

# Ultraiin 共识黄皮书

March 23, 2019

# Contents

<b>1</b>	<b>前言</b>	<b>2</b>
<b>2</b>	<b>贡献</b>	<b>3</b>
<b>3</b>	<b>名词解释</b>	<b>3</b>
3.1	签名验证 . . . . .	3
3.2	摘要函数 . . . . .	3
3.3	可验证随机函数 . . . . .	4
3.4	后量子算法 . . . . .	4
3.5	BLS 算法 . . . . .	5
3.6	区块 . . . . .	5
3.7	节点身份 . . . . .	5
3.8	消息类型 . . . . .	5
<b>4</b>	<b>架构</b>	<b>6</b>
4.1	身份层 . . . . .	6
4.2	VRF 层 . . . . .	6
4.3	共识层 . . . . .	7
4.3.1	分片 . . . . .	7
4.3.2	延时 . . . . .	7
4.4	新节点加入 . . . . .	12
<b>5</b>	<b>安全性证明</b>	<b>12</b>
5.1	定理 . . . . .	12
5.2	安全假设 . . . . .	13
5.3	推论 . . . . .	13

## Abstract

Ultrain 共识层的目的是为了在不损失安全度和去中心化特性的情况下，通过随机数和密码学应用方面的创新，提高区块链底层的性能。

## 1 前言

对于区块链应用场景，需要共识算法具有如下特性

- Throughput : transaction per second 意味着区块链系统每秒能处理的交易笔数，为了能够满足全球用户，交易处理速度必须足够大，否则会造成交易积压，回复时间长。比如 bitcoin 的交易拥堵和以太坊因为 cryptokitties 造成的网络拥堵。
- Latency : 交易从提交到回复成功与否的时间。在 PoW 网络中，因为有分叉可能，一般要等待几个块后确定。在 BFT 共识系统中，因为正确节点都在共识结束后就确定性有相同结果，所以在交易不拥堵的情况下，响应速度与共识时间相同。
- Scalability : 一般默认区块链系统参与的节点数越多，系统安全度越高。对于基于 PoW 共识的系统，安全度与参与的诚实算力相关。对于基于 BFT 共识的系统，参与节点数越多，能容纳的恶意节点越多，恶意节点少于 1/3。但是对于 BFT 系统，节点越多，通信和计算开销越大。Algorand 提出利用 Verifiable Random Function 随机选出一部分节点形成 committee 参与共识，保证了无论总节点数多大，最后的 committee 数目一定，共识速度就不变。
- Security : 对于 BFT 共识，安全度包含 consistence, liveness 和 fairness。Consistence 是系统内所有诚实节点要最终达到相同的状态。Liveness 需要系统在任意情况下都能收敛到确定状态，并且能持续接受交易，产生正确的共识结果。Fairness 在于对于系统的用户，任意合法的交易都不会被拒绝。

对于传统的 PBFT[1, 2] 系统，需要假设网络处于弱同步状态，通过超时和换主来保证 liveness。但是因为换主是确定性地换到预先设定的下一个节点，而且每次换主导致的节点消息同步耗时长，主节点会被连续不断的网络攻击导致瘫痪，最终系统持续换主，永远无法共识交易，相当于停滞状态。对此，Tendermint[3] 提出每一轮共识完成后确定性由下一个节点提出共识请求，从而避免了一直由主节点提交请求，避免了攻击的薄弱点。但是这依然无法避免提交请求节点的可预测性，导致被依次攻击，系统每轮共识都为空。Honey Badger[4] 提出了基于 RBC ( Reliable Broadcast ) 和 BA ( Binary Agreement ) 的协议。核心在于任意节点都可以提出共识消息，通过 RBC 可靠传播到所有节点，并且为了减小广播带宽，通过 erasure code 分割消息为多份。同时所有节点都通过 BA 协议共识所有消息，BA 协议的执行会在任意情况下快速收敛到 1 或者 0，表示对共识消息的接受与否。该协议相当于每一轮，所有节点并行地提出交易、共识，最终得到交易的子集作为共识结果。所以对任意一个节点的攻击，都不会造成整个网络的崩溃，只是会影响网络部分性能。而且能充分利用带宽，适合全球部署。该协议的缺点在于交易响应延时比较高，因为每轮要共识多个节点的交易。同时协议需要预先确定节点的集合，不能动态添加节点，不能支持大规模节点。为了解决节点添加和扩充节点的问题，Algorand[5] 提出将 VRF ( Verifiable Random Function ) [6] 和 BA 结合起来。无论节点数目多少，通过 VRF 和持有的代币数随机选出特定数目节点，然后节点通过 BA 相互发送交易，并对优先级最高 ( VRF 随机数最小 ) 的节点发出的交易共识。为了提高安全性，共识的每一步都通过 VRF 选出新一轮共识节点，从而让攻击者无法预测下一个攻击目标。该协议的主要有点在响应时间短，缺点在于无法做到高吞吐率和小带宽。

Dfinity[7] 主要是通过 threshold 签名来保证 VRF 的每一轮都能确定性地收到 signature。结合 Random Beacon 和 Notaries 来使每一轮的成员都随机选择，并且提交的块按照本轮随机数进行排名。但是一个潜在的经济学博弈问题是，threshold 签名可以通过多个成员的合谋的方法来预测，合谋的成员可以协同计算出群私钥，快速预测出下一轮的随机数。通过作恶 Dos 攻击下一轮的成员的目的，合谋者可以破坏网络的公平性。因为这种攻击是不容易被发现的，所以可以做到零成本收益，因而在现实世界必然会出现。

## 2 贡献

Ultrain 共识具备如下特性

- 引入硬件独特的计算能力评分和历史记录评分，生成无法篡改的硬件指纹，保证参与共识的节点都是高性能和良好信誉的。通过可验证随机函数和硬件指纹、持有代币随机选出共识节点（侧重硬件性能）和验证节点（侧重代币数和信誉评分），提高了系统的安全性和扩展性。
- 通过利用拜占庭共识过程中的并行性，提高了共识的效率，同时保证在少数共识提交者被攻击的时候，网络依然保持活性。
- 通过优化密码学签名验签模块，在共识过程采用轻量级的密码学模块 [8, 9]，同时持久化高安全度的密码学结果，提高了节点响应速度。
- 通过冗余编码 [10] 将消息划分为多份传递，而不是一份消息广播，保证了节点利用有限的网络带宽广播最大的消息数量，提高共识吞吐率。
- 通过阈值签名 [11]，将多个节点编成一个组，提高了节点的活性和响应速度，同时提高了公平性。
- 通过阈值加密 [12]，保证了每个节点在收到足够多消息之前无法知道消息的内容，所以没法做到有倾向地传递共识消息，提高了公平性。

## 3 名词解释

### 3.1 签名验证

Ultrain 选择 ED25519[13] 作为对消息的签名，主要是基于如下考虑

- ED25519 是开源的密码学算法，由 Daniel Bernstein 提出，经过众多密码学专家论证，最后入选 RFC7748 标准。曾经参与过同微软 FourQ[14] 共同竞争下一代更高效更抗侧信道攻击的椭圆曲线算法标准，后因 NIST 直接进入后量子算法的遴选而取消。
- ED25519 有比较高效的开源算法实现，并且汇编实现支持 Intel AVX 指令集。
- ED25519 相对基于 Weistrass 曲线的 NIST P-256 曲线，性能有比较大的提升。

在接下来的数学表达中，

- 签名标记为： $SIG_{sk}(m) = ED25519_{sig}(sk, m)$
- 验证标记为： $VER_{pk}(SIG_{sk}(m), m) = ED25519_{ver}(pk, SIG_{sk}(m), m)$

其中  $(sk, pk)$  为一对私钥、公钥。

### 3.2 摘要函数

Ultrain 采取 SHA256 作为摘要函数，基于如下考虑：

- 虽然 SHA3 标准已经提出，但是目前尚没有明显针对 SHA256 的攻击。
- SHA256 性能相对 SHA3 比较高，而且有众多比较成熟的开源实现。
- SHA256 相对 SHA3 或者其他摘要算法，有更多市场可用的实现。

在接下来的数学表达中，摘要函数标记为： $SHA(m) : hash = SHA256(m)$

### 3.3 可验证随机函数

可验证随机函数要求函数满足如下特性：

- Deterministic, 一旦有随机数被引入, 节点即可不断尝试新的随机数, 直到 VRF 的结果对己方有利为止。
- Non-malleable, 一旦签名结果可被变换为不同的, 那么用户可以通过修改签名, 达到想要的结果。

在接下来的数学表达中, 标记为:  $proof = VRF_{sk}(seed)$

VRF 算法目前应用在系统随机数生成, Layer1 为 dapp 提供的随机源和 Layer2 为链下状态通道提供的随机源中。

### 3.4 后量子算法

量子计算机主要对密码学中的非对称算法和密钥交换影响较大, 比如 RSA 基于的大数分解困难问题可用 Schor 算法高效解出, 同理离散对数问题也可以解决 (如 GEECM)。然而对于对称算法和单向函数影响不大, 只需要为了增加安全性到两倍即可即可。目前 Ultrain 不以后量子算法为主, 基于如下考虑:

- 量子计算机的发展仍然有相当的时间长度 [15], 一些关键性的问题如多个量子的叠加、量子纠错。
- 目前抗量子算法如密钥交换和非对称算法标准尚不成熟, NIST 第一轮的筛选刚刚完成。
- 抗量子算法目前有基于 LWE[16] 和 code[17] 的算法, 效率普遍比较低, 而且安全性的量化评估目前尚无定论 (如 LWE 基于的 CVP 和 SVP 问题一直有效率更高的算法被提出 [18])。
- 传统算法基于的困难问题, 如基于 pairing 的 ABE[19]/IBE[20]、零知识证明 [21], 在抗量子的困难问题基础上构造比较困难, 或者比较难做到高性能。
- 区块链上的数据到后量子的迁移并没有很大难度。主要是因为足够多的时间, 从非抗量子的公钥钱包迁移到抗量子的钱包。
- Merkle tree、存证等基于的 SHA 算法即使安全度降低, 被碰撞的难度依然比较大。

Ultrain 的抗量子钱包, 通过结合最新的 NIST 征集的抗量子加密算法, 并通过多重签名技术达到足够的安全性。并且针对抗量子算法的缺点又以下解决方案

- 针对目前抗量子算法是否真正抗量子的的问题, 即使 NIST 尚无定论, 所以我们采用多个算法 (Crystals-dilithium, Spincs+, Rainbow) + 多重签名的方式。这样即使其中一个算法被攻破, 不至于影响钱包的安全性。
- 针对公钥长度太大的问题, 目前消息传递采用账户名的方式, 所以只是多占用用户的存储资源, 不会带来交易的带宽消耗。
- 针对签名长度太大的问题, 会造成交易大小的增大, 影响系统的 tps, 消耗用户的带宽资源。所以目前只能依赖密码学的进一步发展, 同时建议用户只是用在抵押账户或者不经常使用的存储账户上。
- 针对多密钥的管理问题, 我们提供安全的密钥派生钱包, 用户只需要存储一个密钥, 即可管理所有抗量子的钱包, 并且不会因为一个账户被破解而影响其他任何一个账户的安全性, 用户体验不会受到任何影响。

### 3.5 BLS 算法

采用 BLS 算法，可以对签名结果进行聚合，从而减小传递的消息大小。Ultrairn 目前将此算法此算法应用在 Vote 消息中，减小 Vote 的消息大小，并相应地减小每个块证书的大小，减小侧链向主链传递的跨链消息大小。同时小的块证书大小，也对轻客户端的验证带宽和计算负载比较小。

### 3.6 区块

区块由如下结构构成

- 块高度 bheight, 简写为 bh
- 上一个块的 hash, 定义为  $hprev_{bh}$ 。
- 块包含的交易的 merkle 根  $hroot_{bh}$ 。
- 交易执行结果的 merkle 状态树根, 即  $hstate_{bh}$ 。
- 交易的 Proposers 的 merkle 树根, 即  $hProposer_{bh}$ 。

总的区块格式为

$$(bh, hprev_{bh}, hroot_{bh}, hstate_{bh}, hProposer_{bh}) \quad (1)$$

考虑到创世区块比较特殊，我们定义为

$$(0, 0, hroot_0, hstate_0, 0) \quad (2)$$

其中  $hroot_0$  包含一个创世合约，其中约束了 ultrairn 的规则。 $hstate_0$  包含创世成员的身份和代币分布。

### 3.7 节点身份

在区块链系统运行过程中，任意一个节点属于如下一种身份：

- Proposer: 共识每轮提起交易批
- Voter: 对交易进行确认，发出投票同意打包请求
- Listener : 对消息进行传递，收到足够票数后执行

### 3.8 消息类型

区块链系统运行过程中，主要包含如下两种消息：

- PROPOSE 消息：提议共识中需要包含的交易，由 Proposer ( 对应密钥对为  $(skp, pkp)$  ) 生成，格式为

$$(PROPOSE, shard, txs, bh, round, txhash, proof_{pkp}, SIG_{skp}, pkp) \quad (3)$$

其中  $shard$  为分片,  $txs$  为交易批,  $txhash = SHA(txs)$ ,  $SIG_{skp} = SIG_{skp}(bh, round, txhash)$ ,  $proof_{pkp}$  为节点被选为 Proposer 的证明。

- ECHO 消息：投票消息，由 Voter ( 对应密钥对为  $(skv, pkv)$  ) 生成，格式为

$$(ECHO, shard, bh, round, txhash, proof_{pkp}, SIG_{skv}, pkp, proof_{pkv}, SIG_{skv}, pkv) \quad (4)$$

其中  $SIG_{skv} = SIG_{skv}(bh, round, txhash)$ ,  $proof_{pkv}$  为 Voter 被选中的证明，其余同上。

## 4 架构

Ultrain 每一轮，分为如下几步完成

1. 通过结合硬件指纹和代币，在每一轮共识开始，利用可验证随机函数 (VRF) 选出一定数目的 Proposer 和 Voter。
2. 由 Proposer 提议需要成块的消息，由 Voter 对选中的唯一提议消息相互发送确认。
3. 节点对交易批进行执行，再利用 VRF 选出一定数量 Voter，通过 BA (binary agreement) 协议相互同步对交易批的执行后状态。
4. 节点对交易批和执行后状态独立打包成块。

为了达到并行执行和节省节点资源消耗的目的，Ultrain 采用 sharding 技术随机将节点分到不同 shard 和跨 shard 的区中，使节点在处理本 shard 的时候，从其他 shard 异步接受状态，最后将包含所有 shard 的信息打包成块。

### 4.1 身份层

每个自愿参与共识的节点都有一对密钥 (sk,pk) 作为身份标识。设备被选为共识成员的概率依赖以下三个评分的综合结果。

- 链上代币。所有公钥对应的资产在链上都可以查询到，所以持有代币的百分比作为  $w_{coin}$ 。有的节点可能持有大量代币，但是不愿参与共识，可以代理给其他可靠的硬件厂商对应账户，类似 Dpos。或者持有大量代币的节点可能同时分散在多个账户，这不会影响到公平性，因为多个账户的综合被选中概率与一个账户相同（假设其他两个账户对应的硬件设备指标相同）。
- 硬件特性。根据自愿参与共识的设备硬件指纹信息可以得出硬件特性评分。该评分可能有多个维度，比如 CPU/网卡/硬盘特性，分别适合 committee 中不同的身份。该权重暂定为  $w_{hardware}$ 。
- 声誉。根据节点账户对代币的持有寿命（块高度）、设备历史参与记录、成功提交区块的次数，系统自动计算出节点信誉。该权重为  $w_{reputation}$ 。

每个公钥对应一个权重  $w = r_1 * w_{coin} + r_2 * w_{hardware} + r_3 * w_{reputation}$ ，其中  $r_1, r_2, r_3$  为系统常量，用来设置各个权重的比率。

### 4.2 VRF 层

我们选择独立的 VRF，而不是相对类似 RANDAO 或者 Dfinity 的方式。基于如下两点：

- 该方式，只要有足够多的节点联合起来，或者一个控制了大量代币的用户，只要超过了阈值，那么就可以预测下一轮的随机数，这样可以有时间窗口去 DDOS 攻击下一轮的提议者，以此影响系统的公正性。
- 该方式延时比较高，对于其中一个节点，需要足够多节点响应，才能确定随机数结果。
- 该方式比较高效的非交互式算法，只有基于椭圆曲线 pairing 的 BLS 算法，该算法相对一般椭圆曲线运算如点乘，性能比较低。其他如 RSA 算法需要节点交互。

为了保证 VRF 层的正确性，Ultrain 的 VRF 函数进行一定的设计，满足如下特性

- 每一个持有代币数为  $s$  的节点，通过 VRF 的选举机制，被选中的概率为  $p(s)$ 。节点不能控制 VRF 的结果，而且 VRF 每轮的结果必须是唯一，所以节点每次计算 VRF，类似于对 Random Oracle 的访问。

- 对于任意数  $s_x, s_y$ ,  $p(s_x) + p(s_y) = p(s_x + s_y)$ 。该设计是为了保证公平性，即一个代币持有者通过拆分为多个持有少量代币的身份，获取更大的被选中的概率；同时这也会增加系统的安全性，否则持有小于  $1/3$  代币的人可以通过合谋获取大于  $1/3$  的投票概率，从而影响系统的安全性。
- 系统可以通过静态或者动态的参数设置，控制被选中的节点数。

一个比较直接的例子是  $p(s) = B^{-1}(r; s, \lambda)$ ，其中  $s$  为节点持有代币数， $\lambda \in [0, 1]$  为系统选中节点比例， $r \in [0, 1]$  为随机数， $B$  为二项分布。假设  $r$  是均匀的  $[0, 1]$  概率分布，那么期望  $E(p(s)) = s * \lambda$ 。

因为， $B(v_1 : s_1, \lambda) + B(v_2 : s_2, \lambda) = B(v_1 + v_2 : s_1 + s_2, \lambda)$ ，所以将同一个节点的代币  $s$  拆分为两个节点  $s_1$  和  $s_2$ ，得到的投票权期望分别是  $E(f(s_1)) = s_1 * \lambda$  和  $E(f(s_2)) = s_2 * \lambda$ ，总的为  $s_1 * \lambda + s_2 * \lambda = s * \lambda$ ，和一个节点单独采样是一样的；同理，所有节点总的票数为  $\sum s * \lambda$ 。

## 4.3 共识层

### 4.3.1 分片

考虑到共识过程中，除了 PROPOSE 消息外，ECHO 消息比较小，但是占用时间长，为了不浪费带宽，Ultrain 共识同时存在 16 个 shard，每个 shard，包含不同的交易类型。我们设置了固定 16 个 shard，每个 shard 只选最小 proof 的 Proposer，多个 shard 并行共识，最后一起执行。

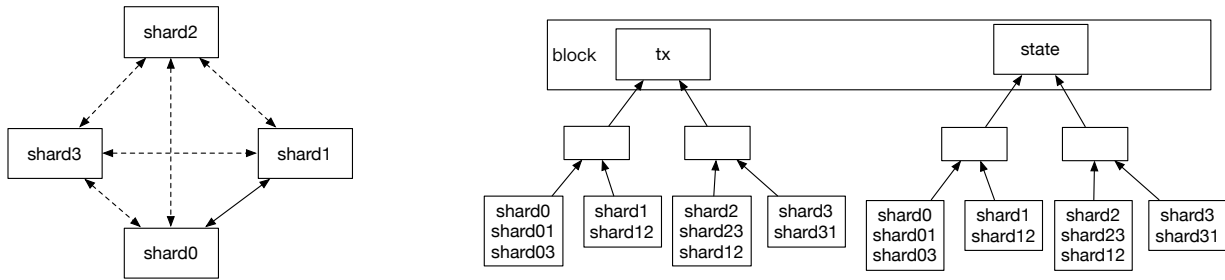


Figure 1: 存储和执行的 shard

对于存储和执行的 shard 我们有以下措施：

- 对于只参与  $shard_i$  的节点，可能无法对其他  $shard$  的状态进行验证，这时只需要监听网络中的消息，对其他  $shard$  的共识执行结果接受即可。
- 当然假如是跨 shard 执行的合约，我们将  $C_N^2$  中跨 shard 执行交易平均分配给网络中不同的 shard，因而每个 shard 执行本 shard 和其余  $C_N^2/N$  个跨 shard 的交易，这样一个节点需要存储约  $((C_N^2)/N + N)/N$  个 shard 的状态。
- 不允许一个交易跨多个 shard。即使支持，这种交易需要在多个共识周期才能完成，每个共识周期完成一次 shard 的跨越。为了方便使用，基本的合约应该在每个 shard 中包含，尽量避免跨 shard 调用。

用户随机被分配到某个的 shard 中，假如可以随意选 shard 的话，网络可能会被一个持有少量代币的用户集中攻击一个 shard[22]。下面只针对一个 shard 中的行为进行描述。

### 4.3.2 延时

传统的 BFT 系统采取两种方式：

- (tx 执行 + 提议 + tx 验证 + BA)，这样延时比较大，因为对执行结果的验证相当于重复执行，总延时包含了两次执行过程。好处在于可以避免打包非法交易进入区块。一般基于 PoW 的公链采取这种形式。



- (tx 提议 + BA+tx 执行) 采取共识后执行的方法，避免不了打包非法交易。同时也对智能合约的 deterministic 有比较强的要求。
- (tx 部分执行 + 共识排序 + 共识状态验证和转移) 这是 hyperledger fabric 的方式，好处在于将 non-deterministic 的部分放在共识前，将 deterministic 的验证签名和状态转移，同时可以并行执行无依赖的状态转移。缺点在于，节点不对执行过程进行验证，依赖受状态影响的各方签名校验。

为了减小节点的延时，我们考虑如下几种方式：

- 多轮 tx 共识 + 一轮执行结果共识，这样或许可以将多轮的 tx 批量执行，效率有一定程度的提高。诚实的客户端，只需要收到 tx 被打包进区块的收据，就可以对执行结果有足够的信心，不必等到执行结果被打包成块。
- Tx 共识 + 上一轮的执行结果共识。流水线处理，共识和执行相对独立，执行落后共识一个成块周期。不会减少延时，但是可以做到在同一时间点上可以并行处理共识和执行。缺点在于，会将非法交易打包成块，浪费区块存储。
- (tx 提议 + voter 执行 + voter 以执行结果 hash 响应 + 对执行结果投票)，延时也可以做到最小。优点在于可以避免打包非法交易，同时识别出哪些交易批没有 deterministic 地执行，因为这些交易批收集不到足够多的票数。从经济学博弈的角度，提议者为了获得激励，不会故意提交 non-deterministic 的程序。

我们的共识采用第三种方式，主要是考虑到方便上层智能合约的设计。

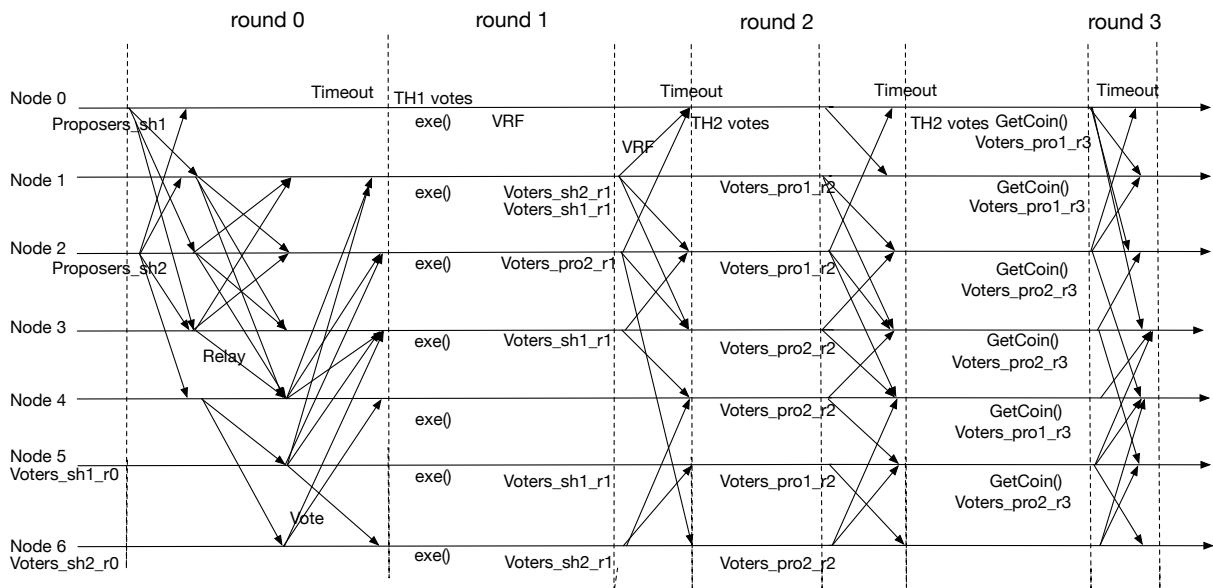


Figure 2: consensus overview

- propose 轮。节点基于上一个 block，独立判断是否被选为 Proposer 或 Voter。该轮保证消息确定性地由 Proposer 到全网大部分节点。
  - Proposer 在广播 PROPOSE 消息的同时为了加速传播，广播带交易 hash 的 ECHO。为了最小化带宽消耗，PROPOSE 消息被冗余编码后，分别传给多个节点。每个 Proposer 只能广播一次，多次广播被视为恶意节点。
  - Voter 等待  $T_{PROP}$  时间后，对收到 PROPOSE 消息校验格式，选择 proof 最小的广播相应 ECHO 消息。

- 节点收到  $TH_1$  个消息后，即使在未收到 PROPOSE 消息的情况下，也广播 ECHO 消息。
- 节点收到  $TH_2$  个 ECHO 后完成此轮，返回。
- 节点等待  $TIMEOUT$  未收集足够数量 ECHO 后，判断超时，将交易批设置为空。
- 执行，所有节点对上一轮的结果执行，并基于原状态  $state_1$  计算输出状态的摘要  $state_2$ 。
- BA 轮，全网基于上一个 block 和轮数，独立判断是否被选为 Voter。该轮保证大部分节点对执行结果的摘要接受与否达成一致。
  - Voter 广播包含  $state_2$  的 ECHO 消息。
  - 节点收到  $TH_2$  个 ECHO 后完成此轮。超时则成空提议，返回。
  - 若上轮超时，则新开始一轮，重新判断是否为 voter，基于 CommonCoin 决定是否提起包含  $state_2$  还是空交易批的 ECHO 消息，然后广播。
  - 节点收到  $TH_2$  个 ECHO 后完成此轮。否则继续利用 CommonCoin 判断下一轮的 ECHO 消息内容。
  - BA 中每一轮时间固定为  $T_{Bar}$ ，节点判断超时后返回空。若超出特定 BA 轮数，则节点将状态回滚到  $state_1$ ，该轮包含空交易批。
- 成块
  - 节点利用本 shard 的执行结果状态  $state_2$  结合来自其他 shard 的执行结果，成块。

在整个共识过程中，Listener 会接收到跟 Voter 和 Proposer 同样的消息，所以除了传递消息外，Listener 同样会校验、处理消息、成块。

节点对不同 shard 的共识输出打包成块。对于只参与部分 shard 的节点，可以先打包成块，然后再向其他节点拉取其他 shard 的交易批次。具体实现过程中，节点会在条件满足的情况下，提

前为下一轮做准备，同时为了提升效率，对已经满足的条件做忽略处理。具体算法描述如下：

---

**Algorithm 2:** Ultrain 节点操作

---

**Data:** 初始代币分布  $I_{stake}$ , 成员列表  $I_{member}$

**Result:** 区块链状态

```

1 Initialization
2    $bh = 1$  /* 初始块高度 */
3    $state_{bh}, block_0 = (I_{stake}, I_{member})$  /* 初始状态和块 */
4    $N_{shard} = 16$  /* shard 个数 */
5    $(sk_i, pk_i) = KeyGen()$  /* 节点身份 */
6    $queue_{txs} = set_{EMPTY}$  /* 初始交易队列为空 */

7 Upon incoming transaction  $tx$ 
8   /* 检查交易消息的签名和格式 */
9   if  $check(tx)$  then
10  |    $queue_{txs} \cup = tx$ 

11 Upon start consensus
12  /* last block */
13   $round = 0$ ;
14  /* 提议轮 propose, 最后获取投票结果和交易批 hash, 以及交易批 */
15   $res, hbatchtx, batchtx =$ 
     $ROUND(PROPOSE, hblock_{bh-1}, shard, queue_{shard1}, block_{r-1}, round)$ 

16  /* BA */
17   $round++$ ;
18  if  $res == TIMEOUT$  then
19  |   /* 若上轮收集不到足够选票, 超时, 则提议空交易批 */
20  |    $res = ROUND(ECHO, shard, set_{EMPTY}, block_{bh-1}, round)$ 
21  else
22  |   /* 保存上一个状态  $state_{bh-1}$  */
23  |    $save(state_{bh-1})$ 
24  |   /* 执行交易批  $batchtx$ , 获得新状态, 计算摘要为  $hstate_{bh}$  */
25  |    $hstate_{bh} = exe(state_{bh-1}, batchtx)$ 
26  |    $res = ROUND(ECHO, shard, hbatchtx, block_{bh-1}, round)$ 
27  while  $res == TIMEOUT \& round < MAX_r$  do
28  |   /* at most  $MAX_r = 7$  rounds */
29  |    $round++$ 
30  |    $CC = SHA(res \parallel hblock_{r-1} \parallel round)$ 
31  |   /* 取 CC 的最低位来决策下一步动作 */
32  |   if  $LSB(CC) == 1$  then
33  |   |    $res = ROUND(ECHO, shard, hbatchtx, block_{bh-1}, round)$ 
34  |   else
35  |   |    $res = ROUND(ECHO, shard, set_{EMPTY}, block_{bh-1}, round)$ 
36  /* 收集其他 shards 的 tx root and state root, 构建块 */
    $build\_block(state_{bh}, hstate_{othershards}, hbatchtx_{othershards})$ 

```

---

节点每轮对消息的响应如下：

---

**Algorithm 3:** Ultrain 节点轮操作

---

**Data:** *Type, shard*, 交易批 *batchtx*, 块摘要 *block<sub>bh-1</sub>*, *round*

**Result:** 返回值 *res*, 交易批摘要 *hbatchtx*, 交易批 *batchtx*

```

1 初始化
2  setECHO = empty /* 响应 hbatchtx */
3  settxs = empty /* HASH(batchtxs) : batchtxs */
4  sentECHO = empty /* sent ECHO HASH(batchtxs) : batchtxs */
5  Upon Start
6  start timing
7  proof = VRFsk(blockprev || round || role)
8  /* 由 proof 的低 4 位判断被选为哪个 shard */
9  shard = proof[3 : 0]
10 if proof[193 : 256] < score /* 判断是否被选中 */
11 then
12     sig = SIGsk(blockbh-1, round, hbatchtxs)
13     if round! = 0 /* BA */
14     then
15         broadcast (ECHO, shard, height, round, batchtxs, proof, sig, pk)
16         break
17     hbatchtxs = SHA(batchtxs)
18     broadcast (PROPOSE, shard, height, round, hbatchtxs, batchtxs, proof, sig, pk)
19     /* Proposer 默认也为 voter, 发送 ECHO 加速消息传播 */
20     broadcast (ECHO, shard, height, round, hbatchtxs, proof, sig, pk)
21 Upon receive PROPOSE
22 if HASH(PROPOSE.batchtxs)! = PROPOSE.hbatchtxs then
23     break
24     /* 是否收到该 ECHO */
25 if PROPOSE.pk ∈ setECHO then
26     settxs[PROPOSE.hbatchtxs] = PROPOSE.batchtxs
27     return 1
28 if receive_ECHO(PROPOSE \ PROPOSE.batchtxs) then
29     settxs[PROPOSE.hbatchtxs] = PROPOSE.batchtxs
30 Upon receive ECHO
31 if VER(msg) then
32     setECHO[ECHO.hbatchtxs] ∪ = ECHO.pk
33     if count(setECHO[ECHO.hbatchtxs]) == TH2 then
34         if round! = 0 || ECHO.hbatchtxs ∈ index(settxs) then
35             return hbatchtxs
36             /* wait for some time to get the batchtxs */
37         if count(setECHO[ECHO.hbatchtxs]) == TH1 then
38             if not ECHO.hbatchtxs ∈ sentECHO then
39                 add_sig_then_send(ECHO, ECHO = {ECHO, hbatchtxs, sigmsg, proof, pk}), sk,
40                 pk)
41                 sentECHO ∪ = ECHO.hbatchtxs
41 Upon TIMEOUT
42 return TIMEOUT

```

---

---

**Algorithm 4: 消息传递**

---

**Data:**  $msg_{in}, sk, pk$   
1 /\* 在原消息添加节点签名 \*/  
2  $broadcast(Type, msg_{in}.msg, msg_{in}.sig \cup SIG(msg_{in}.msg, sk), msg_{in}.proof, msg.pk \cup pk)$

---

**Algorithm 5: 消息验证**

---

**Data:**  $msg_{in}, sk, pk, block_{prev}, shard, role$   
**Result:** 验证结果  $res$   
1 /\* 可并行验证 \*/  
2 **if**  $msg_{in}.proof[193 : 256] \geq score$  **then**  
3 | **return** 0  
4 /\* 通过批验证加速 \*/  
5 **if**  $!VER_{msg_{in}.pk}(proof, block_{prev} \parallel shard \parallel role)$  **then**  
6 | **return** 0  
7 **if**  $!VER_{msg_{in}.pk}(msg_{in}.sig, msg_{in}.msg)$  **then**  
8 | **return** 0  
9 **return** 1

---

证书的保存对新加入的节点同步，有比较大的影响。我们考虑采用如下方式：

- 聚合签名，将多个消息的签名通过类似 BLS 签名的聚合方式 [23]，以  $O(1)$  的存储空间保存所有签名。
- 采用 DAG 的方法，任意一个节点，都要对现存的两个 signature 进行验证。然后只要锁定了 DAG 的根，那么就可以对证书进行确定。

此外，为了实现 sharding，p2p 底层网络也需要修改，在我们的协议中，节点只需要接收被选中 shard 中的消息，对其他 shard 的共识结果被动接受。

## 4.4 新节点加入

新节点需要经过如下几个步骤：

- 观察并接收网络正在传播的消息，确定块高度和共识的轮数，当前块共识结果。为了防止被恶意节点通过网络控制的方式攻击，新节点需要尽量从多个网络地址监听共识消息。
- 根据当前成块，向网络中其他节点拉取历史区块，并验证链的正确性。考虑到链的单向性，新节点可以保证历史区块的正确性。

## 5 安全性证明

### 5.1 定理

**Theorem 5.1 (FLP 不可能性)** 异步系统下，即使只有一个节点崩溃或者作恶，不可能构造在确定有限时间内的共识算法。

**Theorem 5.2 (容错上界)** 对于部分同步网络，最多容忍  $1/3$  的恶意节点。异步网络，确定性共识算法不能容错。强同步网络可以达到 100% 容错。

**Theorem 5.3 (CAP 定理)** 对于网络分区，无法同时保证一致性和可用性。

## 5.2 安全假设

**Assumption 5.4 (网络连通性)** 网络不会被超过  $1/3$  的恶意群体控制，任意两点的消息传播通过 *gossip* 协议，保证消息在确定时间范围内最终大概率到达。同时消息的源头不会被轻易识别出来，被攻击。

**Assumption 5.5 (节点恶意行为)** 网络上的节点可以发送任意恶意消息或者串谋、拒绝响应，只要总持币数不会被  $1/3$  的恶意节点控制，那么区块链系统的安全性和活性就能得到保证。

**Assumption 5.6 (VRF 算法)** *VRF* 算法保证，持有比例为  $x$  的代币，大概率不会产生超过比例  $x$  的投票权。

**Assumption 5.7 (弱同步时钟)** 各个节点依赖本地时钟和网络消息判断何时进入下一轮。不同节点的本地时钟可能存在差异，需要定时跟网络时钟服务（如微软）进行矫正。我们的每轮是固定时间宽度，考虑了消息在不同网络拓扑下的传播时间分布和节点处理消息的时间。

**Assumption 5.8 (恶意节点数)** 假设网络上不可能有两个相同的集合，每个集合包含了相同的签名；即假设了足够的恶意节点持有的代币数。

## 5.3 推论

虽然拜占庭共识能保证全网系统的最终一致性，但是因为在公网上，情况比较特殊：

- 为了扩展性的要求，考虑的是通过委员会 + 阈值的方式，而不是传统的每个节点接收到其他全部节点的消息后做响应。
- 节点间的信息传输非点对点传输，而是通过公网 *gossip* 方式传播，恶意节点可能会将消息只广播到部分网络分区，能在超时范围内，影响该网络其他节点的接收消息的完整度。

恶意节点可能会有以下几种攻击的可能性：

- Case1: 作为 Proposer，有意向一部分网络节点发送 PROPOSE 消息，向另外一部份节点延迟发送，或者根本不发送。
- Case2: 作为 Voter，有意向一部分网络节点发送 ECHO 消息，向另外一部份节点延迟发送，或者根本不发送。

**Corollary 5.8.1** *Case1* 不会影响区块链的一致性和活性。

证明：VRF 保证了 voter 分布的不可预测性和随机性，所以假如 PROPOSE 消息没有大于 TH1 的网络上传播，大部分节点将无法在足够的超时范围内收到足量 ECHO 消息，从而以空交易批的方式进入第二轮；对于少部分能在超时范围内接收到 ECHO 消息的节点，在第二轮将无法得到足够多的对应交易 hash 的 ECHO 消息，而会在第三轮收到空交易批的 ECHO 消息，所以最后所有节点都会以空的交易批结束。网络的一致性得到保证。同理，假如 Proposer 对不同的网络分区广播不同的 PROPOSE 消息，也会导致所有消息都接收不到足量的 ECHO 消息，最终所有节点以空交易批结束。活性：VRF 会选足够多的 Proposer，并且有不同的 shard，另外引入公平的随机性，从而保证所有 Proposer 都是恶意节点的概率非常小，系统会大概率获取一定的交易批次。

**Corollary 5.8.2** *Case2* 不会影响节点的一致性和活性。

证明：voter 在 case2 主要是干扰网络的一致性，VRF 保证了 voter 分布的不可预测性和随机性，所以其实 voter 无法确定其他 voter 的位置，假设全部 voter 在网络的分部是均匀的，恶意 voter 很难通过只广播小部分分区，影响大部分 voter 的目的。即使假定 voter 能控制足够量其他 voter 接收的消息，从而控制分区 1 超过阈值收敛为交易批，分区 2 超时，但是因为我们超时后的行为是随机的，继续提交交易批，或者提交空交易批，所以这种不可预测的行为让 voter 无法控制网络，最后超出固定轮数后，所有 voter 最后以空交易批结束。但是，这属于极端情况，因为恶意节点不能一直作为 voter 存在，所以最后网络还是不受恶意节点控制。

**Corollary 5.8.3** 强同步网络下, *Ultrain* 共识第二轮结束后即完成; 半同步网络下, *Ultrain* 需要多轮可以大概率收敛。

证明: 在强同步网络下, 任何节点在超时范围内, 都能收到其他节点发过来的消息。基于假设, 网络恶意投票数不会超过  $1/3$ , 所以所有节点在第一轮都可以收到所有的共识提议, 无论是恶意还是诚实节点发送的。在第二轮过程中, 只有诚实节点的提议会收到足够多的票数, 所有节点都确定性地执行, 并达到相同的结果。在半同步网络下, 所有节点依然可以保持确定时间收敛和一致性。

**Corollary 5.8.4** *Ultrain* 可以防御女巫攻击。

女巫攻击主要是利用系统的不公平性, 为某一实体谋取利润的方式。我们有如下防御方式

- 针对利用大量的假名, 构造多个伪身份对应一个真实身份的攻击, *Ultrain* 采用硬件指纹的方式, 唯一地对网络上的代币持有者进行标示。
- *Ultrain* 采用 VRF 函数, 保证即使用户拆分为多个伪身份, 依然不能产生超出总持有代币的投票权, 所以不能进行女巫攻击。

**Corollary 5.8.5** *Ultrain* 可以抵御 *Selfish-mining* 攻击

*Ultrain* 基于的 BFT 系统是保证所有节点的一致性, 所以即使外部节点独立生成一条链, 也不会被网络诚实节点接受, 从而无法影响节点一致性。同时为了保证节点

**Corollary 5.8.6** *Ultrain* 可以抵御 *Nothing-at-stake* 攻击

*Ultrain* 中的见证节点相当于投票节点, 每个代币持有者的投票能力与代币成正比, 低于  $1/3$  的持有者不会有能力产生分叉。通过硬件指纹的唯一性, *Ultrain* 也会对双重投票的节点进行一定比例的惩罚。

**Corollary 5.8.7** *Ultrain* 可以抵御 *DoS* 和 *DDoS* 攻击

DDOS 是通过消耗资源使被攻击者的服务机不工作, DOS 是直接利用对方的漏洞。即使共识代码出现了一定的 bug, 系统被攻击崩溃也是有一定难度的, 主要基于如下几点:

- 攻击成本是  $o(n^2)$ , 所以同时攻击多个节点的成本是很高的。
- 基于 gossip 的话更难被锁定信号源进行攻击。
- 假如是放在 sandbox, 签名放在外面的话, 或者用 sgx, 那么即使实现有 bug 不会影响到单个节点的安全性, 比如是代币被花; 应该采用不同的 vote key 和 tx sig key; sandbox 不会让系统其他模块被污染, 所以我们需要隔离好用户的钱包和共识代码, 特别是针对 PoS 场景, 不能采用同样的账户。
- 假如是有足够的时间窗口, 能快速启动的话, 那么能保证节点快速启动, 不会影响到整个系统的 liveness。

网络分区将导致区块链系统的可用性和一致性不能同时满足。一般情况, 区块链系统都选择中止 (比如基于), 或者分叉 (比如比特币)。为了及时发现分区, 我们的 VRF 函数考虑了在不同地域的采样, 比如要求 voter 必须在不同的国家内有一定的比例。因为我们不想在中国和美国都出现脑裂的情况, 系统也出现脑裂。

## References

- [1] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- [2] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.
- [3] Jae Kwon. Tendermint: Consensus without mining. URL [http://tendermint.com/docs/tendermint {\\_} v04. pdf](http://tendermint.com/docs/tendermint_{_} v04. pdf), 2014.
- [4] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.
- [5] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. SOSP, 2017.
- [6] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.
- [7] Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series consensus system, 2018.
- [8] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. Spongint: A lightweight hash function. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 312–325. Springer, 2011.
- [9] Jean-Philippe Aumasson, Luca Henzen, Willi Meier, and María Naya-Plasencia. Quark: A lightweight hash. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 1–15. Springer, 2010.
- [10] Yong Wang, Sushant Jain, Margaret Martonosi, and Kevin Fall. Erasure-coding based routing for opportunistic networks. In *Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 229–236. ACM, 2005.
- [11] Alexandra Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *International Workshop on Public Key Cryptography*, pages 31–46. Springer, 2003.
- [12] Dan Boneh, Xavier Boyen, and Shai Halevi. Chosen ciphertext secure public key threshold encryption without random oracles. In *Cryptographers’ Track at the RSA Conference*, pages 226–243. Springer, 2006.
- [13] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.
- [14] Patrick Longa. Four and four lib.
- [15] Scott Aaronson. The limits of quantum computers. *Scientific American*, 298(3):62–69, 2008.
- [16] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange-a new hope. In *USENIX Security Symposium*, pages 327–343, 2016.



- [17] Pierre-Louis Cayrel and Mohammed Meziani. Post-quantum cryptography: code-based signatures. In *Advances in Computer Science and Information Technology*, pages 82–99. Springer, 2010.
- [18] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Cryptographers’ Track at the RSA Conference*, pages 319–339. Springer, 2011.
- [19] Brent Waters. Ciphertext-policy attribute-based encryption: An expressive, efficient, and provably secure realization. In *International Workshop on Public Key Cryptography*, pages 53–70. Springer, 2011.
- [20] Dan Boneh and Matthew Franklin. Identity-based encryption from the weil pairing. *SIAM journal on computing*, 32(3):586–615, 2003.
- [21] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 238–252. IEEE, 2013.
- [22] <http://www.mangoresearch.co/1-shard-attack-explained-ethereum-sharding-contd/>.
- [23] Dan Boneh, Craig Gentry, Ben Lynn, Hovav Shacham, et al. A survey of two signature aggregation techniques. *RSA cryptobytes*, 6(2):1–10, 2003.