Solana：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）的新区块链架构--一种用于验证事件之间的顺序和时间流逝的证明。PoH被用来将无信任的时间流逝编码到账本中--一个仅有附加的数据结构。当与工作证明（PoW）或赌注证明（PoS）等共识算法一起使用时，PoH可以减少拜占庭式故障托尔复制状态机中的信息传递开销，导致亚秒级的最终时间。本文还提出了两种利用PoH账本的时间保持特性的算法--一种可以从任何规模的分区中恢复的PoS算法和一种高效的流式复制证明（PoRep）。PoRep和PoH的结合在时间（排序）和存储时间方面提供了对账本伪造的防御。该协议在1gbps的网络上进行了分析，本文显示，在目前的硬件条件下，每秒高达710k的交易吞吐量是可能的。

1 简介

区块链是一个容错复制的状态机的实现。目前公开的区块链并不依赖时间，或者对参与者的计时能力做了一个弱的假设[4, 5]。网络中的每个节点通常依赖于他们自己的本地时钟，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消息的时间戳被用来接受或拒绝一个消息时，不能保证网络中的其他参与者会做出完全相同的选择。这里介绍的PoH旨在创建一个具有可验证的时间流逝的账本，即事件和消息排序之间的时间。预计网络中的每一个节点都将能够依赖账本中记录的时间流逝而无需信任。

2 概要

本文的其余部分组织如下。第3节描述了整体系统设计。第4节描述了历史证明的深度描述。第5节描述了拟议的权益证明共识算法的深度描述。第6节描述了拟议的快速复制证明的深度描述。第7节中分析了系统架构和性能限制。第7.5节描述了一个高性能的GPU友好型智能合约引擎。

3 网络设计

如图1所示，在任何给定的时间，一个系统节点被指定为Leader，生成一个历史证明序列，为网络提供全局的读取一致性和可验证的时间段。Leader对用户信息进行排序，并对它们进行排序，使它们能够被系统中的其他节点有效地处理，最大限度地提高吞吐量。它在存储在RAM中的当前状态上执行交易，并将交易和最终状态的签名公布给称为验证器的复制节点。验证者在他们的状态副本上执行相同的交易，并将他们计算出的状态签名作为确认书发布。公布的确认书作为共识算法的投票。

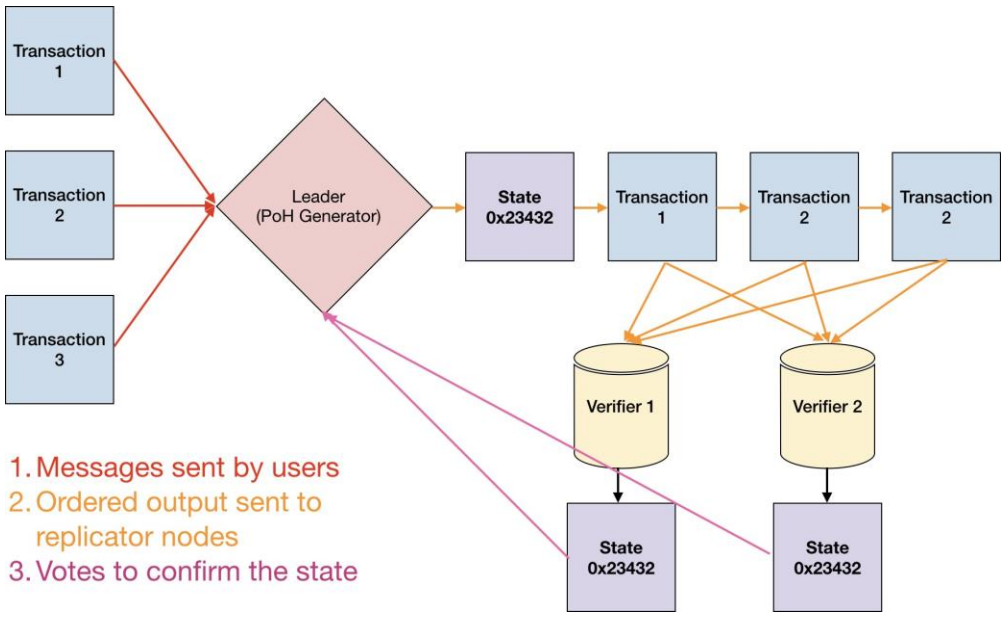


图1： 整个网络的交易流程。

在非分区状态下，在任何时候，网络中都有一个领导者。每个验证者节点具有与领导者相同的硬件能力，可以被选为领导者，这是通过基于PoS的选举完成的。第5.6节将深入介绍拟议的PoS算法的选举。就CAP定理而言，在分区的情况下，一致性几乎总是比可用性更重要。在大分区的情况下，本文提出了一种机制来恢复对网络的控制。任何尺寸。这一点将在第5.12节深入介绍。

4 历史的证明

历史证明是一个计算序列，可以提供一种方法来加密验证两个事件之间的时间流逝。它使用一个加密安全的函数，使输出不能从输入中预测，并且必须完全执行以产生输出。该函数在单个核心上依次运行，其先前的输出为

当前的输入，周期性地记录当前的输出，以及它被调用了多少次。然后，外部计算机可以通过在一个单独的核心上检查每个序列段来重新计算和验证输出。通过将数据（或一些数据的哈希值）附加到函数的状态中，可以将数据的时间标记到这个序列中。记录状态、索引和数据，因为它被附加到序列中，提供了一个时间戳，可以保证数据是在序列中生成下一个哈希值之前的某个时间创建的。这种设计也支持横向扩展，因为多个生成器可以通过将它们的状态混合到彼此的序列中来实现彼此之间的同步。横向扩展的问题在

第4.4节

4.1 描述

该系统被设计成如下工作方式。用一个加密的哈希函数，其输出在不运行该函数的情况下是无法预测的（如sha256，ripemd等），从一些随机的起始值开始运行该函数，并将其输出作为输入再次传入同一函数。记录该函数被调用的次数和每次调用的输出。选择的随机起始值可以是任何字符串，比如当天的《纽约时报》的头条。

比如说。

宝鸡序列

索引	运作	输出哈希值
1	sha256("任何随机起始值")	哈希1
2	sha256(hash1)	哈希2
3	sha256(hash2)	哈希3

其中hashN代表实际的哈希输出。

只需要在一个时间间隔内公布哈希值和指数的子集。

比如说。

宝鸡序列

索引	运作	输出哈希值
1	sha256("任何随机起始值")	哈希1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

只要选择的哈希函数是抗碰撞的，这组哈希值只能由一个计算机线程依次计算。这源于这样一个事实：如果不从起始值开始实际运行算法300次，就没有办法预测索引300处的哈希值会是什么。因此，我们可以从数据结构中推断出，在索引0和索引300之间已经有了真实的时间。

在图2的例子中，哈希62f51643c1产生于计数510144806912，哈希c43d862d88产生于计数510146904064。根据之前讨论的PoH算法的特性，我们可以相信在计数510144806912和计数510146904064之间有真实时间。

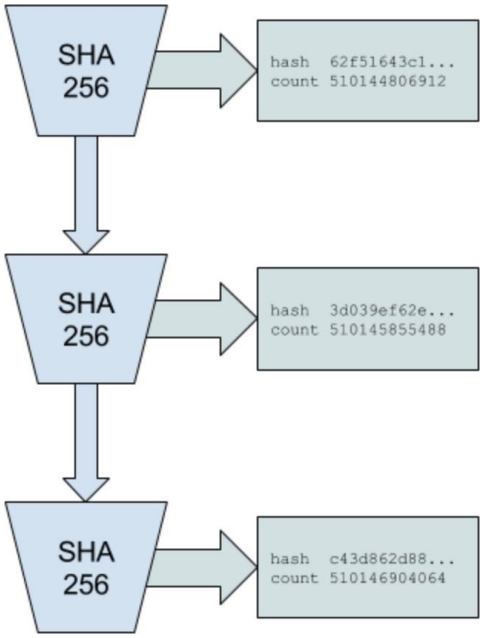


图2： 历史证明序列

4.2 事件的时间戳

这个哈希值序列也可以用来记录某些数据是在某一哈希值索引产生之前创建的。使用 "组合 "函数将这块数据与当前索引的哈希值相结合。该数据可以是任意事件数据的加密的唯一哈希值。组合函数可以是一个简单的数据附加，或任何抗碰撞的操作。下一个生成的哈希值代表了数据的时间戳，因为它只可能在该特定数据被插入后生成。

比如说。

宝鸡序列

索引	操作	输出哈希值
1	sha256("任何随机起始值")	哈希1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

一些外部事件发生了，比如拍摄了一张照片，或者创造了任何任意的数字数据。

有数据的PoH序列

索引	运作	输出哈希值
1	sha256("任何随机起始值")	哈希1
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300
336	sha256(append(hash335, photograph sha256))	hash336

哈希336是由哈希335的附加二进制数据和照片的sha256计算出来的。照片的索引和sha256被记录为序列输出的一部分。所以任何验证这个序列的人都可以重新创建这个序列的变化。验证仍然可以并行进行，其讨论在第4.3节。

因为初始过程仍然是有顺序的，所以我们就可以知道，进入序列的事情一定是在未来的哈希值被计算之前的某个时间发生的。

POH序列

索引	运作	输出哈希值
1	sha256("任何随机起始值")	哈希1
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： PoH序列与2个事件

在表1所代表的序列中，照片2是在哈希600之前创建的，而照片1是在哈希336之前创建的。将数据插入哈希值序列会导致序列中所有后续数值的改变。只要使用的哈希函数是抗碰撞的，而且数据是附加的，那么在计算上就不可能根据事先知道什么数据将被纳入序列而预先计算任何未来的序列。

混入序列的数据可以是原始数据本身，也可以是带有附带元数据的数据散列。

在图3的例子中，输入cfd40df8...被插入到历史证明序列中。它被插入的计数是510145855488，它被插入的状态是3d039eef3。所有未来生成的哈希值都被这个序列的变化所修改，这个变化在图中用颜色的变化表示。

观察这个序列的每个节点可以确定所有事件被插入的顺序，并估计插入之间的实际时间。

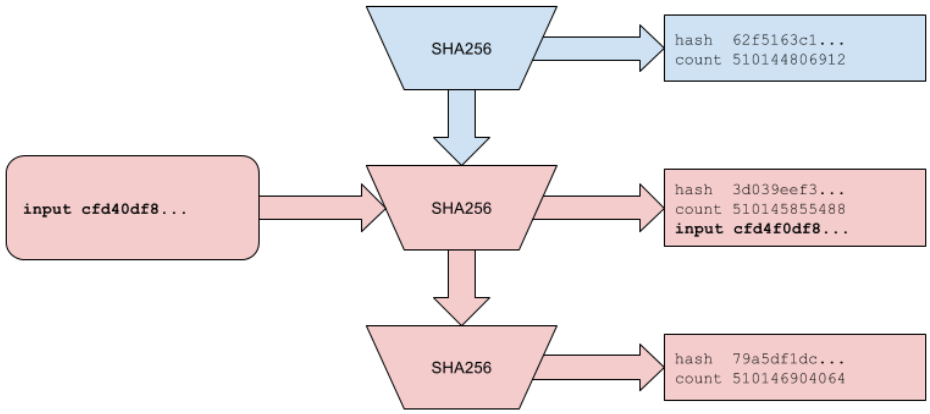


图3： 将数据插入历史证明中

4.3 验证

多核计算机可以在比生成它的时间少得多的时间内验证该序列的正确性。

比如说。

核心1

索引	数据	输出哈希值
200	sha256(hash199)	hash200
300	sha256(hash299)	hash300

核心2

索引	数据	输出哈希值
300	sha256(hash299)	hash300
400	sha256(hash399)	hash400

给定一定数量的内核，比如现代GPU有4000个内核，验证器可以将哈希序列和它们的索引分成4000个片断，并并行地确保每个片断从开始的哈希到片断中的最后一个哈希都是正确的。如果产生该序列的预期时间将是。

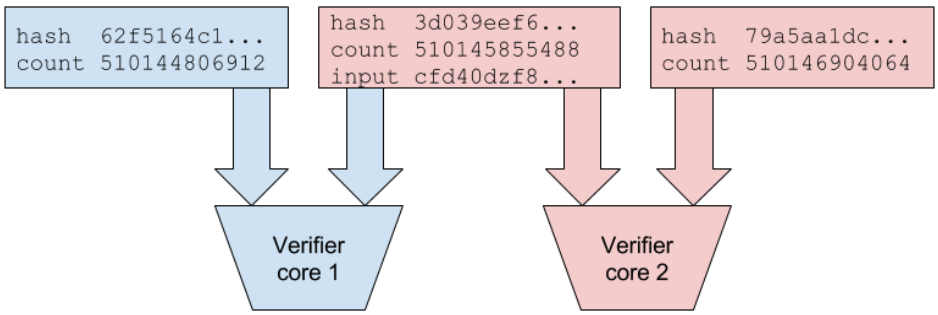


图4： 使用多个核心进行验证

$$\frac{\text{哈希值总数 1个核心每秒的哈希值}}{}$$

验证序列是否正确的预期时间将是。

$$\frac{\text{哈希值的总数}}{\text{(每个核心每秒的哈希值 * 可供验证的核心数量)}}$$

在图4的例子中，每个核都能够并行地验证序列的每个片断。由于所有的输入字符串都被记录到输出中，并附有计数器和状态，验证器可以并行地复制每个片断。红色的哈希值表示序列被数据插入所修改。

4.4 水平缩放

它可以通过将每个生成器的序列状态混合到其他生成器来同步多个历史证明生成器，从而实现历史证明生成器的水平扩展。这种扩展是在没有分片的情况下完成的。两个生成器的输出对于重建系统中事件的完整顺序是必要的。

PoH发电机A		
索引	哈希	数据
1	哈希1a	
2	哈希2a	hash1b
3	哈希3a	
4	hash4a	

波赫发生器B		
索引	哈希	数据
1	hash1b	
2	哈希2b	哈希1a
3	哈希3b	
4	hash4b	

给定生成器A和B，A从B收到一个数据包（hash1b），其中包含生成器B的最后一个状态，以及生成器B从生成器A观察到的最后一个状态。生成器A的下一个状态hash取决于生成器B的状态，所以我们可以推导出hash1b发生在hash3a之前的某个时间。这个属性可以是传递性的，所以如果三个发生器通过一个共同的发生器$A \leftrightarrows B \leftrightarrows C$进行同步，我们可以追踪A和C之间的依赖关系，即使它们没有直接同步。

通过定期同步生成器，每个生成器可以处理一部分外部流量，因此整个系统可以处理更多的事件追踪，但由于生成器之间的网络延迟，真正的时间精度是有代价的。通过选择一些确定的函数来对同步窗口内的任何事件进行排序，例如通过哈希值本身，仍然可以实现一个全局的顺序。

在图5中，两个生成器互相插入对方的输出状态并记录操作。颜色的变化表明，来自对等体的数据修改了序列。混合到每个数据流中的生成的哈希值以黑体字突出显示。

该同步是反式的。$A \leftrightarrows B \leftrightarrows C$ 在A和C之间有一个可证明的事件顺序，通过B。

以这种方式扩展是以可用性为代价的。10×1gbps连接的可用性为0.999，将有$0.999^{10} = 0.99$的可用性。

4.5 一致性

预计用户能够强制执行生成序列的一致性，并通过将他们认为有效的序列的最后观察到的输出插入到他们的输入中，使其能够抵抗攻击。

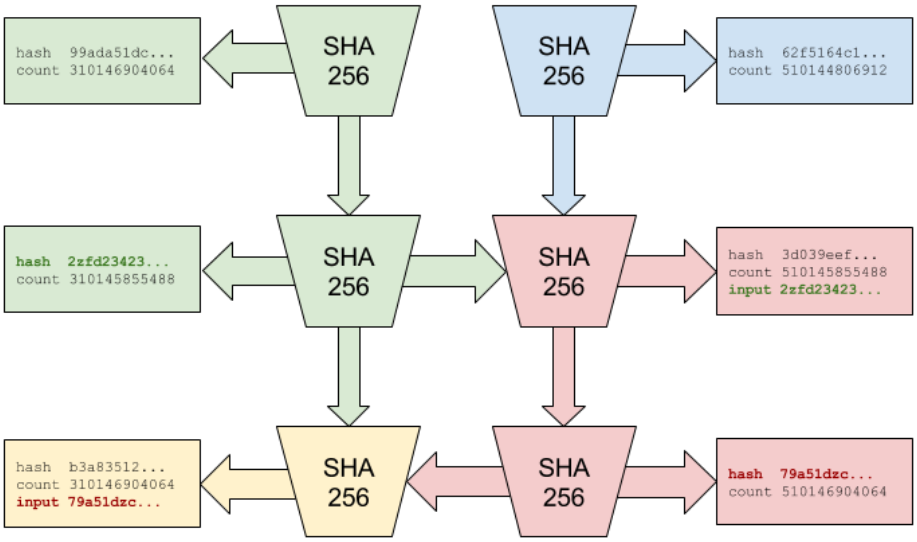


图5： 两台发电机同步运行

波赫序列A		
索引	数据	输出哈希值
10		哈希10a
20	事件1	哈希20a
30	事件2	hash30a
40	活动3	hash40a

坡赫索引	隐藏的数据	序列B输出哈希值
10		hash10b
20	活动3	哈希20b
30	活动2	hash30b
40	事件1	hash40b

恶意的PoH生成器如果能够同时获得所有的事件，或者能够生成一个更快的隐藏序列，就可以产生第二个事件的反向顺序的隐藏序列。

为了防止这种攻击，每个客户端生成的事件本身应该包含客户端从它认为有效的序列中观察到的最新哈希值。因此，当客户端创建 "Event1 "数据时，他们应该附加上他们观察到的最后一个哈希值。

波赫序列A

索引	数据	输出哈希值
10		哈希10a
20	Event1 = append(event1 data, hash10a)	hash20a
30	Event2 = append(event2 data, hash20a)	hash30a
40	Event3 = append(event3 data, hash30a)	hash40a

当序列被发布时，Event3将引用hash30a，如果它在这个事件之前不在序列中，那么序列的消费者就知道它是一个无效的序列。然后，部分重排攻击将被限制在客户端观察事件时产生的哈希值的数量，以及事件被输入的时间。然后，客户应该能够编写软件，在最后观察到的和插入的哈希值之间的短时期内，不认为顺序是正确的。

为了防止恶意的PoH生成器重写客户端的事件哈希值，客户端可以提交事件数据的签名和最后观察到的哈希值，而不仅仅是数据。

波赫序列A

索引	数据	输出哈希值
10		哈希10a
20	Event1 = sign(append(event1 data, hash10a), 客户端私钥)	哈 希
30	Event2 = sign(append(event2 data, hash20a), 客户端私钥)	20A 哈
40	Event3 = sign(append(event3 data, hash30a), 客户端私钥)	希 30A
		哈 希
		40A

对这一数据的验证需要进行签名验证，并在这之前的哈希值序列中查找哈希值。

核实。

(签名，公钥，hash30a，事件3数据）=事件3 验证（签名，公钥，事件3）。

查找(hash30a, PoHSequence)

在图6中，用户提供的输入取决于哈希0xdeadbeef...在其插入前的某个时间，在生成的序列中存在。蓝色的

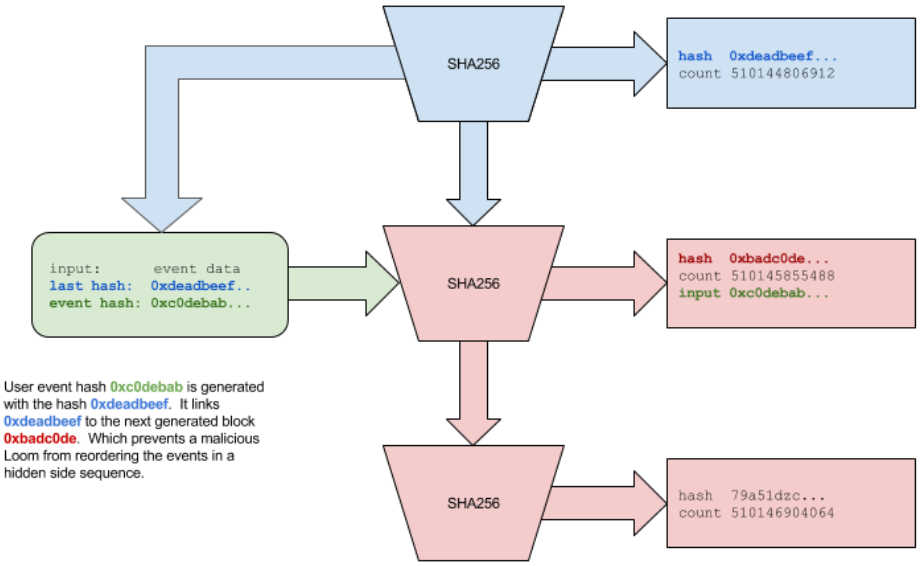


图6： 输入，有一个反向参考。

左上角的箭头表示客户正在引用一个先前产生的哈希值。客户端信息只在包含哈希值0xdeadbeef....的序列中有效。序列中的红色表示该序列已经被客户的数据所修改。

4.6 俯卧撑

每秒4000个哈希值将产生额外的160千字节的数据，并需要访问具有4000个内核的GPU和大约0.25-0.75毫秒的时间来验证。

4.7 攻击

4.7.1 反转

生成一个相反的顺序将需要攻击者在第二个事件后开始恶意的序列。这个延迟应该允许任何非恶意的对等节点对原始订单进行沟通。

4.7.2 速度

拥有多个发生器可能会使部署对攻击更有抵抗力。一个发生器可以是高带宽的，并接收许多事件以混合到它的序列中，另一个发生器可以是高速低带宽的，定期与高带宽发生器混合。

高速序列将产生一个攻击者必须扭转的二级数据序列。

4.7.3 远距离攻击

长距离攻击涉及获取旧的被丢弃的客户私钥，并产生一个伪造的账本[10]。历史证明提供了对长距离攻击的一些保护。获得旧私钥的恶意用户将不得不重新创建一个历史记录，该记录所需的时间与他们试图伪造的原始记录一样多。这需要使用比网络目前使用的更快的处理器，否则攻击者在历史长度上将永远赶不上。

此外，单一的时间来源允许构建一个更简单的复制证明（关于这一点，将在第6节详细介绍）。由于网络被去掉了签名，所以网络中的所有参与者都将依赖单一的历史事件记录。

PoRep和PoH一起应该提供一个空间和时间的防御，以防止伪造的分类帐。

5 桩证共识的证明

5.1 描述

赌注证明的这个具体实例是为了快速确认历史证明生成器产生的当前序列，用于投票和选择下一个历史证明生成器，以及惩罚任何行为不当的验证者。这种算法取决于消息最终在一定的超时时间内到达所有参与节点。

5.2 术语

债券 债券相当于工作证明中的资本支出。矿工购买硬件和电力，并将其投入到工作证明区块链的一个分支中。债券是验证者在验证交易时承诺作为抵押的硬币。

slashing 针对权益证明系统中的一无所有问题提出的解决方案[7]。当一个不同分支的投票证明被公布时，该分支可以破坏验证者的债券。这是一种经济激励措施，旨在阻止验证者确认多个分支。

超级多数 超级多数是指$\frac{2}{3}$ rds的验证者按其债券加权。超级多数票表明网络已经达成共识，至少有$\frac{1}{3}$ rd的网络必须恶意投票，这个分支才会无效。这将使攻击的经济成本达到该币市值的$\frac{1}{3}$ rd。

5.3 粘合

绑定交易需要一定数量的硬币并将其转移到用户身份下的绑定账户。保税账户中的硬币不能被花费，必须留在账户中，直到用户将其删除。用户只能删除已经超时的陈旧硬币。债券在当前利益相关者的超多数确认序列后有效。

5.4 投票表决

预计历史证明生成器将能够在一个预定的时期发布国家的签名。每个绑定的身份必须通过发布他们自己签署的状态签名来确认该签名。投票是一个简单的赞成票，没有反对票。

如果超多数的绑定身份在超时内投票，那么这个分支将被接受为有效。

5.5 解除束缚

缺少N个票数，则标志着该币已过期，不再有资格投票。用户可以发布一个解除绑定的交易来删除它们。

N是一个动态值，基于陈旧票与活跃票的比例。N随着陈旧票数的增加而增加。在发生大型网络分区的情况下，这允许较大的分支比较小的分支恢复得快。

5.6 选举

当检测到PoH生成器故障时，将进行新PoH生成器的选举。拥有最大投票权的验证者，或者在有平局的情况下拥有最高公钥地址的验证者被选为新的PoH发生器。

在新的序列上需要有超级多数的确认。如果在超级多数确认之前，新的领导者失败了，那么就会选择下一个最高的验证者，并且需要一组新的确认。要切换投票，验证人需要在更高的PoH序列计数器上投票，而新的投票需要包含它想要切换的投票。否则，第二张票将被砍掉。票数转换预计将被设计成只能发生在没有超级大多数人。

一旦一个PoH发生器建立起来，就可以选出一个次级发生器来接管交易处理的职责。如果存在一个二级机构，它将被视为一级机构故障时的下一个领导者。

该平台的设计是这样的：如果检测到异常或在预先定义的时间表上，二级变成一级，较低等级的发电机被提升。

5.7 选举的触发因素

5.7.1 分叉的历史证明生成器

PoH生成器被设计为有一个身份来签署生成的序列。只有在PoH生成器的身份被泄露的情况下才会发生分叉。检测到分叉是因为同一PoH身份上发布了两条不同的历史记录。

5.7.2 运行时异常

硬件故障或错误，或者PoH生成器的故意错误，都可能导致它生成一个无效的状态，并发布一个与本地验证者结果不一致的状态签名。验证者将通过流言蜚语发布正确的签名，这一事件将触发新一轮的选举。任何接受无效状态的验证者将被砍掉他们的债券。

5.7.3 网络超时

网络超时将触发一次选举。

5.8 切割

当一个验证者对两个独立的序列进行投票时，就会发生割裂。恶意投票的证明者会将绑定的硬币从流通中移除，并将其添加到矿池中。

包括之前对竞争序列的投票，不符合恶意投票的证明。这种投票不是砍掉债券，而是移除对争夺序列的当前投票。

如果对PoH生成器生成的无效散列值进行投票，也会发生削权。预计生成器会随机生成一个无效的状态，这将触发回落到二级状态。

5.9 二级选举

二级和低级的历史证明生成器可以被提议和批准。一个提案被投在主要生成者序列上。该提案包含一个超时，如果该动议在超时前被超多数票通过，则二级发生器被视为当选，并将按计划接管职责。主网可以通过在生成的序列中插入一个消息，表明将发生交接，或插入一个无效的状态，迫使网络回退到副网，从而向副网进行软交接。

如果一个二级学院被选出，而一级学院失败，二级学院将被视为选举期间的第一后备。

5.10 可利用性

处理分区的CAP系统必须要选择一致性或可用性。我们的方法最终选择了可用性，但是因为我们有一个客观的时间衡量标准，所以在合理的人为超时的情况下，一致性被选中。

股权证明验证人在股权中锁定了一定数量的硬币，这使得他们可以为一组特定的交易投票。锁定币是一种交易，它被输入到PoH流中，就像其他交易一样。为了投票，PoS验证者必须签署状态的哈希值，因为它是在处理所有交易到PoH账本的特定位置后计算出来的。这个投票也会作为交易进入PoH流。看一下PoH账本，我们就可以推断出每次投票之间经过了多少时间，如果发生了分区，每个验证人有多长时间无法使用。

为了处理具有合理的人类时间框架的分区，我们提出了一种动态的方法来解除不可用的验证者的身份。当验证人的数量较多且高于$\frac{2}{3}$，解锁过程就会很快。在不可用的验证者股权完全解除抵押之前，必须生成到账本中的哈希值的数量很低，他们不再被计入共识。当验证器的数量低于$\frac{2}{3}$ rds，但高于$\frac{1}{3}$，解锁计时器就会被激活。速度较慢，需要生成更多的哈希值，然后才会有缺失的核查员被解锁。在一个大的分区中，比如一个缺少$\frac{1}{3}$或更多验证者的分区，解锁过程是非常非常缓慢的。交易仍然可以被输入到数据流中，验证者仍然可以投票，但直到产生了非常多的哈希值和不可用的验证者被解除锁定，才会达成完全的$\frac{2}{3}$ rds共识。网络恢复活泼性的时间差异使我们作为网络的客户可以在人类的时间范围内挑选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，世界上任何一个人都可以在账本中随机挑选任何一个点，通过添加新生成的哈希值和交易来创建一个有效的分叉。如果这个分叉中缺少所有的验证者，那么任何额外的债券都需要非常非常长的时间才能变得有效，并且这个分支要达到$\frac{2}{3}$ rds超级多数共识。因此，在可用验证人为零的情况下，完全恢复需要将大量的哈希值追加到账本上。

而只有在所有不可用的验证器被解密后，任何新的债券才能够验证分类账。

5.12 终结性

PoH允许网络的验证者观察过去发生的事情，并对这些事件的时间有一定程度的把握。由于PoH发生器产生的是一个消息流，所有的验证者都需要在500ms内提交他们对状态的签名。根据网络条件，这个数字可以进一步减少。由于每个验证都被输入到信息流中，网络中的每个人都可以验证每个验证者是否在规定的超时时间内提交了他们的投票，而不需要实际直接观察投票情况。

5.13 攻击

5.13.1 下议院的悲剧

PoS验证器只是确认PoH生成器所生成的状态哈希值。在经济上，他们有动力不做任何工作，只是简单地批准每个生成的状态散列。为了避免这种情况，PoH生成者应该在一个随机的时间间隔内注入一个无效的哈希值。这个哈希值的任何选民都应该被砍掉。当哈希值生成后，网络应该立即促进二次选举的PoH生成器。

每个验证者都需要在一个小的超时内做出反应--比如说500ms。该超时应设置得足够低，以使恶意验证者观察到另一个验证者的投票并将其投票快速输入数据流的概率很低。

5.13.2 与PoH发生器勾结

与PoH生成器串通的验证者会事先知道何时会产生无效的哈希值，而不会投票给它。这种情况实际上与PoH身份拥有更大的验证者权益没有什么不同。PoH生成器仍然需要做所有的工作来生成状态哈希值。

5.13.3 审查制度

当[1] rd的债券持有人拒绝验证任何带有新债券的序列时，审查或拒绝服务就会发生。该协议可以通过动态调整债券变质的速度来防御这种形式的攻击。在拒绝服务的情况下，更大的分区将被设计为分叉和审查拜占庭债券持有人。随着拜占庭债券随着时间的推移变得陈旧，更大的网络将重新覆盖。较小的拜占庭分区将无法在较长的时间内向前推进。

该算法的工作方式如下。网络中的大多数人将选出一个新的领导者。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，直到有足够多的拜占庭债券变得陈旧，使更大的网络拥有超级多数。债券变旧的速度将动态地基于债券的活跃比例。因此，网络的拜占庭少数分叉将不得不比多数分叉等待更长的时间来恢复超级多数。一旦建立了超级多数，就可以用削价来永久地惩罚拜占庭债券持有者。

5.13.4 远距离攻击

PoH对远距离攻击提供了天然的防御。从过去的任何一点恢复账本，都需要攻击者通过超越PoH生成器的速度及时超越有效账本。

共识协议提供了第二层防御，因为任何攻击都必须花费比解除所有验证者身份的时间更长的时间。它还在账本的历史上创造了一个可用性缺口。当比较两个相同高度的账本时，具有最小最大分区的账本可以被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--在分段期间，以及在Finality中作弊超时。

对于分区期间的ASIC攻击，债券解密的速度是非线性的，对于具有大分区的网络，该速度要比ASIC攻击的预期收益慢几个数量级。

对于Finality期间的ASIC攻击，该漏洞允许拥有绑定股权的拜占庭验证者等待其他节点的确认，并向合作的PoH生成器注入他们的投票。然后，PoH生成器可以使用其更快的ASIC在更短的时间内生成价值500ms的哈希，并允许PoH生成器和合作节点之间进行网络通信。但是，如果PoH生成器也是杂乱无章的，那么杂乱无章的生成器没有理由不在他们预计插入故障时通报准确的计数器。这种情况与PoH生成器和所有合作者共享相同的身份，拥有单一的联合股权，并且只使用1套硬件没有什么不同。

6 流动复制的证明

6.1 描述

Filecoin提出了一个复制证明的版本[6]。这个版本的目标是对复制证明进行快速和流式验证，通过跟踪历史证明产生的序列中的时间来实现。复制并不是作为一种共识算法，而是一种有用的工具，用于说明在高可用性下存储区块链历史或状态的成本。

6.2 算法

如图7所示，CBC加密对每个数据块进行顺序加密，使用之前加密的数据块对输入数据进行XOR。

每个复制身份通过签署已生成的历史证明序列的哈希值来产生一个密钥。这将密钥与复制者的身份和特定的历史证明序列联系在一起。只有特定的哈希值可以被选择。(见第6.5节哈希值的选择）。

该数据集被逐块完全加密。然后，为了生成证明，密钥被用来作为伪随机数发生器的种子，从每个块中随机选择32个字节。

计算出一个Merkle哈希值，并将选定的PoH哈希值预加到每个片断中。

根被发布，同时发布的还有密钥，以及生成的选定哈希值。 复制节点需要发布另一个证明

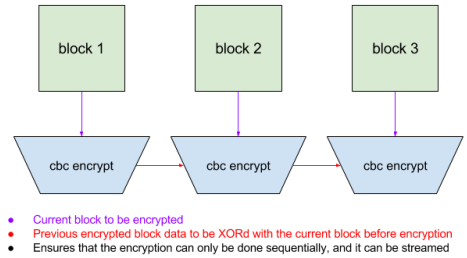


图7： 序列CBC加密

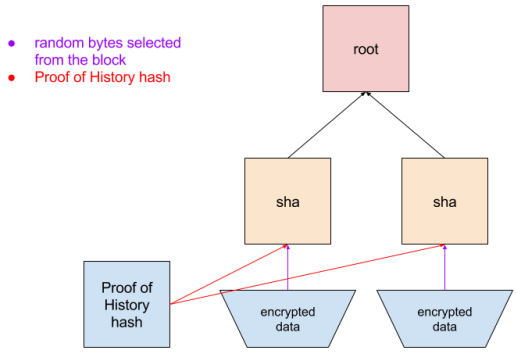


图8：快速复制证明

在N个哈希值中，因为它们是由历史生成器的证明产生的，其中N约为 $\frac{1}{2}$ 的时间来加密数据。 证据的证明历史生成器将在预定的时间段内为复制证明发布特定的哈希值。复制者节点必须选择下一个公布的哈希值来生成证明。同样，哈希值被签名，并从区块中选择随机的片断来创建Merkle根。

经过N次证明后，用新的CBC密钥对数据进行重新加密。

6.3 验证

有了N个核，每个核可以对每个身份进行流式加密。需要的总空间是2blocks $*$ *Ncores*，因为前一个加密块是生成下一个加密块所必需的。然后，每个核心可以用来生成所有从当前加密块衍生出来的证明。

验证证明的总时间预计与加密的时间相等。证明本身从区块中消耗了很少的随机字节，所以需要哈希的数据量大大低于加密区块的大小。可以同时验证的复制身份的数量与可用的内核数量相等。现代GPU有3500多个
他们可用的核心，尽管是在 $\frac{1}{2}$ - $\frac{1}{3}$ CPU的时钟速度。

6.4 按键旋转

如果没有密钥轮换，同一个加密的复制可以为多个历史证明序列产生廉价的证明。密钥是定期轮换的，每次复制都用新的密钥重新加密，该密钥与唯一的历史证明序列挂钩。

旋转的速度需要足够慢，以便在GPU硬件上验证复制证明，而GPU的每核速度比CPU慢。

6.5 杂烩选择

历史证明生成器发布一个哈希值，供整个网络用于加密复制证明，并作为快速证明中的字节选择的伪随机数生成器使用。

哈希值在一个周期性的计数器上发布，该计数器大致等于 $\frac{1}{2}$ ，即加密数据集的时间。每个复制身份必须使用相同的

哈希值，并使用哈希值的签名结果作为字节选择的种子，或加密密钥。

每个复制者必须提供证明的周期必须小于加密时间。否则，复制者可以对加密进行流式处理，并对每个证明进行删除。

恶意的生成器可以在这个哈希值之前的序列中注入数据以生成一个特定的哈希值。这种攻击将在5.13.2中进一步讨论。

6.6 证明验证

历史证明节点预计不会验证提交的复制证明。它被期望跟踪由复制者身份提交的未决和已验证证明的数量。当复制者能够由网络中的超级多数验证者签署证明时，预计证明将得到验证。

验证由复制者通过p2p八卦网络收集，并作为一个数据包提交，其中包含网络中的超级多数验证者。这个数据包验证了由历史证明序列生成的特定哈希值之前的所有证明，并且可以同时包含多个复制者的身份。

6.7 攻击

6.7.1 垃圾邮件

一个恶意的用户可以创建许多复制者的身份，用坏的证明来扰乱网络工作。为了促进更快的验证，要求节点在请求验证时向网络的其他部分提供加密数据和整个梅克尔树。

本文设计的复制证明可以廉价验证任何额外的证明，因为它们不占用额外的空间。但每个身份将消耗1个核心的加密时间。复制目标应设置为现成的核心的最大规模。现代GPU出厂时有3500多个内核。

6.7.2 部分擦除

一个复制者节点可以尝试部分擦除一些数据，以避免存储整个状态。证明的数量和随机性的

种子应该使这次攻击变得困难。

例如，一个存储1兆字节数据的用户从每个1兆字节的块中擦除一个字节。一个从每兆字节中取样1个字节的单一证明，与任何被擦除的字节发生碰撞的可能性是1-（1-*1*，000，0000）^{1,000,000} = 0.63。经过5次证明，可能性为0.99。

6.7.3 与PoH发生器勾结

签名的哈希值预计将被用来作为样本的种子。如果复制者可以提前选择一个特定的哈希值，那么复制者就可以擦除所有不打算被采样的字节。

与历史证明生成器串通的复制者身份可以在随机字节选择的预先定义的哈希值生成之前，在序列的末端注入一个特定的交易。有了足够的内核，攻击者可以生成一个对复制者身份更有利的哈希值。这种攻击只能使单一的复制者身份受益。由于所有的身份都必须使用相同的哈希值，并通过ECDSA（或同等的）进行加密签名，因此产生的签名对每个复制者的身份都是唯一的，并且可以抗碰撞。一个单一的复制者身份将只有边际收益。

6.7.4 拒绝服务

增加一个额外的复制者身份的成本预计等于存储的成本。增加额外计算能力以验证所有复制者身份的成本预计等于每个复制身份的CPU或GPU核心的成本。

这就为通过创建大量有效的复制者身份而对网络进行拒绝服务攻击创造了机会。

为了限制这种攻击，为网络选择的共识协议可以选择一个复制目标，并授予满足所需特征的复制证明，如网络的可用性、带宽、地理位置等。

6.7.5 下议院的悲剧

PoS核查人员可以简单地确认PoRep而不做任何工作。经济激励措施应与PoS核查员的工作相挂钩。

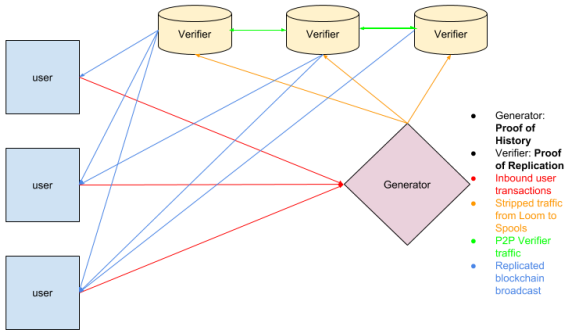


图9： 系统结构

比如通过在PoS验证器和PoRep复制节点之间分配采矿报酬。

为了进一步避免这种情况，PoRep验证者可以在一小部分时间内提交虚假证明。他们可以通过提供产生虚假数据的函数来证明该证明是虚假的。任何确认虚假证明的PoS验证者都会被砍掉。

7 系统结构

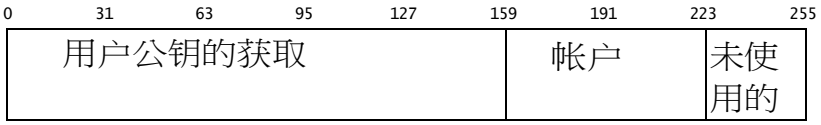
7.1 组成部分

7.1.1 领导人，历史的证明发生器

领导是一个当选的历史证明生成器。它消耗任意的用户交易，并输出所有交易的历史证明序列，以保证系统中唯一的全球秩序。在每一批交易之后，领导者输出一个状态的签名，这个状态是按该顺序运行交易的结果。这个签名是以领导者的身份签名的。

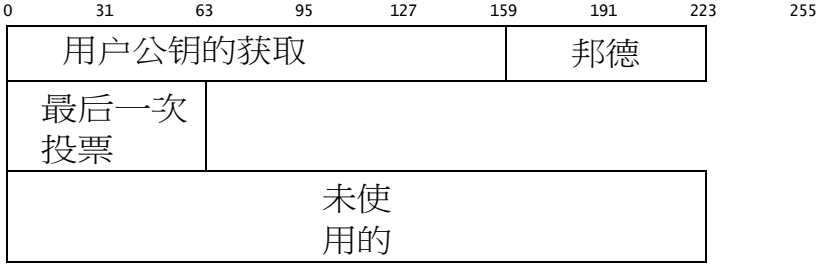
7.1.2 国家

一个以用户地址为索引的天真哈希表。每个单元包含完整的用户地址和该计算所需的内存。例如，交易表包含。



总共32个字节。

股票债券证明表包含。



总共64个字节。

7.1.3 验证人，状态复制

验证者节点复制区块链状态，提供区块链状态的高可用性。复制目标由共识算法选择，共识算法中的验证者根据链外定义的标准选择并投票表决他们认可的复制证明节点。

网络可以被配置为最小的Stake债券规模证明，以及每个债券需要一个复制者身份的要求。

7.1.4 验证人

这些节点正在消耗验证者的带宽。它们是虚拟节点，可以与验证器或领导者在同一台机器上运行，也可以在单独的机器上运行，这些机器是专门为这个网络配置的共识算法。

7.2 网络限制

预计Leader能够接收传入的用户数据包，以最有效的方式排序，并将其排序为历史证明序列，发布给下游验证者。效率是基于

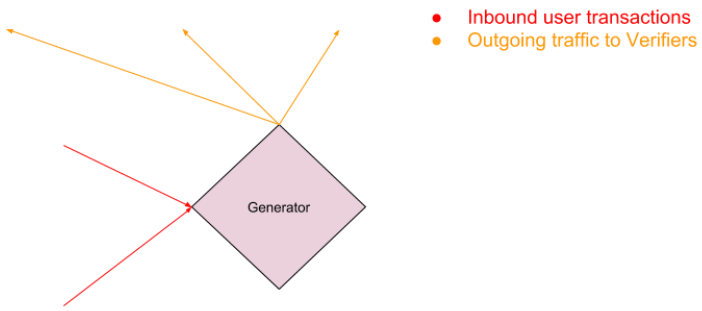
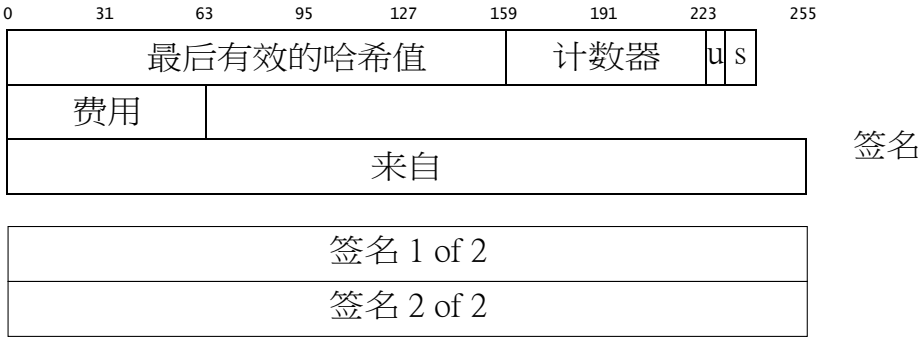


图10：发电机网络限制

交易的内存访问模式，所以交易的排序是为了最大限度地减少故障和最大限度地预取。

入站数据包格式。



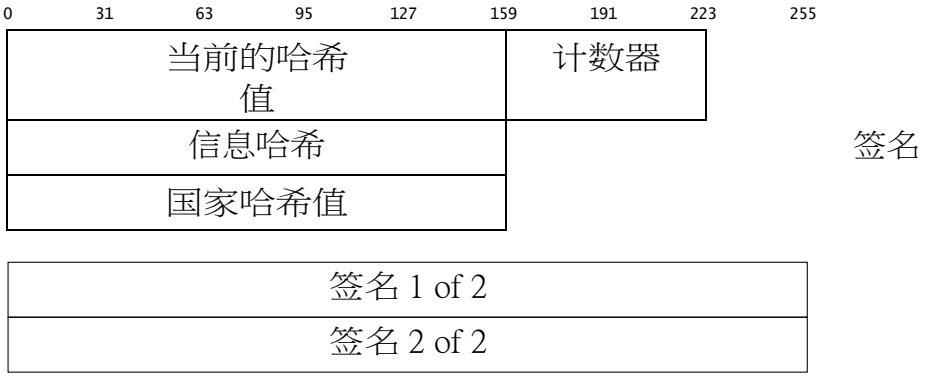
大小20+8+16+8+32+3232=148字节。

可以支持的最小有效载荷是1个目的地计数。带有有效载荷。

带有有效载荷的最小尺寸：176字节

历史证明序列数据包包含当前的哈希值、计数器和添加到PoH序列的所有新消息的哈希值以及处理所有消息后的状态签名。这个数据包每广播N条消息就会发送一次。

历史证明包。



输出数据包的最小尺寸是。132字节

在1gbps的网络连接上，最大的交易数量为

可能是每秒1吉比特/176字节=最大710k tps。由于以太网成帧，预计会有1-4%的损失。超过网络目标量的剩余容量可用于提高可用性，方法是用里德-所罗门编码对输出进行编码，并将其带入可用的下游验证器。

7.3 计算的局限性

每个交易都需要一个摘要验证。这个操作不使用交易信息本身以外的任何内存，并且可以独立进行并行化。因此，预计吞吐量将受到系统中可用的核心数量的限制。

基于GPU的ECDSA验证服务器的实验结果为每秒90万次操作[9]。

7.4 内存限制

一个天真的状态实现是一个50%的全散列表，每个账户有32个字节，理论上可以在640GB中容纳100亿个账户。对该表的稳态随机访问测量为每秒$1.1*10^7$ 次写或读。基于每个交易有2个读和2个写，内存吞吐量可以处理每秒275万个交易。这是在亚马逊网络服务1TB x1.16xlarge实例上测量的。

7.5 高性能智能合约

智能合约是交易的一种普遍形式。这些是在每个节点上运行并修改状态的程序。这种设计利用扩展的伯克利包过滤字节码作为快速和容易分析的代码，并利用JIT字节码作为智能合约语言。

其主要优势之一是零成本的外国函数接口。内在函数，或直接在平台上实现的函数，是可以被程序调用的。调用内在函数会暂停该程序，并将内在函数安排在高性能服务器上。内在函数被分批在GPU上并行执行。

在上面的例子中，两个不同的用户程序调用同一个内在逻辑。每个程序都被暂停，直到本征的批量执行被

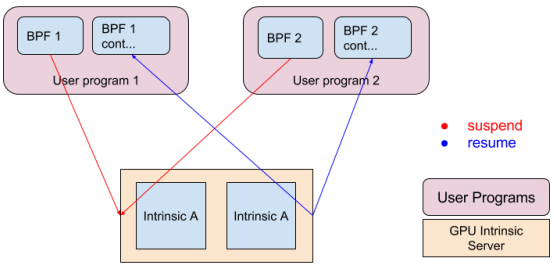


图11：执行BPF程序。

完整的。一个内在的例子是ECDSA验证。分批在GPU上执行这些调用可以将吞吐量提高数千倍。

这个蹦床不需要本地操作系统的线程上下文切换，因为BPF字节码对所有的内存都有一个明确定义的上下文。

它正在使用。

eBPF后端自2015年以来一直包含在LLVM中，因此任何LLVM前端语言都可以用来编写智能合约。它自2015年以来一直在Linux内核中，而字节码的第一次迭代自1992年以来一直存在。单一通道可以检查eBPF的正确性，确定其运行时间和内存要求，并将其转换为x86指令。

参考文献

[1] 利斯科夫，时钟的实际使用
http://www.dainf.cefetpr.br/ tacla/SDII/PracticalUseOfClocks.pdf

[2] 谷歌Spanner TrueTime的一致性
https://cloud.google.com/spanner/docs/true-time-external-consistency

[3] 用订购王解决协议的问题
http://www.inf.usi.ch/faculty/pedone/Paper/2002/2002EDCCb.pdf

[4] Tendermint:没有采矿的共识
https://tendermint.com/static/docs/tendermint.pdf

[5] Hedera。一个管理委员会和公共哈希图网络
https://s3.amazonaws.com/hedera-hashgraph/hh-whitepaper-v1.0-180313.pdf

[6] Filecoin，复制的证明。
https://filecoin.io/proof-of-replication.pdf

[7] Slasher, A punative Proof of Stake算法
https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorith

[8] 比特沙尔的委托权益证明
https://github.com/BitShares/bitshares/wiki/Delegated-Proof-of-Stake

[9] 带有GPU加速的高效椭圆曲线加密法签名服务器
http://ieeexplore.ieee.org/document/7555336/

[10] 友好的终结者卡斯帕的小工具
https://arxiv.org/pdf/1710.09437.pdf