{\mathrm{r}}) \tag{288}$$

Reiterating from previously:

$$T_{\mathrm{r}} = r \tag{289}$$

$$T_{\mathrm{s}} = s \tag{290}$$

T_{w} is either the recovery identifier or 'chain identifier doubled plus 35 or 36'. In the second case, where v is the chain identifier doubled plus 35 or 36, the values 35 and 36 assume the role of the 'recovery identifier' by specifying the parity of y, with the value 35 representing an even value and 36 representing an odd value.

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

$$S(T) \equiv \mathcal{B}_{96..255}\big(\texttt{KEC}\big(\texttt{ECDSARECOVER}(h(T), v_0, T_{\mathrm{r}}, T_{\mathrm{s}})\big)\big) \tag{291}$$

$$v_0 \equiv \begin{cases} T_{\mathrm{w}} & \text{if } T_{\mathrm{w}} \in \{27, 28\} \\ 28 - (T_{\mathrm{w}} \bmod 2) & \text{otherwise} \end{cases} \tag{292}$$

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

$$\forall T : \forall p_{\mathrm{r}} : S(G(T, p_{\mathrm{r}})) \equiv A(p_{\mathrm{r}}) \tag{293}$$

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.

[5] A signature of a transaction can be valid not only with a recovery identifier but with some other numbers. See how the component T_w of a transaction is interpreted.

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.
$G_{quaddivisor}$	100	The quadratic coefficient of the input sizes of the exponentiation-over-modulo precompiled contract.

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:

(294)

$$C(\boldsymbol{\sigma}, \boldsymbol{\mu}, I) \equiv C_{mem}(\boldsymbol{\mu}'_{\mathrm{i}}) - C_{mem}(\boldsymbol{\mu}_{\mathrm{i}}) + \begin{cases} C_{\mathrm{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 \\ & \quad \text{CODECOPY} \vee \text{RETURNDATACOPY} \\ 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_{\mathrm{CALL}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{if } w = \text{CALL} \vee \text{CALLCODE} \vee \\ & \quad \text{DELEGATECALL} \\ C_{\mathrm{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_{\mathrm{low}} & \text{if } w \in W_{\mathrm{low}} \\ G_{mid} & \text{if } w \in W_{mid} \\ G_{\mathrm{high}} & \text{if } w \in W_{\mathrm{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}$$

(295)

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

where:

(296)

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

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

$W_{zero} = \{\text{STOP, RETURN, REVERT}\}$
$W_{base} = \{\text{ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, RETURNDATASIZE, POP, PC, MSIZE, GAS}\}$
$W_{verylow} = \{\text{ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}\}$
$W_{\mathrm{low}} = \{\text{MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}\}$
$W_{mid} = \{\text{ADDMOD, MULMOD, JUMP}\}$
$W_{\mathrm{high}} = \{\text{JUMPI}\}$
$W_{extcode} = \{\text{EXTCODESIZE}\}$

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, $\boldsymbol{\mu}_{\mathrm{i}}$ 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. $\boldsymbol{\mu}'_{\mathrm{i}}$ 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:

$$M(s, f, l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f + l) \div 32 \rceil) & \text{otherwise} \end{cases} \tag{297}$$

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

$$L(n) \equiv n - \lfloor n/64 \rfloor \tag{298}$$

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:

$$O(\boldsymbol{\sigma}, \boldsymbol{\mu}, A, I) \equiv (\boldsymbol{\sigma}', \boldsymbol{\mu}', A', I) \quad \text{with exceptions, as noted} \tag{299}$$

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

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

Value	Mnemonic	δ	α	Description
				10s: Comparison & Bitwise Logic Operations
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 the Nth byte, we count from the left (i.e. N=0 would be the most significant in big endian).

Value	Mnemonic	δ	α	Description
				20s: SHA3
0x20	SHA3	2	1	Compute Keccak-256 hash. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \texttt{Keccak}(\boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$ $\boldsymbol{\mu}'_{\mathbf{i}} \equiv M(\boldsymbol{\mu}_{\mathbf{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$

30s: Environmental Information				
Value	**Mnemonic**	δ	α	**Description**
0x30	ADDRESS	0	1	Get address of currently executing account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{a}}$
0x31	BALANCE	1	1	Get balance of the given account. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0]]_{\mathrm{b}} & \text{if } \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}] \neq \varnothing \\ 0 & \text{otherwise} \end{cases}$
0x32	ORIGIN	0	1	Get execution origination address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{o}}$ This is the sender of original transaction; it is never an account with non-empty associated code.
0x33	CALLER	0	1	Get caller address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{s}}$ This is the address of the account that is directly responsible for this execution.
0x34	CALLVALUE	0	1	Get deposited value by the instruction/transaction responsible for this execution. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{v}}$
0x35	CALLDATALOAD	1	1	Get input data of current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \quad \text{with} \quad I_{\mathbf{d}}[x] = 0 \quad \text{if} \quad x \geqslant \|I_{\mathbf{d}}\|$ This pertains to the input data passed with the message call instruction or transaction.
0x36	CALLDATASIZE	0	1	Get size of input data in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{d}}\|$ This pertains to the input data passed with the message call instruction or transaction.
0x37	CALLDATACOPY	3	0	Copy input data in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{d}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{d}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ This pertains to the input data passed with the message call instruction or transaction.
0x38	CODESIZE	0	1	Get size of code running in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|I_{\mathbf{b}}\|$
0x39	CODECOPY	3	0	Copy code running in current environment to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo.
0x3a	GASPRICE	0	1	Get price of gas in current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{\mathrm{p}}$ This is gas price specified by the originating transaction.
0x3b	EXTCODESIZE	1	1	Get size of an account's code. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_{\mathrm{c}}\|$
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 $\mathbf{c} \equiv \boldsymbol{\sigma}[\boldsymbol{\mu}_{\mathbf{s}}[0] \mod 2^{160}]_{\mathrm{c}}$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[1], \boldsymbol{\mu}_{\mathbf{s}}[3])$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[2] + i$ are not subject to the 2^{256} modulo.
0x3d	RETURNDATASIZE	0	1	Get size of output data from the previous call from the current environment. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \|\boldsymbol{\mu}_{\mathbf{o}}\|$
0x3e	RETURNDATACOPY	3	0	Copy output data from the previous call to memory. $\forall_{i \in \{0 \dots \boldsymbol{\mu}_{\mathbf{s}}[2]-1\}} \boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] + i] \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{o}}[\boldsymbol{\mu}_{\mathbf{s}}[1] + i] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] + i < \|\boldsymbol{\mu}_{\mathbf{o}}\| \\ 0 & \text{otherwise} \end{cases}$ The additions in $\boldsymbol{\mu}_{\mathbf{s}}[1] + i$ are not subject to the 2^{256} modulo. $\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[2])$

40s: Block Information

Value	Mnemonic	δ	α	Description
0x40	BLOCKHASH	1	1	Get the hash of one of the 256 most recent complete blocks. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv P(I_{\mathbf{H}_\mathrm{p}}, \boldsymbol{\mu}_{\mathbf{s}}[0], 0)$ 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. $P(h,n,a) \equiv \begin{cases} 0 & \text{if} \quad n > H_\mathrm{i} \vee a = 256 \vee h = 0 \\ h & \text{if} \quad n = H_\mathrm{i} \\ P(H_\mathrm{p}, n, a+1) & \text{otherwise} \end{cases}$ and we assert the header H can be determined from its hash h unless as its hash is the parent h is zero.
0x41	COINBASE	0	1	Get the block's beneficiary address. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{c}}$
0x42	TIMESTAMP	0	1	Get the block's timestamp. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{s}}$
0x43	NUMBER	0	1	Get the block's number. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{i}}$
0x44	DIFFICULTY	0	1	Get the block's difficulty. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{d}}$
0x45	GASLIMIT	0	1	Get the block's gas limit. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv I_{H\mathrm{l}}$

50s: Stack, Memory, Storage and Flow Operations				
Value	**Mnemonic**	δ	α	**Description**
0x50	POP	1	0	Remove item from stack.
0x51	MLOAD	1	1	Load word from memory. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x52	MSTORE	2	0	Save word to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \dots (\boldsymbol{\mu}_{\mathbf{s}}[0] + 31)] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 32) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x53	MSTORE8	2	0	Save byte to memory. $\boldsymbol{\mu}'_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv (\boldsymbol{\mu}_{\mathbf{s}}[1] \bmod 256)$ $\boldsymbol{\mu}'_{\mathrm{i}} \equiv \max(\boldsymbol{\mu}_{\mathrm{i}}, \lceil (\boldsymbol{\mu}_{\mathbf{s}}[0] + 1) \div 32 \rceil)$ The addition in the calculation of $\boldsymbol{\mu}'_{\mathrm{i}}$ is not subject to the 2^{256} modulo.
0x54	SLOAD	1	1	Load word from storage. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]]$
0x55	SSTORE	2	0	Save word to storage. $\boldsymbol{\sigma}'[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \equiv \boldsymbol{\mu}_{\mathbf{s}}[1]$ $C_{\mathrm{SSTORE}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{sset} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \wedge \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] = 0 \\ G_{sreset} & \text{otherwise} \end{cases}$ $A'_r \equiv A_r + \begin{cases} R_{sclear} & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \wedge \boldsymbol{\sigma}[I_{\mathrm{a}}]_{\mathbf{s}}[\boldsymbol{\mu}_{\mathbf{s}}[0]] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0x56	JUMP	1	0	Alter the program counter. $J_{\mathrm{JUMP}}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_{\mathbf{s}}[0]$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x57	JUMPI	2	0	Conditionally alter the program counter. $J_{\mathrm{JUMPI}}(\boldsymbol{\mu}) \equiv \begin{cases} \boldsymbol{\mu}_{\mathbf{s}}[0] & \text{if } \boldsymbol{\mu}_{\mathbf{s}}[1] \neq 0 \\ \boldsymbol{\mu}_{\mathrm{pc}} + 1 & \text{otherwise} \end{cases}$ This has the effect of writing said value to $\boldsymbol{\mu}_{\mathrm{pc}}$. See section 9.
0x58	PC	0	1	Get the value of the program counter *prior* to the increment corresponding to this instruction. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathrm{pc}}$
0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_i$
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. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv c(\boldsymbol{\mu}_{\mathrm{pc}} + 1)$ where $c(x) \equiv \begin{cases} I_{\mathbf{b}}[x] & \text{if } x < \lVert I_{\mathbf{b}} \rVert \\ 0 & \text{otherwise} \end{cases}$ The bytes are read in line from the program code's bytes array. The function c ensures the bytes default to zero if they extend past the limits. The byte is right-aligned (takes the lowest significant place in big endian).
0x61	PUSH2	0	1	Place 2-byte item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 2)\big)$ with $\boldsymbol{c}(\boldsymbol{x}) \equiv (c(\boldsymbol{x}_0), ..., c(\boldsymbol{x}_{\lVert x \rVert - 1}))$ with c as defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).
⋮	⋮	⋮	⋮	⋮
0x7f	PUSH32	0	1	Place 32-byte (full word) item on stack. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{c}\big((\boldsymbol{\mu}_{\mathrm{pc}} + 1) \ldots (\boldsymbol{\mu}_{\mathrm{pc}} + 32)\big)$ where $\boldsymbol{c}$ is defined as above. The bytes are right-aligned (takes the lowest significant place in big endian).

80s: Duplication Operations

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

90s: Exchange Operations

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

a0s: Logging Operations

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

$$\text{A }1A'_{\mathbf{l}} \equiv A_{\mathbf{l}} \cdot (I_{\mathrm{a}}, \mathbf{t}, \boldsymbol{\mu}_{\mathbf{m}}[\boldsymbol{\mu}_{\mathbf{s}}[0] \ldots (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1] - 1)])$$

and to update the memory consumption counter:
$\boldsymbol{\mu}'_{\mathrm{i}} \equiv M(\boldsymbol{\mu}_{\mathrm{i}}, \boldsymbol{\mu}_{\mathbf{s}}[0], \boldsymbol{\mu}_{\mathbf{s}}[1])$
The entry's topic series, $\mathbf{t}$, differs accordingly:

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

f0s: System operations

Value	Mnemonic	δ	α	Description
0xf0	CREATE	3	1	Create a new account with associated code. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[1] \dots (\boldsymbol{\mu}_\mathbf{s}[1] + \boldsymbol{\mu}_\mathbf{s}[2] - 1)]$ $(\boldsymbol{\sigma}', \boldsymbol{\mu}'_\mathrm{g}, A^+, \mathbf{o}) \equiv \begin{cases} \Lambda(\boldsymbol{\sigma}^*, I_\mathrm{a}, I_\mathrm{o}, L(\boldsymbol{\mu}_\mathrm{g}), I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[0], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \text{if } \boldsymbol{\mu}_\mathbf{s}[0] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \\ & \wedge\, I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, \boldsymbol{\mu}_\mathrm{g}, \varnothing) & \text{otherwise} \end{cases}$ $\boldsymbol{\sigma}^* \equiv \boldsymbol{\sigma}$ except $\boldsymbol{\sigma}^*[I_\mathrm{a}]_\mathrm{n} = \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{n} + 1$ $A' \equiv A \Cup A^+$ which abbreviates: $A'_\mathbf{s} \equiv A_\mathbf{s} \cup A^+_\mathbf{s} \quad \wedge \quad A'_\mathbf{l} \equiv A_\mathbf{l} \cdot A^+_\mathbf{l} \quad \wedge$ $A'_\mathbf{t} \equiv A_\mathbf{t} \cup A^+_\mathbf{t} \wedge \quad A'_\mathbf{r} \equiv A_\mathbf{r} + A^+_\mathbf{r}$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\boldsymbol{\sigma}' = \varnothing$, or $I_\mathrm{e} = 1024$ (the maximum call depth limit is reached) or $\boldsymbol{\mu}_\mathbf{s}[0] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (balance of the caller is too low to fulfil the value transfer); and otherwise $x = A(I_\mathrm{a}, \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{n})$, the address of the newly created account. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[1], \boldsymbol{\mu}_\mathbf{s}[2])$ $\boldsymbol{\mu}'_\mathbf{o} \equiv ()$ Thus the operand order is: value, input offset, input size.
0xf1	CALL	7	1	Message-call into an account. $\mathbf{i} \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[3] \dots (\boldsymbol{\mu}_\mathbf{s}[3] + \boldsymbol{\mu}_\mathbf{s}[4] - 1)]$ $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}, I_\mathrm{a}, I_\mathrm{o}, t, t, C_\mathrm{CALLGAS}(\boldsymbol{\mu}), & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge \\ \quad I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \quad I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ $n \equiv \min(\{\boldsymbol{\mu}_\mathbf{s}[6], |\mathbf{o}|\})$ $\boldsymbol{\mu}'_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[5] \dots (\boldsymbol{\mu}_\mathbf{s}[5] + n - 1)] = \mathbf{o}[0 \dots (n-1)]$ $\boldsymbol{\mu}'_\mathbf{o} = \mathbf{o}$ $\boldsymbol{\mu}'_\mathrm{g} \equiv \boldsymbol{\mu}_\mathrm{g} + g'$ $\boldsymbol{\mu}'_\mathbf{s}[0] \equiv x$ $A' \equiv A \Cup A^+$ $t \equiv \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}$ where $x = 0$ if the code execution for this operation failed due to an exceptional halting (or for a REVERT) $\boldsymbol{\sigma}' = \varnothing$ or if $\boldsymbol{\mu}_\mathbf{s}[2] > \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}$ (not enough funds) or $I_\mathrm{e} = 1024$ (call depth limit reached); $x = 1$ otherwise. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[3], \boldsymbol{\mu}_\mathbf{s}[4]), \boldsymbol{\mu}_\mathbf{s}[5], \boldsymbol{\mu}_\mathbf{s}[6])$ Thus the operand order is: gas, to, value, in offset, in size, out offset, out size. $C_\mathrm{CALL}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_\mathrm{CALLGAS}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}) + G_{callstipend} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}) & \text{otherwise} \end{cases}$ $C_\mathrm{GASCAP}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} \min\{L(\boldsymbol{\mu}_\mathrm{g} - C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu})), \boldsymbol{\mu}_\mathbf{s}[0]\} & \text{if } \boldsymbol{\mu}_\mathrm{g} \geq C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \\ \boldsymbol{\mu}_\mathbf{s}[0] & \text{otherwise} \end{cases}$ $C_\mathrm{EXTRA}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{call} + C_\mathrm{XFER}(\boldsymbol{\mu}) + C_\mathrm{NEW}(\boldsymbol{\sigma}, \boldsymbol{\mu})$ $C_\mathrm{XFER}(\boldsymbol{\mu}) \equiv \begin{cases} G_{callvalue} & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$ $C_\mathrm{NEW}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv \begin{cases} G_{newaccount} & \text{if } \texttt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_\mathbf{s}[1] \mod 2^{160}) \wedge \boldsymbol{\mu}_\mathbf{s}[2] \neq 0 \\ 0 & \text{otherwise} \end{cases}$
0xf2	CALLCODE	7	1	Message-call into this account with an alternative account's code. Exactly equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_\mathrm{a}, I_\mathrm{o}, I_\mathrm{a}, t, C_\mathrm{CALLGAS}(\boldsymbol{\mu}), & \text{if } \boldsymbol{\mu}_\mathbf{s}[2] \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge \\ \quad I_\mathrm{p}, \boldsymbol{\mu}_\mathbf{s}[2], \boldsymbol{\mu}_\mathbf{s}[2], \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \quad I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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. $H_\mathrm{RETURN}(\boldsymbol{\mu}) \equiv \boldsymbol{\mu}_\mathbf{m}[\boldsymbol{\mu}_\mathbf{s}[0] \dots (\boldsymbol{\mu}_\mathbf{s}[0] + \boldsymbol{\mu}_\mathbf{s}[1] - 1)]$ This has the effect of halting the execution at this point with output defined. See section 9. $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$

0xf4	DELEGATECALL	6	1	Message-call into this account with an alternative account's code, but persisting the current values for *sender* and *value*. Compared with CALL, DELEGATECALL takes one fewer arguments. The omitted argument is $\boldsymbol{\mu}_\mathbf{s}[2]$. As a result, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ in the definition of CALL should respectively be replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. Otherwise it is equivalent to CALL except: $(\boldsymbol{\sigma}', g', A^+, \mathbf{o}) \equiv \begin{cases} \Theta(\boldsymbol{\sigma}^*, I_\mathrm{s}, I_\mathrm{o}, I_\mathrm{a}, t, \\ \quad \boldsymbol{\mu}_\mathbf{s}[0], I_\mathrm{p}, 0, I_\mathrm{v}, \mathbf{i}, I_\mathrm{e}+1, I_\mathrm{w}) & \text{if} \quad I_\mathrm{v} \leqslant \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \wedge \\ I_\mathrm{e} < 1024 \\ (\boldsymbol{\sigma}, g, \varnothing, ()) & \text{otherwise} \end{cases}$ 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.
0xfa	STATICCALL	6	1	Static message-call into an account. Exactly equivalent to CALL except: The argument $\boldsymbol{\mu}_\mathbf{s}[2]$ is replaced with 0. The deeper argument $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$, $\boldsymbol{\mu}_\mathbf{s}[5]$ and $\boldsymbol{\mu}_\mathbf{s}[6]$ are respectively replaced with $\boldsymbol{\mu}_\mathbf{s}[2]$, $\boldsymbol{\mu}_\mathbf{s}[3]$, $\boldsymbol{\mu}_\mathbf{s}[4]$ and $\boldsymbol{\mu}_\mathbf{s}[5]$. The last argument of Θ is $\bot$.
0xfd	REVERT	2	0	Halt execution reverting state changes but returning data and remaining gas. The effect of this operation is described in (131). For the gas calculation, we use the memory expansion function, $\boldsymbol{\mu}'_\mathrm{i} \equiv M(\boldsymbol{\mu}_\mathrm{i}, \boldsymbol{\mu}_\mathbf{s}[0], \boldsymbol{\mu}_\mathbf{s}[1])$
0xfe	INVALID	$\varnothing$	$\varnothing$	Designated invalid instruction.
0xff	SELFDESTRUCT	1	0	Halt execution and register account for later deletion. $A'_\mathbf{s} \equiv A_\mathbf{s} \cup \{I_\mathrm{a}\}$ $\boldsymbol{\sigma}'[r] \equiv \begin{cases} \varnothing & \text{if } \boldsymbol{\sigma}[r] = \varnothing \ \wedge \ \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} = 0 \\ (\boldsymbol{\sigma}[r]_\mathrm{n}, \boldsymbol{\sigma}[r]_\mathrm{b} + \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b}, \boldsymbol{\sigma}[r]_\mathbf{s}, \boldsymbol{\sigma}[r]_\mathrm{c}) & \text{if } r \neq I_\mathrm{a} \\ (\boldsymbol{\sigma}[r]_\mathrm{n}, 0, \boldsymbol{\sigma}[r]_\mathbf{s}, \boldsymbol{\sigma}[r]_\mathrm{c}) & \text{otherwise} \end{cases}$ where $r = \boldsymbol{\mu}_\mathbf{s}[0] \bmod 2^{160}$ $\boldsymbol{\sigma}'[I_\mathrm{a}]_\mathrm{b} = 0$ $C_{\text{SELFDESTRUCT}}(\boldsymbol{\sigma}, \boldsymbol{\mu}) \equiv G_{selfdestruct} + \begin{cases} G_{newaccount} & \text{if} \quad n \\ 0 & \text{otherwise} \end{cases}$ $n \equiv \texttt{DEAD}(\boldsymbol{\sigma}, \boldsymbol{\mu}_\mathbf{s}[0] \mod 2^{160}) \wedge \boldsymbol{\sigma}[I_\mathrm{a}]_\mathrm{b} \neq 0$

Appendix I. Genesis Block

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

$$\big((0_{256}, \texttt{KEC}\big(\texttt{RLP}\big(()\big)\big), 0_{160}, \mathit{stateRoot}, 0, 0, 0_{2048}, 2^{17}, 0, 0, 3141592, \mathit{time}, 0, 0_{256}, \texttt{KEC}\big((42)\big)\big), (), ()\big) \tag{300}$$

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 list in RLP, both empty lists.

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

Appendix J. Ethash

J.1. **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.
$J_{\text{hashbytes}}$	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.

$$E_{epoch}(H_{\text{i}}) = \left\lfloor \frac{H_{\text{i}}}{J_{epoch}} \right\rfloor \tag{301}$$

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

$$d_{size} = E_{\text{prime}}(J_{datasetinit} + J_{datasetgrowth} \cdot E_{epoch} - J_{mixbytes}, J_{mixbytes}) \tag{302}$$

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

$$c_{size} = E_{\text{prime}}(J_{cacheinit} + J_{cachegrowth} \cdot E_{epoch} - J_{\text{hashbytes}}, J_{\text{hashbytes}}) \tag{303}$$

$$E_{\text{prime}}(x, y) = \begin{cases} x & \text{if} \quad x/y \in \mathbb{P} \\ E_{\text{prime}}(x - 2 \cdot y, y) & \text{otherwise} \end{cases} \tag{304}$$

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

J.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:

$$\mathbf{s} = C_{seedhash}(H_{\text{i}}) \tag{305}$$

$$C_{seedhash}(H_{\text{i}}) = \begin{cases} \mathbf{0}_{32} & \text{if} \quad E_{epoch}(H_{\text{i}}) = 0 \\ \texttt{KEC}(C_{seedhash}(H_{\text{i}} - J_{epoch})) & \text{otherwise} \end{cases} \tag{306}$$

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:

$$\mathbf{c}_i = \begin{cases} \texttt{KEC512}(\mathbf{s}) & \text{if} \quad i = 0 \\ \texttt{KEC512}(\mathbf{c}_{i-1}) & \text{otherwise} \end{cases} \tag{307}$$

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

$$\mathbf{c}'[i] = \mathbf{c}_i \quad \forall \quad i < n \tag{308}$$

$$n = \left\lfloor \frac{c_{size}}{J_{\text{hashbytes}}} \right\rfloor \tag{309}$$

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

$$\mathbf{c} = E_{cacherounds}(\mathbf{c}', J_{cacherounds}) \tag{310}$$

$$E_{cacherounds}(\mathbf{x}, y) = \begin{cases} \mathbf{x} & \text{if} \quad y = 0 \\ E_{\text{RMH}}(\mathbf{x}) & \text{if} \quad y = 1 \\ E_{cacherounds}(E_{\text{RMH}}(\mathbf{x}), y - 1) & \text{otherwise} \end{cases} \tag{311}$$

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

$$E_{\text{RMH}}(\mathbf{x}) = \big(E_{rmh}(\mathbf{x}, 0), E_{rmh}(\mathbf{x}, 1), ..., E_{rmh}(\mathbf{x}, n - 1)\big) \tag{312}$$

$$E_{rmh}(\mathbf{x}, i) = \texttt{KEC512}(\mathbf{x}'[(i - 1 + n) \mod n] \oplus \mathbf{x}'[\mathbf{x}'[i][0] \mod n]) \quad \text{with} \quad \mathbf{x}' = \mathbf{x} \quad \text{except} \quad \mathbf{x}'[j] = E_{rmh}(\mathbf{x}, j) \quad \forall \quad j < i \tag{313}$$

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 $J_{\text{hashbytes}}$ bytes in size:

$$\mathbf{d}[i] = E_{datasetitem}(\mathbf{c}, i) \quad \forall \quad i < \left\lfloor \frac{d_{size}}{J_{\text{hashbytes}}} \right\rfloor \tag{314}$$

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

$$E_{\text{FNV}}(\mathbf{x}, \mathbf{y}) = (\mathbf{x} \cdot (\text{0x01000193} \oplus \mathbf{y})) \mod 2^{32} \tag{315}$$

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

$$E_{datasetitem}(\mathbf{c}, i) = E_{\text{parents}}(\mathbf{c}, i, -1, \varnothing) \tag{316}$$

$$E_{\text{parents}}(\mathbf{c}, i, p, \mathbf{m}) = \begin{cases} E_{\text{parents}}(\mathbf{c}, i, p + 1, E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1)) & \text{if} \quad p < J_{\text{parents}} - 2 \\ E_{mix}(\mathbf{m}, \mathbf{c}, i, p + 1) & \text{otherwise} \end{cases} \tag{317}$$

$$E_{mix}(\mathbf{m}, \mathbf{c}, i, p) = \begin{cases} \texttt{KEC512}(\mathbf{c}[i \mod c_{size}] \oplus i) & \text{if} \quad p = 0 \\ E_{\text{FNV}}\big(\mathbf{m}, \mathbf{c}[E_{\text{FNV}}(i \oplus p, \mathbf{m}[p \mod \lfloor J_{\text{hashbytes}}/J_{wordbytes} \rfloor]) \mod c_{size}]\big) & \text{otherwise} \end{cases} \tag{318}$$

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

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

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

$$\texttt{PoW}(H_{\not H}, H_{\text{n}}, \mathbf{d}) = \{\mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_{\text{n}}, \mathbf{d}), \texttt{KEC}(\mathbf{s}_{\text{h}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_{\text{n}}) + \mathbf{m}_{\text{c}}(\texttt{KEC}(\texttt{RLP}(L_H(H_{\not H}))), H_{\text{n}}, \mathbf{d}))\} \tag{319}$$

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

$$\mathbf{m}_{\text{c}}(\mathbf{h}, \mathbf{n}, \mathbf{d}) = E_{compress}(E_{\text{accesses}}(\mathbf{d}, \sum_{i=0}^{n_{mix}} \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), \mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}), -1), -4) \tag{320}$$

The seed hash being:

$$\mathbf{s}_{\text{h}}(\mathbf{h}, \mathbf{n}) = \texttt{KEC512}(\mathbf{h} + E_{revert}(\mathbf{n})) \tag{321}$$

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

$$E_{revert}(\mathbf{n})[i] = \mathbf{n}[\|\mathbf{n}\| - i] \tag{322}$$

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:

$$n_{mix} = \left\lfloor \frac{J_{mixbytes}}{J_{\text{hashbytes}}} \right\rfloor \tag{323}$$

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

$$E_{\text{accesses}}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = \begin{cases} E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) & \text{if} \quad i = J_{\text{accesses}} - 2 \\ E_{\text{accesses}}(E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i), \mathbf{s}, i + 1) & \text{otherwise} \end{cases} \tag{324}$$

$$E_{mixdataset}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) = E_{\text{FNV}}(\mathbf{m}, E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i) \tag{325}$$

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

$$E_{newdata}(\mathbf{d}, \mathbf{m}, \mathbf{s}, i)[j] = \mathbf{d}[E_{\text{FNV}}(i \oplus \mathbf{s}[0], \mathbf{m}[i \mod \left\lfloor \frac{J_{mixbytes}}{J_{wordbytes}} \right\rfloor]) \mod \left\lfloor \frac{d_{size}/J_{\text{hashbytes}}}{n_{mix}} \right\rfloor \cdot n_{mix} + j] \quad \forall \quad j < n_{mix} \tag{326}$$

The mix is compressed as follows:

$$E_{compress}(\mathbf{m}, i) = \begin{cases} \mathbf{m} & \text{if} \quad i \geqslant \|\mathbf{m}\| - 8 \\ E_{compress}(E_{\text{FNV}}(E_{\text{FNV}}(E_{\text{FNV}}(\mathbf{m}[i + 4], \mathbf{m}[i + 5]), \mathbf{m}[i + 6]), \mathbf{m}[i + 7]), i + 8) & \text{otherwise} \end{cases} \tag{327}$$

Appendix K. Anomalies on the Main Network

K.1. **Deletion of an Account Despite Out-of-gas.** At block 2675119, in the transaction 0xcf416c536ec1a19ed1fb89e4ec7ffb3cf73aa413b3aa9b77d60e4fd81a4296ba, an account at address 0x03 was called and an out-of-gas occurred during the call. Against the equation (195), this added 0x03 in the set of touched addresses, and this transaction turned $\boldsymbol{\sigma}[0x03]$ into $\varnothing$.

Appendix L. List of mathematical symbols

Symbol	Latex Command	Description
$\bigvee$	`\bigvee`	This is the least upper bound, supremum, or join of all elements operated on. Thus it is the greatest element of such elements (Davey and Priestley [2002]).