

说明书

发明名称: 实现二层网络卷叠的方法、二层网络和证明器

[0001] 本申请要求于2023年11月16日提交中国国家知识产权局、申请号为202311538590.9、申请名称为“实现二层网络卷叠的方法、二层网络和证明器”的中国专利申请的优先权，其全部内容通过引用结合在本申请中。

技术领域

[0002] 本说明书实施例属于区块链技术领域，尤其涉及一种实现二层网络卷叠的方法、二层网络和证明器。

背景技术

[0003] 分布式系统中存在经典的CAP定理——Consistency(一致性)、Availability(可用性)、Partition tolerance(分区容错性)，三者不可兼得，简称“不可能三角”问题。区块链中也存在一个不可能三角：效率、去中心化、安全。尽管这里的区块链“不可能三角”还没有明确的理论论证，但是是对现有区块链的总结。这里的效率、去中心化和安全，分别定义如下：

[0004] 效率：每秒处理交易的数量，即TPS(Transaction per Second)。

[0005] 去中心化：参与节点门槛足够低，保证系统中有大量分布式的节点。

[0006] 安全：对区块链发动攻击的难度足够大。

[0007] 针对上述区块链“不可能三角”的问题，Ethereum(也就是以太坊，也称为“主网”或“一层”，Mainnet或Layer1；此外，目前较为流行的还有币安的BSC、波场TRON等的公链项目，也是Layer1/主网的范畴)选择了安全性和去中心化，牺牲了效率，目前只有大约12~15TPS。当有大量交易需要处理时，尤其是很多操作复杂的交易时，Layer1就会拥堵。除了普通的转账交易外，Layer1还作为DeFi(去中心化金融)和NFT(同质化代币)等火爆应用的主要平台，在交易盛行时，Layer1拥堵的问题就变得十分严重，带来较大的交易延迟，以致于影响到用户的使用。

[0008] 此外，高昂的Gas fee(汽油费，或交易费)很多时候成为交易的障碍。尽管已从PoW(工作量证明)转向了PoS(权益证明)，但这只是改变了区块链上记账权的获取

方式，而需要支付给获得记账权的节点的交易费没有明显减少。这是因为Gas费的设计针对的是节点执行交易(包括执行智能合约代码)的消耗，这部分消耗不改变的情况下Gas费并不会降低。例如，一笔去中心化交易所的货币兑换交易的交易费在拥堵时可能会超过100美元，这让很多用户望而却步。

[0009] Layer 2技术是构建在Layer1之上的扩展性解决方案，旨在解决Layer1的效率低和交易费用高的问题。如图1所示，交易可以在Layer2上快速执行，Layer2按照一定的时机将最终状态同步回Layer1。Layer2用于专门提供高速的交易处理，而安全性和去中心化交由Layer1去保障。这样，减轻Layer1压力的同时，Layer2上执行交易的Gas费可以大幅下降，将Layer2上交易执行的最终状态(而不是Layer2上所有的状态)同步回Layer1所需的Gas费也可以大幅下降。在Layer1之上架构Layer2的技术，实际上是将交易或合约的执行过程与最终状态的保存分离开，执行过程放到了Layer2，Layer1只需要保存最终的状态。这个设计中，需要保证Layer2上交易或合约的执行过程的正确性，并且写入Layer1的状态与Layer2上正确执行的结果是一致的。

[0010] Ethereum的Layer2包括一种称为Rollup的机制。Rollup的核心思路如图1所示，是在Layer1上保存能够验证交易过程的凭证，而将交易过程(计算过程)还有状态存储运行在Layer2中。所谓交易过程的凭证，包括一组交易执行前的状态(Pre-State)和该组交易执行后的状态(Post-State)以及该组交易，可以用于验证这组交易对应的状态转移是否正确，还可以用于还原出Layer2上所有交易的执行过程和所有账户的状态，从而消除Layer2上由于数据可用性带来的安全风险。

[0011] 发明内容

[0012] 本说明书在于提供一种实现二层网络卷叠的方法、二层网络和证明器，通过以下方式实现：

[0013] 一种实现二层网络卷叠的方法，其中二层网络中的执行器和证明器均采用可信执行环境实现，所述方法包括：

[0014] 二层网络的排序器从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SP

V证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；

- [0015] 所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名；将所述写集及第一签名发送至证明器；
- [0016] 所述证明器验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，将树根作为前状态根，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；
- [0017] 所述证明器采用自身可信执行环境中的私钥对所述前状态根和后状态根生成第二签名，将第二签名作为证明并与所述前状态根和后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [0018] 一种二层网络，包括交易池、排序器、执行器和证明器，所述执行器和证明器均采用可信执行环境实现，其中：
- [0019] 交易池，用于接收二层网络上用户发送的交易；
- [0020] 排序器，从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；
- [0021] 执行器，在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名；将所述写集及第一签名发送至证明器；
- [0022] 证明器，验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；并采用自身可信执行环境中的私钥对所述后状态根生成第二签名，将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [0023] 一种证明器，采用可信执行环境实现，并包括：

- [0024] 接收单元，用于接收排序器发送的打包交易执行前的全部初始状态/打包交易读集的SPV证明，以及接收执行器发送的写集及第一签名；
- [0025] 验证单元，用于验证第一签名；
- [0026] 执行单元，所述验证单元验证通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；；
- [0027] 签名单元，采用自身可信执行环境中的私钥对所述后状态根生成第二签名采用可信执行环境中的私钥对所述后状态根生成第二签名；
- [0028] 发送单元，将所述第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网的区块链账本。
- [0029] 如上，所述二层网络中的执行器和证明器均采用可信执行环境实现，由执行器生成打包交易执行所产生的写集并通过TEE签名，由证明器生成打包交易执行后的后状态根，可以进一步降低Prover的验证负担，从而提升Prover的验证效率，因此可以带来更低的延迟。

附图说明

- [0030] 为了更清楚地说明本说明书实施例的技术方案，下面将对实施例描述中所需要使用的附图作简单地介绍，显而易见地，下面描述中的附图仅仅是本说明书中记载的一些实施例，对于本领域普通技术人员来讲，在不付出创造性劳动性的前提下，还可以根据这些附图获得其他的附图。
- [0031] 图1是一实施例中Layer2的原理示意图；
- [0032] 图2是一实施例中Layer2的原理示意图；
- [0033] 图3为一实施例中Layer2的原理示意图；
- [0034] 图4为一实施例中Layer2的原理示意图；
- [0035] 图5为一实施例中Layer2的原理示意图；
- [0036] 图6为一实施例中Layer2的原理示意图；
- [0037] 图7为一实施例中MPT树的原理示意图；
- [0038] 图8为一实施例中MPT树的原理示意图；
- [0039] 图9为一实施例中Layer2的原理示意图；

[0040] 图10为一实施例中Layer2的原理示意图。

具体实施方式

[0041] 为了使本技术领域的人员更好地理解本说明书中的技术方案，下面将结合本说明书实施例中的附图，对本说明书实施例中的技术方案进行清楚、完整地描述，显然，所描述的实施例仅仅是本说明书一部分实施例，而不是全部的实施例。基于本说明书中的实施例，本领域普通技术人员在没有作出创造性劳动前提下所获得的所有其他实施例，都应当属于本说明书保护的范围。

[0042] Rollup的原理具体可以如图2和图3(图3中主要是下文中的OP-Rollup)所示。

[0043] 创建智能合约的交易发送到Layer1上，经过Layer1的共识之后，Layer1上各节点可以执行这个交易，从而完成合约的部署。具体的，可以由区块链节点的EVM/WASM来执行这个交易。EVM是一个图灵完备的虚拟机，这意味着可以通过它实现各种复杂的逻辑，这也是以太坊作为区块链2.0的代表相对于区块链1.0的最大改进之一。用户在以太坊中发布和调用智能合约可以在EVM上运行。执行该部署智能合约的交易之后，Layer1上产生一个与该智能合约对应的合约账户，该合约账户具有链上的地址，并且包括余额balance、计数器nonce、合约代码的hash值Codehash，合约存储的根Storage_Root。通过Codehash和Storage_Root，该合约的代码和账户存储可以保存在该合约账户中。智能合约的行为由合约代码控制，而智能合约的账户存储则保存了合约的状态。换句话说，智能合约使得区块链上产生包含合约代码和账户存储(Storage)的虚拟账户。后续，Layer1上的节点可以接收调用部署的智能合约的交易请求，该交易请求可以包括调用的合约的地址、调用的合约中的函数和输入的参数。一般的，该交易请求经过共识后，区块链各个节点可以各自独立执行指定调用的智能合约。

[0044] 以太坊中的Storage_root是一个MPT树根节点的哈希，这个MPT树对合约账户的状态的存储进行组织。MPT全称为Merkle Patricia Tree，是结合了Merkle Tree(默克尔树)和Patricia Tree(压缩前缀树，一种更节省空间的Trie树，字典树)的一种树形结构。Merkle Tree算法对每个交易都计算一个Hash值，然后两两连接再次计算Hash，一直到最顶层的Merkle根。以太坊中采用改进的MPT树，例如是16叉树的结构，通常也简称为MPT树。实际上，以太坊区块头中包括三颗MPT

树的树根，分别是Transaction_Root(简写为Tx_Root)、State_Root、Receipt_Root。State_Root当前区块中所有账户的状态组成的MPT树的根的哈希值，即指向State_Root的为一颗MPT形式的状态树state trie。从这个MPT的根节点到叶子节点中每个节点的一部分值按照顺序串联起来可以构成账户地址并作为key，叶子节点中存储的账户信息为这个账户地址对应的value，这样，构成了key-value键值对。具体的，这个key可以是sha3(Address)，即账户地址的hash值(hash算法例如采用sha3算法)，其存储的值value可以为rlp(Account)，即账户信息的rlp(Recursive Length Prefix，递归长度前缀)编码。其中账户信息是[nonce,balance,storageRoot,codeHash]构成的四元组。对于外部账户(Externally Owned Account，简称EOA)来说，一般只有nonce和balance两项，而storageRoot、codeHash字段默认存储空字符串/全0字符串。也就是说，外部账户不存储合约，也不存储合约执行后产生的状态变量。合约账户一般包括Nonce,Balance,Storage root,CodeHash。不论是外部账户还是合约账户，其账户信息一般都位于一个单独的叶子节点(Leaf Node)中。

[0045] 其中，对于state trie中的一个合约账户(Contract Account，简写为CA)来说，其storage_Root指向另一颗同为MPT形式的树，其中存储了合约执行涉及的状态变量(state variable)的数据。这个storage_Root指向的MPT形式的树为Storage Trie，即storage_Root中存储Storage Trie的根节点的hash值。一般的，这个Storage Trie树存储的也是key-value键值对。key表明状态变量的地址，其取值可以是合约中的状态变量声明的位置(从0开始计数的值)经过一定规则处理后得到的结果，例如是sha3(状态变量声明的位置)，或者是sha3(合约名称+状态变量声明的位置)。value用于存储状态变量的取值(例如是经RLP编码的值)。从根节点到叶子节点的路径上存储的一部分数据连起来构成key，叶子节点中存储value。State trie和其中的Storage trie中，除了叶子节点外的根节点、中间节点采用图3中云朵状来表示其中灵活和复杂多变的结构。

[0046] 如图2所示，Rollup中可以在Layer1上部署Rollup Contract。Layer2中，可以接收发起的交易，如图中的Tx_21、Tx_22等。一个批次的交易称为一个Batch，作为一个整体按序执行并更新状态。如图1和2中所示，Batch 1假设可以包括Tx_21、Tx_22、Tx_23这三笔交易，Batch 2假设可以包括Tx_24、Tx_25、Tx_26这三笔交

易。Layer2中可以按照一种规则对这些交易进行排序，进而可以按照顺序执行排序后的这些交易，例如按照交易中的时间戳排序。一个Batch中的交易在按序执行后，产生的状态可以包括若干，例如Batch 1中的交易分别为：

- [0047] Tx_21: Alice→Bob 20; 表示Alice转给Bob 20个单位的资产，如以太币；
- [0048] Tx_22: Alice→Charlie 10; 表示Alice转给Charlie 10个单位的资产，如以太币；
- [0049] Tx_23: Bob→Charlie 10; 表示Bob转给Charlie 10个单位的资产，如以太币；
- [0050] Batch 1中的交易执行前，假设共有4个状态，这里称为Batch 1的初始状态，即图3中由Pre State Root锁定的状态值，分别是：
 - [0051] Alice: 50, Bob: 100, Charlie: 150, David: 200
- [0052] Batch 1中的交易执行后，这4个状态更新为Batch 1的终局状态，即图3中由Post State Root锁定的状态值，分别是：
 - [0053] Alice: 20, Bob: 110, Charlie: 170, David: 200
- [0054] Layer2中可以采用Merkle树来组织Batch 1中的交易按序执行前、后的状态。如图3所示，在Layer2中可以将一个Batch的交易及其相关的状态变化组织在一起，例如将Batch 1中的交易的hash值按照Merkle树组织后得到的Merkle根锁定在Batch的头(Header)中，这个头类似Layer 1中的区块头，这个Merkle根即Batch的头中的Tx_Root。同样的，如前所述，可以将Batch 1中的交易在执行前的所有状态值按照Merkle树组织后得到的Merkle根锁定在Batch的头中，即Pre State Root；可以将Batch 1中的交易在按序执行后的所有状态值的hash按照Merkle树组织后得到的Merkle根锁定在Batch的头中，即Post State Root。此外，对于Batch M来说，在其Header中还可以设置一个时间戳(Timestamp)用于记录该Batch生成的时间，当然还可以设置其它字段的信息。对于第M+1个Batch来说，其区块头中的Pre State Root，与第M个Batch的区块头中的Post State Root相等，如图3中这两个根之间的连线所表示。这样，Batch之间形成了一个链式的结构。
- [0055] Layer2的Rollup机制，可以包括两种：一种是Optimistic-Rollup(乐观Rollup，简称OP-Rollup)，另一种是ZK-Rollup(零知识证明Rollup，ZK指的是zero knowledge，即零知识)。这两种Rollup机制，以太坊上都部署有如前所述的智能合约——Rollup Contract。

- [0056] OP-Rollup中，在Layer2里可以至少设置TxPool(交易池)、Relayer(中继器)、Sequencer(定序器)。
- [0057] 如图3中所示，假设TxPool中收集到Tx21、Tx23、Tx25、Tx22、Tx26、Tx24这些交易。Sequencer按照这些交易中的时间戳对这些交易排序并打包，例如将Tx21、Tx22、Tx23打包到Batch M中，并将这些交易的hash值组织为Merkle后生成Merkle Root，并将该Merkle Root的hash值填入生成的Batch M中Header内的Tx_Root中。需要说明的是，在计算Merkle树的树根时，一般对于小于 2^n 的情况，可以将最后一个交易赋值多份以用于补齐 2^n ，从而计算 2^n 叉的Merkle树的树根。如图3中，按照时间顺序排序为最末的Tx23重复一次后补齐了4个交易，即使交易的数量达到 2^2 ，进而可以计算得到 2^n 叉的Merkle树的树根。类似的，Sequencer按照交易中的时间戳对交易排序后将Tx21、Tx22、Tx23打包到Batch M+1中，不再赘述。
- [0058] 在Batch M执行前，所有的状态包括如前所述，Alice: 50, Bob: 100, Charlie: 150, David: 200。Sequencer可以将这些状态组织成Merkle树，并将该Merkle树根节点的hash值存储于Batch M Header的Pre State Root字段中。进而，Sequencer可以按照顺序执行Tx21、Tx22、Tx23这3笔交易。执行这3笔交易的结果，生成了新的状态，如前所述，Alice: 20, Bob: 110, Charlie: 170, David: 200。Sequencer可以将这些更新后的状态组织成Merkle树，并将该Merkle树根节点的hash值存储于Batch M Header的Post State Root字段中。
- [0059] 进而，Sequencer可以通过Relayer发送一笔调用Rollup Contract的交易1至Layer 1，该交易1中包含了Batch M Header中的各字段以及经过压缩的Tx21、Tx22、Tx23。Tx21、Tx22、Tx23经过压缩后，所占空间可以大大降低。调用的Rollup Contract合约中的接口，可以包括将压缩后的交易数据(Tx21、Tx22、Tx23)存储于较便宜(Gas费用较低)的位置calldata上。在以太坊智能合约中，calldata是一种特殊的数据位置(data location)，用于存储通过函数调用的输入数据。具体的，calldata数据一般直接存储在以太坊的交易数据中，其通常比其它数据位置(如memory和storage)更加节省gas。

[0060] 关于压缩，一笔简单的以太坊交易(发送ETH)大小约为110字节，而在Rollup上进行ETH转账，目前的设计中经最大压缩后大小可以降至约12字节。下表是一个例子中以太坊的一笔交易各字段(参数)所占体积与Rollup中一笔交易的各字段：

参数	以太坊（字节数）	Rollup（字节数）
Nonce	~3	0
Gasprice	~8	0-0.5
Gas	3	0-0.5
To	21	4
Value	~9	~3
Signature	~68 (2 + 33 +33)	~0.5
From	0（从签名中恢复）	4
Total	~112	~12

[0061] 表1

[0062] 这里略去压缩的实现细节。值得注意的是，在一笔以太坊交易中，签名部分占了整体的约一半的体积，其From字段可以是0字节，因为该From字段可以从签名中恢复出来。而在Rollup中，可以采用BLS聚合，平均每笔交易的签名被压缩到约0.5字节，相应的，Rollup交易中由于无法从这种聚合签名中恢复出交易的From字段，因此这里需要单独采用4字节来表示。

[0063] 在密码学中，公开钥匙密码学，简称公钥密码学，又称非对称密码学，是使用一对公钥和私钥(公钥-私钥记为pk-sk，其中pk表示public key，sk表示secret key)的密码学，与只用一个私钥的密码学相对应。公钥密码学包括加密算法和数字签名算法。公钥-私钥密码对是现代密码学安全的基石，很多的应用都是基于pk-sk，例如基于https(Hypertext Transfer Protocol Secure，安全的超文本传输协议)应用层加密传输协议以及区块链等。

[0064] 私钥通常代表拥有该私钥的一方的身份，其只能由私钥的拥有者持有，不能公开，而对应的公钥可以公开出去。使用私钥进行的签名可以表示私钥拥有者对数字世界的某种信息进行的认可，签名的信息在协议的消息中也可以表示私钥拥有者的某种行为。一般的，一个拥有者单独拥有一个私钥，则这个拥有者可

以采用自身私钥对某个信息进行签名后发给其它方。接收方收到这个签名后，可以采用对应的公钥对该签名进行验证。验证通过，则接收方可以确认是拥有者对该信息进行了签名，且所签名的信息没有被篡改。

[0065] BLS签名方案最初是由斯坦福大学教授Dan Boneh等人于2001年提出的一种签名方案，基于双线性映射构造。一般一个用户对自身账户的控制，体现在用自身私钥对一笔交易进行签名。单个签名如上表格中所示，可能达到68字节，占用较大体积。多个账户发起的多笔交易，则每个账户需要对自身的每笔交易分别签名。而签名聚合，可以把多个签名聚合成一个签名，因此大大节省了签名的字节空间。这种聚合可以是同一个消息由不同签名者签名的聚合，也可以是不同消息的签名聚合。公钥聚合，可以将多个不同公钥聚合成一个总的公钥，由该总公钥即可对聚合签名完成验证，而不需要在验证总签名时由每个公钥参与计算。通常可以采用Schnorr签名算法、BLS签名算法实现签名聚合。相对于Schnorr签名算法来说，BLS签名算法实现的聚合签名体积更小，约是Schnorr的二分之一。

[0066] 上述压缩方案和采用BLS进行签名聚合，是理想中的设计方案。尽管如此，实际中的几种Rollup方案中，目前一笔压缩后的交易仍然占用40-50字节的空間，且其中尚不包括签名，也就是说还未采用BLS签名聚合。

[0067] 接前文，Layer1中接收到该调用Rollup Contract的交易1后，经过共识，在一些节点上执行这个交易1，并将该交易1及其执行结果连同其它一些交易及执行结果一并生成一个区块，如图3中的Block N。所述其它一些交易，可以包括普通转账交易和涉及其它智能合约的交易。其中，普通转账交易和涉及其它智能合约的交易都可以涉及交易的执行。特别是其它智能合约的交易，可能涉及较为复杂的执行逻辑。而这里的交易1，调用Rollup Contract中存入calldata位置的函数，主要涉及存入指定位置，即不涉及复杂合约逻辑的执行。也就是说，Layer1里的Rollup Contract，主要作用是将Layer2上发来的交易中指定的内容存入指定位置，此外不涉及复杂的执行逻辑。例如，作为一个例子，Tx21、Tx22、Tx23存入由Block N Header中Tx_Root锁定到交易树中的Tx_12中。此外，作为一种实现，交易1中的Batch M Header中的Pre State Root、Post State Root可以通过Ro

llup Contract分别存入Layer1上合约状态中的状态1和状态2中。如前所述，状态1和状态2包含在由Block N Header中State_Root锁定的state trie中，具体例如是storage trie中。为了加以区分，Layer1上合约状态中的状态1和状态2例如分别称为Last State root和Current State root。此外，为了简化，也可以在Layer1上合约状态中仅保存Current State root，而不保存Last State root，如下面的代码示例所示。在另一些实现中，Last State root和Current State root也可以与Layer2中的交易一样存入Layer1中的calldata内，这里不再赘述。对于前者，例如是调用如下的Layer1中的commitBatch()合约接口：

[0068] function CommitBatch(bytes[] calldata transactions, bytes32 _preStateRoot, bytes32 _postStateRoot)

[0069] CommitBatch()合约接口内部，可以设置基本的校验逻辑，例如包括验证当前交易1中的Pre State Root是否与合约存储中的Current State Root相等，如果相等则将交易1中传入的Layer2上的交易存入calldata中，并将合约存储中的Current State Root更新为Post State Root，示例代码如下：

```
1.      // 检查提交的状态根是否与预期的状态根匹配
2.      require(_preStateRoot == currentStateRoot, "Invalid pre-state root");
3.
4.      // 将交易存储到 calldata
5.      for(uint i = 0; i < _transactions.length; i++) {
6.          // 在这里假设交易已经被编码为字节，并可以直接存储
7.          // 在实际的实现中，可能需要对交易进行解码和验证
8.          calldata.push(_transactions[i]);
9.      }
10.
11.     // 更新状态根
12.     currentStateRoot = _postStateRoot;
13.
14.     // 触发一个事件，以便外部观察者可以跟踪状态的变化
15.     emit BatchCommitted( preStateRoot,  postStateRoot,  transactions).
```

[0070] 通过上述方式，OP-Rollup可以在Layer2中执行各个交易，并且可以在一个Batch中捆绑几十、数百甚至更多的Layer2交易，且只发布最低限度的信息至Layer1上。所谓最低限度的信息，包括如前所述的压缩后的Layer2上的交易和这批交

易执行前后的状态根的hash值。如前所示，这里例如可以通过一个Layer1上的交易发送，如图3中的交易1。可见，这相当于一次性在Layer2上执行了一批交易，但是在Layer1上只执行了一个交易。同时，OP-Rollup并不附带任何证明。这是出于假设提交的Layer2上的信息不存在欺诈或恶意行为，因此其是“Optimistic”（乐观）这样的名字。虽然存在这样的假设，为了预防和震慑，仍然会要求提交Layer2信息的交易者在OP-Rollup的Rollup Contract中质押一部分Layer1上的资产（典型的是链上原生的token，如以太坊上的以太币）。OP-Rollup尽管是乐观的，但Layer1上仍然可以设置一个挑战期，任何人都可以在挑战期内使用欺诈证明提出挑战。典型的原因包括挑战者证明该Batch内的某个交易没有被正确的执行，即该交易的执行没有产生正确的状态。如果一个Batch内的某个交易被证明是无效的，则该Batch也是无效的，Layer1上的这个Rollup合约可以将其中的Layer2的Batch链进行回滚，该无效Batch及其后面的所有Batch都会变成孤块。一旦欺诈证明成功，保证金中的一部分会支付给挑战者，剩余部分则销毁。反之，如果直到挑战期结束都没有人提交欺诈证明，Rollup合约可以确定这个Batch。

- [0071] 欺诈证明是Optimistic Rollup方案采用的一种数据有效性的验证方式。目前，欺诈证明可分为单轮交互型和多轮交互型两种。
- [0072] 一种单轮交互型欺诈证明的实现中，在Layer1上重放一遍有争议的交易，检测是否存在无效状态后进行提交。例如，Alice作为验证人将OP-Rollup压缩后的数据同步到Layer1，同时质押保证金。如果挑战者Bob对该数据存在争议，则需在窗口期(挑战期)内发起挑战且同样质押保证金。OP-Rollup的Rollup Contract将会在Layer1上重新计算该区块中交易来判定对错。错误方的保证金将被罚没，正确方将获得奖励。
- [0073] 例如，Layer2上的Sequencer此前打包并执行了Tx21、Tx22、Tx23，并生成了Batch M的头Header，其中包括按序执行Tx21、Tx22、Tx23前后的全局状态的Merkle根Pre State Root和Post State Root，以及其它。Sequencer可以通过Relayer发送一笔调用Rollup Contract的交易1至Layer1，该交易1例如是调用Layer1中Rollup Contract合约的CommitBatch()合约接口，其中包含了Batch M Header中的各字段以及经过压缩的Tx21、Tx22、Tx23。Bob是Layer2上的一位用户，假设其参与了Laye

r2上Batch M中的一些交易，例如前述的Tx_21：Alice→Bob 20和Tx_23：Bob→Charlie10。

[0074] 假设Bob认为Sequencer执行Tx_23中Bob转给Charlie10之后得到的Bob的状态有误，那么Bob可以在Layer1上在Batch M的挑战期内发起挑战。Bob作为Layer2上的用户，其可以获得Layer2上的每个Batch执行前的全部状态以及执行后的全部状态，以及每个Batch中的所有交易，这是前提。进而，Bob可以向Layer1中Rollup Contract合约发起挑战，例如调用的是Challenge()合约接口。通过该Challenge()合约接口，Bob可以发送挑战的Batch编号batchIndex、其中的交易序号txIndex、证明。证明可以包括挑战的Batch M中的全部交易的原文(包括每个交易的签名；这里以存入calldata中的压缩后的交易不带签名为例)和Batch M执行前的全局状态。这样，Layer1上的Rollup Contract合约在执行时，可以执行一个验证函数，这个验证函数的执行逻辑例如包括：

- [0075] 1、验证Bob发起的挑战中，全部交易原文的签名是否正确；
- [0076] 2、验证Bob发起的挑战中，基于全部交易原文去除交易签名并压缩后组织成Merkle树的根hash与之前Sequencer通过Relayer调用CommitBatch()发来的Batch M Header中的Tx_Root(即Batch M中的全部交易组织成Merkle树的根hash)相同；
- [0077] 3、验证Bob发起的挑战中的Batch M执行前的全局状态组织成Merkle树后得到的根hash与之前Sequencer通过Relayer调用CommitBatch()发来的Pre State Root相同；
- [0078] 4、如果1、2、和3都通过验证，进一步模拟Layer2按序执行Bob发起的挑战中的Batch M中的全部交易，生成Batch M执行后的Post State Root，并与之前Sequencer通过Relayer调用CommitBatch()发来的Post State Root验证是否相同；
- [0079] 第4步执行的结果如果不同，则说明Sequencer有欺诈，可以通过Rollup Contract合约将保证金中的一部分会支付给挑战者。
- [0080] 这里需要说明的是，Layer1上的calldata中尽管有Layer2上所有的压缩后的交易，但在验证欺诈证明过程中，并不直接将calldata中Batch M中压缩后的交易直接取出执行，而是基于挑战者传入的Batch M中的所有交易原文模拟执行。这是因为，calldata中的压缩后Batch M的交易一般仅用于还原出Layer2上所有交易的执

行过程和所有账户的状态，从而消除Layer2上由于数据可用性带来的安全风险；且calldata中的压缩后Batch M的交易，不带签名的情况下本身的真实性是有待确定的，而挑战者传入的交易是完整交易，可以验证交易的有效性；再者，还因为无法验证初始状态，合约存储中只是状态根而不是状态，除非从Batch 0开始重新顺序执行所有的交易直至Batch M-1执行后得到的全部状态，而这显然是不经济不高效的。

[0081] 上述单轮交互型欺诈证明，增加了必须在链上发布的数据，且需要在Layer1上重新执行Batch M中的全部交易，并且重放这些交易还会产生巨大的燃料成本，因此，OP-Rollup正在转向多轮交互式证明，以更高的效率实现相同的目标。

[0082] 多轮交互型欺诈证明中，当Bob对Sequencer同步的数据提出挑战后，Sequencer需将争议的Batch范围二等分为前、后两段范围并通过例如dispute()合约接口返回，然后由Bob进一步选择继续挑战的二分后的一个范围，再由Sequencer将争议范围二等分，如此循环，直至将争议范围缩减到一个具体的交易。这里Bob会发送挑战的交易的原文(包括每个交易的签名)及该交易执行前的全局状态，Sequencer会发送挑战的交易执行前的Pre State Root(这里也称为状态承诺)和执行后的Post State Root，交由Layer1的Rollup Contract合约进行计算和判定。具体的，例如Rollup Contract合约基于所挑战交易执行前的Pre State Root验证挑战交易执行前的全局状态，并在该挑战交易执行前的全局状态基础上执行挑战的交易，生成执行后的状态，在挑战交易执行后的全局状态的基础上生成Post State Root，与Sequencer发送的Post State Root相比，如果不同，则说明挑战成功，反之挑战失败。在实际应用中，Arbitrum不是使用二分的方式，而是使用了K分，即每次将N个指令分为N/K组去寻找欺诈指令，效率更高。

[0083] 因此，相比单轮交互型，多轮交互型能以更低的成本来解决争议，例如二等分协议可以减少发布在链上的数据量(无需为每笔交易发布状态提交)，也更容易支持复杂的智能合约和处理更高要求的争议。不过，随着交互次数的增加，它的争议窗口期也会更长一些。

[0084] 整体来说，由于OP-Rollup的设计中需要把与状态更新和验证相关的数据都放到Layer1上，因此OP-Rollup的拓展性和对吞吐量的提升程度最为有限，且需要

7天甚至更长的等待期。这意味着，从OP-Rollup中提取资产会产生延迟，需要等待窗口期过后。

- [0085] ZK-Rollup不同于Optimistic Rollup，因为ZK-Rollup使用零知识(zero knowledge)证明技术。ZK指的是在不披露必要信息的情况下向另一方证明某事(一项交易或状态)的能力。ZK-Rollup中，与OP-Rollup类似的，将Tx21、Tx22、Tx23打包到Batch M中，并将这些交易的hash值组织为Merkle树后生成Merkle Root，并将该Merkle Root的hash值填入生成的Batch M中Header内的Tx_Root中，以及生成Pre State Root、Post State Root和Timestamp等并填入到Batch M Header中。而且，Layer2中还会为这个batch生成一个证明(包括proof)(例如采用ZK-SNARK等的密码学证明算法)，这个证明中的proof可以证明Post State Root是在Pre State Root基础上按序正确执行batch中的交易后得到的，只需要验证proof即可以确认这一点，而不需要重新执行Batch中的交易来验证。
- [0086] 在生成证明前可以创建计算电路，该电路实际上是一种特定的算法，可以预先针对验证交易及状态的逻辑编写。这个预先编写的过程包括在称为可信设置的过程中，通过该可信设过程还可以生成创建密钥(Proving Key)和证明密钥(Verifying Key)。创建的电路可以设置在Layer2上的一个proposer(或称为prover，证明器)中。需要说明的是，针对验证交易及状态的逻辑创建的电路，与其Proving Key和Verifying Key是绑定的。Proving Key、Verifying Key都是可以公开的，且是与电路是绑定的。
- [0087] 生成证明的过程，可以是由Sequencer将Layer2中一个Batch中所有交易(包括发送方、接收方、金额)以及这个Batch中的交易执行前后的Pre State Root、Post State Root和和Proving Key等信息打包发送至Layer2上的一个证明器，由证明器将这些信息转换为一种特殊的形式，称为witness(见证者；选择其中的一部分作为公共输入，另一部分作为隐私输入)。进而，证明器采用其内布置的零知识证明的计算电路(一种特定的算法)对公共输入+隐私输入的有效性和正确性进行验证。如果验证正确，电路执行成功，否则电路执行失败。电路执行成功的情况下，证明器可以输出一个proof，该proof用以证明打包的数据是有效且正确的。而

且，这个proof的体积较小，一般只有几十字节。上述生成证明的过程，类似于重新执行的过程，并且包含大量的电路计算，更为复杂，所以一般比较耗时。

[0088] 需要说明的是，由proof本身无法倒推出隐私输入。只有知道zk-SNARK的验证密钥的一方才能验证用对应创建密钥加密的proof。而公共输入可以用来验证打包的数据的有效性和正确性，但无法用来获取打包的数据的具体内容。

[0089] 采用零知识证明，证明器可以通过Relayer将较小的proof发布到Layer1上，并可以由Rollup Contract快速验证。proof体积较小，这进一步降低了Layer1上Gas的消耗。而且，在ZK-Rollup中这个验证过程通常由智能合约自动完成，且不用像OP-Roll那样再设置窗口期，提取资产也会变得迅速，这是OP-Rollup和ZK-Rollup之间的主要特性差异之一。

[0090] 具体的验证过程，首先，将公共输入(可以从Block N中提交的交易1计算得到，例如是在交易1提交至Layer1的Rollup Contract时由Rollup Contract中的自动执行计算逻辑计算得到)、proof和验证密钥(验证密钥可以通过Layer1上的治理合约保存并提供至Rollup Contract)输入到智能合约中的验证逻辑(例如包括ZK-SNARK的验证算法)，由验证逻辑使用验证算法检查proof是否有效。如果proof有效，那么公共输入对应的打包数据就被认为是有效且正确的。如果验证成功，验证算法会输出一个确认信息，证明公共输入对应的打包数据是有效且正确的；否则，验证算法会输出一个错误信息，标示数据的有效性或错误性。

[0091] 然而，ZK-SNARK这类的零知识证明在生成proof过程中的计算量非常大，也意味着生成proof将比较耗时。假设如图4中所示，通过交易1将经过压缩的Tx21、Tx22、Tx23存入Layer1中calldata中，并将Post State Root存入Layer1上合约状态中的Current State Root。这个交易1如图4位于Layer1的Block N中。则Layer2中生成对应proof所需要的复杂计算量带来的延迟，可能在一段时间通过交易2将Batch M的proof传入Layer1上的Block N+412中。从Block N到Block N+412，显然也带来了一定的延迟，这个延迟可能接近小时级。

[0092] 针对计算复杂、生成证明所需时间久这个问题，目前行业里主要的解决方案主要包括通过工程上的大规模并行化证明来实现，或通过硬件优化来加速。但是

这并不容易。电路的实现相对复杂，且没有高级语言支持，很多情况下都是手写R1CS(电路中的一种常用形式)。

[0093] 进一步，为了利用ZK证明系统，为了优化电路的实现，整个Layer2的状态经常优化为电路友好结构(merkle树)。所以，ZK-kRollup的系统需要考虑电路的结构，从而约束了Layer2交易以及账户模型。ZK-Rollup中，包括zksync、zkswap、loopring等方案，都只实现了特定交易场景，做到兼容以太坊虚拟机和支持图灵完备就更困难，也就是说，很难支持较为复杂的合约。

[0094] 以下是OP-Rollup和ZK-Rollup两种Layer2中的Rollup方案的特点对比：

	OP-Rollup	ZK-Rollup
证明类型	欺诈证明	有效证明
链上计算复杂度	高	低
TPS	100（有BLS签名可以达到500）	2000
链下计算复杂度	低（能跑EVM就行）	非常高
图灵完备	可以实现	较难实现
资金撤出时间	一至两周	小时级

[0095] 表2

[0096] 本申请一种实现二层网络卷叠的方法，在Layer2上所依赖的架构如图5所示。这里至少包括Relayer、TxPool、Sequencer、Prover。如前所述，Relayer的主要作用是将Layer2上的交易或操作提交到Layer1上，即起到桥接Layer1和Layer2的作用，可以包括将Sequencer打包的Layer2上的用户交易提交到Layer1上，以及将Layer2上Prover生成的证明提交到Layer1上。当然，也可以不包括Relayer，而是由Sequencer将打包的Layer2上的用户交易提交到Layer1上，以及由Prover将生成的证明提交到Layer1上。实际上，后者是将Relayer的功能集成在Sequencer、Prover中。为了简略，后续不再着重提及Relayer在整个方案中的作用。

[0097] TxPool用于接收Layer2上用户发送的交易。

[0098] Sequencer从TxPool中拉取交易，并按照拉取交易中的时间戳对这些交易排序并打包，并将这些交易的hash值组织为Merkle树后生成Merkle Root，并将该Merkle Root的hash值填入生成的Batch M中Header内的Tx_Root中。例如图4中所示，将Tx21、Tx22、Tx23打包到Batch M中，按照时间顺序排序为最末的Tx23重复

一次后补齐了4个交易，进而计算得到2叉的Merkle树的树根。并且，Sequencer还在Batch M中交易执行前状态的基础上顺序执行Tx21、Tx22、Tx23这3笔交易。执行这3笔交易的结果，生成了新的状态。Sequencer将执行前的状态组织成Merkle树，并将该Merkle树根节点的hash值存储于Batch M Header的Pre State Root字段，还将Batch M中交易执行后的状态组织成Merkle树，并将该Merkle树根节点的hash值存储于Batch M Header的Post State Root字段中。

- [0099] Prover可以采用TEE(Trusted Execution Environment, 可信执行环境)实现。
- [0100] TEE是基于CPU硬件的安全扩展，且与外部完全隔离的可信执行环境。目前工业界十分关注TEE的方案，几乎所有主流的芯片和软件联盟都有自己的TEE解决方案，比如软件方面的TPM(Trusted Platform Module, 可信平台模块)以及硬件方面的Intel SGX(Software Guard Extensions, 软件保护扩展)、ARM Trustzone(信任区)和AMD PSP(Platform Security Processor, 平台安全处理器)等。TEE可以起到黑箱作用，在TEE中执行的代码和数据即便是操作系统层都无法偷窥，只有通过代码中预先定义的接口才能对其进行操作。在效率方面，由于TEE的黑箱性质，在TEE中进行运算的是明文数据，而不是同态加密中复杂的密码学运算，计算过程效率几乎没有损失。因此，采用TEE技术可以在性能损失相对较小的前提下很大程度上满足区块链场景下的可信计算需求。
- [0101] 本申请一种实现二层网络卷叠的方法，包括如下：
- [0102] S110：二层网络的排序器从交易池中拉取交易，对所述拉取的交易排序并打包，生成打包交易的第一交易根，基于所述打包交易执行前的初始状态按序执行所述打包的交易并生成第一后状态根。
- [0103] S120：所述排序器发送打包的交易、第一交易根、所述打包交易执行前的初始状态以及初始状态对应的第一前状态根和第一后状态根至二层网络的证明器。
- [0104] 与前述类似，Rollup中可以在Layer1上部署Rollup Contract。Layer2中，交易池TxPool可以接收发起的Layer2交易。假设TxPool收集到Tx21、Tx23、Tx25、Tx22、Tx26、Tx24这些Layer2交易。
- [0105] Sequencer可以从交易池中拉取交易，并例如按照这些交易中的时间戳对这些交易排序并打包。例如如图4所示，将Tx21、Tx22、Tx23排序并打包到Batch M

中，并将这些交易的hash值组织为Merkle树后生成Merkle Root，并将该Merkle Root的hash值填入生成的Batch M中Header内的Tx_Root中。该Merkle Root即为Batch M的第一交易根。

- [0106] 打包的交易执行前，例如状态为：
- [0107] Alice: 50, Bob: 100, Charlie: 150, David: 200
- [0108] 这些状态即可以为打包交易执行前的初始状态。
- [0109] Sequencer可以将Batch M中的交易在执行前的所有状态值按照Merkle树组织并得到Merkle根，即第一前状态根。还可以将Batch M中的交易在执行前的所有状态值按照Merkle树组织后得到的Merkle根锁定在Batch M的头中，即图4中的Pre State Root。
- [0110] Sequencer可以在初始状态的基础上按序执行打包的交易，从而产生执行后的状态，例如：
- [0111] Alice: 20, Bob: 110, Charlie: 170, David: 200
- [0112] Sequencer可以将Batch M中的交易按序执行后得到的所有状态按照Merkle树组织并得到Merkle根，即第一后状态根。还可以将Batch 1中的交易在按序执行后的所有状态值的hash按照Merkle树组织后得到的Merkle根锁定在Batch的头中，即Post State Root。
- [0113] 进而，所述排序器可以发送打包的Layer2交易、第一交易根、所述打包交易执行前的初始状态以及初始状态对应的第一前状态根和第一后状态根至二层网络的证明器。
- [0114] 此外，在S110或S120中，Sequencer可以(或通过Relayer)发送一笔调用Rollup Contract的交易1(第一主网交易)至Layer1，该交易1中包含了Batch M Header中的各字段以及经过压缩的Tx21、Tx22、Tx23。Tx21、Tx22、Tx23经过压缩后，所占空间可以大大降低。调用的Rollup Contract合约中的接口，可以包括将压缩后的交易数据(Tx21、Tx22、Tx23)存储于较便宜(Gas费用较低)的位置calldata上。在以太坊智能合约中，calldata是一种特殊的数据位置(data location)，用于存储通过函数调用的输入数据。具体的，calldata数据一般存储在以太坊的交易数据中，这些

数据在以太坊成块后无法更改，即是只读的，其通常比其它数据位置(如memory和storage)更加节省gas。

[0115] 像其它合约一样，Rollup Contract合约可以在虚拟机(例如Ethereum Virtual Machine, EVM)中执行，当然也可以是在容器(例如docker)中执行，这里并不限定。Sequencer可以(或通过Relayer; Sequencer/Relayer通过主网上的外部账户发起)向区块链发起一笔调用主网上Rollup Contract的交易，可以触发该合约的执行。交易的内容例如包括from字段、to字段、value字段、data字段。from字段可以是交易发起方的账户地址(Sequencer/Relayer的主网外部账户地址)，to字段可以代表被调用的智能合约的地址，value字段可以是区块链上原生的通证(例如在以太坊中是以太币的值)，data字段可以包含的调用智能合约的方法和参数。通过在to字段指明调用的智能合约的地址，可以表明是对区块链上某个智能合约的调用。智能合约中一般可以包括一个或多个函数，每个函数可以包括一些输入的参数。交易中通过data字段可以指明所要调用的智能合约中的某个函数，并在data字段中填入需要传入的参数。如前所述，data字段中可以指明调用commitBatch()合约接口。这样，在Rollup Contract合约正常执行时，可以实现将第一主网交易(交易1)中经压缩的Tx21、Tx22、Tx23存入calldata中，并将合约存储中的Current State Root更新为Post State Root并传入参数，不再赘述。

[0116] 合约执行的结果，一方面可以改变合约的存储，即合约的世界状态，另一方面，该交易的执行结果或相关信息可以记录在区块链的收据(receipt)中。具体的，合约执行结果/相关信息可以表现为收据中的事件(event)。事件的结构例如为如下格式：

[0117] Event:

[0118] [topic][msg]

[0119] [topic][msg]

[0120]

[0121] 在上述示例中，事件的数量可以为一个或多个。每个事件可以包括主题(topic)和数据(data)等字段。交易执行时输出的事件的格式，可以在合约中指定。通过内置的SDK，区块链客户端或区块链节点可以监听特定topic的事件，并在监听到

特定topic事件的情况下，拉取相应的msg的内容，以及可以监听到特定topic或对应msg中的某些内容后执行预设的处理。

- [0122] 通过这种事件机制中，主网上的区块链节点可以将执行结果存放到某个topic对应的msg中，从而Sequencer/Relayer可以通过自身内置的区块链客户端监听该topic，进而可以获得相应的执行结果，这里例如是第一主网交易在主网Rollup Contract合约上已成功执行。
- [0123] 上述主网上的Rollup Contract合约执行后，可以生成特定topic的事件。Prover/Relayer监听到相应topic后，可以确定之前调用主网Rollup Contract合约的交易已在主网上执行。这样，Prover可以执行下面的S130步骤。这样，Prover可以通过监听特定事件确认主网上的区块链节点的执行结果，之后再进行S130的验证，从而避免可能由于某些原因第一主网交易并没有在主网上执行的情况下进行验证，例如是第一主网交易没有发送成功，或者是主网Rollup Contract合约对第一主网交易本身的验证失败等。当然，也可以是在S120后直接执行S130。
- [0124] S130：所述证明器验证第一交易根与第一前状态根；两个验证通过后，基于所述初始状态按序执行所述打包交易，生成第二后状态根，验证第二后状态根与第一后状态根相等后，对所述打包交易的信息签名，并将签名作为证明通过第二主网交易发送至主网的区块链账本。
- [0125] 所述证明器验证第一交易根，包括：所述证明器生成打包交易的第二交易根，验证第一交易根是否与第二交易根相等。具体的，所述证明器可以将打包的交易Tx21、Tx22、Tx23的hash值组织为Merkle树后生成Merkle Root，即与Sequencer相同的方式将这些交易的hash值组织为Merkle树后生成Merkle Root，该Merkle Root即为第二交易根。这样，所述证明器可以验证第二交易根是否与Sequencer发来的第一交易根相等。
- [0126] 所述证明器验证第一前状态根，包括：所述证明器生成初始状态的第二前状态根，验证第二前状态根是否与第一状态根相等。具体的，所述证明器可以将Sequencer发来的初始状态的hash值按照Merkle树组织并生成Merkle根，即第二前状态根。这样，所述证明器可以验证第二前状态根是否与Sequencer发来的第一前状态根相等。

- [0127] 如果所述证明器验证第二交易根与Sequencer发来的第一交易根相等，且第二前状态根与Sequencer发来的第一前状态根相等，则说明Sequencer发来的第一交易根与第一前状态根是正确的，进而可以执行后续的内容。
- [0128] 所述证明器可以基于所述Sequencer发来的初始状态按序执行所述打包的交易，生成更新后的状态，并将这些更新后的状态组织成Merkle树，从而得到更新后的状态的Merkle树根节点的hash值，即第二后状态根。进而，所述证明器可以验证第二后状态根与第一后状态根是否相等。如果相等，则说明Sequencer的执行过程是正确的，因为从相同的初始状态开始，Sequencer按序执行打包交易的执行结果与所述证明器独立按序执行相同打包交易的执行结果是相同的。
- [0129] 这样，所述证明器验证可以对所述打包交易的信息签名。具体的，所述证明器验证可以对打包交易的Batch编号、打包交易的交易根、打包交易对应的Post State Root中的至少一个签名。对这些内容中的至少一个进行签名，表明证明器对Sequencer执行本Batch所产生结果的认可。
- [0130] 如前所述，私钥通常代表拥有该私钥的一方的身份，其只能由私钥的拥有者持有，不能公开，而对应的公钥可以公开出去。使用私钥进行的签名可以表示私钥拥有者对数字世界的某种信息进行的认可，签名的信息在协议的消息中也可以表示私钥拥有者的某种行为。一般的，一个拥有者单独拥有一个私钥，则这个拥有者可以采用自身私钥对某个信息进行签名后发给其它方。接收方收到这个签名后，可以采用对应的公钥对该签名进行验证。验证通过，则接收方可以确认是拥有者对该信息进行了签名，且所签名的信息没有被篡改。
- [0131] 所述证明器可以采用TEE实现。TEE具有与外部完全隔离的可信执行环境，因此可以以保密的方式持有私钥。这样，证明器在验证上述第二后状态根与第一后状态根相等后，可以用自身私钥对所述打包交易的信息签名，以表示对签名内容的认可。
- [0132] TEE技术中，包括Intel SGX(Intel Software Guard Extension，以下简称SGX)、AMD SEV等。下面以SGX技术为例加以说明。Layer2中的Prover可以基于SGX技术创建Enclave(围圈或飞地)，以作为用于执行验证的TEE。其中，Prover可以利用CPU中新增的处理器指令，在内存中可以分配一部分区域EPC(Enclave Page Ca

che, 围圈页面缓存或飞地页面缓存), 以用于驻留上述的Enclave。上述EPC对应的内存区域被CPU内部的内存加密引擎MEE(Memory Encryption Engine)加密, 该内存区域中的内容(Enclave中的代码和数据)只有在CPU内核中才能够被解密, 且用于加解密的密钥只有在EPC启动时生成并存储在CPU中。可见, Enclave的安全边界只包含其自身和CPU, 无论是特权软件或非特权软件都无法访问Enclave, 即便是操作系统管理员和VMM(Virtual Machine Monitor, 虚拟机监视器; 或称为Hypervisor)也无法影响Enclave中的代码和数据, 因而具有极高的安全性。在上述安全性保障的前提下, CPU能够在Enclave中对数据进行处理, 具有极高的运算效率, 从而兼顾了数据安全性和计算效率。

[0133] Intel从第六代CPU开始提供对SGX的支持。而且, 这类支持SGX的CPU在出厂前, 会由厂商将部署密钥(Provisioning Key)和信封密钥(Sealing Key)烧录在CPU中的熔丝(fuse)寄存器内。熔丝寄存器是一种一次性编程的寄存器, 一经烧入数据后即熔断熔丝, 使得后续该寄存器中的内容仅可读而不再可写。Intel承诺烧录至熔丝寄存器内的密钥是随机生成的; 此外, 还承诺一经烧录后关于烧入的密钥的所有备份都会被销毁, 即连Intel自身都不知道烧入的密钥。Provisioning Key可以代表CPU中的部分信息, 例如CPU的代号(例如第六代酷睿、第七代酷睿等)、型号(如桌面型、移动型等)之类。出于安全考虑, 关于加密、签名等操作, 并不直接使用Provisioning Key, 而是采用由Provisioning Key衍生出的一个证明密钥(attestation key)来进行, 因此Provisioning Key起部署作用。

[0134] 基于用户的指令, 可以创建User Enclave, 以执行用户指定的程序代码(例如载入TEE中的交易执行代码, 包括虚拟机及智能合约)。同时, 由于上述Enclave的隔离特性, 用户创建的User Enclave很多时候需要首先对外证明自身是可信的, 这就涉及Enclave的认证, 如下面所说的本地认证和远程认证。

[0135] 发起远程认证前, CPU可以检测是否存在attestation key。如果没有, 则发起初始化。初始化的过程, 可以基于密钥生成协议, 通过与Intel服务器的交互(具体可以是SGX中的Provisioning Enclave, 是一种架构级的Enclave), 并根据Provisioning Key的生成规则生成EPID(Enhanced Privacy Identification), 该EPID作为attestation key, 一般用作非对称加密密钥中的私钥。生成的该EPID可以存放于一个Enc

lave中用于后续的签名，这个Enclave一般称为Quoting Enclave。此外，EPID也可以是存储在CPU的寄存器中，只有架构级别的Enclave才有权限从CPU中获取，这样，Quoting Enclave也可以是从CPU的寄存器中获取该EPID。上述提到，生成EPID的过程中，SGX(具体如上述提到的Provisioning Enclave)可以与Intel服务器进行交互，从而，Intel可以通过交互过程得到该EPID对应的公钥。特别的是，该EPID对应的公钥并不会公开，仅由Intel服务器保管。这样的特性适于后续远程认证过程中由Intel的服务器(也称为IAS，即Intel Attestation Server)进行认证。

[0136] 应用程序的开发者(或后文所称的颁发者)可以将需要载入Enclave的代码/数据(例如载入TEE中的交易执行代码，包括虚拟机及智能合约)用开发者的私钥签名后通过应用程序传至SGX。此外，还可以将颁发者的公钥传至SGX(也可以是上传颁发者的证书，该证书中包括用颁发者签名的信息，该信息中可以包括颁发者的公钥；所述信息还可以包括要载入Enclave的代码/数据的hash值，该hash值可以用于SGX对所载入代码/数据的完整性校验)。所述应用程序执行的过程中，可以通过CPU中集成的指令集向SGX申请创建一个Enclave，这个Enclave一般称为Application Enclave(相当于上述的User Enclave)。SGX在创建Application Enclave时，需要进行页面分配、复制程序代码/数据和度量操作。所述度量，具体的，可以由CPU度量加载到每个页面的代码/数据的hash值，从而在加载完毕后，得到加载了代码/数据的Application Enclave的hash值，这个hash值一般称为MREnclave，哈希算法一般采用SHA256。MREnclave可以存放于称为SECS(SGX enclave control structure)的结构体中。SECS位于所属Enclave的一个EPC页中，用于记录所述Enclave的元数据。所述元数据包括如Enclave密码学测度(即MREnclave)等敏感信息，因此这段结构体只能被CPU的SGX管理机制访问修改。

[0137] 基于Hash算法的特性，篡改代码/数据，分配多余的页，将恶意代码复制进来，这三者中任一项的发生，哪怕只是改变了一点，都会导致MREnclave的结果产生很大的不同；不同的Application Enclave，该MREnclave值也不同。基于这样的性质，MREnclave一方面可以标识这个Enclave，另一方面也可以用于Application Enclave的完整性验证。通过度量操作，可以采用上述证书中包含的hash值对Ap

plication Enclave的完整性进行验证(证书中包含的hash值与MREnclave的生成规则相同,因此正常情况下应当是一致的),以判断特权软件在创建过程中是否篡改了程序/数据,如是否分配了多余的页、是否将恶意代码复制进来,或是否篡改了复制的数据等。

[0138] 具体的,SGX创建Application Enclave时可以通过一条初始化指令将前述提到的MREnclave与Application Enclave颁发者签名的证书中的hash值进行比较,如果匹配,则说明载入Application Enclave的代码/数据与预期的一致,即是颁发者的代码/数据;反之,如果不匹配,则说明创建过程存在问题,返回失败结果。如果匹配,SGX还将所述证书中的颁发者公钥进行哈希,从而得到MRSigner。类似的,MRSigner可以用于标识颁发者。该MRSigner也可以保存在前述SECS中。

[0139] 这样,SECS结构体从某种角度来说相当于Enclave的身份,它同时拥有两种Enclave的身份:MREnclave为Enclave的测度,MRSigner为颁发者的测度。

[0140] Application Enclave中的内容需要进行远程认证,以证明自身是合法的SGX中创建的可信的Enclave。如前所述,Application Enclave中的内容需要经Quoting Enclave中的EPID签名后进行远程认证,这就需要Application Enclave与Quoting Enclave的交互。在此之前,Application Enclave需要向Quoting Enclave证明自身是同一SGX上的Enclave,是可信的。Application Enclave向Quoting Enclave证明自身是同一SGX上的Enclave的过程,即是本地证明。

[0141] 本地证明的过程可以采用消息认证码(Message Authentication Code, MAC)算法。Application Enclave可以获得Quoting Enclave的MREnclave,并可以生成签名的报告结构(Report)。具体的,Application Enclave可以调用CPU提供的EREPORT指令,CPU可以用一个根秘钥并结合Application Enclave获得的Quoting Enclave的MREnclave来生成一个对称秘钥——REPORT Key。同时,通过调用EREPORT指令,可以获得Application Enclave的身份信息和属性、平台硬件TCB信息,并作为Report的内容;此外,Report中还可以附加用户希望交互的数据。调用EREPORT指令,还将采用MAC算法将REPORT Key和上述Report计算得到第一MAC码,第一MAC码可以作为Report的签名,以用于完整性验证。上述Application Enclave调用CPU的EREPORT指令,CPU可以在内部直接获取Application E

nclave的身份信息和属性、平台硬件TCB信息，从而保证这些信息不会受Application Enclave的干预，从而不会被篡改；而且，CPU用根秘钥并结合Quoting Enclave的MREnclave生成REPORT Key，这个过程也是在CPU内部执行的，即使Application ENclave也不会得知该根秘钥，因此不会泄露根秘钥。Quoting Enclave接收到所述Report和第一MAC码后，可以调用CPU提供的EGETKEY指令，并将自身的MREnclave作为输入参数，从而由CPU根据与EREPORT指令中相同的根秘钥和相同的MREnclave得到相同的REPORT Key。进而，Quoting Enclave可以采用相同的MAC算法将该得到的REPORT Key和接收到的Report计算得到第二MAC码，如果该第二MAC码与收到的第一MAC码相同，说明Application Enclave发来的Report的完整性没有被破坏，也就说明Report中包含的Application Enclave的身份信息和属性、平台硬件TCB信息是完整的。由于只有同一SGX上的两个Enclave才能分别通过CPU提供的EREPORT指令和EGETKEY指令结合相同的CPU根秘钥生成相同的Report Key，因此，当验证MAC码通过后，Quoting Enclave可以确认Application Enclave是同一SGX上的Enclave。

[0142] 通过上述过程，完成本地证明，即Quoting Enclave可以确认Application Enclave是同一SGX上的Enclave。在本地证明基础之上，SGX外部的软件可以向Application Enclave发起挑战(该SGX外部的软件也称为挑战者)，要求该Application Enclave向所述挑战者证明自身是合法的Enclave，且其中运行的是可信的程序。在完成远程认证之后，挑战者可以使用Application Enclave提供的服务。

[0143] 远程认证中，挑战者可以首先向Application Enclave发起挑战，要求该Application Enclave证明其自身是合法的Enclave，且其中运行的是可信的程序。Application Enclave接收到挑战后，可以生成一个报告，这个报告一般称为Quote。这个Quote报告中可以包括Application Enclave中加载的代码/数据的Hash值，即前述MREnclave，还可以包括前述MRSigner。通过前述的本地证明后，Quote中的内容可以由Quoting Enclave采用EPID签名，例如记为Signature1。这个签名可以是Quoting Enclave采用EPID对所述MREnclave所做的签名，或对MREnclave和MRSigner所做的签名。之后，Application Enclave可以将Quote报告发送至挑战者。由于挑战者并不持有EPID对应的公钥，挑战者无法验证签名，而所述EPID对应的公钥由

英特尔认证服务(Intel Attestation Services, IAS)持有,因此,挑战者将Quote发送至IAS,由IAS进行验签。IAS收到所述Quote后,可以用EPID对应的公钥验证Quote中的签名的正确性,并返回认证验证报告(Attestation Verification Report, AVR)。AVR报告中,可以包括Quote报告和对Quote中的签名的验证结果,例如签名验证是正确或错误的。IAS可以对该AVR报告的内容用Intel的私钥进行签名,并将该签名(例如记为Signature2)和证书附加在AVR报告中,将AVR报告返回至挑战者。挑战者收到AVR报告后,可以采用证书中的公钥(与Intel的私钥对应)对Intel的签名进行验证。此外,AVR报告中也可以不包括证书,而是由挑战者自行下载Intel的证书(Intel的官方网站后其他权威网站可以提供该证书的下载),并用下载的证书中的公钥对Signature2进行验证。通过对Signature2的验证,可以确认该AVR报告确实是由Intel发出,且是完整的。AVR报告中包括的对Quote中的签名的验证结果,可以证明经由IAS验证后Quote报告所代表的SGX是否合法。进一步的,挑战者可以验证Quote中的MREnclave与MRSigner,例如挑战者可以预先从颁发者处获得加载入SGX中的代码/数据的MREnclave以及颁发者的MRSigner,从而分别与Quote中的MREnclave与MRSigner进行比较,以确认所述SGX中加载的代码/数据是可信的,且是由合法的颁发者颁发的。

- [0144] 在S130中证明器对所述打包交易的信息签名之前,Layer2上交易的发起方可以作为挑战者对证明器中的TEE发起挑战,这个过程中包括证明器中的TEE与IAS交互,以及挑战者与IAS的交互。根据IAS返回的AVR报告,交易的发起方可以确认证明器中的TEE是合法的,且中加载的代码/数据是可信的。从而,Layer2上交易的发起方可以信任证明器中的TEE。
- [0145] 此外,在S130中证明器对所述打包交易的信息签名之后,其它验证者也可以作为挑战者对证明器中的TEE发起挑战,类似上述过程,不再赘述。而且,TEE可以对输入的代码/数据进行hash后签名,从而用自身的可信来担保执行的输入也是可信的;TEE还可以对输出(也就是执行的结果)的hash值签名,也是用自身的可信来担保执行的输入也是可信的。
- [0146] 这样,所述证明器验证基于所述初始状态按序独立执行(也可以称为重放)所述打包交易后,对所述打包交易的信息签名,即是用自身的可信来担保输出结果

是可信的。进而，所述证明器可以将签名作为证明通过第二主网交易发送至主网的区块链账本。任何人对该签名产生质疑，可以通过对证明器中的TEE发起挑战来验证，这个过程如上所述也需要与IAS交互，从而从IAS返回的报告确认TEE及其中加载的代码/数据是可信的，并基于对TEE的信任，确信其签名的交易信息是可信的。

[0147] 需要说明的是，一旦通过远程认证，通过TEE的机制，CPU能够在Enclave中对数据进行处理，具有极高的运算效率，从而兼顾了数据安全性和计算效率。这就比ZK-Rollup的方式还要快很多，例如同样情况下，ZK-Rollup生成proof需要间隔30~50分钟量级(或称小时级)的时间(图4中Block N到Block N+412)缩短到5~10分钟量级的时间(图5中Block N到Block N+62)，甚至在不采用监听模式，而是在S120后直接执行S130的情况下，时间将缩短到更少。并且，生成的证明是一个签名的大小，约为几十字节，与ZK-Rollup类似。

[0148] 对签名的验证即是采用TEE的公钥进行验证签名的过程，只要保证公钥是可信的，这个验证也将是可信且快速的。所述证明器可以将签名作为证明并通过第二主网交易发送至主网的区块链账本，包括证明器将签名作为证明并通过第二主网交易发送至主网，调用主网上的卷叠合约；所述卷叠合约接收到所述交易后，通过合约中的验证逻辑验证所述签名的正确性。

[0149] 上述方案也称为EPID方案。在上述EPID方案中，每次挑战都需要挑战者及TEE与远程的IAS通信，这会带来较大延迟，甚至很多运行实体无法在运行时接入Intel服务。因此，改进的SGX DCAP认证协议提供将IAS本地化，例如一些云服务平台、数据中心将IAS本地化在管控机器上，本地化的管控机器也称为本地化的认证服务(Verify Service)。SGX DCAP以证书链的方式提供信任，唯一的信任根为Intel SGX Root CA。

[0150] Intel SGX DCAP，全名为Intel Software Guard Extensions Data Center Attestation Primitives，是一种专为数据中心环境设计的远程校验(attestation)技术。如前所述，Intel SGX在硬件层面上提供安全保护的技术，它能够创建一个被称为"enclave"的安全区域，在这个区域内的数据和代码都被硬件加密保护，即使操作系统、虚拟机管理器或者BIOS遭到攻击，也无法窃取或篡改这个区域内的数据。在In

tel SGX中, "attestation"是一种机制, 用于证明代码正在一个真实的enclave中运行, 并且代码和数据没有被篡改。在传统的SGX环境中, 这种校验过程需要通过Intel的远程认证服务(IAS)来进行。然而, 在数据中心环境中, 可能需要进行大量的校验操作, 因此Intel引入了DCAP, 它允许数据中心自行承载校验服务。DCAP将校验过程从Intel的服务中解耦出来, 使得数据中心可以在本地进行校验, 从而提高效率, 增强可控性, 并降低对Intel服务的依赖。简单来说, SGX DCAP就是一种在数据中心环境中使用SGX技术的解决方案, 它提供了一种更高效和可控的校验机制。

- [0151] 数据中心的认证服务(Verify Service)可以从Intel IAS获得证书, 该证书一般具有一个有效期。在有效期内, 部署的SGX实例可以在不与Intel交互的情况下, 通过与本地的验证服务(这时相当于本地的IAS)通过交互完成远程报告的生成。当过了有效期之后, 数据中心的验证服务需要从Intel IAS获得新的证书。证书中一般包括Intel签名的验证服务的公钥, 提供由Intel的公钥验证Intel的签名来保证证书中验证服务的公钥的可信。而验证服务可以采用自身私钥对其它内容签名, 并提供由验证服务的公钥验证Intel的签名来保证证书中验证服务的公钥的可信。此外, 证书中可以包括有效期, 有效期的作用如前所述。
- [0152] 实现SGX DCAP认证协议的过程简略来说可以包括如下(详细部分请参见上述内容):
- [0153] 首先, 证明器中的TEE在本地创建一个受保护的执行环境(User Enclave), 并在其中生成一个密钥对。这个密钥对包括一个公钥和一个私钥, 其中公钥将被用来生成一个证明报告(Quote)。
- [0154] 然后, 证明器中的TEE将公钥和一些关于User Enclave的信息(例如其度量值)一起提交给本地的Quoting Enclave(QE), 请求生成一个证明报告。QE会对这些信息进行签名, 并将签名和其他信息打包到证明报告中。
- [0155] 接着, 证明器中的TEE将这个证明报告提交给认证服务(Verify Service), 此服务将验证报告的有效性, 并对其中包含的公钥进行签名, 生成一个证书。这个证书包含认证服务对证明器中TEE的签名外, 还可以包括其它内容, 与前述远程认证报告类似, 因此也可以视为一种由认证服务生成的远程认证报告。

- [0156] 最后，生成的证书将被返回给证明器的TEE，后续证明器可以采用自身私钥对执行结果签名，并可以提供该证书。这样，其他人可以通过证书中公钥对签名进行验证以确保执行结果是可信的。而该证书是由认证服务颁发的，即由认证服务采用自身私钥的签名来保证的，该证书的可信可以由Intel IAS为认证服务颁发的证书进一步来保证。上述多个证书和签名的信任传递关系即证书链原理。
- [0157] 总之，证明器可以将证书通过第二主网交易发送至主网的区块链账本。一种典型的方式中，证明器可以将证书与所述证明器的签名一并通过第二主网交易发送至主网的区块链账本。当采用上述EPID方案时，所述证书包括IAS为所述证明器生成的远程认证报告，这样，验证者可以通过验证该证书来确定证明器对交易信息的签名是合法的。当采用上述DCAP方案时，所述证书可以包括认证服务为所述证明器颁发的证书。如果基于对认证服务的信任，则可以延伸到认证服务为证明器颁发的证书，从而验证者可以采用该证书中的公钥对证明器的签名进行验证。如果需要更高层次的信任，除了需要认证服务为证明器颁发的证书外，可能还需要包括IAS为所述认证服务生成的远程认证报告，这样，通过Intel权威的认证(即远程认证报告，相当于Intel为认证服务颁发的证书)可以保证认证服务签名的可信，加上认证服务为证明器颁发的证书，可以将信任向下传导，从而保证证明器签名的可信。这即是SGX DCAP以证书链的方式提供信任。
- [0158] 上述验证者执行的验证逻辑可以内置于Rollup Contract合约中，对外可以以合约接口的方式提供，如验证接口。这样，所述证明器将签名作为证明并通过第二主网交易发送至主网，具体可以是在第二主网交易交易中调用的主网上的Rollup Contract合约，包括证明器将签名作为证明并通过第二主网交易发送至主网，调用主网上的Rollup Contract合约的验证接口，通过该接口可以将所述签名传入。这样，验证接口内部可以执行验证逻辑，具体包括验证所述签名的合法性。验证签名合法性的过程，主要包括采用证明器的公钥验证签名的合法性。
- [0159] 此外，通过验证接口还可以将所述签名对应的证明器的证书一并通过第二主网交易传入，所述证书中可以包括所述证明器的公钥。这样，除了采用证明器的公钥验证签名的合法性之外，还可以包括验证证明器证书的合法性。所述证书，如前所述，可以包括IAS为所述证明器生成的远程认证报告，或包括认证服

务为所述证明器颁发的证书(还可以包括IAS为所述认证服务生成的远程认证报告)。相应的,验证证书,可以包括采用Intel的权威公钥验证证明器证书的合法性,或者可以包括采用Intel的权威公钥验证认证服务的证书的合法性,并采用认证服务的证书中的公钥验证证明器证书的合法性。

- [0160] 如上,采用TEE作为二层网络上的证明器,由于TEE中重放交易并验证接近于原生执行的速度,因此可以带来更低的延迟。
- [0161] 上述图4中例举了Layer2上的全局状态为4个的情况,即包含Alice、Bob、Charlie、David四个全局状态的情况。实际上,Layer2上的一个Batch中可能包含数百甚至上千笔的交易,这些交易可能涉及更多的状态,例如上万个状态。如果Layer2中的交易包括涉及Layer2上合约的交易,则一个合约可能涉及更多的状态,则整体上Layer2中的状态更多,例如数万个状态。这样,在S120中,排序器将全部的状态的初始状态发送至二层网络中的证明器,数据量较大。因此,本申请一种实现二层网络卷叠的方法,包括如下:
- [0162] S210: 二层网络的排序器从交易池中拉取交易,对所述拉取的交易排序并打包,生成打包交易的第一交易根,基于所述打包交易执行前的初始状态按序执行所述打包的交易并生成第一后状态根。
- [0163] S210具体过程与前述S110类似,不再赘述。
- [0164] S220: 所述排序器发送打包的交易、第一交易根、所述打包交易执行前读集的SPV证明以及第一前状态根和第一后状态根至二层网络的证明器。
- [0165] 为了加以说明,假设如图6所示,Layer2中的交易(包括普通转账交易和涉及合约的交易)共涉及8个状态,如图6左下角中8、9、10、11、12、13、14、15这8个hash值对应的状态。并假设打包到Batch M中的交易,按序执行前后影响到的状态是8、9对应的Alice和Bob,而不涉及10、11、12、13、14、15这6个hash值对应的状态。如图所示,这样,打包到Batch M中的交易按序执行后发生改变的是Alice和Bob的余额,Charlie和David的余额并未改变(也就是10、11两个树节点的hash值没有改变,这里示出了10和11两个节点对应的内容;12、13、14、15对应的内容也没有改变,这里省略其内容)。

- [0166] 所述排序器除了发送打包的交易、第一交易根、初始状态对应的第一前状态根和第一后状态根至二层网络的证明器以外，可以发送所述打包交易执行前该打包交易读集的SPV证明至二层网络的证明器。
- [0167] 如图6中的例子，所述排序器可以发送树节点8、9对应的状态，以及由节点5的hash值、节点3的hash值组成的SPV证明至二层网络的证明器。这样，可以避免把10、11、12、13、14、15这6个叶子节点对应的状态发送至证明器。
- [0168] SPV，本身来自于虚拟货币中的概念，全称是Simplified Payment Verification，即简单支付验证。SPV证明使用Merkle树来工作。这里，Batch M中涉及的状态组成一颗Merkle树，树的根(即Merkle根)包含在Batch M的Header中。Merkle树的结构使得可以通过一个很短的路径(Merkle路径)来验证任何状态是否存在于某个特定的Merkle根下。这个路径仅包含一些必要的节点，而不是所有的叶子节点对应的状态内容。
- [0169] 如图6中例子所示，一个SPV证明包括两个叶子节点(用灰色表示)对应的状态，一个Merkle路径(用黑色表示)。
- [0170] SPV证明中的Merkel路径，提到的Merkle是一种树形的数据结构，可以是虚拟货币中的Merkle树，也可以是以太坊中的MPT树，此外还可以是SMT等。
- [0171] MPT全称为Merkle Patricia Tree，是结合了Merkle Tree(默克尔树)和Patricia Tree(压缩前缀树，一种更节省空间的Trie树，字典树)的一种树形结构。Merkle Tree，默克尔树算法对每个交易都计算一个Hash值，然后两两连接再次计算Hash，一直到最顶层的Merkle根。以太坊中采用改进的MPT树，例如是16叉树的结构，通常也简称为MPT树。以前述提到的以太坊中的状态树(state trie)为例，状态树中包含以太坊网络中每个账户所对应的存储内容的键值对(key and value pair，也写作key-value，简称k-v或kv)。状态树中的“键”(key)可以是一个的160bits的标识符(例如以太坊账户的地址或地址的hash值的一部分，下面统称为账户地址)，这个账户地址分布于从状态树的根节点开始到叶子节点的存储中。状态树中的“值”是通过对以太坊账户的信息进行编码(使用递归长度字典编码(Recursive-Length Prefix encoding, RLP)方法)生成的。如前所述，对于外部账户来说，

细则 91,
18.12.2024

值包括nonce和balance；对于合约账户来说，值包括nonce、balance、codehash和storage_root。

[0172] 如前所述，state_root是当前区块中所有账户的状态组成的MPT树的根的哈希值，即指向state_root的为一颗MPT形式的状态树state trie。这个MPT树的根节点一般为一个扩展节点(Extension Node)或一个分支节点(Branch Node)，state_root中存储的一般为这个根节点的hash值。如图7所示，根节点可以与下面一层或多层的Extension Node/Branch Node相连，这些多层的树节点可以统称为中间节点(Internal Node)。从这个MPT的根节点到叶子节点中每个节点中的一部分值按照顺序串联起来可以构成账户地址并作为key，叶子节点中存储的账户信息为这个账户地址对应的value，这样，构成了key-value键值对。这个key也可以是sha3(Address)后取一部分，即账户地址的hash值(hash算法例如采用sha3算法)的一部分，其存储的值value可以为rlp(Account)，即账户信息的rlp编码。其中账户信息是[nonce,balance,storageRoot,codeHash]构成的四元组。如前所述，对于外部账户来说，一般只有nonce和balance两项，而storageRoot、codeHash字段默认存储空字符串/全0字符串。也就是说，外部账户不存储合约，也不存储合约执行后的产生的状态变量。合约账户一般包括Nonce,Balance,Storage root,CodeHash。其中Nonce是该合约账户的交易计数器；Balance是账户余额；Storage root对应另外一个MPT，通过Storage root能链接到合约相关的状态的信息；CodeHash是合约代码的hash值。不论是外部账户还是合约账户，其账户信息一般都位于一个单独的叶子节点(Leaf Node)中。从根节点的Extension Node/Branch Node到每个账户的Leaf Node，可能中间会经过若干个分支节点以及扩展节点。

[0173] state trie可以是MPT形式的树，一般是16叉树，即每一层最多可以有16个孩子节点。对于Extension Node，用于存储共同前缀，其一般有1个孩子节点，这个孩子节点可以是Branch Node。对于Branch Node，其最多可以有16个孩子节点，其中可能包括Extension Node和/或Leaf Node。

[0174] 其中，对于state trie中的一个合约账户来说，其storage_Root指向另一颗同为MPT形式的树，其中存储了合约执行涉及的状态变量(state variable)的数据。这个storage_Root指向的MPT形式的树为Storage Trie，即Storage Trie的根节点的hash

值。一般的，这个Storage Trie树存储的也是key-value键值对。key表明状态变量的地址，其取值可以是合约中的状态变量声明的位置(从0开始计数的值)经过一定规则处理后得到的结果，例如是sha3(状态变量声明的位置)，或者是sha3(合约名称+状态变量声明的位置)。value用于存储状态变量的取值(例如是经RLP编码的值)。从根节点经中间节点到叶子节点的路径上存储的一部分数据连起来构成key，叶子节点中存储value。前面提到，这个Storage trie也可以是MPT形式的树，一般也是16叉树，即对于Branch Node，其最多可以有16个孩子节点，这些孩子节点可能包括Extension Node和/或Leaf Node。而对于Extension Node，其一般可以有1个孩子节点，这个孩子节点可以是Branch Node或Leaf Node。

[0175] 例如图7中的state Trie的Leaf Node Account P，该账户是一个合约账户，其Storage Root锁定了该合约存储中的所有状态。这些状态组织为MPT树，树形结构如该Storage Root链接的Storage trie。这个链接的Storage trie中，以Leaf Node State Variable N为例，例如为前述合约代码示例的storedData的值，则其key为sha3(storedData的声明位置，后面将详述)，其value值为s(为了简洁，这里省略了对value的编码格式，例如是RLP，后续类似，不再赘述)。其中，key的值顺序的分布于storage Trie的根节点到叶子节点(即Leaf Node Variable N)中。

[0176] 再例如，图7中的state Trie中的Leaf Node Account C，该账户是一个外部账户，其key为sha3(Address C)，即账户C的地址的hash值(hash算法例如采用sha3算法)，其存储的值value可以为(Account)，其中账户信息Account是[nonce,balance]构成的二元组。如前所述，由于Account C为外部账户，因此其账户信息是nonce和balance两项(这里省略了codehash和storage root，以下类似)。例如一个外部账户，其nonce为20，Balance为4550，则Leaf Node State Variable C这个叶子节点中即存储nonce=20，balance=4550。而Account C的地址为key，其值顺序的分布于state Trie的根节点到叶子节点(即Leaf Node Variable C)中。

[0177] 这些状态，包括外部账户的k-v和合约账户的k-v，最终存储于数据库中。数据库中的存储，并不是直接存储这些账户的状态，即不是直接存储这些账户的k-v，而是存储每个树节点本身的k-v值。

[0178] 如图8的示例中所示，上一级的MPT结构中，对于叶子节点A1，通过根节点A8(Extension Node)中shared nibble的a7—中间节点A7(Branch Node)的槽位1—叶子节点A1中key-end的1335，顺序组合起来构成该叶子节点的key，即为a711335，在该叶子节点中存储Balance=45.0ETH，Nonce=n1。对于叶子节点A2，通过根节点A8(Extension Node)中shared nibble的a7—中间节点A7(Branch Node)的槽位7—节点A6(Extension Node)中shared nibbles的d3—中间节点A5(Branch Node)中的槽位3—叶子节点A2中key-end的7，顺序组合起来构成该叶子节点的key，即为a77d337，在该叶子节点中存储Balance=1.00WEI，Nonce=n2。对于叶子节点A3，通过根节点A8(Extension Node)中shared nibble的a7—中间节点A7(Branch Node)的槽位f—叶子节点A3中key-end的9365，顺序组合起来构成该叶子节点的key，即为a7f9365，在该叶子节点中存储Balance=1.1ETH，Nonce=n3。对于叶子节点A4，通过根节点A8(Extension Node)中shared nibble的a7—中间节点A7(Branch Node)的槽位7—节点A6(Extension Node)中shared nibbles的d3—中间节点A5(Branch Node)中的槽位9—叶子节点A4中key-end的7，顺序组合起来构成该叶子节点的key，即为a77d397，在该叶子节点中存储Balance=0.12ETH，Nonce=n4，CodeHash=c1，Storage root=s1。s1可以为H(A10)，即下一层树的根节点A10的hash至。其中，A1、A2和A3的叶子节点中存储的是外部账户的信息，A4的叶子节点中存储的是合约账户的信息。对于合约账户，其包含下一级MPT，构成Storage Trie，用于存储该合约账户中的状态变量。

[0179] 如图8的示例中所示，下一级的MPT结构中，对于叶子节点A11，通过根节点A10(Branch Node)中的槽位3—叶子节点A11中key-end的35b2e4，顺序组合起来构成该叶子节点的key，即为335b2e4，在该叶子节点中存储“张三_A=20”，例如表示在合约中定义的A类型数字资产归属于张三的份额为20，即张三的A类资产的余额为20。对于叶子节点A12，通过根节点A10(Branch Node)中的槽位7—叶子节点A12中key-end的c25988，顺序组合起来构成该叶子节点的key，即为7c25988，在该叶子节点中存储“李四_B=20”，例如表示在合约中定义的B类型数字资产归属于李四的份额为50，即李四的B类资产的余额为50。对于叶子节点A15，通过根节点A10(Branch Node)中的槽位f—中间节点A13(Extension Node)中shar

ed nibble的a—中间节点A14(Branch Node)的槽位6—叶子节点A15中key-end的be33，顺序组合起来构成该叶子节点的key，即为fa6be33，在该叶子节点中存储“storedData=s”。对于叶子节点A16，通过根节点A10(Branch Node)中的槽位f—中间节点A13(Extension Node)中shared nibble的a—中间节点A14(Branch Node)的槽位9—叶子节点A16中key-end的9365，顺序组合起来构成该叶子节点的key，即为fa99365，在该叶子节点中存储“王五_A=35”，例如表示在合约中定义的A类型数字资产归属于王五的份额为35，即王五的A类资产的余额为35。

[0180] 上述MPT树的节点构成中，用前缀prefix表示树节点类型，例如0表示包含偶数个shared nibbles(共享的半字节)的Extension Node，用1表示包含奇数个shared nibble(s)的Extension Node，用2表示包含偶数个nibbles的Leaf Node，用3表示包含奇数个nibble(s)的Leaf Node。

[0181] 上述节点构成中，下一个树节点的整体内容的hash值，填入上一个树节点的对应位置中。数据库中，实际上存储每个树节点的key-value的映射，其中value包括这个树节点中存储的内容，对应的key是这个树节点整体内容的hash值。这样，数据库中实际存储的树节点k-v如下表：

Key	Value (逻辑示意, 且持久化存储中需要经过 RLP 编码)
H(A9)	Prev Hash: , Nonce: ,Timestamp: , Block Num: ,State Root: H(A8), Transaction Root: ,Receipt Root: ,...
H(A8)	prefix:0, shared nibble(s): a7, next node: H(A7)
H(A7)	0: ,1: H(A1) ,2: ,3: ,4: ,5: ,6: ,7: H(A6) ,8: ,9: ,a: ,b: ,c: ,d: ,e: ,f: H(A3) , value:
H(A1)	prefix:2, Key-end:1335, balance: 45.0ETH, nonce: n1
H(A6)	value: prefix:0, shared nibble(s):d3, next node: H(A5)
H(A3)	prefix:2, Key-end:9365, balance: 1.1ETH, nonce: n3
H(A5)	0: ,1: ,2: ,3: H(A2) ,4: ,5: ,6: ,7: ,8: ,9: H(A4) ,a: ,b: ,c: ,d: ,e: ,f: , value:
H(A2)	prefix:0, Key-end:7, balance: 1.00WEI, nonce: n2
H(A4)	prefix:0, Key-end:7, balance: 0.12ETH, nonce: n4, codhash:c1, storage root: H(10)
H(A10)	0: ,1: ,2: ,3: H(A11) ,4: ,5: ,6: ,7: H(A12) ,8: ,9: ,a: ,b: ,c: ,d: ,e: ,f: H(A13) , value:
H(A11)	prefix:2, Key-end:35b2e4, value: 张三_A=20
H(A12)	prefix:2, Key-end:c25988, value: 李四_B=50
H(A13)	prefix:1, shared nibble(s):a, next node: H(A14)
H(A14)	0: ,1: ,2: ,3: ,4: ,5: ,6: H(A15) ,7: ,8: ,9: H(A16) ,a: ,b: ,c: ,d: ,e: ,f: , value:
H(A15)	prefix:2, Key-end: be33, value: storedData=s
H(A16)	prefix:2, Key-end: 9365, value: 王五_A=50

- [0182] 表3、数据库中实际存储的树节点k-v
- [0183] 上表3中, 用H()表示hash计算。这样, 下一个树节点的hash值锚定在了上一个树节点中。通过这样的层层hash, 得到整颗state trie树的根hash, 并将该根hash锁定到了区块头的state root字段中。
- [0184] 普通的MPT树, 叶子节点里存储具体的内容(状态)。而采用上述的具体结构, 在有新的状态产生时, 需要在树上插入新的叶子节点, 随之产生到根节点的路径上的一系列变化, 包括Branch Node和Extension Node的分裂、合并以及hash值重新计算并更新等复杂操作。类似的, 有状态删除时, 需要在树上删除已有的叶子节点, 也会产生Branch Node和Extension Node的分裂、合并以及hash值重新计算并更新等复杂操作。有状态发生改变时, 除了叶子节点中内容的改变, 其也会影响到根节点的Branch Node和Extension Node的分裂、合并以及hash值重新

计算并更新等复杂操作。总体来说，树的深度和具体的Branch Node和Extension Node的结构，会随着叶子节点中状态的变化而发生较多的改变。

- [0185] 虚拟货币中的交易Merkle树也是类似的，当一个区块打包有4笔交易时，它的Merkle树的叶子节点就由这四笔交易组成，即它的叶子节点只会有四个，树的深度是2层。如果一个区块打包有1024笔交易，那就有1024个叶子节点，深度是11层，这显然是一棵规模更大的树。
- [0186] Libra提出的SMT(Sparse Merkle Tree，稀疏默克尔树)也是一种Merkle类型的数据结构，但也具有其特殊性，有些地方与一般的Merkle树不同。最主要的区别点在于，叶子节点的数量、树的深度和形状预先就固定了。前述虚拟货币中的交易Merkle树和以太坊中的MPT树，树的深度、规模等是变化的，即是动态、不固定的。而SMT是存预先确定的数量且数量不会改变的状态，用一个固定大小的树来存。例如，以太坊中账户的地址长度是20字节，总计有 2^{160} 个不同的地址，即 2^{160} 个Account。实际上，Ethereum的账户是可以用SMT来存的(尽管它没有这么做)。但是，SMT中很可能不会存在 2^{160} 个Account，可能明显小于这个数量。那么，对于空缺的账户，可以用预设值(例如空白)来填充，以表示这是个未启用的空白账户。在底层数据库中，可以不存储空白(是预设值)的叶子节点，而只存储非空白的叶子节点。
- [0187] 采用SMT的好处在于，很多时候可以压缩Merkle路径的大小，例如采用1bit来表示预设值，这就比用完整的hash值(例如hash256中的hash值长度是256bit)小很多，从而使得SPV证明的大小大大降低。
- [0188] S230：所述证明器验证第一交易根，并在打包交易读集的SPV证明基础上验证第一前状态根；两个验证通过后，按序执行所述打包交易并生成第二后状态根；验证第二后状态根与第一后状态根相等后，采用可信执行环境中的私钥对所述打包交易的信息签名，将签名作为证明并通过第二主网交易发送至主网的区块链账本。
- [0189] 所述证明器验证第一交易根，类似前述S130，不再赘述。
- [0190] 所述证明器在所述SPV证明基础上验证第一前状态根，具体包括：所述证明器基于SPV证明的读集以及Merkle路径计算Merkle根，得到第二前状态根，验证第

二前状态根是否与第一状态根相等。如果相等，则说明所述证明器发来的第一前状态根是正确的。Hash计算的特性：每个计算中的值发生一点小的改变，都会使得最终的结果产生很大的不同。基于这个特性，使得证明器无法造假，即无法提供错误的叶子节点中的状态或是提供错误的Merkle树节点的hash值的基础上产生一个正确的结果，即无法产生正确的Merkle根。具体的，所述证明器可以计算图6中左下角节点8对应内容的hash值作为节点8的值，计算节点9对应内容的hash值作为节点9的值；将节点8和节点9的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值，作为节点4的hash值；将节点4和SPV证明中节点5的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值，作为节点2的hash值；将节点2和节点3的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值，作为节点1的hash值，即作为第二前状态根，从而验证第二前状态根是否与第一前状态根相等。

[0191] 所述证明器按序执行所述打包交易并生成第二后状态根，可以是所述证明器基于所述该打包交易涉及的读集按序执行所述打包交易，并在执行过程中更新写集，以及在所述SPV证明基础上根据更新的写集重算Merkle根，以生成第二后状态根。具体的，如图9所示，并假设处理的一个Batch中包含3个交易，分别是：

[0192] Tx_21: Alice→Bob 20

[0193] Tx_22: Bob→Alice 10

[0194] Tx_23: Alice→Bob 20

[0195] 这个Batch中的交易涉及的状态是Alice和Bob，因此，这个Batch涉及的读集可以包括Alice和Bob这两个账户。读集的SPV证明中，除了可以包括Alice: 50和Bob: 100这两个读集之外，还可以包括所在Merkle树的节点5的hash值，节点3的hash值。这样，证明器在这个Batch读集的基础上，也就是在Alice: 50和Bob: 100基础上，按序执行Batch中的所有交易。

[0196] 例如，Tx_21、Tx_22、Tx_23按序执行后产生的读写集包括如下：

[0197] Tx_21执行前的读集包括Alice: 50和Bob: 100，Tx_21执行后Alice的账户余额变化为30，Bob的账户余额变化为120，则产生的写集为Alice: 30和Bob: 120。

- [0198] Tx_22执行前的读集包括Alice: 30和Bob: 120, Tx_22执行后Alice的账户余额变化为40, Bob的账户余额变化为110, 则产生的写集为Alice: 40和Bob: 110。
- [0199] Tx_23执行前的读集包括Alice: 40和Bob: 110, Tx_23执行后Alice的账户余额变化为20, Bob的账户余额变化为130, 则产生的写集为Alice: 20和Bob: 130。
- [0200] 这个Batch中的交易按序执行完毕后, 产生的这个Batch的终局状态是Alice: 20和Bob: 130这两个写集。所述证明器结合该写集所在Merkle树的节点5的hash值, 节点3的hash值, 可以得到更新后的状态所在Merkle树根节点的hash值, 即第二后状态根。这里, 例如产生的写集仍然是Merkle树中第8、9两个节点对应的内容, 例如节点8对应的状态, Alice的余额相对于初始状态的50修改为20, 节点9对应的状态, Bob的余额相对于初始状态的100修改为130。所述证明器可以基于Alice:20重新计算叶子节点8的hash值, 并基于Bob:130重新计算叶子节点9的hash值, 从而产生新的hash值。则类似的, 所述证明器可以在产生的写集基础之上, 结合所述SPV证明中Merkle路径(即黑色3、5两个节点的hash值)计算得到Merkle根的hash值, 从而作为第二后状态根。
- [0201] 具体的, 所述证明器可以计算节点8对应的写集内容的hash值作为节点8的值, 计算节点9对应写集内容的hash值作为节点9的值; 将节点8和节点9的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值, 作为节点4的hash值; 将节点4和SPV证明中节点5的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值, 作为节点2的hash值; 将节点2和节点3的两个hash值例如做顺序拼接后再次以相同的hash算法计算hash值, 作为节点1的hash值, 即作为第二后状态根, 从而验证第二后状态根是否与第一后状态根相等。
- [0202] 进而, 所述证明器可以验证第二后状态根与第一后状态根是否相等。如果相等, 则说明Sequencer的执行过程是正确的, 因为从相同的初始状态开始, Sequencer按序执行打包交易的执行结果与所述证明器独立按序执行相同打包交易的执行结果是相同的。
- [0203] 这样, 所述证明器验证可以对所述打包交易的信息签名。具体的, 所述证明器验证可以对打包交易的Batch编号、打包交易的交易根、打包交易对应的Post Stat

细则 91,
18.12.2024

e Root中的至少一个签名。对这些内容中的至少一个进行签名，表明证明器对Sequencer执行本Batch所产生结果的认可。

- [0204] 如前所述，所述证明器可以采用TEE实现。TEE具有与外部完全隔离的可信执行环境，因此可以以保密的方式持有私钥。这样，证明器在验证上述第二后状态根与第一后状态根相等后，可以用自身私钥对所述打包交易的信息签名，以表示对签名内容的认可。
- [0205] 所述证明器可以采用TEE实现。TEE具有与外部完全隔离的可信执行环境，因此可以以保密的方式持有私钥。这样，证明器在验证上述第二后状态根与第一后状态根相等后，可以用自身私钥对所述打包交易的信息签名，以表示对签名内容的认可。
- [0206] 在S230中证明器对所述打包交易的信息签名之前，Layer2上交易的发起方可以作为挑战者对证明器中的TEE发起挑战。如果挑战成功，交易的发起方可以确认证明器中的TEE是合法的，且中加载的代码/数据是可信的。从而，Layer2上交易的发起方可以信任证明器中的TEE。
- [0207] 此外，在S230中证明器对所述打包交易的信息签名之后，其它验证者也可以作为挑战者对证明器中的TEE发起挑战，类似前述过程，不再赘述。而且，TEE可以对输入的代码/数据进行hash后签名，从而用自身的可信来担保执行的输入也是可信的；TEE还可以对输出(也就是执行的结果)的hash值签名，也是用自身的可信来担保执行的输入也是可信的。
- [0208] 这样，所述证明器验证基于所述初始状态按序独立执行(也可以称为重放)所述打包交易后，对所述打包交易的信息签名，即是用自身的可信来担保输出结果是可信的。进而，所述证明器可以将签名作为证明通过第二主网交易发送至主网的区块链账本。任何人对该签名产生质疑，可以通过对证明器中的TEE发起挑战来验证；如果验证通过，则可以基于对TEE的信任，确信其签名的交易信息是可信的。
- [0209] 对签名的验证即是采用TEE的公钥进行验证签名的过程，只要保证公钥是可信的，这个验证也将是可信且快速的。所述证明器可以将签名作为证明并通过第二主网交易发送至主网的区块链账本，包括证明器将签名作为证明并通过第二

主网交易发送至主网，调用主网上的卷叠合约；所述卷叠合约接收到所述交易后，通过合约中的验证逻辑验证所述签名的正确性。

[0210] 总之，证明器可以将证书通过第二主网交易发送至主网的区块链账本。一种典型的方式中，证明器可以将证书与所述证明器的签名一并通过第二主网交易发送至主网的区块链账本。当采用上述EPID方案时，所述证书包括IAS为所述证明器生成的远程认证报告，这样，验证者可以通过验证该证书来确定证明器对交易信息的签名是合法的。当采用上述DCAP方案时，所述证书可以包括认证服务为所述证明器颁发的证书。如果基于对认证服务的信任，则可以延伸到认证服务为证明器颁发的证书，从而验证者可以采用该证书中的公钥对证明器的签名进行验证。如果需要更高层次的信任，除了需要认证服务为证明器颁发的证书外，可能还需要包括IAS为所述认证服务生成的远程认证报告，这样，通过Intel权威的认证(即远程认证报告，相当于Intel为认证服务颁发的证书)可以保证认证服务签名的可信，加上认证服务为证明器颁发的证书，可以将信任向下传导，从而保证证明器签名的可信。这即是SGX DCAP以证书链的方式提供信任。

[0211] 上述验证者执行的验证逻辑可以内置于Rollup Contract合约中，对外可以以合约接口的方式提供，如验证接口。这样，所述证明器将签名作为证明并通过第二主网交易发送至主网，具体可以是在第二主网交易交易中调用的主网的Rollup Contract合约，包括证明器将签名作为证明并通过第二主网交易发送至主网，调用主网上的Rollup Contract合约的验证接口，通过该接口可以将所述签名传入。这样，验证接口内部可以执行验证逻辑，具体包括验证所述签名的合法性。验证签名合法性的过程，主要包括采用证明器的公钥验证签名的合法性。

[0212] 此外，通过验证接口还可以将所述签名对应的证明器的证书一并通过第二主网交易传入，所述证书中可以包括所述证明器的公钥。这样，除了采用证明器的公钥验证签名的合法性之外，还可以包括验证证明器证书的合法性。所述证书，如前所述，可以包括IAS为所述证明器生成的远程认证报告，或包括认证服务为所述证明器颁发的证书(还可以包括IAS为所述认证服务生成的远程认证报告)。相应的，验证证书，可以包括采用Intel的权威公钥验证证明器证书的合法

性，或者可以包括采用Intel的权威公钥验证认证服务的证书的合法性，并采用认证服务的证书中的公钥验证证明器证书的合法性。

- [0213] 如上，采用TEE作为二层网络上的证明器，由于TEE中重放交易并验证接近于原生执行的速度，因此可以带来更低的延迟。一旦本地的TEE通过远程认证，通过TEE的机制，CPU能够在Enclave中对数据进行处理，具有极高的运算效率，从而兼顾了数据安全性和计算效率。这就比ZK-Rollup的方式还要快很多，例如同样情况下，从ZK-Rollup生成proof需要间隔30~50分钟量级(或称小时级)的时间(图4中Block N到Block N+412)，缩短到本申请(可以称为TEE-Rollup)5~10分钟量级的时间(图6中Block N到Block N+62)，甚至在不采用监听模式，而是在S220后直接执行S230的情况下，时间将缩短到更少。并且，生成的证明是一个签名的大小，约为几十字节，与ZK-Rollup类似。
- [0214] 与前述类似的，所述对所述打包交易的信息签名，可以是对打包交易的Batch编号、打包交易的交易根、打包交易对应的后状态根中的至少一个签名。
- [0215] 所述对所述打包交易的信息签名，可以是对打包的若干个连续Batch的编号、打包交易的交易根、打包交易对应的后状态根中的至少一个签名；或是对打包的若干个连续Batch分组后的每个组中Batch编号、打包交易的交易根、打包交易对应的后状态根中的至少一个签名。
- [0216] 所述证明器验证第一交易根与第一前状态根之前，还可以包括：所述排序器发送一笔调用卷叠合约的第一主网交易至主网，该第一交易中包含所述打包交易的Batch Header中的各字段，以及经过压缩的交易或经过压缩的读写集(术语称为标准术语是State Diff)；
- [0217] 进而，所述证明器可以监听主网上的合约执行执行结果。具体与前述实施例类似，不再赘述。
- [0218] 而且，所述证明器还可以将所述证书与所述签名一并通过第二主网交易发送至主网的区块链账本。所述证书中可以包括IAS为所述证明器生成的远程认证报告，或包括所述证书包括认证服务为所述证明器颁发的证书，或还包括IAS为所述认证服务生成的远程认证报告。

- [0219] 所述证明器将签名作为证明并通过第二主网交易发送至主网的区块链账本，包括证明器将签名作为证明并通过第二主网交易发送至主网，调用主网上的卷叠合约。并还可以包括：所述卷叠合约接收到所述交易后，通过合约中的验证逻辑验证所述签名的正确性。所述卷叠合约接收到所述交易后，通过合约中的验证逻辑验证所述签名的正确性，还可以包括：验证证书的合法性，所述证书中包括所述证明器的公钥。
- [0220] 所述验证证书的合法性，具体可以包括：
- [0221] 采用Intel的权威公钥验证证明器证书的合法性；或者，
- [0222] 采用Intel的权威公钥验证认证服务的证书的合法性，并采用认证服务的证书中的公钥验证证明器证书的合法性。
- [0223] 上述方案中，整体来说，集成TEE的Prover要全部重新执行一遍交易，代价较大。而且，可以将Sequencer中执行交易的部分解耦，例如解耦为Executor，如图10所示。图10中，Layer2中的TxPool、Sequencer、Executor三者很可能是同一参与方，Prover很可能是另一参与方。且前一参与方一般还会具有一组数据库，这里称为Data Availability(数据可用性，DA)，用于持久化存储Layer2中的数据，这个数据一般包括原始交易和状态，以及还经常包括收据(Receipts)。这里的收据与主网上的收据类似，每一交易(包括普通转账交易和涉及智能合约的交易)被执行后都会生成相应的收据，包括该交易的执行结果或相关信息。具体的，对于普通转账交易，例如是交易状态(成功或失败)、交易的总Gas消耗、交易产生的日志信息、带有日志信息的事件列表。再例如，合约执行结果/相关信息可以表现为收据中的事件(event)事件的结构例如为如下格式：
- [0224] Event:
- [0225] [topic][msg]
- [0226] [topic][msg]
- [0227]
- [0228] 在上述示例中，事件的数量可以为一个或多个。每个事件可以包括主题(topic)和数据(data)等字段。

- [0229] Layer2上的客户端在发起Layer2的交易后，可以通过SDK与Layer2保持连接(例如是TCP长连接)，从而查询关注的交易的执行执行结果或相关信息。Layer2上的客户端在发起Layer2的交易后，一般都会期望尽快查询到Layer2的收据中关于该交易的执行结果信息，从而可以确认自身发起的交易在Layer2上已经被接受。
- [0230] 图10中用颜色指示出了该情况的参与方。而且，Executor也可以采用TEE实现。这样，Layer2中归属于不同参与方的Executor和Prover，都采用TEE实现。两个TEE设备可以通过互相挑战来确认对方的TEE属性，具体例如前述详细描述远程认证过程，包括EPID和DCAP方案，因此后续可以基于相应的IAS证书和证书链(包括IAS证书+DCAP证书)来确认有效期内TEE签名的正确性。
- [0231] 在该前提下，由于可以确认两个经过认证的TEE设备的合法身份，因此可以在建立对TEE设备的信任的基础上通过流程优化来降低Prover的验证负担。本申请提供一种实现二层网络卷叠的方法实施例，其中二层网络中的执行器和证明器均采用可信执行环境实现，所述方法包括如下：
- [0232] S310：二层网络的排序器从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器。
- [0233] 与前述类似，Rollup中可以在Layer1上部署Rollup Contract。Layer2中，交易池TxPool可以接收Layer2上的客户端发起的交易。Sequencer可以从交易池中拉取交易，并例如按照这些交易中的时间戳对这些交易排序并打包。
- [0234] 另一方面，排序器可以从DA中拉取当前打包交易执行前的全部状态。进而，排序器可以将打包交易执行前的全部初始状态发送至证明器；或者，为了降低发送至证明器的数据量，可以将当前打包交易读集的SPV发送至证明器，而不是发送打包交易执行前的全部初始状态。
- [0235] 所述打包交易的读集，可以通过解析所述打包交易得到。例如对于普通转账交易，可以是转账涉及的账户地址。账户地址与其账户余额构成key-value形式的键值对，数据库(DA)中一般会按照key-value的方式存储账户地址和其余额，也会以同样的key-value的方式存储涉及智能合约中的状态地址及其值。这样，

排序器可以将交易中的账户作为key，查询DA中打包交易执行前该key对应的value。查询的交易执行前的key-value，也称为读集。同样的，对于涉及智能合约的交易，可以通过解析合约的代码来得到状态变量的key，进而可以根据这个key在DA中查询对应的value，这也是读集。

[0236] 例如对于打包并排序后1个Batch中的100笔交易，其执行前假设共有5000个状态(假设包括外部账户、合约账户和合约存储中的状态变量)，而这100笔交易涉及其中的800个状态的读或写。也就是说其余4200个状态并不涉及读或写。因此，为了降低发送至证明器的数据量，可以采用SPV证明。SPV证明的例子，简单的形式如图6中的例子，可以省去不涉及读(写操作也会涉及首先读的操作，因此可以统称为读集)的叶子节点中的内容，而在保留涉及读的叶子节点的前提下，尽可能取较少的Merkle树上的中间节点或叶子节点的hash值。这样，可以是将当前打包交易读集的SPV发送至证明器，而不是发送打包交易执行前的全部初始状态。

[0237] 此外，例如执行器执行1个Batch中的100笔交易，只需在涉及的读集的基础上执行交易，产生写集，因此排序器可以发送所述读集至执行器，而不是发送全部的初始状态至执行器。当然，这是优化的实现；相对来说，发送打包交易执行前的全部初始状态至执行器，也是可以的。

[0238] 此外，所述排序器可以将该打包交易通过第一主网交易发送至主网上的卷叠合约，具体可以是通过Relayer发送。所述打包的交易可以存储于calldata中，具体与前述类似，不再赘述。而且，所述排序器可以将第一交易根也通过第一主网交易发送至主网上的卷叠合约，或所述排序器将第一交易根连同所述打包交易通过第一主网交易发送至主网上的卷叠合约，则所述第一交易根可以存储在Rollup Contract的合约状态中，具体与前述类似，也不再赘述。

[0239] 而且，可以是由所述排序器根据所述全部初始状态/读集的SPV构建Merkle树，将树根作为前状态根，并将该前状态根也通过第一主网交易发送至主网上的卷叠合约，所述前状态根可以存储在Rollup Contract的合约状态中。

- [0240] S320: 所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易, 并采用自身可信执行环境中的私钥对产生的写集生成第一签名; 将所述写集及第一签名发送至证明器。
- [0241] 所述执行器, 可以接收到发来的读集和所述打包交易, 所述读集具体可以包括读集的key及对应的value。由于读集是打包交易涉及的读集, 也就是说, 打包交易中涉及的读、写操作都在这些读集的基础上操作。因此所述执行器可以在打包交易涉及的读集基础上按序执行所述打包交易, 产生写集。所述写集, 具体可以包括写集的key及对应的value。
- [0242] 延续上述例子, 打包并排序后1个Batch中的100笔交易, 共涉及800个状态的读(或写), 即涉及这800个读集的读/写操作。因此, 执行器可以在这个800个读集的基础上按序执行这100笔交易, 产生相应的逐笔交易的写集。
- [0243] 例如, 按序执行这100笔交易后产生的逐笔的写集如下:
- [0244] tx2001: {写集key1-写集value1, 写集key2-写集value2, ...};
- [0245] tx2002: {写集key1-写集value1, 写集key2-写集value2, ...};
- [0246]
- [0247] tx2100: {写集key1-写集value1, 写集key2-写集value2, ...}。
- [0248] 执行器可以采用自身可信执行环境中的私钥对上述写集生成第一签名, 进而可以将该写集及第一签名发送至证明器。
- [0249] 此外, 也可以是执行器可以在这个800个读集的基础上按序执行这100笔交易, 产生汇总后的写集, 例如按序执行这100笔交易后产生的汇总写集如下:
- [0250] {写集key1-写集value1, 写集key2-写集value2, 写集key3-写集value3, ...}
- [0251] 也就是说, 在后的写集中的value, 可能对在前相同key的写集value构成覆盖, 最终的写集中只会存在最后一次写的结果, 而不存有中间的变化过程中的写的结果。显然, 这样的好处是写集中元素个数会大大减少。
- [0252] 类似的, 执行器可以采用自身可信执行环境中的私钥对上述汇总的写集生成第一签名, 进而可以将该汇总写集及第一签名发送至证明器。

- [0253] 此外，还可以是执行器在打包交易执行前的全部初始状态的基础上按序执行这100笔交易，产生逐笔交易的写集或汇总后的写集，结果与上述类似，生成第一签名也类似，不再赘述。
- [0254] 除了所述排序器将该打包交易通过第一主网交易发送至主网上的卷叠合约外，也可以是所述执行器将该打包交易通过第一主网交易发送至主网上的卷叠合约，具体可以是通过Relayer发送。所述打包的交易可以存储于calldata中，具体与前述类似，不再赘述。而且，所述执行器可以将第一交易根也通过第一主网交易发送至主网上的卷叠合约，例如连同所述打包交易通过第一主网交易发送至主网上的卷叠合约，则所述第一交易根可以存储在Rollup Contract的合约状态中，具体与前述类似，也不再赘述。
- [0255] 而且，可以是由所述执行器根据所述全部初始状态/读集的SPV构建Merkle树，将树根作为前状态根，并将该前状态根也通过第一主网交易发送至主网上的卷叠合约，所述前状态根可以存储在Rollup Contract的合约状态中。
- [0256] 需要说明但是，由于所述执行器采用了可信执行环境，而可信执行环境可以保存有证明自身身份的私钥，因此，所述执行器可以采用自身可信执行环境中的私钥对第一主网交易中的内容签名并将该签名一并通过第一主网交易发送至主网上的卷叠合约。具体的，所述执行器例如采用可信执行环境的私钥对所述打包交易签名并通过第一主网交易发送所述打包交易及签名至主网上的卷叠合约；或者所述执行器可以采用可信执行环境的私钥对所述打包交易、第一交易根签名并通过第一主网交易发送所述打包交易、第一交易根及签名至主网上的卷叠合约；还可以是所述执行器可以采用可信执行环境的私钥对所述打包交易、第一交易根和前状态根签名并通过第一主网交易发送所述打包交易、第一交易根和前状态根及签名至主网上的卷叠合约；等。当然，对于排序器与执行器同属于同一参与方，可以由排序器发送部分内容，而由执行器发送另一部分内容；除了通过不同笔主网交易这种明显的方式外，为了降低在Layer1上的Gas消耗，可以均由Relayer通过同一个主网交易发送，例如第一主网交易。采用执行器发送主网交易至主网上的卷叠合约，好处在于可以包含执行器自身可信执行环境的签名，从而可以在Layer1的Rollup Contract中验证该签名。

- [0257] S330: 所述证明器验证所述第一签名通过后, 根据所述全部的初始状态/读集的SPV构建Merkle树, 并将所述执行器发来的写集更新至所述Merkle树的叶子节点中, 并更新Merkle根, 将更新后的Merkle根作为后状态根。
- [0258] 对于发来的是打包交易执行前的全部初始状态, 该初始状态包括key和对应的value, 则所述证明器可以将全部的这些初始状态作为叶子节点构建Merkle树(如前述的MPT或SMT树), 得到的Merkle树的根作为前状态根。进而, 所述证明器可以在所述全部初始状态基础上按序将逐笔交易的写集进行更新至所述Merkle树的叶子节点中, 并更新Merkle根, 直至打包的交易按序更新完毕, 则更新后的Merkle根作为后状态根。
- [0259] 此外, 所述证明器可以在所述全部初始状态基础上将所述汇总写集更新至所述Merkle树的叶子节点上, 并更新Merkle根, 则更新后的Merkle根作为后状态根。显然的, 这样的方式可以一次性的更新全部涉及的叶子节点, 速度更快, 而不需要像前者那样逐笔交易更新叶子节点。
- [0260] 另外, 所述证明器可以在所述读集SPV基础上构建Merkle树(如前述的MPT或SMT树), 得到的Merkle树的根作为前状态根。进而, 所述证明器可以在所述全部初始状态基础上按序将逐笔交易的写集进行更新至所述Merkle树的叶子节点上, 并更新Merkle根, 直至打包的交易按序更新完毕, 则更新后的Merkle根作为后状态根。
- [0261] 类似的, 所述证明器可以在所述读集SPV基础上将所述汇总写集更新至所述Merkle树的叶子节点上, 并更新Merkle根, 直至打包的交易按序更新完毕, 则更新后的Merkle根作为后状态根。同样的, 这样的方式可以一次性的更新全部涉及的叶子节点, 速度更快, 而不需要像前者那样逐笔交易更新叶子节点。
- [0262] S340: 所述证明器采用自身可信执行环境中的私钥对所述后状态根生成第二签名, 将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [0263] 所述证明器执行完S330后, 采用自身可信执行环境中的私钥对所述后状态根签名, 得到第二签名。

- [0264] 进而，所述证明器可以将第二签名作为证明并通过第二主网交易发送至主网的Rollup Contract合约。一种典型的方式中，证明器可以将证书与所述证明器的签名一并通过第二主网交易发送至主网的区块链账本。当采用上述EPID方案时，所述证书包括IAS为所述证明器生成的远程认证报告，这样，验证者可以通过验证该证书来确定证明器对交易信息的签名是合法的。当采用上述DCAP方案时，所述证书可以包括认证服务为所述证明器颁发的证书。如果基于对认证服务的信任，则可以延伸到认证服务为证明器颁发的证书，从而验证者可以采用该证书中的公钥对证明器的签名进行验证。如果需要更高层次的信任，除了需要认证服务为证明器颁发的证书外，可能还需要包括IAS为所述认证服务生成的远程认证报告，这样，通过Intel权威的认证(即远程认证报告，相当于Intel为认证服务颁发的证书)可以保证认证服务签名的可信，加上认证服务为证明器颁发的证书，可以将信任向下传导，从而保证证明器签名的可信。这即是SGX DCAP以证书链的方式提供信任。
- [0265] 上述验证者执行的验证逻辑可以内置于Rollup Contract合约中，对外可以以合约接口的方式提供，如验证接口。这样，所述证明器将签名作为证明并通过第二主网交易发送至主网，具体可以是在第二主网交易交易中调用的主网的Rollup Contract合约，包括证明器将签名作为证明并通过第二主网交易发送至主网，调用主网上的Rollup Contract合约的验证接口，通过该接口可以将所述签名传入。这样，验证接口内部可以执行验证逻辑，具体包括验证所述签名的合法性。验证签名合法性的过程，主要包括采用证明器的公钥验证签名的合法性。
- [0266] 此外，通过验证接口还可以将所述签名对应的证明器的证书一并通过第二主网交易传入，所述证书中可以包括所述证明器的公钥。这样，除了采用证明器的公钥验证签名的合法性之外，还可以包括验证证明器证书的合法性。所述证书，如前所述，可以包括IAS为所述证明器生成的远程认证报告，或包括认证服务为所述证明器颁发的证书(还可以包括IAS为所述认证服务生成的远程认证报告)。相应的，验证证书，可以包括采用Intel的权威公钥验证证明器证书的合法性，或者可以包括采用Intel的权威公钥验证认证服务的证书的合法性，并采用认证服务的证书中的公钥验证证明器证书的合法性。

- [0267] 前述提到，所述证明器可以在所述读集SPV基础上构建Merkle树(如前述的MPT或SMT树)，得到的Merkle树的根作为前状态根，相应的，这里所述证明器还可以将所述前状态根通过第二主网交易发送至主网上的卷叠合约，所述签名包括对该前状态根的签名。
- [0268] 如上，二层网络中的执行器和证明器均采用可信执行环境实现，由执行器生成打包交易执行所产生的写集并通过TEE签名，由证明器生成打包交易执行后的后状态根，可以进一步降低Prover的验证负担，从而提升Prover的验证效率，因此可以带来更低的延迟。
- [0269] 为了降低在Layer1上的Gas消耗，可以通过一笔第二主网交易发送多个连续Batch的各个前状态根和后状态根，此时方案如下：
- [0270] S410：二层网络的排序器从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；所述打包交易包括至少两个连续的Batch；
- [0271] S420：所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的至少两个连续Batch的写集生成第一签名；将所述写集及第一签名发送至证明器；
- [0272] S430：所述证明器验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，将所述执行器发来的写集更新至每个Batch的所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为该Batch的后状态根；
- [0273] S440：所述证明器采用自身可信执行环境中的私钥对所述至少两个连续Batch的各后状态根生成第二签名，将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [0274] 除了打包交易包含至少两个连续的Batch的情况下，第一交易根可以包括每个Batch的交易根或多个Batch的总交易根，所述前、后状态根可以包括每个Batch的前、后状态根或该至少两个连续的Batch的总前、后状态根外，其它实现细节与上述类似，不再赘述。

- [0275] 上述过程中，当Executor TEE的性能较高时，整体的性能瓶颈可能在于Prover TEE验证的过程。这时，可以将Prover TEE扩充为多个，从而并行执行上述过程。显然的，多个Prover TEE并行执行可以提升Layer1上证明的速度，从而进一步降低Layer1上交易确认的延迟。
- [0276] 以下介绍本申请一种二层网络实施例，包括交易池、排序器、执行器和证明器，所述执行器和证明器均采用可信执行环境实现，其中：
- [0277] 交易池，用于接收二层网络上用户发送的交易；
- [0278] 排序器，从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；
- [0279] 执行器，在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名；将所述写集及第一签名发送至证明器；
- [0280] 证明器，验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；并采用自身可信执行环境中的私钥对所述后状态根生成第二签名，将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [0281] 所述证明器可以为至少两个且并行执行。
- [0282] 以下介绍本申请一种证明器，采用可信执行环境实现，并包括：
- [0283] 接收单元，用于接收排序器发送的打包交易执行前的全部初始状态/打包交易读集的SPV证明，以及接收执行器发送的写集及第一签名发送至证明器；
- [0284] 验证单元，用于验证第一签名；
- [0285] 执行单元，所述验证单元验证通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；

- [0286] 签名单元，采用自身可信执行环境中的私钥对所述后状态根生成第二签名采用可信执行环境中的私钥对所述后状态根生成第二签名；
- [0287] 发送单元，将所述第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网的区块链账本。
- [0288] 在20世纪90年代，对于一个技术的改进可以很明显地区分是硬件上的改进(例如，对二极管、晶体管、开关等电路结构的改进)还是软件上的改进(对于方法流程的改进)。然而，随着技术的发展，当今的很多方法流程的改进已经可以视为硬件电路结构的直接改进。设计人员几乎都通过将改进的方法流程编程到硬件电路中来得到相应的硬件电路结构。因此，不能说一个方法流程的改进就不能用硬件实体模块来实现。例如，可编程逻辑器件(Programmable Logic Device, PLD)(例如现场可编程门阵列(Field Programmable Gate Array, FPGA))就是这样一种集成电路，其逻辑功能由用户对器件编程来确定。由设计人员自行编程来把一个数字系统“集成”在一片PLD上，而不需要请芯片制造厂商来设计和制作专用的集成电路芯片。而且，如今，取代手工地制作集成电路芯片，这种编程也多半改用“逻辑编译器(logic compiler)”软件来实现，它与程序开发撰写时所用的软件编译器相类似，而要编译之前的原始代码也得用特定的编程语言来撰写，此称之为硬件描述语言(Hardware Description Language, HDL)，而HDL也并非仅有一种，而是有许多种，如ABEL(Advanced Boolean Expression Language)、AHD L(Altera Hardware Description Language)、Confluence、CUPL(Cornell University Programming Language)、HDCal、JHDL(Java Hardware Description Language)、Lava、Lola、MyHDL、PALASM、RHDL(Ruby Hardware Description Language)等，目前最普遍使用的是VHDL(Very-High-Speed Integrated Circuit Hardware Description Language)与Verilog。本领域技术人员也应该清楚，只需要将方法流程用上述几种硬件描述语言稍作逻辑编程并编程到集成电路中，就可以很容易得到实现该逻辑方法流程的硬件电路。
- [0289] 控制器可以按任何适当的方式实现，例如，控制器可以采取例如微处理器或处理器以及存储可由该(微)处理器执行的计算机可读程序代码(例如软件或固件)的计算机可读介质、逻辑门、开关、专用集成电路(Application Specific Integrated C

ircuit, ASIC)、可编程逻辑控制器和嵌入微控制器的形式,控制器的例子包括但不限于以下微控制器: ARC 625D、Atmel AT91SAM、Microchip PIC18F26K20 以及Silicone Labs C8051F320,存储器控制器还可以被实现为存储器的控制逻辑的一部分。本领域技术人员也知道,除了以纯计算机可读程序代码方式实现控制器以外,完全可以通过将方法步骤进行逻辑编程来使得控制器以逻辑门、开关、专用集成电路、可编程逻辑控制器和嵌入微控制器等的形式来实现相同功能。因此这种控制器可以被认为是一种硬件部件,而对其内包括的用于实现各种功能的装置也可以视为硬件部件内的结构。或者甚至,可以将用于实现各种功能的装置视为既可以是实现方法的软件模块又可以是硬件部件内的结构。

[0290] 上述实施例阐明的系统、装置、模块或单元,具体可以由计算机芯片或实体实现,或者由具有某种功能的产品来实现。一种典型的实现设备为服务器系统。当然,本申请不排除随着未来计算机技术的发展,实现上述实施例功能的计算机例如可以为个人计算机、膝上型计算机、车载人机交互设备、蜂窝电话、相机电话、智能电话、个人数字助理、媒体播放器、导航设备、电子邮件设备、游戏控制台、平板计算机、可穿戴设备或者这些设备中的任何设备的组合。

[0291] 虽然本说明书一个或多个实施例提供了如实施例或流程图所述的方法操作步骤,但基于常规或者无创造性的手段可以包括更多或者更少的操作步骤。实施例中列举的步骤顺序仅仅为众多步骤执行顺序中的一种方式,不代表唯一的执行顺序。在实际中的装置或终端产品执行时,可以按照实施例或者附图所示的方法顺序执行或者并行执行(例如并行处理器或者多线程处理的环境,甚至为分布式数据处理环境)。术语“包括”、“包含”或者其任何其他变体意在涵盖非排他性的包含,从而使得包括一系列要素的过程、方法、产品或者设备不仅包括那些要素,而且还包括没有明确列出的其他要素,或者是还包括为这种过程、方法、产品或者设备所固有的要素。在没有更多限制的情况下,并不排除在包括所述要素的过程、方法、产品或者设备中还存在另外的相同或等同要素。例如若使用到第一,第二等词语用来表示名称,而并不表示任何特定的顺序。

- [0292] 为了描述的方便，描述以上装置时以功能分为各种模块分别描述。当然，在实施本说明书一个或多个时可以把各模块的功能在同一个或多个软件和/或硬件中实现，也可以将实现同一功能的模块由多个子模块或子单元的组合实现等。以上所描述的装置实施例仅仅是示意性的，例如，所述单元的划分，仅仅为一种逻辑功能划分，实际实现时可以有另外的划分方式，例如多个单元或组件可以结合或者可以集成到另一个系统，或一些特征可以忽略，或不执行。另一点，所显示或讨论的相互之间的耦合或直接耦合或通信连接可以是通过一些接口，装置或单元的间接耦合或通信连接，可以是电性，机械或其它的形式。
- [0293] 本发明是参照根据本发明实施例的方法、装置(系统)、和计算机程序产品的流程图和/或方框图来描述的。应理解可由计算机程序指令实现流程图和/或方框图中的每一流程和/或方框、以及流程图和/或方框图中的流程和/或方框的结合。可提供这些计算机程序指令到通用计算机、专用计算机、嵌入式处理机或其他可编程数据处理设备的处理器以产生一个机器，使得通过计算机或其他可编程数据处理设备的处理器执行的指令产生用于实现在流程图一个流程或多个流程和/或方框图一个方框或多个方框中指定的功能的装置。
- [0294] 这些计算机程序指令也可存储在能引导计算机或其他可编程数据处理设备以特定方式工作的计算机可读存储器中，使得存储在该计算机可读存储器中的指令产生包括指令装置的制造品，该指令装置实现在流程图一个流程或多个流程和/或方框图一个方框或多个方框中指定的功能。
- [0295] 这些计算机程序指令也可装载到计算机或其他可编程数据处理设备上，使得在计算机或其他可编程设备上执行一系列操作步骤以产生计算机实现的处理，从而在计算机或其他可编程设备上执行的指令提供用于实现在流程图一个流程或多个流程和/或方框图一个方框或多个方框中指定的功能的步骤。
- [0296] 在一个典型的配置中，计算设备包括一个或多个处理器(CPU)、输入/输出接口、网络接口和内存。
- [0297] 内存可能包括计算机可读介质中的非永久性存储器，随机存取存储器(RAM)和/或非易失性内存等形式，如只读存储器(ROM)或闪存(flash RAM)。内存是计算机可读介质的示例。

- [0298] 计算机可读介质包括永久性和非永久性、可移动和非可移动媒体可以由任何方法或技术来实现信息存储。信息可以是计算机可读指令、数据结构、程序的模块或其他数据。计算机的存储介质的例子包括，但不限于相变内存(PRAM)、静态随机存取存储器(SRAM)、动态随机存取存储器(DRAM)、其他类型的随机存取存储器(RAM)、只读存储器(ROM)、电可擦除可编程只读存储器(EEPROM)、快闪记忆体或其他内存技术、只读光盘只读存储器(CD-ROM)、数字多功能光盘(DVD)或其他光学存储、磁盒式磁带，磁带磁磁盘存储、石墨烯存储或其他磁性存储设备或任何其他非传输介质，可用于存储可以被计算设备访问的信息。按照本文中的界定，计算机可读介质不包括暂存电脑可读媒体(transitory media)，如调制的数据信号和载波。
- [0299] 本领域技术人员应明白，本说明书一个或多个实施例可提供为方法、系统或计算机程序产品。因此，本说明书一个或多个实施例可采用完全硬件实施例、完全软件实施例或结合软件和硬件方面的实施例的形式。而且，本说明书一个或多个实施例可采用在一个或多个其中包含有计算机可用程序代码的计算机可用存储介质(包括但不限于磁盘存储器、CD-ROM、光学存储器等)上实施的计算机程序产品的形式。
- [0300] 本说明书一个或多个实施例可以在由计算机执行的计算机可执行指令的一般上下文中描述，例如程序模块。一般地，程序模块包括执行特定任务或实现特定抽象数据类型的例程、程序、对象、组件、数据结构等等。也可以在分布式计算环境中实践本说明书一个或多个实施例，在这些分布式计算环境中，通过通信网络而被连接的远程处理设备来执行任务。在分布式计算环境中，程序模块可以位于包括存储设备在内的本地和远程计算机存储介质中。
- [0301] 本说明书中的各个实施例均采用递进的方式描述，各个实施例之间相同相似的部分互相参见即可，每个实施例重点说明的都是与其他实施例的不同之处。尤其，对于系统实施例而言，由于其基本相似于方法实施例，所以描述的比较简单，相关之处参见方法实施例的部分说明即可。在本说明书的描述中，参考术语“一个实施例”、“一些实施例”、“示例”、“具体示例”、或“一些示例”等的描述意指结合该实施例或示例描述的具体特征、结构、材料或者特

点包含于本说明书的至少一个实施例或示例中。在本说明书中，对上述术语的示意性表述不必须针对的是相同的实施例或示例。而且，描述的具体特征、结构、材料或者特点可以在任一个或多个实施例或示例中以合适的方式结合。此外，在不相互矛盾的情况下，本领域的技术人员可以将本说明书中描述的不同实施例或示例以及不同实施例或示例的特征进行结合和组合。

[0302] 以上所述仅为本说明书一个或多个实施例的实施例而已，并不用于限制本说明书一个或多个实施例。对于本领域技术人员来说，本说明书一个或多个实施例可以有各种更改和变化。凡在本说明书的精神和原理之内所作的任何修改、等同替换、改进等，均应包含在权利要求范围之内。

权利要求书

- [权利要求 1] 一种实现二层网络卷叠的方法，其中二层网络中的执行器和证明器均采用可信执行环境实现，所述方法包括：
- 二层网络的排序器从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；
- 所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名；将所述写集及第一签名发送至证明器；
- 所述证明器验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；
- 所述证明器采用自身可信执行环境中的私钥对所述后状态根生成第二签名，将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。
- [权利要求 2] 如权利要求1所述的方法，所述方法还包括：
- 所述排序器将该打包交易通过第一主网交易发送至主网上的卷叠合约。
- [权利要求 3] 如权利要求2所述的方法，所述方法还包括：
- 所述排序器生成所述打包交易的第一交易根，将该打包交易通过第一主网交易发送至主网上的卷叠合约。
- [权利要求 4] 如权利要求2所述的方法，所述方法还包括：
- 所述排序器根据所述全部初始状态/读集的SPV构建Merkle树，将树根作为前状态根，并将该前状态根也通过第一主网交易发送至主网上的卷叠合约。
- [权利要求 5] 如权利要求1所述的方法，所述方法还包括：

所述执行器将该打包交易通过第一主网交易发送至主网上的卷叠合约。

[权利要求 6]

如权利要求5所述的方法，所述方法还包括：

所述执行器生成所述打包交易的第一交易根，将该打包交易通过第一主网交易发送至主网上的卷叠合约。

[权利要求 7]

如权利要求5所述的方法，所述方法还包括：

所述执行器根据所述全部初始状态/读集的SPV构建Merkle树，将树根作为前状态根，并将该前状态根通过第一主网交易发送至主网上的卷叠合约。

[权利要求 8]

如权利要求5-7中任一项所述的方法，所述方法还包括：

所述执行器采用自身可信执行环境中的私钥对第一主网交易中的内容签名并将该签名一并通过第一主网交易发送至主网上的卷叠合约。

[权利要求 9]

如权利要求1所述的方法，所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名，包括：

所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，生成逐笔交易的写集；

所述执行器采用自身可信执行环境中的私钥对产生的逐笔交易的写集生成第一签名。

[权利要求 10]

如权利要求1所述的方法，所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名，包括：

所述执行器在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，生成汇总的写集；

所述执行器采用自身可信执行环境中的私钥对产生的汇总写集生成第一签名。

- [权利要求 11] 如权利要求1所述的方法，所述证明器将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，包括：
所述证明器在所述全部初始状态/所述读集SPV基础上按序将逐笔交易的写集进行更新至所述Merkle树的叶子节点中，并更新Merkle根，直至打包的交易按序更新完毕。
- [权利要求 12] 如权利要求1所述的方法，所述证明器将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，包括：
所述证明器在所述全部初始状态/所述读集SPV基础上将所述汇总写集更新至所述Merkle树的叶子节点上，并更新Merkle根。
- [权利要求 13] 如权利要求1所述的方法，所述证明器还将所述全部初始状态/读集的SPV构建Merkle树的树根作为前状态根。
- [权利要求 14] 如权利要求13所述的方法，所述证明器还将所述前状态根通过第二主网交易发送至主网上的卷叠合约，相应的，所述签名包括对该前状态根的签名。
- [权利要求 15] 如权利要求1-14中任一项所述的方法，所述打包交易包括包括至少两个连续的Batch。
- [权利要求 16] 如权利要求15所述的方法，所述第一交易根包括每个Batch的交易根或多个Batch的总交易根；和/或，
所述前、后状态根包括每个Batch的前、后状态根或该至少两个连续的Batch的总前、后状态根。
- [权利要求 17] 如权利要求1-16中任一项所述的方法，所述证明器为至少两个且并行执行。
- [权利要求 18] 一种二层网络，包括交易池、排序器、执行器和证明器，其中：
交易池，用于接收二层网络上用户发送的交易；
排序器，从交易池中拉取交易，对所述拉取的交易排序并打包，所述排序器从数据可用性中读取打包交易执行前的全部初始状态/打包交易读集的SPV证明至证明器，并发送所述打包交易执行前的全部初始状态/读集和所述打包交易至执行器；

执行器，在所述打包交易执行前的全部初始状态/读集的基础上按序执行所述打包的交易，并采用自身可信执行环境中的私钥对产生的写集生成第一签名；将所述写集及第一签名发送至证明器；证明器，验证所述第一签名通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；并采用自身可信执行环境中的私钥对所述后状态根生成第二签名，将第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网上的卷叠合约。

- [权利要求 19]如权利要求18所述的二层网络，所述证明器为至少两个且并行执行。
- [权利要求 20]一种证明器，采用可信执行环境实现，并包括：
接收单元，用于接收排序器发送的打包交易执行前的全部初始状态/打包交易读集的SPV证明，以及接收执行器发送的写集及第一签名；
验证单元，用于验证第一签名；
执行单元，所述验证单元验证通过后，根据所述全部初始状态/读集的SPV构建Merkle树，并将所述执行器发来的写集更新至所述Merkle树的叶子节点中，并更新Merkle根，将更新后的Merkle根作为后状态根；
签名单元，采用自身可信执行环境中的私钥对所述后状态根生成第二签名采用可信执行环境中的私钥对所述后状态根生成第二签名；
发送单元，将所述第二签名作为证明并与所述后状态根一并通过第二主网交易发送至主网的区块链账本。