Solana：一种高性能区块链的新架构（v0.8.13）

Anatoly Yakovenko
anatoly@solana.io

摘要

本文提出了一种基于历史证明（PoH，验证事件之间的顺序和时间流逝的证明）的新型区块链基础架构。PoH 发挥作用的地方在于将不可靠的时间段编码到一个只能增加数据的账本中。当它结合某种共识算法例如工作量证明（PoW）或权益证明（PoS）的时候，PoH 可以减少拜占庭式容错复制状态机中的消息传递负载，从而实现毫秒级的确认时间。另外，本文还提出了两种利用 PoH 账本的时间保持特性的算法：一种可以从任何大小的分区恢复的 PoS 算法和一种高效的流式复制证明（PoRep）。PoRep 和 PoH 的结合为账本提供了一种在时间（序列）和存储防止伪造的措施。该协议在 1 gbps 网络上进行了测试，本文显示了在当今的硬件环境下，其吞吐量每秒可处理的交易能达 71 万笔。

1 介绍

区块链本质上是一个容错复制状态机。目前的公共区块链都不依赖时间，或者对参与者保持时间的能力做了弱假设[4,5]。网络中的每个节点通常依

赖于自身的本地时钟，而不知道网络中任何其他参与者的时钟。缺乏可信的时间来源意味着当使用消息时间戳接受或拒绝某个消息时，不能保证网络中的每个其他参与者都会做出完全相同的选择。本文介绍的 PoH 旨在创建一个可验证时间流逝的账本，即事件和消息序列之间的持续时间。按照设定，网络中的每个节点都可以不受信任地依赖账本记录的时间流逝。

2 提纲

本文剩余章节结构如下。第 3 章介绍系统总体设计。第 4 章深入剖析历史证明。第 5 章对本文提出的权益证明共识算法进行深入介绍。第 6 章将详细分析本文提出的高速复制证明。第 7 章分析了统架构和性能限制。第 7.5 章则介绍一个高性能、GPU 友好的智能合约引擎。

3 网络设计

如图 1 所示，在任何给定的时间，某个系统节点将被指定为领导者，它负责生成一个历史证明序列，提供网络全局读取一致性以及可验证的时间流逝。领导者搜集用户消息并对其进行排序，以便系统中的其他节点有效地处理这些消息，从而最大限度地提高吞吐量。它对存储在 RAM 中的当前状态交易进行处理，并将它和最终状态的签名发布到称为验证节点（Verifier）的复制节点。验证节点在其状态副本上执行相同的交易，并发布其处理的状态签名，作为交易的确认。该发布的确认结果视为共识算法的投票。

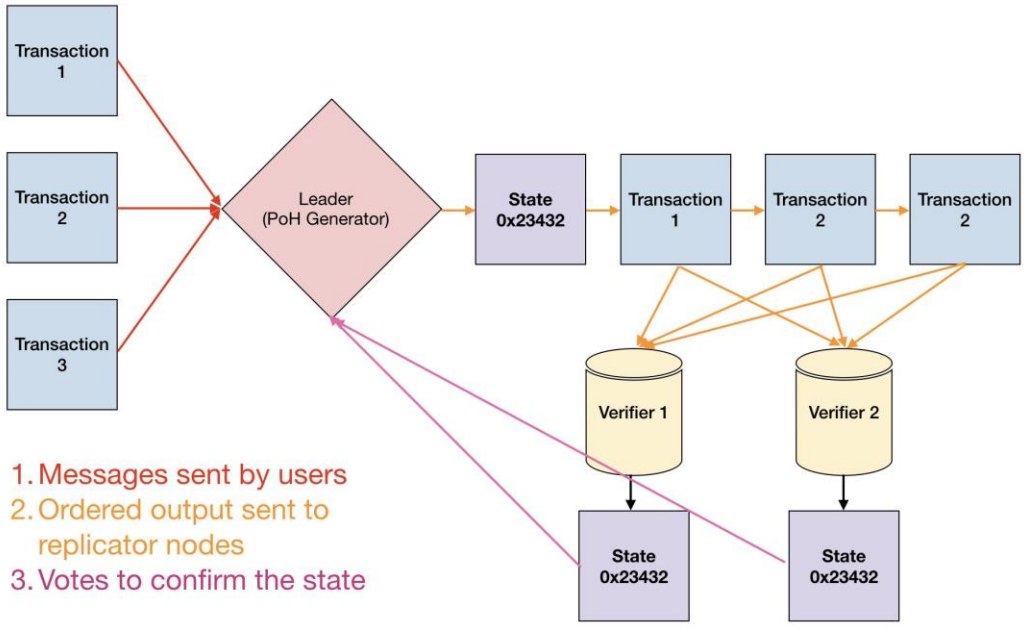


图 1：整体网络的交易流向

在非分区状态下，网络在任何给定时间都有一个领导者。每个验证节点节点的硬件配置都和领导者的相同，并且可以被选为领导者，这是通过基于 PoS 的选举来实现的。5.6 章节将对本系统的 PoS 算法选择进行详细介绍。

根据 CAP 定理，在分区事件中，一致性的重要性基本总是高于可用性。在分区较大的情况下，本文提出了一种从任意大小分区恢复网络控制的机制。该机制将在 5.12 节进行详细介绍。

4 历史证明

历史证明是一个计算序列，它能够提供一种通过密码学验证两个事件之间时间推移的方法。它使用了一个加密的安全函数，因此无法通过输入预测输出，用户必须完全执行函数才能生成输出。该函数在某个核心上按顺序运行，它的上一个输出作为当前的输入，定期记录当前输出以及被调用的

次数。然后，外部计算机可以通过检查单独核上的每个序列段来并行重新计算和验证输出。

通过将数据（或某些数据的哈希）添加到函数的状态中，数据能够封装成时间戳加入该序列中。状态、索引和数据附加到序列中，生产一个时间戳，以此确保数据是在下一个序列产生的哈希之前创建的。因为多个生成器可以通过将它们的状态混合到各自的序列中来彼此同步，该设计还支持横向扩容。4.4 章节将深入讨论横向扩容。

4.1 说明

系统设计如下。对于一个加密哈希函数（例如 sha256、ripemd 等），如果它不执行运算，那么就无法预测其输出，则从某个随机起始值开始运行该函数，并将其输出作为输入再次传递给同一个函数。同时记录函数被调用的次数以及每次调用时的输出。所选的起始随机值可以是任何字符串，例如《纽约时报》某一天的标题。

例如：

PoH Sequence

Index	Operation	Output Hash
1	sha256(”any random starting value”)	`hash1`
2	sha256(`hash1`)	`hash2`
3	sha256(`hash2`)	`hash3`

其中 hashN 表示实际的哈希输出。

只需要每隔一段时间发布哈希和索引的子集。

例如：

PoH Sequence

Index	Operation	Output Hash
1	sha256("any random starting value")	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

只要选择的哈希函数是抗冲突的，那么这组哈希就只能由一个计算机线程按顺序计算。这是由于：如果不从起始值开始实际运行 300 次算法，就无法预测索引 300 处的哈希值是什么。因此，我们可以从数据结构推断出索引 0 和索引 300 之间的实时时间。

在图 2 中的示例中，哈希 62f51643c1 在计数 510144806912 上生成，哈希 c43d862d88 在计数 510146904064 上生成。根据前面讨论的 PoH 算法的属性，我们可以相信在计数 510144806912 和计数 510146904064 之间传递的实时性。

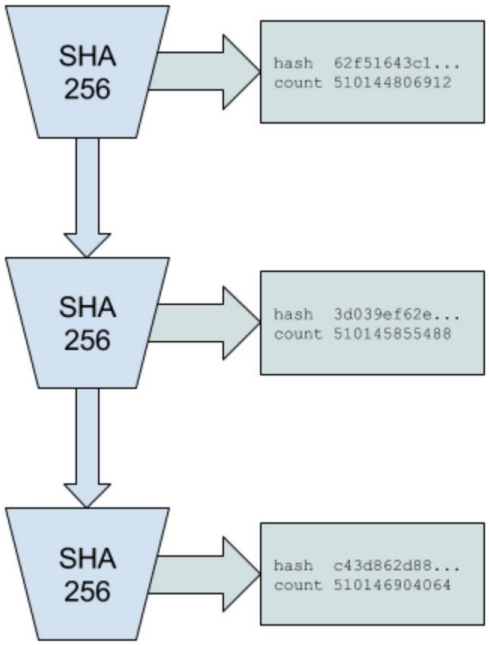


图 2：历史证明序列

4.2 事件的时间戳

该哈希序列还可以用来记录在生成特定哈希索引之前创建的一些数据。通过一个“combine”函数可以将数据段与当前索引处的当前哈希相结合。数据可以是任意事件数据的唯一加密哈希值。combine 函数可以是简单的添加数据，也可以是任何抗冲突的运算。下一个生成的哈希表示数据的时间戳，因为它可能只在插入特定的数据段之后生成。

例如：

PoH Sequence

Index	Operation	Output Hash
1	sha256(”any random starting value”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`

发生了一些外部事件，如拍摄照片或创建任意数字数据：

PoH Sequence With Data

Index	Operation	Output Hash
1	sha256(”any random starting value”)	`hash1`
200	sha256(`hash199`)	`hash200`
300	sha256(`hash299`)	`hash300`
336	sha256(append(`hash335`, photograph_sha256))	`hash336`

Hash336 通过 hash335 的附加二进制数据和照片的 sha256 计算出来。索引和照片的 sha256 作为序列输出的一部分被记录下来。因此，任何验证此序列的人都可以重新创建对序列的更改。验证仍然可以并行进行，详细讨论请见第 4.3 节

因为初始过程仍然是连续的，所以我们可以判断，输入到序列中的内容一定在计算未来哈希值之前发生。

POH Sequence

Index	Operation	Output Hash
1	sha256(”any random starting value”)	hash1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300
336	sha256(append(hash335, photograph1_sha256))	hash336
400	sha256(hash399)	hash400
500	sha256(hash499)	hash500
600	sha256(append(hash599, photograph2_sha256))	hash600
700	sha256(hash699)	hash700

表 1：两个事件的 PoH 序列

在表 1 所示的序列中，photograph2 是在 hash600 之前创建的，photograph1 是在 hash336 之前创建的。将数据插入哈希序列会引起序列中所有后续值的改变。只要所使用的哈希函数是抗冲突的，并且数据是附加上去的，那么基于将要集成到序列中的数据的先验知识，不可能预先计算出任何未来的序列。

混合到序列中的数据可以是原始数据本身，也可以只是数据的哈希以及附带的元数据。

在图 3 中的示例中，输入值 cfd40df8…被插入到历史证明序列中。插入时的计数为 510145855488，插入状态为 3d039eef3。所有未来生成的哈希都会因为这个序列的改变而变化，该变化通过图中的颜色改变变现出来。

观察这个序列的每个节点都可以确定所有事件插入的顺序，并估计插入之间的实时时间。

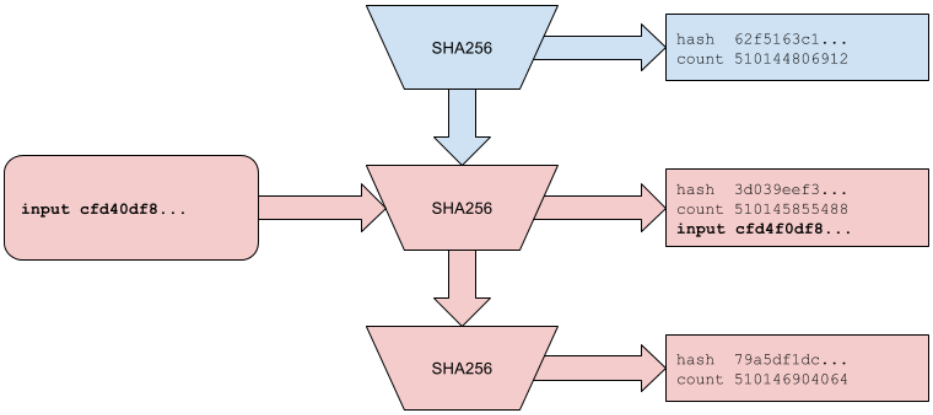


图 3：将数据插入历史证明

4.3 验证

多核计算机可以在比生成序列所需的时间更短的情况下验证序列的正确性。

例如：

Core 1		
Index	Data	Output Hash
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300
Core 2		
Index	Data	Output Hash
300	sha256(hash299)	hash300
400	sha256(hash399)	hash400

对于一定数量的核心（如一个 4000 核的现代 GPU），验证节点可以将哈希

序列及其索引分成4000个片段，同时并行确保每个片段从起始哈希到最后一个哈希都是正确的。如果产生序列的预期时间为：

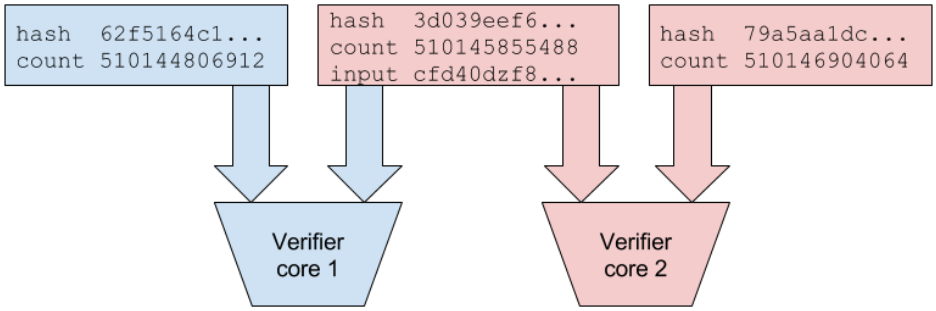


图 4：使用多核进行验证

$$\frac{\text{哈希总数}}{\text{每核每秒哈希数}}$$

验证序列是否正确的预期时间为：

$$\frac{\text{哈希总数}}{\text{（每核每秒的哈希数*可验证的核数）}}$$

在图 4，每个切片的顺序都能够验证。由于所有的输入字符串都被记录到输出中，并且带有附加到它们的计数器和状态，所以验证节点可以并行地复制每个片段。红色哈希表示有一段数据插入了序列，进行了修改。

4.4 横向扩容

通过将每个生成器的序列状态混合到另一个生成器，来同步多个历史证明生成器，能够实现历史证明生成器的横向扩容（并且这种扩容是无需分片）。对于重建系统中事件的完整顺序，两个生成器的输出是必需的。

PoH 生成器 A		
索引	哈希	数据
1	哈希 1a	
2	哈希 2a 哈希 1b	
3	哈希 3a	
4	哈希 4a	

PoH 生成器 B		
索引	哈希	数据
1	哈希 1b	
2	hash2b 哈希 1a	
3	哈希 3b	
4	哈希 4b	

给定生成器 A 和生成器 B，A 从 B 接收一个数据包（哈希 1b），其中包含来自生成器 B 的最后一个状态，以及从生成器 A 观察到的生成器 B 的最后一个状态。然后，生成器 A 中的下一个状态哈希取决于生成器 B 的状态，因此我们可以得出哈希 1b 发生在哈希 3a 之前的某个时间。该属性是可以传递的，因此如果三个生成器通过某一个公共生成器来同步 A↔ B↔ C，我们就能够追踪 A 和 C 之间的依赖关系，即使它们没有直接同步。

通过周期性地同步生成器，每个生成器都能够处理一部分外部通信量，因此由于生成器之间的网络延迟，通过牺牲一小部分的实时准确性整个系统可以处理大量的事件来跟踪。全局顺序则仍然可以通过某个确定性函数（如哈希本身的值）对同步窗口内的任何事件进行排序来实现。

在图 5 中，两个生成器互相插入输出状态并记录操作。颜色变化表明对方的数据序列发生了改变。混合到每个流中的生成哈希以粗体显示。

同步性是可传递的。例如 A ↔ B ↔ C 序列，通过 B， A 和 C 之间存在着一个可证明的事件顺序。

通过这种方式扩展的代价是可用性。可用性为 0.999 的 10×1 gbps 连接将有 $0.999^{10} = 0.99$ 可用性。

4.5 一致性

按照设想，用户能够加强生成序列的一致性，并通过将他们所认定的有效序列的最后一次观察到的输出插入到输入中，从而让其能够抵抗攻击。

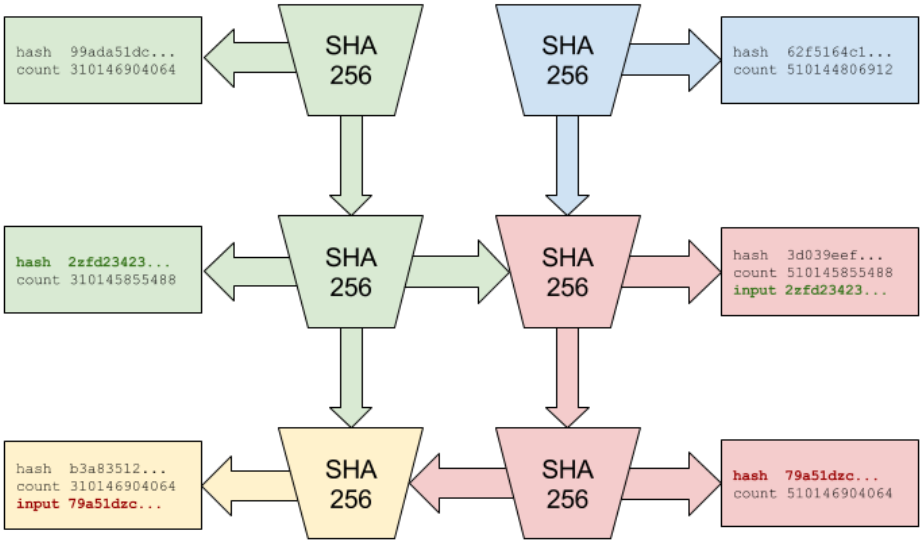


图 5：两个生成器的同步过程

PoH 序列 A		
索引	数据	输出哈希
10		哈希 10a
20	事件 1	哈希 20a
30	事件 2	哈希 30a
40	事件 3	哈希 40a

PoH 隐藏序列 B		
索引	数据	输出哈希
10		哈希 10b
20	事件 3	哈希 20b
30	事件 2	哈希 30b
40	事件 1	哈希 40b

如果某个作恶的 PoH 生成器可以一次过访问所有事件，或者能够生成一个更快的隐藏序列，那么它就可以按事件的相反顺序生成第二个隐藏序列。

为了防止这种攻击，每个客户端生成的事件自身都应该包含客户端观察到的、认为是有效序列的最新哈希。因此，当某个客户端创建“Event1”数据时，他们应该加上其观察到的最后一个哈希。

PoH Sequence A

Index	Data	Output Hash
10		`hash10a`
20	Event1 = append(event1 data, `hash10a`)	`hash20a`
30	Event2 = append(event2 data, `hash20a`)	`hash30a`
40	Event3 = append(event3 data, `hash30a`)	`hash40a`

当序列公布后，Event3 将引用 hash30a，并且如果它不在这个事件之前的序列中，序列的使用者就知道它是一个无效的序列。接着，局部重新排序攻击将会被限制到客户端观察到某个事件以及事件被输入时生成的哈希数。然后，客户端能够编写一个软件，这个软件在最后观察到的哈希和插入的哈希之间的短时间内，不假定交易顺序是正确的。

为了防止恶意 PoH 生成器重写客户端事件哈希，客户端可以提交事件数据和上次观察到的哈希签名（而不仅仅是数据）。

PoH Sequence A

Index	Data	Output Hash
10		hash10a
20	Event1 = sign(append(event1 data, hash10a), Client Private Key)	hash20a
30	Event2 = sign(append(event2 data, hash20a), Client Private Key)	hash30a
40	Event3 = sign(append(event3 data, hash30a), Client Private Key)	hash40a

验证这个数据需要一个签名确认以及在此之前的哈希序列中查找哈希。验证：

```
(Signature, PublicKey, hash30a, event3 data) = Event3
Verify(Signature, PublicKey, Event3)
Lookup(hash30a, PoHSequence)
```

在图 6 中，用户提供的输入依赖于哈希 0xdeadbeef...已经出现在生成序列之前的某个时间。蓝色的

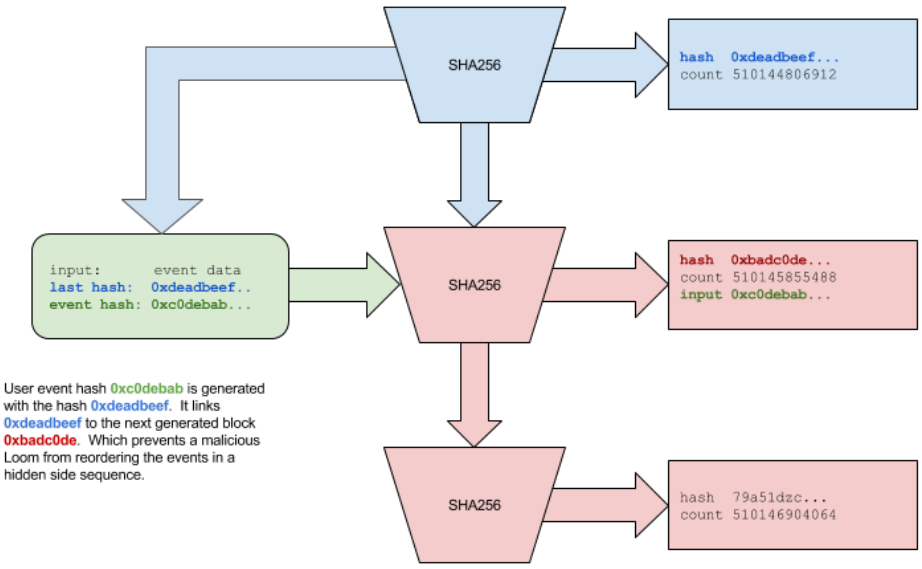


图 6：带有反向引用的输入。

左上箭头指示客户端正在引用以前生成的哈希。客户端消息仅在包含哈希 0xdeadbeef...的序列中有效。序列中的红色表示序列已经经过客户端数据修改。

4.6 开销

每秒 4000 个哈希将产生 160KB 的额外数据，并且需要访问一个 4000 核的 GPU，大约需要 0.25-0.75 毫秒的时间进行验证。

4.7 攻击

4.7.1 逆转

想要生成逆转的序列，攻击者要在第二个事件之后启动恶意序列。这个延迟会让任何非恶意对等节点对原始的序列进行交流。

4.7.2 速度

多个生成器可能会让部署抵抗攻击的能力更强。一个生成器可以是高带宽的，并且可以接收许多事件来混合到它的序列中，另一个生成器可以是高速低带宽的，它定期与高带宽生成器混合。

高速序列将创建一个辅助数据序列，攻击者必须反转该序列。

4.7.3 远程攻击

远程攻击包括获取旧的、丢弃的客户端私钥，并生成伪造账本[10]。历史证明提供了一些防止远程攻击的保护。某个作恶用户想要获得旧私钥的访问权，就必须重新创建一个历史记录，其时间与他们试图伪造的原始记录所用的时间相同。这就要求访问比当前网络更快的处理器，否则攻击者将永远无法赶上历史长度。

此外，单一的时间源能让构建一个复制证明更加简单（更多信息请参见第 6 节）。由于网络的设计使得所有参与者都依赖于单一的历史事件记录。

PoRep 结合 PoH，能够更好地在空间和时间上的抵抗复制账本的攻击。

5 质押一致性证明

5.1 说明

该权益证明的特定实例旨在快速确认由历史证明生成器生成的当前序列，用于投票和选择下一个历史证明生成器，以及惩罚任何不合格的验证程序。这个算法依赖于在一定时间内最终传递到所有参与节点的消息。

5.2 术语

债券（bond） 债券相当于工作证明的资本支出。矿工购买硬件和电力，并将其交付给工作证明区块链中的一个分支。债券是验证节点在验证交易时作为抵押品提交的代币。

罚没（slashing） 权益证明系统中质押问题的解决方案[7]。当一个不同分支的投票证明发布时，该分支会破坏验证节点的债券。这是一种经济激励措施，旨在阻止验证节点确认多个分支。

绝大多数 指超过 2/3 质押债券的验证节点。绝大多数投票表明网络已经达成共识，至少 1/3 的网络恶意投票才能使这个分支无效。这使得攻击的经济成本达到代币市值的 1/3。

5.3 绑定

进行一笔绑定交易首先需要一定数量的代币，然后将其转移到用户身份下的绑定账户。债券账户中的代币无法使用，必须保留在账户中，直到用户将它们取出来。并且用户只能移除某个时间段内的过期代币。债券在当前绝大多数的质押节点确认了顺序之后才生效。

5.4 投票

预计历史证明生成器能够在预定义的时间内发布一个签名。每一个债券身份都必须公布自己签署的声明进行确认。投票只能投赞成票，没有反对票。

如果绝大多数绑定身份在规定时间内投票，则此分支被视为有效。

5.5 解除绑定

缺少 N 轮票数标志着代币过时，它们就失去了投票资格。用户可以发送一笔解除绑定交易移除它们。

N 是一个动态值，基于无效投票与活跃投票的比率。N 随着过期票数的增加而增加。在网络分区较大的情况下，它能让较大分支比较小分支恢复得更快。

5.6 选举

当检测到 PoH 生成器故障后，网络就会选择新的 PoH 生成器。该过程会选择具有最大投票权或最高公钥地址（如果票数相同）的验证节点作为新的 PoH 生成器。

新的序列需要绝大多数确认。如果新的领导者在获得绝大多数确认之前失败，则会选择下一个最大票数的验证程节点，同时启动新一组确认。

如果要切换投票，验证节点需要在更高的 PoH 序列计数器进行投票，而新的投票需要包含它想要切换的投票。否则第二次投票将面临罚没的危险。预计投票转换的设计只会发生在没有达成大多数投票的高度。

一旦生成了 PoH 生成器，就可以选择一个辅助生成器来接管交易处理职责。如果发生意外，则在发生重大故障期间，它将作为下一个领导者。

该平台的设计目的在于，如果检测到异常或按照预定义的时间表，则辅助服务器将成为主服务器，而低级生成器的级别也会得到提升。

5.7 选举触发因素

5.7.1 分叉的历史证明生成器

PoH 生成器的设计有一个生成序列标识。只有在 PoH 生成器标识被破坏的情况下才会发生分叉。如果在同一个 PoH 身份上发布了两个不同的历史记录，就会检测到分叉。

5.7.2 运行异常

硬件故障、错误或 PoH 生成器中的蓄谋错误都可能导致生成无效状态，然后发布与本地验证节点结果不匹配的状态签名。验证节点将通过问询（gossip）发布正确的签名，该事件将引发新一轮选举。任何接受了无效状态的验证节点都要面临着质押罚没的惩罚。

5.7.3 网络超时

网络超时将触发选举。

5.8 罚没

当验证节点对两个独立的序列进行投票时，就会发生罚没。恶意投票被发现后，将会把债券代币移出流通中，并将其添加到一个矿池。

如果某个节点之前对竞争顺序进行了投票，那么它就没有决定恶意投票的资格。这次投票并不会没收债券，只是取消了对目前竞争顺序的投票资格。

如果对 PoH 生成器生成的无效哈希进行了投票，也会导致罚没。生成器将随机生成一个无效状态，这将触发到辅助服务器的回退。

5.9 二次选举

二级和低级别的历史证明生成器可以被提议并批准。可以对一次生成器序列提出提议。提案中有一个暂停时间，如果倡议在暂停前获得过半数票通过，则视为二次当选，并将如期接任职务。主服务器可以通过在生成的序列中插入一条消息，指示将发生切换，或者插入一个无效状态并强制网络回退到辅助服务器，从而实现到辅助服务器的软切换。

如果第二次选举产生，而初选失败，则第二次选举将被视为选举期间的第一次退选。

5.10 可用性

处理分区的 CAP 系统必须选择一致性或可用性。我们的方法最终会选择可用性，但是因为我们有一个客观的时间度量，所以一致性是根据合理的人工超时来选择的。

利益证明验证节点在一个股份中锁定了一定数量的代币，这允许他们投票支持一组特定的交易。锁定代币是一个进入 PoH 流的交易，就像任何其他交易一样。为了投票，PoS 验证节点必须对状态的哈希值进行签名，因为它是在将所有交易处理到 PoH 分类账中的特定位置之后计算出来的。此投票也作为一项交易输入 PoH 流。通过查看 PoH 分类账，我们可以推断每次投票之间经过了多少时间，如果发生分区，每个验证节点都不可用的时间有多长。

为了处理具有合理人工时间框架的分区，我们提出了一种动态方法来取消不可用验证节点。当验证节点的数量大于或等于较高时，解卡过程可以很快。在“不可用验证节点”的股份完全不稳定之前，必须生成到账本中的哈希数很低，而且它们不再被计算为共识。当验证节点的数量低于 rds 但高于 rds 时，取消生成计时器的速度较慢，需要在未停止丢失的验证节点之前生成更大数量的哈希。在一个大的分区中，比如一个缺少或多个验证节点的分区，脱离过程非常缓慢。交易仍然可以进入流中，并且验证节点仍然可以投票，但是直到生成了大量的哈希并且不关闭不可用的验证节点，

否则将无法达成完全的 rds 共识。网络恢复活跃的时间差允许我们作为网络的用户，选择一个我们想继续使用的分区。

5.11 恢复

在我们提出的系统中，账本可以从任何故障中完全恢复。这意味着，世界上任何人都可以在账本中随机选择一个点，并通过附加新生成的哈希和交易来创建有效的 fork。如果所有的验证节点都从这个分支中消失了，那么任何额外的债券都将需要很长的时间才能生效，并且这个分支机构也需要很长的时间才能达成 rds 的超级多数共识。因此，使用零可用验证节点进行完全恢复需要将大量哈希添加到账本中，并且只有在所有不可用的验证节点都被取消之后，任何新的债券才能验证账本。

5.12 终局性

PoH 允许网络的验证节点观察过去发生的事情，并在一定程度上确定这些事件发生的时间。由于 PoH 生成器生成一个消息流，所有验证节点都需要在 500ms 内提交状态签名。这一数字可以根据网络状况进一步减少。由于每个验证都被输入到流中，网络中的每个人都可以验证每个验证节点是否在所需的超时内提交了他们的投票，而无需直接观察投票。

5.13 攻击

5.13.1 公地悲剧

PoS 验证节点只需确认由 PoH 生成器生成的状态哈希。有一种经济激励，让他们不做任何工作，只需批准每个生成的状态哈希。为了避免这种情况，PoH 生成器应该以随机间隔注入一个无效的哈希。任何支持这种做法的选民都应该被大幅削减。当哈希产生时，网络应立即提升二次选择的 PoH 生成器。

每个验证节点都需要在很短的超时时间内响应，例如 500 毫秒。超时应该设置得足够低，以便恶意验证节点观察到另一个验证节点投票并足够快地将他们的投票放入流中的概率很低。

5.13.2 与 PoH 生成器串通

一个与 PoH 生成器串通的验证节点会事先知道什么时候会产生无效的哈希，而不会投票给它。这个场景与拥有更大验证节点的 PoH 身份没有什么不同。PoH 生成器仍然需要完成生成状态哈希的所有工作。

5.13.3 审查制度

当债券持有人拒绝验证新债券的任何序列时，可能会出现审查或拒绝服务的情况。协议可以通过动态调整债券失效的速度来抵御这种形式的攻击。在拒绝服务的情况下，更大的分区将被设计成分叉和审查拜占庭债券持有人。随着时间的推移，拜占庭债券变得陈旧，更大的网络将恢复。较小的拜占庭分区将无法在较长的时间内向前推进。

算法的工作原理如下。网络中的大多数人将选出新的领导者。然后，领导者将审查拜占庭债券持有人的参与。历史证明生成器将不得不继续生成一个序列，以证明时间的流逝，直到足够多的拜占庭债券变得过时，因此更大的网络拥有了绝大多数。债券失效的速度将根据活跃债券的百分比动态变化。因此，网络中的拜占庭少数派分支要比多数派分支等待更长的时间才能恢复超级多数。一旦确立了绝对多数，大幅削减可能被用来永久惩罚拜占庭债券持有人。

5.13.4 远程攻击

PoH 为远程攻击提供了自然防御。从过去的任何时候恢复账本都需要攻击者通过超过 PoH 生成器的速度来及时超过有效的账本。

一致性协议提供了第二层防御，因为任何攻击都需要比解除所有有效验证节点所需时间更长的时间。这也造成了账本历史上的可用性缺口。当比较两个相同高度的矿柱时，最大隔板最小的一个可以客观地认为是有效的。

5.13.5 ASIC 攻击

在这个协议中存在两个 ASIC 攻击的机会-在划分期间，以及在最终阶段作弊超时。

对于分区过程中的 ASIC 攻击，键的未固定速率是非线性的，对于具有大分区的网络，该速率比 ASIC 攻击的预期收益慢几个数量级。

对于最终阶段的 ASIC 攻击，该漏洞允许拥有绑定股份的拜占庭验证节点等待来自其他节点的确认，并将其投票注入协作的 PoH 生成器。然后，PoH 生成器可以使用其更快的 ASIC 在更短的时间内生成 500ms 的哈希值，并允许 PoH 生成器和协作节点之间进行网络通信。但是，如果 PoH 生成器也是拜占庭式的，那么当他们期望插入故障时，没有理由不通知精确的计数器。这个场景与 PoH 生成器和所有的协作者共享同一个身份、拥有一个组合的股份并且只使用一组硬件没有什么不同。

6 流式复制证明

6.1 说明

Filecoin 提出了复制证明的一个版本[6]。此版本的目标是通过在历史证明生成的序列中跟踪时间，来对复制证明进行快速的流式验证。复制不是一种一致的算法，而是一种有用的工具，可以用来解释以高可用性存储区块链历史或状态的成本。

6.2 算法

如图 7 所示，CBC 加密使用先前加密的区块来异或输入数据，按顺序加密每个数据块。

每个复制标识通过签名已生成的历史证明序列的哈希来生成密钥。这将密钥与复制器标识和特定的历史证明序列关联在一起。其中只能选择特定的哈希。（见第 6.5 节哈希选择）

数据集逐块完全加密。然后，为了生成一个证明，密钥被用来生成一个伪随机数生成器，该生成器从每个区块中随机选择 32 个字节。

一个 merkle 哈希是通过在每个切片前面加上选定的 PoH 哈希来计算的。

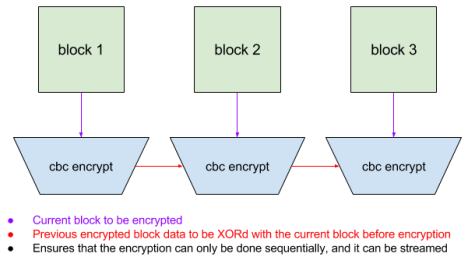


图 7：CBC 加密序列

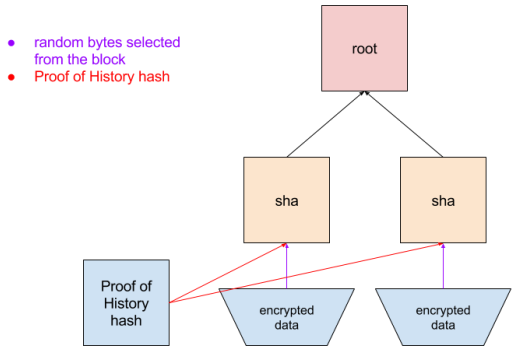


图 8：快速复制证明

根目录、密钥和生成的选定哈希一起发布。复制节点需要在由历史证明生成器生成的 N 个哈希中发布另一个证明，N 大约是加密数据所需的时间。历史证明生成器将在预定义的时间段发布用于复制证明的特定哈希值。复

制器节点必须选择下一个发布的哈希来生成证明。同样，哈希被签名，并从区块中选择随机切片来创建 merkle 根。

经过 N 次证明后，数据将使用新的 CBC 密钥重新加密。

6.3 验证

对于 N 个核心，每个核心可以对每个身份进行流加密。由于上一个加密区块是生成下一个区块必需的，因此所需的总空间为 2blocks*N cores。然后每个核心可以用来生成从当前加密区块派生的所有证明。

验证证明的总时间预计等于加密所需的时间。证明本身从区块中消耗的随机字节非常少，因此哈希的数据量明显低于加密区块的大小。同时，可以验证的复制标识数量等于可用核心的数量。现代的 GPU 有 3500 多个可用的核心，尽管其时钟速度为 CPU 的 ½-⅓。

6.4 密钥轮换

在不经过密钥轮换的情况下，相同的加密复制可以为多个历史证明序列生成廉价的证明。密钥会周期性地轮换，每个复制都用一个新的密钥重新加密，该密钥与一个唯一的历史证明序列关联在一起。

轮换的速度要足够慢，以便在 GPU 硬件上验证复制证明是可行的，它比 CPU 每核的速度更慢。

6.5 哈希选择

历史证明生成器发布一个供整个网络进行加密复制证明的哈希，同时用于快速证明中字节选择的伪随机数生成器。

哈希发布在一个周期性的计数器，该计数器大约等于加密数据集所需的时间。每个复制标识必须使用相同的哈希，并将哈希的签名结果用于字节选择的种子或加密密钥。

每个复制器必须提供证明的周期必须小于加密时间。否则，复制器可以流式传输加密并删除每个证据。

恶意生成器可以在此哈希之前将数据注入序列以生成特定哈希。该攻击方式已经在 5.13.2 章节进行了详细讨论。

6.6 验证证明

历史证明节点无需验证提交的复制证明。它将跟踪复制器身份提交的待决和已核实证据的数量。当复制器能够通过网络中的绝大多数验证节点对证明进行签名时，就可以对证明进行验证。

复制器通过 p2p 八卦网络收集验证信息，作为一个包含网络中绝大多数验证节点的数据包提交。该数据包在历史证明序列生成的特定哈希之前验证所有的证明，并且可以同时包含多个复制器身份。

6.7 攻击

6.7.1 垃圾邮件（Spam）

恶意用户可以创建许多复制器身份，然后用不可靠的证据向网络发送垃圾邮件。为了加快验证速度，节点在请求验证时需要向网络的其余部分提供加密数据和整个 merkle 树。

本文设计的复制证明允许任何附加证明的廉价验证，因为它们不占用额外空间。但是每个身份都会消耗 1 个核心的加密时间。复制目标应设置为可用核心的最大大小。配备 3500+核心的现代 GPU。

6.7.2 部分擦除

复制器节点可以尝试删除部分数据，来避免存储整个状态。证明的数量和种子的随机性会使这种攻击变得困难。

例如，存储 1TB 数据的用户会从每个 1MB 块中删除一个字节。一个单一的证明，从每兆字节中抽取 1 个字节，就有可能与任何被删除的字节 $1–（1−1/10000000）^{1,000,000}=0.63$ 发生冲突。.经过 5 次证明的可能性为 0.99。

6.7.3 与 PoH 生成器串通

签名的哈希应该用于为样本设定种子。如果一个复制器可以预先选择一个特定的哈希值，那么复制器就可以删除所有不需要采样的字节。

与历史证明生成器串通的复制器标识可以在生成用于随机字节选择的预定义哈希之前，在序列末尾注入特定交易。只要有足够的核心，攻击者就可以生成一个比复制器身份更好的哈希。

这种攻击只会受益于一个复制器身份。由于所有标识都必须使用相同的哈希值，而该哈希值是用 ECDSA（或等效的）加密进行签名，因此生成的签名对于每个复制器标识都是唯一的，并且能够应对串通。单一的复制器身份只会具备边际收益。

6.7.4 拒绝服务

添加额外的复制器标识的成本预计等于存储成本。添加额外的计算能力来验证所有复制器标识的成本预计等于每个复制标识的 CPU 或 GPU 核心的成本。

这一点为通过创建大量有效的复制器标识，为网络上的拒绝服务攻击创造了机会。

为了限制这种攻击，为网络选择的一致性协议可以选择一个复制目标，并授予满足所需特性（如网络可用性、带宽、地理位置等）的复制证明...

6.7.5 公地悲剧

PoS 验证节点可以在不做任何工作的情况下简单地对 PoRep 进行确认。经济激励措施应与 PoS 验证节点付出工作相一致，比如在 PoS 验证节点和 PoRep 复制节点之间分离挖矿收益。

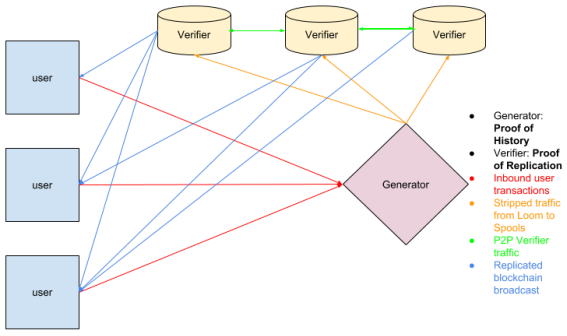


图 9：系统架构

为了进一步避免这种情况，PoRep 验证节点可以在一段很短的时间内提交虚假证明。他们可以通过提供生成虚假数据的函数来证明该验证是假的。任何确认虚假证明的 PoS 验证节点都将面临罚没的惩罚。

7 系统架构

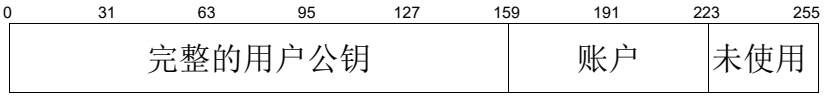
7.1 组件

7.1.1 领导者，历史证明生成器

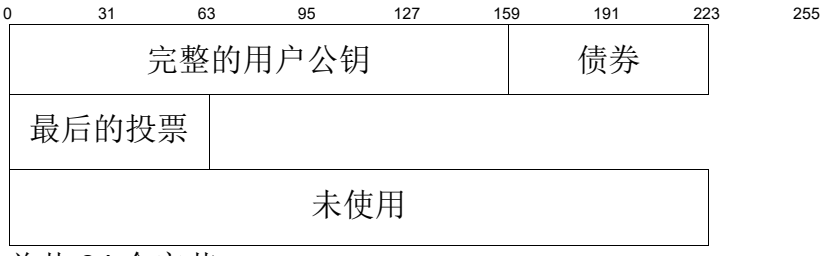
领导者是一个选举产生历史证明的角色。它通过任意用户交易并输出所有交易的历史证明序列，以保证系统中唯一的全局顺序。在每一批交易之后，领导者输出一个状态签名，该签名是按该顺序运行交易的结果。该签名通过领导者的身份进行签名。

7.1.2 状态

按用户地址索引的原始哈希表。每个单元包含完整的用户地址和此计算所需的内存。例如交易表包含：



总共 32 个字节。
质押证明债券表包含：



总共 64 个字节。

7.1.3 验证程序，状态复制

验证程序节点复制区块链状态，并提供区块链状态的高可用性。复制目标由共识算法选择，共识算法中的验证节点根据链下定义的标准选择并投票批准的复制证明节点。

网络可以配置为最小的质押证明债券规模，并要求每个债券只有一个复制器身份。

7.1.4 验证节点

这些节点消耗验证程序的带宽。它们是虚拟节点，可以运行在与验证节点或领导者相同的机器上，也可以运行在为该网络配置的共识算法特定的单独计算机上。

7.2 网络限制

领导者被期望能够接收传入的用户数据包，以最有效的方式对它们进行排序，并将它们排列成历史证明序列，然后发布给下游验证节点。效率由交易内存访问模式决定，因此对交易进行排序以最小化错误并最大化预取。

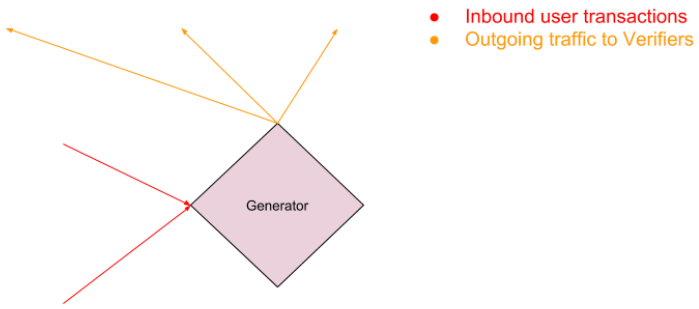
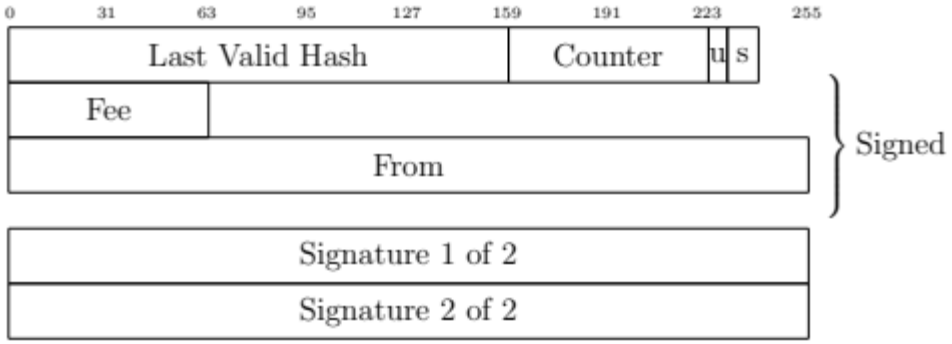


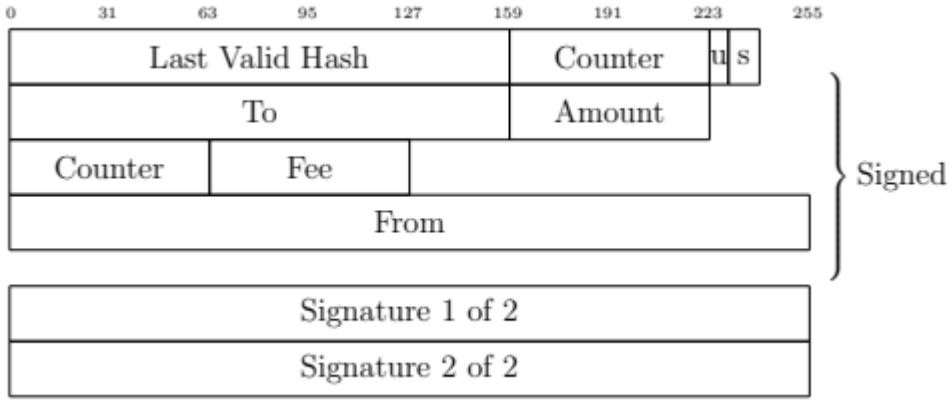
图 10：生成器网络限制

传入数据包格式：



大小为 32+8+32+8+148 字节。

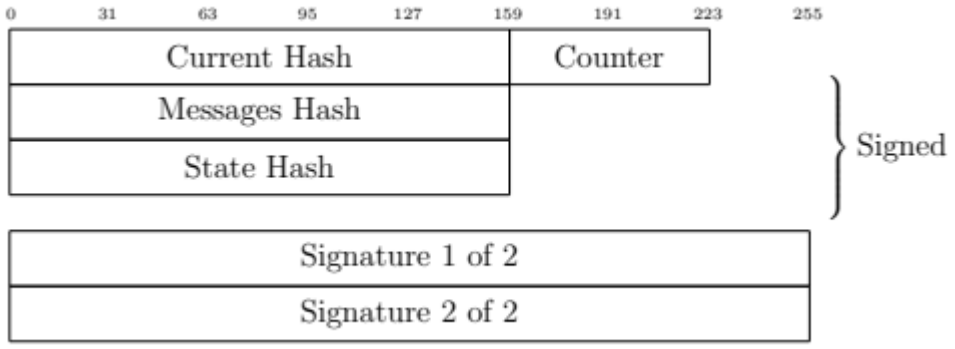
可支持的最小负载为 1 个目标账户。有效载荷：



有效负载的最小大小：176 字节

历史证明序列包包含当前哈希、计数器、添加到 PoH 序列中的所有新消息的哈希以及处理所有消息后的状态签名。每广播 N 条消息，就发送一次该数据包。

历史证明数据包：



输出数据包的最小大小为：132 字节

在 1gbps 网络连接上，最大可能交易数为每秒 1 千兆位/176 字节=最大 710k tps。由于以太网帧，预计会有 1-4%的损失。通过使用 Reed-Solomon 码对输出进行编码并将其分条到可用的下游，可以使用网络超过目标量的空闲容量来提高可用性验证节点。

7.3 计算限制

每个交易都需要摘要验证。此操作不使用交易消息本身之外的任何内存，并且可以独立地并行化。因此，吞吐量将受到系统可用核心数量的限制。

基于 GPU 的 ECDSA 验证服务器的实验结果是每秒 900k 次操作[9]。

7.4 内存限制

一个简单的状态实现是一个 50%完整的哈希表，每个账户有 32 个字节的条目，理论上可以将 100 亿个账户放入 640GB 中。此表的稳态随机访问量为 $1.1*10^7$ 次写入或读取。基于每个交易 2 次读取和两次写入，内存吞吐量每秒可处理 2750 万个交易。这是在亚马逊服务器 1TB x1.16 xlarge 实例上测量的数据。

7.5 高性能智能合约

智能合约是一种通用的交易形式。它们是在每个节点上运行并修改状态的程序。此设计利用扩展的 Berkeley 包过滤字节码作为快速和易于分析的字节码，并利用 JIT 字节码作为智能合约语言。

它的主要优点之一是零成本的外部函数接口。内部函数或直接在平台上实现的函数可由程序调用。调用内部函数会挂起该程序并在高性能服务器上调度内部函数。内部函数被批处理在一起，以便在 GPU 上并行执行。

在上面的例子中，两个不同的用户程序调用相同的内部函数。在批处理执行内部函数之前，每个程序都被挂起完成。ECDSA 验证就是一个例子。在 GPU 上批量执行这些调用可以将吞吐量提高数千倍。

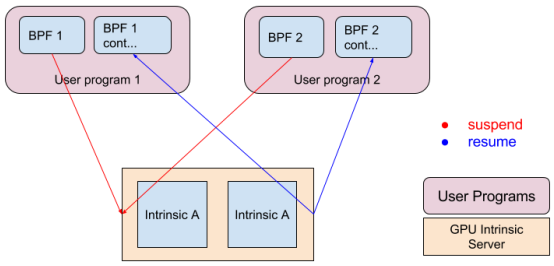


图 11：执行 BPF 程序。

这个流程不需要原生操作系统线程上下文切换，因为 BPF 字节码对它所使用的所有内存都有一个定义良好的背景。

自 2015 年以来，eBPF 后端已经包含在 LLVM 中，因此任何 LLVM 前端语言都可以用来编写智能合约。也是从 2015 年起，它就在 Linux 核心中，字节码的第一次迭代也是从 1992 年开始的。单次通过可以检查 eBPF 的正确性，确定其运行时和内存需求，并将其转换为 x86 指令。

参考文献

[1] Liskov, Practical use of Clocks
http://www.dainf.cefetpr.br/ tacla/SDII/PracticalUseOfClocks.pdf
[2] Google Spanner TrueTime consistency
https://cloud.google.com/spanner/docs/true-time-external-consistency
[3] Solving Agreement with Ordering Oracles
http://www.inf.usi.ch/faculty/pedone/Paper/2002/2002EDCCb.pdf
[4] Tendermint: Consensus without Mining
https://tendermint.com/static/docs/tendermint.pdf
[5] Hedera: A Governing Council & Public Hashgraph Network
https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.0-180313.pdf
[6] Filecoin, proof of replication,
https://filecoin.io/proof-of-replication.pdf
[7] Slasher, A punative Proof of Stake algorithm
https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/
[8] BitShares Delegated Proof of Stake
https://github.com/BitShares/bitshares/wiki/Delegated-Proof-of-Stake
[9] An Efficient Elliptic Curve Cryptography Signature Server With GPU Acceleration
http://ieeexplore.ieee.org/document/7555336/
[10] Casper the Friendly Finality Gadget
https://arxiv.org/pdf/1710.09437.pdf