Solana: 高性能区块链的一种新构架

v0.8.14

Anatoly Yakovenko

anatoly@solana.io

翻译 t.chen@205x.org【Draft1】

摘要

本文提出了基于PoH的新的区块链架构 - 验证事件之间的顺序和时间流逝的证明。PoH用于将无需信任的时间流转信息编码进仅能追加数据的账本中。当与共识算法PoW或PoS一起使用时，PoH可以减少BFT【拜占庭容错算法Byzantine Fault Tolerant】复制状态机中的消息量，从而将终局确认时间减少到秒级以内。本文还提出了两种算法，加强了PoH账本的时间记录机制：一种可以从任何大小的分区中恢复的PoS 算法，以及一种高效的流式复制证明（PoRep）。PoRep和PoH的结合在存储和时间维度上提供了的防止账本篡改的防御机制。本文展示了，在使用当今的通用硬件和1gbps网络上运行时，这种协议构架的TPS可以高达710k交易/秒。

1 简介

区块链是一种容错复制状态机的实现。当前公开可用的区块链不依赖于时间，或对参与者保持时间的能力做了少许的假设 [4, 5]。网络中的每个节点通常依赖于自己的本地时钟，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，当一个消息的时间戳用于接收或拒绝消息时，无法保证网络中的其他所有参与者都会做出完全相同的选择。此处介绍的PoH在创建账本时带有可验证的时间信息，比如事件之间的时长和消息的次序。网络中的每个节点将能够无需其他信任信息源，而确信账本中的时间记录。

2 文本组织

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节对PoH进行了详细阐述。第5节对拟议的"PoS共识"算法作了深入描述。第6节详细描述了拟定的Proof of Replication。系统架构和性能限制在第7节中进行了分析。7.5中描述了可使用高性能GPU 的智能合约引擎。

3 系统网络设计

如图 1所示，在任何给定时间，一个系统节点被指定为"领导者"以生成PoH序列，从而提供全网络一致的可验证时间流逝片段。领导者对用户消息进行排序并编号，以便系统中的其他节点可以有效地处理它们，从而最大限度地提高吞吐量。节点执行存储在RAM中的当前状态的交易，并将交易和最终状态的签名并广播发布到同类的节点。这些节点称之为Verifier-验证器。验证器在其状态副本上执行相同的交易，发布其计算结果的签名作为确认。发布确认的过程就是共识算法的投票。在非分区状态下，在任何给定时间，网络中都有一个领导者。每个验证器节点具有与领导者相同的硬件功能，都可以当选为领导者，这是通过基于PoS的选举完成的。PoS

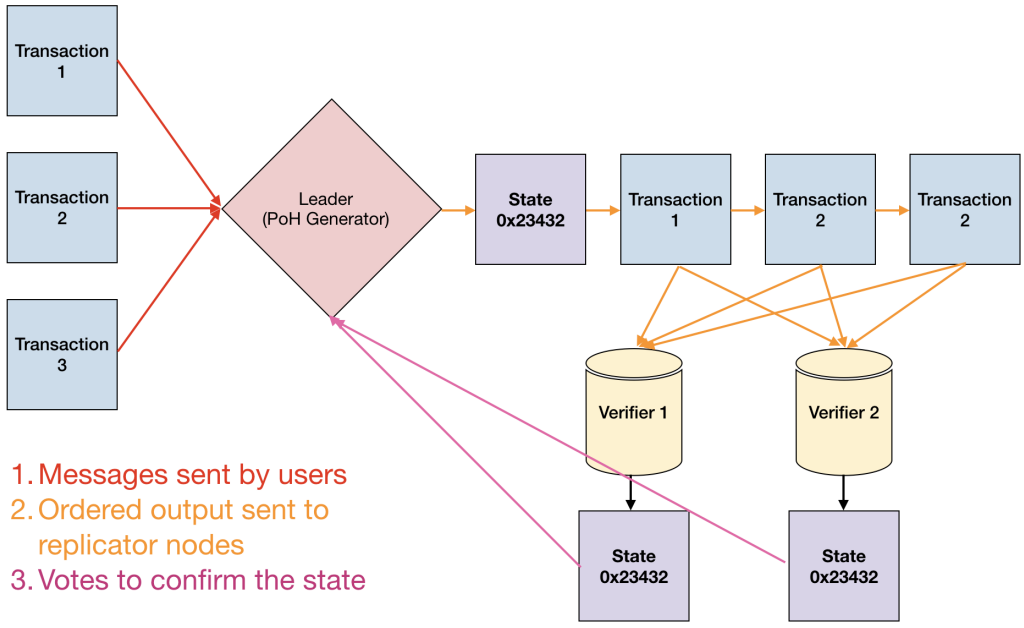


图 1: Transaction flow throughout the network.

的选举算法在第 5.6节中进行了深入的描述。

就CAP定理而言，一致性【C】总是被置于局部【P】有效性【A】之上。在大面积分割的情况下，本文提出了一种机制，可以从任何大小的分区中恢复对网络的控制。第 5.12小节对此进行了深入的探讨。

4 Proof of History

PoH是一个计算结果的序列。这种计算可以验证两个事件之间的时间流逝次序。这里采用了加密运算函数做计算，输入不能由输出反向推算，并且必须完全执行运算函数才能生成输出。该函数在单个内核上按顺序运行，其上次的计算输出为当前的输入数据，不停地记录输出数据，以及调用的次数。其他计算机可以在单个计算内核上复现一个计算并验证结果，然后使用多核进行并行数据验证。

可以通过将特定数据（或数据的哈希）附加到函数的状态中来给特定数据打上时间戳。状态、序号和序列中添加的特定数据一起提供了一个时间戳，可以保证特定数据是在序列中生成下一个输出之前的某个时间创建的。此设计还支持横向扩展，因为多个PoH生成器可以汇合状态然后同步彼此的序列。第 4.4节深入讨论了横向扩展的细节。

4.1 概述

PoH系统设计为工作如下。使用一个必须实际计算才能得出输出的加密哈希函数（如 `sha256`, `ripemd`等），用随机输入值运行该函数，并将其输出作为输入再次传递到同一函数中。记录调用函数的次数和每次调用的输出。选择的开始随机值可能是任何字符串，如当天的纽约时报标题。
示例:

PoH Sequence

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

`hashN` 代表了实际的哈希运算结果

实际上，只需要发布哈希运算结果的片段，给运算编上索引号即可。

示例:

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碰撞的，这组哈希就只能由单个计算机线程按顺序计算。如果不从开始值实际运行算法300次，就无法预测索引300的哈希值将是什么。因此，我们可以由索引0和到300之间的数据结构推断期间有时间的流逝。

在图2中的例子中，哈希`62f51643c1`是按索引510144806912运算的，哈希`c43d862d88`是按索引510146904064计算的。按照之前讨论的PoH算法的属性，我们可以确信，索引510144806912和索引510146904064之间有时间的流逝。

4.2 事件的时间戳

哈希序列也可用于记录事件：某些数据是在生成特定的哈希索引之前创建的。使用“组合”功能将数据片段与当前索引中的当前哈希数据相结合。这些数据可以简单地是任意事件数据的唯一哈希。组合功能可以是数据的简单附录，也可以是任何抗碰撞运算。下一个生成的哈希表示数据的时戳，因为只有在插入该特定数据之后这个哈希值才能生成。

示例:

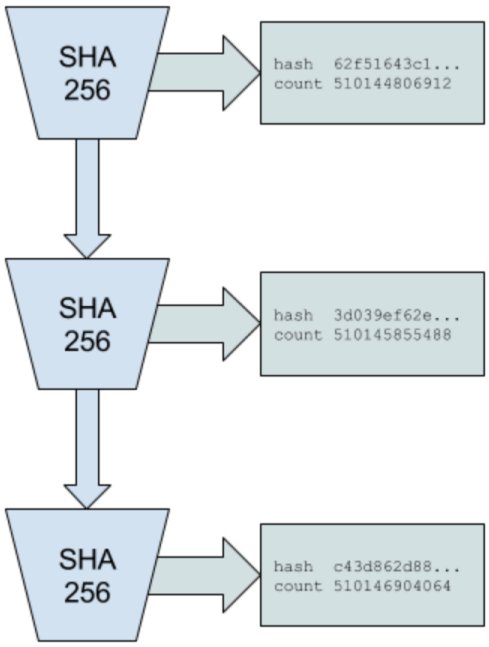


图 2: Proof of History sequence

PoH Sequence

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

在有外部事件发生，比如照片拍摄或者任何数据的产生时：

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: PoH Sequence With 2 Events

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`是根据哈希 `hash335`和照片的`sha256`计算的。索引和照片的`sha256`记录为序列输出的一部分。因此，任何验证此序列的人都可以重新生成包含这种更改的序列。验证仍可以并行进行，将在第 4.3 节中讨论。

由于初始过程仍按顺序排列，因此我们可以判断进入序列的事件一定发生在运算此后的哈希值之前的某个时间。

在表 1所表示的序列中，照片2是在哈希`hash600`之前创建的，照片1是在哈希`hash336`之前创建的。将数据插入哈希序列会导致序列中所有后续值的更改。只要使用的哈希函数具有抗碰撞能力，数据被附加后，根据数

据信息无法预先计算未来的哈希。

混入序列的数据可以是原始数据本身，也可以带有元数据的数据哈希。

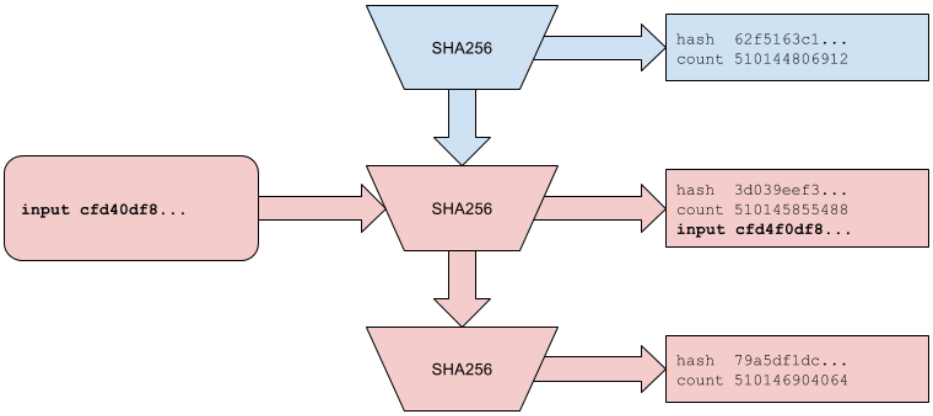


图 3: Inserting data into Proof of History

在图 3中的示例中，输入`cfd40df8...`被插入PoH序列。插入的索引为510145855488，插入状态为 `3d039eef3`。所有其后生成的哈希都会因此而改变，这个改变在图中以颜色的变化来示意。

每个观察此序列的节点都可以确定插入事件的顺序，并估计数据插入之间时间。

4.3 验证

多核计算机可以在比生成序列的时间短得多的时间内验证该序列的正确性。

示例:

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`

对于多核的计算平台，如具有4000个内核的现代GPU，验证器可以将哈希及其索引的序列拆分为4000片，同时验证每个切片从开始哈希到切片中的最后一个哈希是正确的。如果生成序列的预期时间是：

$$\frac{\text{Total number of hashes}}{\text{Hashes per second for 1 core}}$$

则验证序列是否正确的预期时间是：

$$\frac{\text{Total number of hashes}}{\text{(Hashes per second per core * Number of cores available to verify)}}$$

在图 4中的示例中，每个内核能够并行验证序列的每一片。由于所有输入字符串都记录在输出中，并附有计数器并附状态，验证器可以并行验证每个切片。红色哈希表示序列通过数据插入进行了修改。

4.4 横向扩展

通过汇合序列，可以同步多个生成器的序列状态，实现PoH生成器的横向扩展。此扩展无需分片即可完成。多个生成器的输出需要重构，以形成全系统的事件序列。

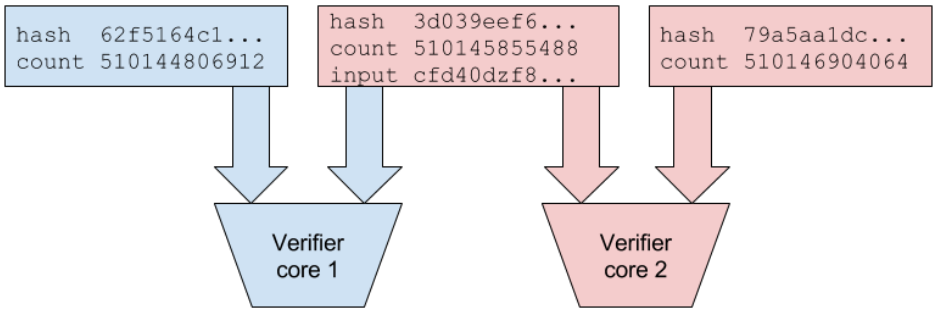


图 4: Verification using multiple cores

PoH Generator A			PoH Generator B		
Index	Hash	Data	Index	Hash	Data
1	hash1a		1	hash1b	
2	hash2a	hash1b	2	hash2b	hash1a
3	hash3a		3	hash3b	
4	hash4a		4	hash4b	

给定PoH产生器 A 和B，A接收来自 B （hash1b）的数据包，其中包含生成器B 的最后一个状态，以及B观察到的A的最后一个状态。生成器 A 中的下一个状态哈希，取决于来自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。此判定是有传递性的，因此，如果三个生成器同步到单个生成器 $A \leftrightarrow B \leftrightarrow C$, 我们可以跟踪A和C之间的依赖性，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部分外部流量，因此，由于生成器之间的网络延迟，整个系统可以真实时间精度处理大量事件。可以通过选择一些决定性函数来排序同步窗口内的任何事件（例如通过哈希本身的值）来实现全系统次序。

在图 5中，两个生成器插入彼此的输出状态并记录操作。颜色变化表明来自对方的数据对序列进行了修改。以粗体突出显示混合到每个流中的生成的哈希。

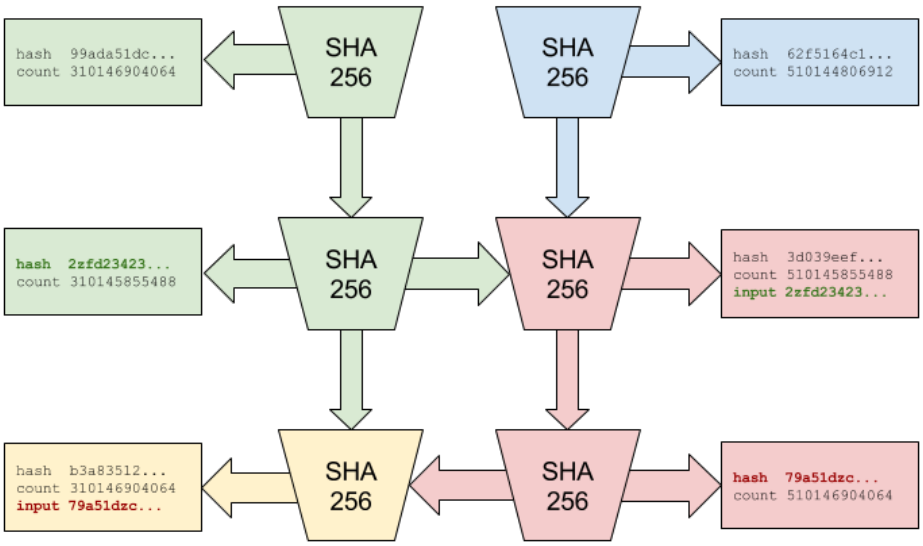


图 5: Two generators synchronizing

同步是有传递性的。$A \leftrightarrow B \leftrightarrow C$ 中， A和C之间通过B可确定事件顺序。

这种扩展会带来可用性的缩减。10个1gbps连接，如果单个可用性是0.999，系统可用性将会是 $0.999^{10} = 0.99$。

4.5 一致性

通过加入观测到的最后序列输出值到事件数据中，PoH使用者可以保证PoH的一致性，从而抵御攻击。

PoH Sequence A		
Index	Data	Output Hash
10		`hash10a`
20	Event1	`hash20a`
30	Event2	`hash30a`
40	Event3	`hash40a`

PoH Hidden Sequence B		
Index	Data	Output Hash
10		`hash10b`
20	Event3	`hash20b`
30	Event2	`hash30b`
40	Event1	`hash40b`

如果一个恶意的PoH可以访问到所有事件，或者能够生成更快的序列。它可以生成第二个隐藏序列，含有不同的事件次序。

为防止此类攻击，每个单点生成的事件内容应包含它观察到的最新哈希。因此，当节点创建"Event1"数据时，它们应该附加他们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`

当序列发布时，Event3将引用哈希30a，如果30a不在该事件之前的序列中，则序列的使用者就知道它是一个无效序列。因此，部分重新排列攻击将限于节点观察到事件以及事件进入时产生的哈希数。节点可以编写软件，不要假设最后观察到的和新插入的哈希之间的短时间内顺序是正确的。

为防止恶意 PoH 生成器重写事件哈希，节点可以提交事件数据和最后观察到的哈希的签名，而不仅仅是哈希数据。

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

验证此数据需要验证签名，以及需要在之前序列中找到对应的哈希。验证:

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

在图 6中，用户提供的输入附加了之前某个时间存在于PoH序列中的哈希0xdeadbeef…。左上蓝色箭头表示节点引用以前生成的哈希值。节点的消息仅在包含哈希0xdeadbeef的序列中有效...序列中的红色表示序列已被节点数据修改。

4.6 开销

每秒4000个哈希将产生额外的 160KB的数据。具有4000个核的GPU需要大约0.25-0.75秒进行验证。

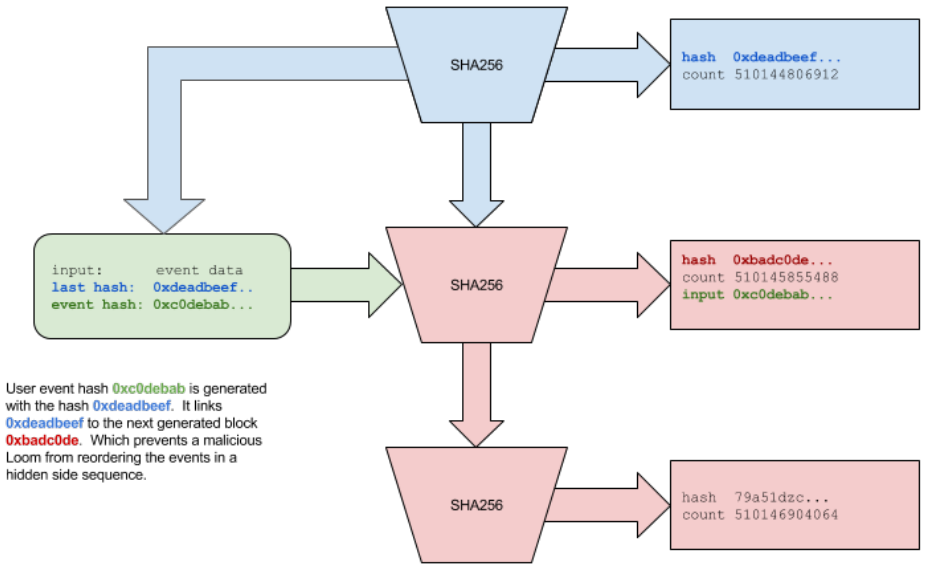


图 6: Input with a back reference.

4.7 抵御攻击

4.7.1 更改次序

生成更改的次序需要攻击者在第二个事件后启动恶意序列。此延迟期间非恶意节点之间已经交换了原有次序的数据。

4.7.2 速度

多个生成器对攻击更具抵抗力。一台生成器可能是高带宽，并接收许多事件混合到其序列中，另一个生成器可能是高速低带宽，定期与高带宽发生器混合。高速序列将创建攻击者必须反转的辅助数据序列。

4.7.3 远程攻击

远程攻击涉及获取旧丢弃的节点私钥，以生成伪造的账本 [10]. PoH为抵御远程攻击提供了一些保护。获得旧私钥访问的恶意用户必须重现历史

记录，该记录需要的时间与他们试图伪造的原始记录一样多。这将需要比网络当前使用的更快的处理器，否则攻击者将永远不会赶上历史长度。

此外，单一的时间来源允许构建更简单的复制证明Proof of Replication（更多参见第 6节）。网络设计为所有参与者将依赖于单一的事件历史记录。

PoRep和PoH 一起为空间和时间提供了防御机制，以抵御伪造的账本。

5 PoS共识机制

5.1 概述

此特定POS的实现旨在快速确认当前PoH生成器产生的序列，用于投票和选择下一个PoH生成器，并惩罚任何行为异常的验证器。此算法需要确保在一定时间范围内，消息最终到达所有的节点。

5.2 术语

bonds 验证抵押相当于POW中的资本支出。POW矿工购买硬件和电力，并加入到一个POW区块链中。验证抵押是验证者在验证交易时作为抵押品的币。

slashing 扣罚方案用以解决POS质押的问题 [7]. 当验证者为其他分支投票的证明公布时，可能会导致验证者的抵押币被销毁。这是一种经济措施，旨在阻止验证者确认多个分支。

super majority 超级多数是按照抵押加权计算的验证器的2/3。超级多数表明，该网络已达成共识。要使得这个分支无效，1/3以上的网络节点必须恶意投票。这将使攻击的经济成本达到质押币的总规模的1/3。

5.3 抵押使用

提交交易需要一定数量的币，币会转到节点名下的抵押帐户。抵押账户中的币不能使用，必须留存在帐户中，直到用户移除它们。节点只能移除已过时的无效币。在超级多数验证者确认对应序列后，抵押变得再次有效。

5.4 投票

PoH生成器会在预定的周期内发布一个状态的签名。每个有BOND的验证者必须发布自己对此状态的签名来确认该PoH生成器的签名。投票是简单的赞成票，没有反对票。如果绝大多数验证者在该时间段投票，则该分支数据将被接受为有效。

5.5 抵押移除

错过N次投票标志着验证抵押币过时，不再有资格投票。用户可以发出UNBONDING交易来移除它们。

N是基于过时投票与有效投票的比例的动态值。N 随着过时投票数目的增加而增加。在发生大型网络分区时，这允许较大的分支恢复速度快于较小的分支。

5.6 选举

检测到PoH发生器故障时，将选举新的PoH 生成器。具有最大投票权的验证器将被选为新的 PoH 生成器。

新PoH 生成器需要超级多数的确认。如果新领导者在获得超级多数确认之前出现故障，则选择下一个最大投票权的验证器，并且需要新的超级多数确认。

要进行切换投票时，验证器需要给最高的PoH 序列投票，而且新的投票需要包含它想要切换的前投票。否则，第二次投票将是有惩罚的。投票

切换被设计为仅在没有超级多数的确认高度进行。

一旦PoH生成器确立，可以选择一个备选者来接管交易处理职责。如果存在备选者，主PoH生成器失效时，它将被视为下一位领导者。

平台的设计是，如果检测到异常或在预先定义的流程，备选者成为领导者，低阶的生成器同步获得提升。

5.7 选举触发机制

5.7.1 生成器分支

PoH生成器的都有一个ID，可以签署生成的序列。只有在PoH 生成器的身份被泄露的情况下，才能发生序列分支。以同一PoH身份发布两个不同的历史记录，网络会检测到分支。

5.7.2 运行时异常

PoH生成器的硬件故障、软件错误或有意的犯错，可能导致其生成无效状态，并发布与本地验证器结果不符的状态签名。验证者将通过广播发布正确的签名，此事件将触发新一轮选举。任何接受无效状态的验证者都会被惩罚，削减其BOND。

5.7.3 网络超时

网络超时将触发选举。

5.8 扣罚

当验证器投票给两个不同序列时，就会发生系统惩罚。恶意行为的投票将取消其BOND币的流通，并将其添加到矿池。

包含先前对竞争序列的投票不算恶意行为投票。这次投票不会触发惩罚，而是取消了对竞争顺序的投票。

如果投票确定PoH生成器生成了无效哈希，也会发生惩罚。生成器可能会生成无效状态，这时会触发备选者继任。

5.9 备选者选举

可以提出和批准备选和低阶PoH生成器。提案投在主生成器的序列上。该提案包含超时，如果该动议在超时前获得绝大多数票数通过，则认为备选者当选，并将如期接管职责。主选者可以主动发起软切换到备选者，插入消息到生成的序列，指示将发生交接即可。或插入无效状态并强制网络切换到备选者。

在选举过程中，如果主选者失效，并且之前有备选者，备选者自动接手。

5.10 可用性

处理局部性问题【部分验证节点无效】的 CAP 系统必须选择一致性或可用性。我们的方法最终选择了可用性。但是，由于我们有一个客观的时间衡量标准，在合理的人类感知的超时内，一致性优先。

POS验证人将一定数量的币锁定在质押态中，从而允许他们投票给一组特定的交易。锁定币是一种交易，进入PoH流，就像任何其他交易一样。要进行投票时，PoS 验证器签署交易状态的哈希。哈希是在将所有交易处理到 PoH账本中的特定位置后计算的。此投票也作为交易输入 PoH 流。查看 PoH 账本，我们可以推断出每次投票之间经过了多少时间，如果出现局部性问题，可以推断每个验证器无效的时间。

为了在合理的时间内处理局部性问题，我们提出一种动态的移除无效节点的方法。当有效节点超过2/3时，移除流程会比较快。账本包含的哈希数会比较少，其后无效验证节点会移除，不再计入共识统计。如果有效节点低于2/3但是高于1/2，移除过程会长一点，需要更多的HASH产生，以移除无效节点。在大面积无效情况下，比如缺失1/2或更多验证节点，移除过程将会花费很长时间。这时，交易还可以继续进入PoH流，验证节点还可

以继续投票，但是2/3共识数量无法达到。直到大量哈希产生，无效的节点被移除为止。这不同的时长的网络活动恢复机制，使得我们在合适的时长内，可以选择一个有效的局部节点范围以继续使用区块链网络。

5.11 恢复

在我们设计的系统中，交易记账可以完全地从任何失效状态恢复。这意味着，任何人可以随机选择账本的任意一点，由此开始创建链的分支，产生哈希并添加新的交易。如果所有验证节点忽略了这一个分支，此分支要达到2/3绝对多数的共识，将会花费很长很长时间。一个0验证的分支恢复，将会有大量的HASH产生插入到账本，直到无效的节点移除，新的节点最后验证了这个账本。

5.12 确定性

PoH使得验证节点可以观测过去的事件，得知一定的时间上的确定性。由于PoH 生成器负责生成消息流，因此所有验证节点都必须在 500ms 内提交其状态签名。此数字可以根据网络条件还可以进一步减少。由于每次验证都进入数据流，网络中的每个人都可以验证每个验证节点在规定时间是否投票，而无需直接观察投票。

5.13 抵御攻击

5.13.1 懒惰节点

PoS 验证节点只需确认 PoH生成器产生的状态哈希。从经济动机上讲，他们可以不做任何工作，只是简单批准每一个生成的状态哈希。为了避免这种情况，PoH 生成器应随机注入无效的哈希。任何支持这种无效哈希的投票节点都应该被惩罚。产生无效哈希后，网络应立即切换到备选PoH 生成器。

每个验证节点都需要在较短时间内响应 -例如，500ms。超时应设置得足够低，恶意验证节点作恶的可能性就很低。它们没有时间去观察其他验证节点的投票，无法很快进行恶意投票。

5.13.2 串谋

与 PoH 生成器串通的验证节点前知道何时将生产无效的哈希，同时不投票。这种情况和PoH生成器身份产生问题相同，PoH生成器必须有最大的质押份额。 PoH生成器仍必须完成其他所有事项才有资格产生状态哈希。

5.13.3 审查

审查或拒绝服务可能发生在1/3的BOND持有节点拒绝投给任何有新BOND的序列。系统协议通过动态调整BOND过期时间，可以抵御这种攻击。在拒绝服务出现后，多数节点产生分支并审查拒绝服务节点。多数节点在将会随着拒绝服务节点的BOND过期而获得恢复。而少额的拒绝新BOND节点，则需要很长时间继续发展分支。

该算法工作流程如下。大多数网络节点将选出新的领导者。然后，领导人将审查少额拜占庭节点。PoH发生器继续生成序列，以证明时间的流逝，直到足够的拒绝服务节点的BOND过时，然后多数节点网络会获得超级多数共识。BOND过时的速度将根据活跃BOND的百分比动态地计算。因此，大多数节点分支要比小团体拜占庭共识分支更快的达到超级多数共识。一旦建立了超级多数共识，网络可以惩罚小群体拒绝服务节点。

5.13.4 远程攻击

PoH提供了抵御远程攻击的天然防御机制。从过去的任何时候发展分支将要求攻击者超过PoH生成器的速度，从而及时超越正常记账。

共识协议提供了第二层防御机制，因为任何攻击都需要比移除有效验证节点所需的时间更长的时间。攻击就会在账本的历史中留下这种差异。

在比较两个相同高度的账本时，可以客观地认为最短时间的最大共识的账本有效。

5.13.5 ASIC攻击

此协议中存在两个ASIC 攻击机会–局部性网络期间，以及确定性超时作弊。

对于局部性网络期间的 ASIC 攻击，移除BOND质押的比率是非线性的，对于分区较大的网络，该比率比 ASIC 攻击的预期收益要慢几个数量级。

对于最终账本确定阶段的 ASIC 攻击，恶意验证节点等待其他节点的确认，并将其投票注入协调作恶的 PoH 生成器。然后，PoH生成器可以使用其速度更快的 ASIC 在更短的时间内生成对应500ms 的哈希，并与作恶节点之间通信。但是，如果 PoH 生成器也是作恶的，则当Byzantine 生成器期望插入异常数据时，它要和确定数量的验证节点通信。这种情形下，PoH生成器和协同者作恶与单个硬件持有合并的Stake权益没有什么不同。

6 流式复制证明

6.1 简述

Filecoin提出一个Proof of Replication【复制证明】 [6]的版本。此版本的目标是对复制证明进行快速和流式验证。这一目标可以通过持续验证PoH序列而实现。Solana中，复制证明不参与共识算法，而是作为一个工具，在高可用性环境下存储区块链历史或记录状态。

6.2 算法

如图 7所示，使用CBC按次序加密每个区块数据，使用前一个块的加密结果对输入数据进行 XOR操作。

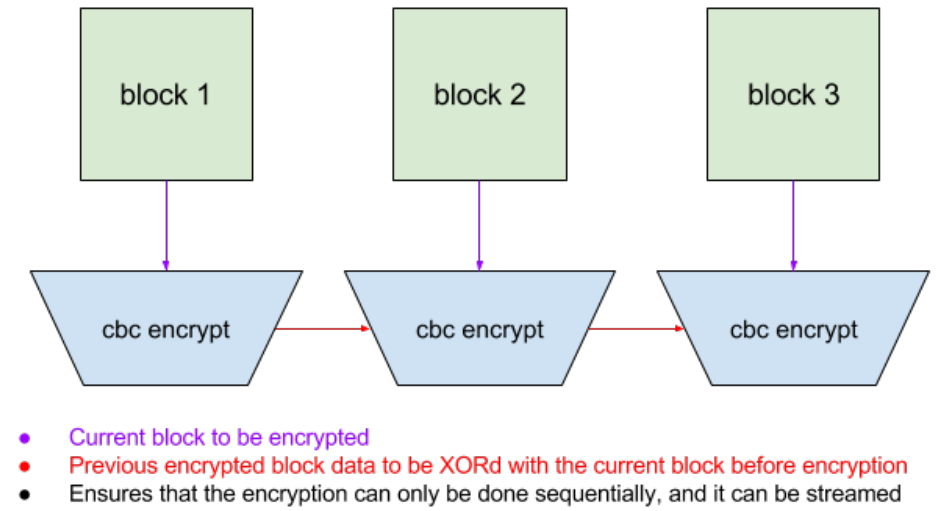


图 7: Sequential CBC encryption

每个复制器都会对PoH序列生成的一个哈希签名而生成一个密钥。这个密钥因此关联了复制器的身份和特定的PoH序列。只有特定的PoH哈希会被使用（参见部分 6.5关于哈希选择）。

区块数据按照次序逐个加密。其后，为了生成复制证明，上面生成的密钥会作为一个伪随机数产生器的种子。随机地从区块中选择 32 字节数据片段。再对每个数据片段前置选定的 PoH 哈希，然后计算一个Merkle哈希。

最后，跟密钥和所选的PoH哈希一起，发布Merkle哈希树的根节点。N个PoH哈希后，复制器需要发布另一个证明。N大约等于加密时间的1/2。PoH产生器会给复制器在预定的时间周期内发布哈希值。复制器必须选择新的哈希用来产生复制证明。和前述相同，哈希签名后用于选择区块中的随机数据片段来产生Merkle根节点。

每N个证明周期之后，系统使用一个新的CBC Key来加密数据。

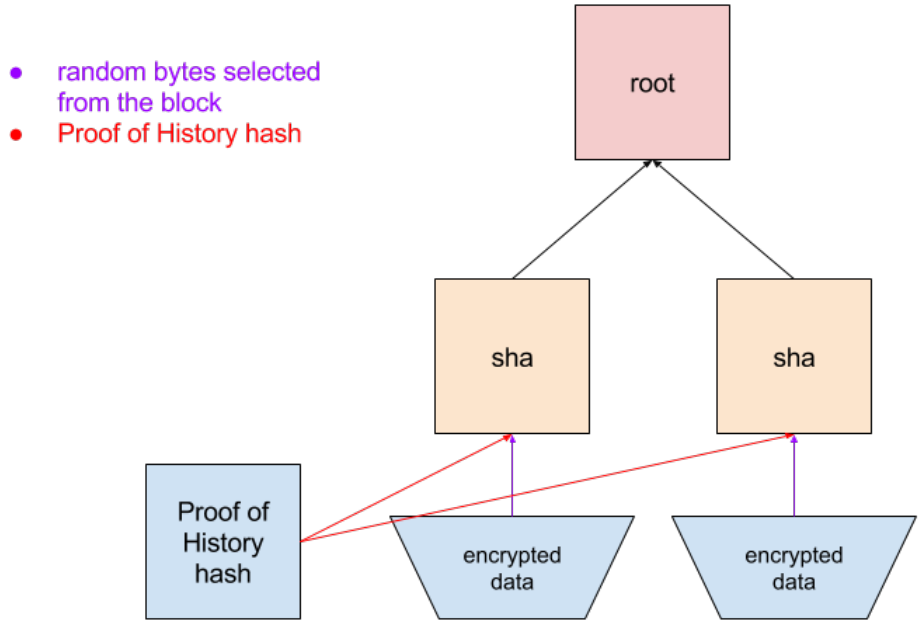


图 8: Fast Proof of Replication

6.3 验证

使用N内核的计算设备，每核能按照区块链流为每个复制器加密数数据。因为需要前一个区块加密下一个区块，系统所需的总空间是2区块*N核。每个内核都从当前区块产生其后的所有复制证明。

复制证明的验证时间将会和加密所需的时间相同。验证本身只需要从区块中取很少的随机数据片段用于计算哈希，相比整个区块，这些片段的数据量非常小。可以同时验证的复制器身份数量等于可用内核的数量。现代 GPU 有 3500+的内核，尽管它们的时钟速度是CPU的一半到1/3。

6.4 Key更替

CBC如果保持不变，生成多个历史证明序列的会相对容易。密钥会根据PoH序列定期更换。

更换周期要足够长，确保在GPU硬件上能够实际应用，因为 GPU 内核的运算速度比 CPU慢。

6.5 哈希选择

PoH生成器发布哈希，供整个网络工作用于复制证明的加密，并作用于伪数字生成器，用于选择快速验证所使用的区块字节片段。

哈希的发布周期大致等于数据加密时间的1/2。每个复制器必须使用相同PoH哈希，签名后，作为随机数生成器的种子和加密密钥。

每个复制器提供复制证明的周期必须小于加密时间。否则，复制器可以在提供证明期间重组和删除加密数据。

恶意PoH生成器可能在哈希生成之前将数据注入序列以生成特定的哈希。这种攻击在 5.13.2中有具体的讨论。

6.6 复制证明的终验

PoH节点不会验证提交的复制证明。它只记录待定和已验证证明的数量。复制证明在签署由有效多数Validator发布的验证数据包后完成。

验证结果通过p2p广播网络收集，它会是包含网络中有效多数Validator的一个数据包。此数据包验证在特定PoH哈希之前的所有复制证明，并可以同时包含多个复制器标识。

6.7 抵御攻击

6.7.1 垃圾数据

恶意用户可能会创建许多复制器身份，并使用不良证据向网络发送垃圾数据。为了加快验证速度，在请求验证时，需要节点向网络的其他部分提供已加密数据和整个 merkle 树。

本文中设计的复制证明可以使用非常简易的方法验证很多额外的证据，因为它们无需占用额外的空间。但每个恶意复制器将消耗1个运算内核的加

密时间。目标复制器应选择使用最大可用内核数量。现今的GPU有3500以上的内核。

6.7.2 局部擦除

复制器节点可能会尝试部分擦除某些数据以避免存储整个状态。复制证明的数量和种子的随机性使得这种攻击难以实施。

例如，从存储的1TB数据中，每1MB数据擦除1个字节。单个复制证明，产生冲突的可能性为 $1-(1-1/1,000,0000)^{1,000,000}=0.63$。5个证明后，可能性是0.99.

6.7.3 串谋

签过名的PoH哈希用于选择数据片段。如果复制器可以提前选择特定的哈希，则复制器可以擦除数据片段外的字节。

与PoH生成器串通的复制器可以在确定数据片段选择的哈希产生之前，在序列末尾注入特定的交易。如果有足够多的内核，攻击者可以生成一个合适的哈希。

此攻击只能影响单个复制器。由于所有复制器必须使用ECDSA（或同类）签名同一个哈希，生成的签名对于每个复制器来说是独一无二的，具有抗碰撞性。单个复制器攻击的威胁有限。

6.7.4 拒绝服务

添加额外复制器的成本预计将等于存储成本。增加额外的计算能力以验证所有复制器身份的成本将等于每个复制器对应一个CPU或GPU内核。

通过创建大量有效的复制器身份，可以由可能进行拒绝服务攻击。

为了压制此攻击，网络共识协议可以选择复制器，奖励符合特定条件的复制证明，如网络可用性、带宽、地理位置等。

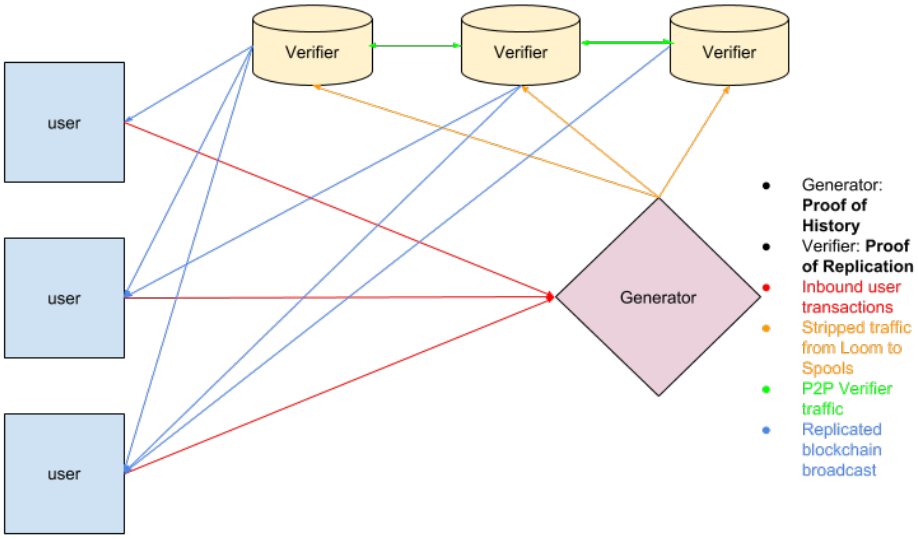


图 9: System Architecture

6.7.5 懒惰节点

PoS验证器可以无条件确认PoRep，而不执行真正的验证工作。系统经济激励机制通常会奖励PoS验证器所作的工作。比如，区块挖矿收益分配给PoSe验证器和PoRep复制器。

为了进一步避免这种情况，PoRep验证器可以在一小部分时间提交虚假证明。他们可以通过提供生成虚假数据的运算来证明证据是假的。任何确认假证明为真的PoS 验证器都会受到惩罚。

7 系统构架

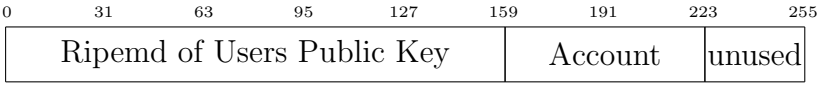
7.1 部件

7.1.1 领导者, PoH生成器

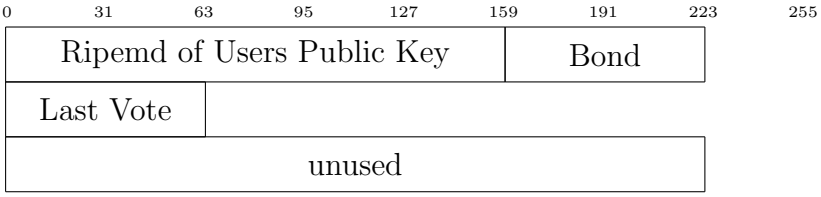
领导者是通过选举产的PoH生成器。它打包任意的交易，并产生对应的一个PoH序列，保证了系统中的全局次序。每批交易后，领导者都会按该顺序运行交易的结果输出一个状态的签名。

7.1.2 状态

状态是一个简单的哈希表，按用户的地址索引。每个单元格都包含完整用户的地址和此计算所需的内存。例如,
交易表包含:



共32字节.
POS验证抵押表包含:



共64字节.

7.1.3 验证器, 状态复制

验证器节点保存区块链状态的高可靠性副本。复制目标内容由共识算法决定，共识算法中的投票节点【Validator】根据预定义链下规则投票选择验证器节点。

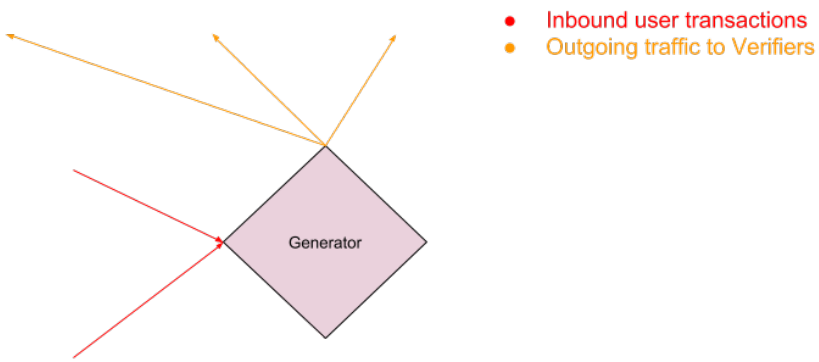


图 10: Generator network limits

Solana网络可以配置最低的POS验证抵押，一个验证抵押需要对应的一个验证器。

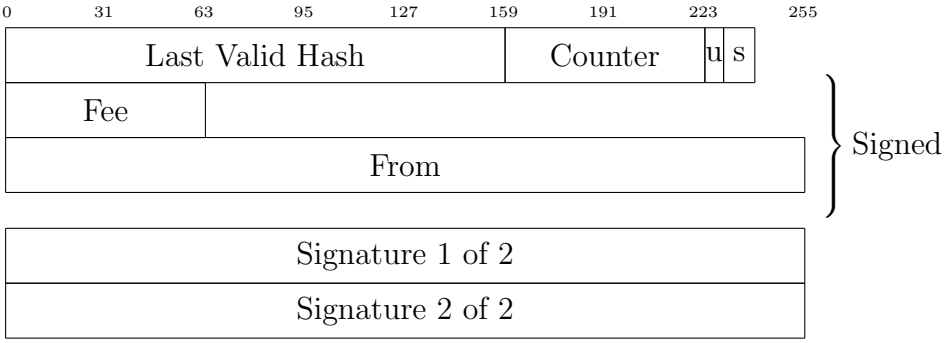
7.1.4 共识节点

这些投票节点会读取验证器的数据。它们是虚拟节点，可以与验证器或领导者运行在同一台机器上，也可以专门运行在一个机器上运行共识协议。

7.2 网络限制

领导者接收传入的交易数据包，以尽可能高效的方式排序，然后产生PoH序列。然后发布到下游的验证器。这个过程的效率取决于数据包的内存访问方式，因此领导者排序交易数据，以减少故障并提高处理量。

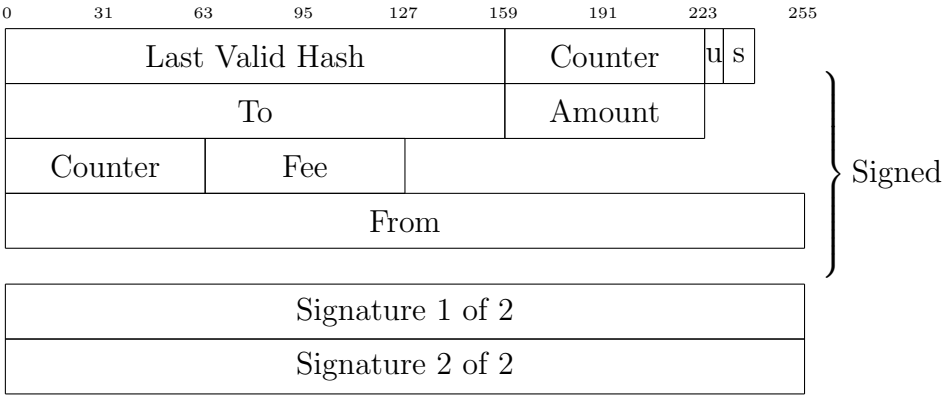
Incoming packet format:



Size $20 + 8 + 16 + 8 + 32 + 32 + 32 = 148$ bytes.

最小数据是单个接收人的交易，大小为 176 字节。

With payload:

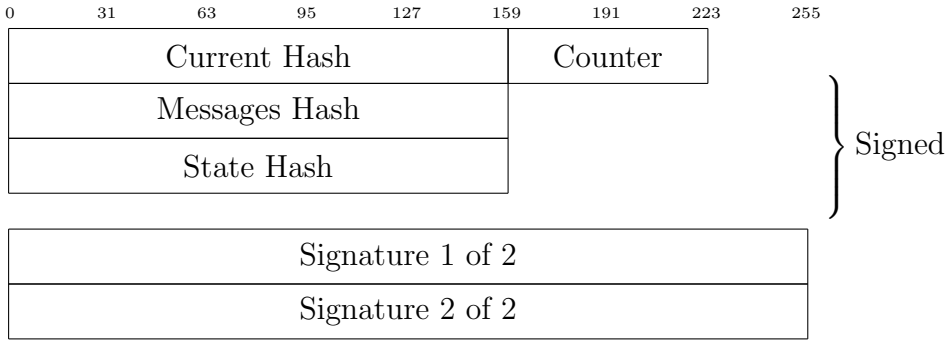


With payload the minimum size: 176 bytes

PoH序列数据包含当前哈希值、计数器、添加到 PoH序列中的所有新

消息的哈希值，以及数据处理后的状态签名。处理 N个消息后，数据包就会发送。

Proof of History 数据包:



最小数据包长度: 132 bytes

在1gbps网络连接中，可能的最大交易数量为： 1gbps/176字节=710k tps。在以太网的构架中，会出现一些数据丢包(1−4%)。使用 Reed-Solomon编码可以额外增加数据容量，最后再发给验证器。

7.3 算力限定

每个交易都需要进行验证。此操作不使用交易消息本身以外的任何数据，并且可以独立并行运算。因此，吞吐量仅受制于系统可用内核数量。

实验数据显示，使用 GPU 的 ECDSA 验证服务器，验证可以达到每秒90万次 [9]。

7.4 存储限定

如果50%的全哈希表的条目是32字节的账户数据，640G可以管理100亿个帐户。这个大Hash表每秒可以支持 $1.1 * 10^7$ 读写操作。基于每交易2次写操作、一次读操作计算，内存操作每秒可以处理275万次交易。以上性能

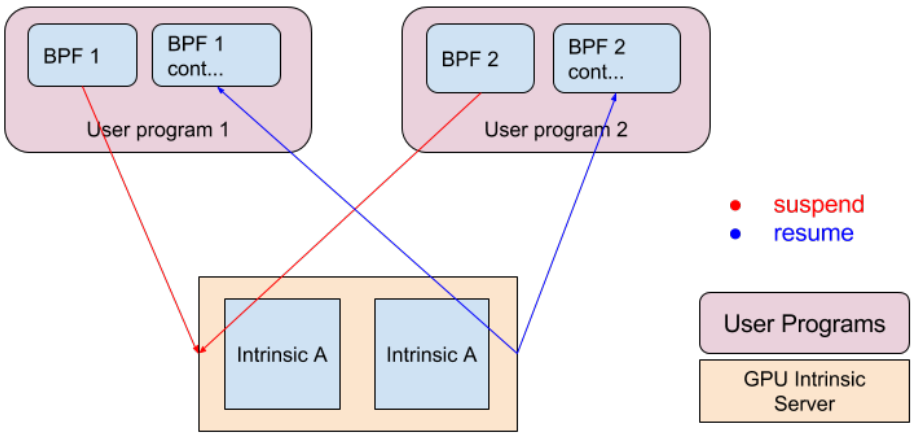


图 11: Executing BPF programs.

在亚马逊的云服务器上做了测试，使用了AWS的1TBx1.16xlarge云服务器实例。

7.5 高性能智能合约

智能合约可归结为通用形式的交易。智能合约程序在每个节点上运行并修改状态。此设计利用扩展的Berkeley Packet Filter【BPF】字节代码，该字节码快速易于分析，JIT即时编译字节代码作为智能合约。

BPF的主要优势之一是零成本的对外功能接口。内部固有功能或函数直接由平台实现，由智能合约调用。调用固有函数会暂停该智能合约，并再高性能服务器上启动运行固有函数。使用GPU，固有函数可以分批并行运行。

在示例中，两个不同的智能合约调用同一个固有函数。两个智能合约都会暂停，等待固有函数运行结束。固有函数的一个例子是ECDSA椭圆加密的验证功能。再 GPU上执行这些固有函数，可使系统吞吐量增加数千倍。

BPF这种调用与原生操作系统线程上下文无关，因为 BPF 字节代码

明确定义了它所内存的上下文关联。

eBPF 后端自2015年以来已包含在LLVM中，因此任何LLVM 前端语言都可以用于编写SOLANA智能合同。BPF字节代码始于1992年，自2015 年以来，它就存在于Linux 内核中。单次检视就可以确认eBPF的正确性，确定其运行环境和内存要求，并将其转换为 x86指令集代码。

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-algorith

[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