Zcash Protocol 说明书（译）

原文：https://zips.z.cash/protocol/protocol.pdf

翻译：By Ola

注：本译文包含了原文的第1～5章的重点内容（包含了Zcash协议的设计思想和具体实现），这5个章节同时描述了Sprout、Sapling和Orchard三个版本的协议，为了简化翻译工作，忽略了原文中提及的版本间的微小差异。

> Zcash是去中心化匿名支付方案Zerocash的一个具体实现，不仅修复了Zerocash的部分安全问题，还提升了性能和丰富了功能。它在类似Bitcoin的透明支付方案的基础上增加了基于*zk-SNARKs*的隐私支付方案，同时解决PoW挖矿的中心化问题。

1 介绍

目前Zcash已经发布了多个版本：Sprout、OverWinter、Sapling、Blossom、Heartwood、Canopy、Orchard(NU5)，本书重点介绍了Sprout、OverWinter和Sapling三个版本。

本书章节的结构如下：

- 符号 - 文档中定义的符号
- 概念 - 理解Zcash需要的主要抽象概念
- 抽象协议 - 根据理想的加密组件描述的高级协议
- 具体协议 - 抽象协议中的函数和编码的具体实现
- 网络升级 - Zcash协议升级的策略
- 和比特币的共识变化 - Zcash和Bitcoin在共识层的区别，包括PoW
- 和Zerocash协议的区别 - 与Zerocash协议相比的变化总结
- 附录：电路设计 - 二次约束程序在Sapling电路中的设计细节
- 附录：批处理优化 - 验证多个签名和多个证明的效率提升方案

1.1 警告

Zcash协议的安全依赖于共识，如果一个程序在和Zcash网络交互的时候发生了共识层面的分叉，它的安全性就被削弱甚至销毁了。分叉的原因并不重要：可能是程序bug或网络上其他软件故障，丢了钱的用户并不关心具体的原因是什么。话虽如此，规范预期行为对安全性分析、理解协议和维护Zcash

以及相关软件来说非常重要，所以如果发现了任何本书的错误，请在github上创建一个issue或者联系security@z.cash。

1.2 概述

Zcash中所有的资金（value）都属于某个链资金池（chain value pool），Zcash协议中仅有一个透明的链资金池（transparent chain value pool），每个隐私协议各有一个隐私的链资金池（译者注：如Sprout、Sapling和Orchard算三个不同的隐私协议，所以有三个不同的隐私的链资金池）。透明的链资金池中发生的转账机制类似于在Bitcoin中转账，和Bitcoin有相同的隐私属性。在隐私的链资金池中的转账通过notes携带资金，notes指定了具体的金额和隐私支付地址（译者注：即收款方的隐私地址），将notes发送到这个隐私支付地址即完成转账。类似于Bitcoin，这个地址有一个关联的私钥可以花费发送来的notes，在Zcash中叫spending key。

每个note都有一个关联的note commitment和nullifier（不同note的nullifier不能重复），所有note commitment构成一棵Merkle树，当创建note的交易被打包进区块，则这个note的commitment在树上的位置（note position）就确定了。计算出nullifier需要note的spending key，不知道spending key就无法将nullifier和note commitment或note position关联起来（译者注：即无法通过note commitment或note position计算出nullifier）。一个有效的未花费note就是指它的note commitment和位置已经被公开在区块里，但是nullifier还没有公开出来。

一个Zcash交易可能包含公开输入、输出和脚本（类似于Bitcoin中的脚本），还可能包含JoinSplit descriptions、Spend descriptions、Output descriptions或Action descriptions（译者注：不同版本的协议中包含不一样的description）。同时交易中还包含转账信息，即输入一些旧的隐私notes，产生新的隐私notes。Sprout中的每个JoinSplit descriptions最多包含2个输入和2个输出notes；Sapling中每个输入和输出的note都有一个description，Orchard中的每个Action description最多包含1个输出和1个输出note。不同的链资金池之间也可以互相转账，且必须将转账金额公开出来。

每个隐私转账中，输入的note的nullifier必须被公开（防止双花），输出的note的commitment也必须被公开（为了之后花费这个note），交易中还包含了一些zk-SNARK的证明和签名，来保证交易是真实有效的：

每个隐私输入：

- 链上有一个公开的note commitment，承诺的金额和note中的金额相等
- 如果金额不是0，就存在一个公开的note commitment，对应这个note
- prover知道这个note的proof authorizing key
- nullifier和note commitment都是计算正确的

每个隐私输出：

- 有一个公开的value commitment，对应输出note中的金额
- note commitment计算正确
- 输出note的nullifier跟其他的nullifier不会碰撞

隐私地址中包含一个传输密钥（transmission key），用来实现对称加密。除了发送方和接收方，其他人都不知道具体的传输密钥（译者注：发送方和接收方之间有一个对称加密的密钥交换方案），所以只有接收方能通过生成传输密钥来解密，从而知道发送的note是什么。

Zcash的隐私是基于当一个note被花费的时候，发送方只需要证明这个note对应的note commitment已经被公开了，而不需要说出具体是哪一个，这样就不知道是哪个交易创建了这个被花费的note。对攻击者来说，一个交易的input notes的溯源集合包含了所有这个攻击者不知道的notes（攻击者不能花费的notes）。对比其他一些只是将部分交易混合起来的隐私支付系统，它们的溯源集合就小很多。

2 符号

符号定义请参考原文。

3 概念

3.1 支付地址和密钥

用户必须有一个隐私支付地址（shielded payment address）才能收到别人的隐私支付，隐私支付地址由spending key生成。

Sprout、Sapling和Orchard中的密钥系统各有不同，如图1所示，箭头表示通过已有的密钥可以生成其他密钥，两条线表示使用同一个密钥。

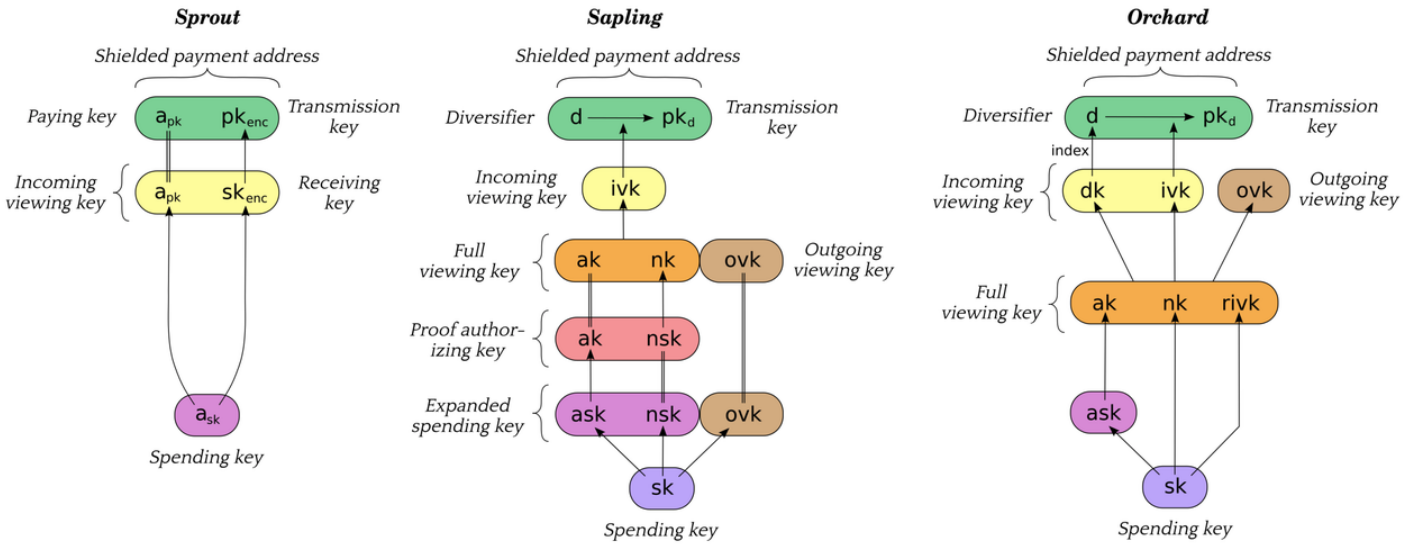


图1：密钥系统

不应该将隐私支付地（shielded payment address）、incoming viewing keys、full viewing keys和spending keys（译者注：expanded spending keys）这些协议细节暴露给普通用户，但是需要用户提供一个spending key，来生成其他密钥。用户利用ivk（incoming viewing keys）很方便地创建多个支付地址，最好给每个付款方创建一个支付地址，防止多个付款方发现是同一个接收地址，同一个ivk也不会增加在链上扫块的成本。

3.2 Notes

Sprout、Sapling和Orchard中都使用note进行转账，一个note表示一笔金额，可以被note的持有者花费掉，note的持有者就是当这个note发送到一个隐私支付地址，拥有这个地址对应的spending key的用户。

【Sprout】中的note是一个元组 (a_{pk}, v, ρ, rcm)：

- a_{pk} 是支付地址对应的paying key
- v 的范围是 $\{0..MAXMONEY\}$，表示这个note里金额的数量，单位是zatoshi
  - $1ZEC = 10^8\ zatoshi$
- ρ 是用来生成nullifier的输入信息
- rcm 表示 $NoteCommit^{Sprout}.Trapdoor$ （译者注：陷门指用来生成commitment的随机数）
- note的nullifier通过 ρ 和 收款人的spending key a_{sk} 生成
  - 花费这个note需要通过零知识证明用户知道 (ρ, a_{sk}) 的知识来计算并公开note对应的nullifier

【Sapling】中的note是一个元组 (d, pk_d, v, rcm)：

- d 是收款人的隐私支付地址里的多样化参数（diversifier）
- pk_d 是收款人的隐私支付地址里的多样化传输密钥（diversified transmission key）
- v 金额，同【Sprout中】
- rcm 同【Sprout】
- note的nullifier通过 ρ 和 收款人的spending key $nk(nullifier\ deriving\ key)$ 生成
  - 花费note就是通过零知识证明出知道 (ρ, ak, nsk) 的知识来计算并公开note对应的nullifier
  - 花费note还需要spend authorization signature，证明用户知道 ask

译者注：d 用来和ivk一起生成不同 pk_d ，一组 (d, pk_d) 就是一个隐私支付地址。

【Orchard】中的note是一个元组 $(d, pk_d, v, \rho, \psi, rcm)$：

- d、pk_d、v、ρ、rcm 同上
- ψ 是生成这个note的nullifier的额外随机数
- note的nullifier通过 ρ、ψ 和 收款人的spending key $nk(nullifier\ deriving\ key)$ 以及note commitment生成
  - 花费note就是通过零知识证明出知道 (ρ, ak, nk) 的知识来揭露nullifier
  - 花费note还需要spend authorization signature，证明知道 ask

3.2.1 Note明文和Memo字段

Note传输的时候是以密文的形式和cm（note commitment）一起存储在区块链上。每个description里的note明文通过对应的传输密钥 $pk^{new}_{enc,1..N^{new}}$ 加密成密文。

【Sprout】每个note明文包括：

- leadByte：一个前导字节（译者注：用来区分note类型，使用不同加解密算法）
- v：金额
- ρ：用于生成nullifier
- rcm：陷门（译者注：用于生成commitment的随机数）
- memo：备忘录，512字节以内的自定义字符串

【Sapling和Orchard】每个note明文包括：

- leadByte：一个前导字节（译者注：用来区分note类型，使用不同加解密算法）
- d：用于生成不同隐私支付地址
- v：金额
- rseed：用来生成rcm
- memo：备忘录，512字节以内的自定义字符串

3.3 区块链

在一个给定的时间点，每个完整的validato可能会收到一组可选的区块，这些区块构成了一棵以创世区块为root的树状结构，每个节点都通过区块头里的hashPrevBlock字段指向父节点。

从root到叶子节点的一条包含一个或多个有效区块的路径被称为有效区块链（valid block chain）。每个区块都有一个高度，创世区块高度是0，后续每个区块高度加1 （高度不能超过 $2^{31} - 1$）。

Validator会在所有有效区块链中选择工作量最大的那条作为最优有效区块链（best valid block chain）。共识协议被设计成确保给定一个区块高度，大多数节点最终都会达成同一个最优有效区块链的共识。完整的validator应该从多个来源接收区块，增加它找到最终反应共识状态的区块的概率（译者注：即找到的区块出现在最优区块链上）。

对给定网络的进行网络升级会从某个激活的区块哈希开始，并对这个达成社会共识。一个完整节点必须只在包含最新的网络升级被激活的区块的链上展示信息或进行资金交易，也应该对回滚的区块数做限制，例如zcashd和zebra限制为100个区块。

3.4 交易和Treestates

每个区块都包含一个或多个交易。

每个交易都有一个交易ID（transaction ID），交易ID被用来从区块里的交易树的叶子节点中的tx_out字段唯一标识一笔交易，交易树的root存储在hashMerkleRoot字段（译者注：区块中只存储交易树的root，交易树的叶子节点包含交易ID，具体的交易数据存储在其他地方）。

交易中的公开输入（transparent inputs）往公开交易资金池（transparent transaction value pool）中增加金额，而公开输出（transparent outputs）从公开交易资金池中移除金额。和Bitcoin中

一样，非coinbase交易里公开交易资金池中剩余的金额会作为矿工的费用，coinbase交易里公开交易资金池中的剩余金额会销毁。

共识规则：公开交易资金池中的剩余金额不能为负数。

每个交易都有一些关联的初始treestates，每个treestate包含：

- 一棵note commitment树（Merkle树）
- 一个nullifier集合

一个anchor是note commitment树的树根（root），它唯一确定了一棵note commitment树的状态，同时确定了nullifier集合的状态。

在一个给定的区块链中，treestates按以下方式串联起来：

- 创世区块的输入treestate是空的
- 每个区块第一笔交易的输入treestate是前一个区块的最终treestate
- 每个区块的其他交易的输入treestate是前一笔交易的输出treestate
- 每个区块的最终treestate是最后一笔交易的输出treestate

3.5 JoinSplit Transfers and Descriptions

一个JoinSplit description是包含在交易中的数据，描述了一个JoinSplit transfer，例如隐私转账。Sprout中资金转移的操作通过交易里的Zcash特定的JoinSplit transfer完成。

一个JoinSplit transfer花费 N^{old} 个notes和一个transparent input v_{pub}^{old}，创建出 N^{new} 个notes和一个 v_{pub}^{new}。这些关联在一个JoinSplit statement实例里，这个statement可以生成一个zk-SNARK proof。

Sprout中的一笔交易包含多个JoinSplit descriptions，分别对应多个JoinSplit transfer操作。v_{pub}^{new} 的总和会往这个交易的公开交易资金池中加钱，反之 v_{pub}^{old} 会从这个池子里减钱。每个JoinSplit description中的anchor字段都指向一个Sprout treestate。 N^{old} 个隐私输入note的nullifiers都会公开到链上，用来检测note是否被双花。

一笔交易中的每个JoinSplit description都会构建一个插页式输出treestate（interstitial output treestate），构建方法是将description中所有的note commitment和nullifier添加到anchor指定的输入treestate上。每个JoinSplit description的输出treestate都可以作为同一笔交易内后续其他JoinSplit descriptions的输入treestate。所以在一笔交易内，所有descriptions的treestates集合构成了一棵树，树中每个节点的父节点都由它们的anchor字段决定。

插业式treestate是必要的，因为当一个交易被创建出来的时候，它不知道最终会被打包进哪个区块里，所以它的anchors字段必须和交易的最终位置是独立的（译者注：只有区块里每笔交易的第一个description的anchor是由前一笔交易决定，后续其他descriptions的anchors只能由交易内的treestates决定）。

每个JoinSplit transfer的输入和输出金额必须是平衡的，虽然不能直接检查输入和输出金额，但是可以通过JoinSplit statement保证。

3.6 Spend Transfers, Output Transfers, and their Descriptions

Sapling中的notes不使用JoinSplit transfer，而是用一个Spend transfer表示一个隐私输入，用一个Output transfer表示一个隐私输出。交易里包含一组Spend descriptions和一组Output descriptions，分别表示Spend transfer和Output transfer。

一个Spend transfer花费一个note，一个Output transfer创建一个note，他们对应的Spend description和Output description都包含一个Pedersen value commitment，对这个note的value进行了承诺。且分别关联了一个Spend statement和Output statement的实例，分别可以生成一个证明。

我们使用Pedersen承诺的同态属性（译者注：承诺本质是椭圆曲线上的点）来保证余额是平衡的，把金额v1的承诺加上金额v2的承诺，得到的结果是一个新的金额的Pedersen承诺，且新的金额是v1+v2。减法类似。

对余额的保证就是把所有隐私输入的承诺加起来减去所有隐私输出的承诺，得到一个新的承诺，通过使用Sapling bindling signature证明新的承诺的金额等于网络中公开金额的变化。这个方法允许所有的zk-SNARK statements独立于彼此，增加了预计算的可能性。

一个Spend description指定了一个anchor，它是前面block的输出treestate。这个description也公开了一个nullifier用来防止note被双花。

共识规则：

- Spend transfers和Action transfers必须在这个交易的 $v^{balanceSapling}$ 上保持一致。
- 每个Spend description的anchor都必须是一个早期block的最终treestate。

3.7 Action Transfers and their Descriptions

Orchard中引入了Action transfers的概念，每个Action transfer都可选地花费一个note，以及可选地产生一个输出note。Action descriptions是包含在交易里的数据，描述了Action transfers。

一个Action transfer花费一个note n^{old} ，又创建一个新的note n^{new} ，它的Action description中包含一个Pedersen value commitment，承诺一个金额数值，比如花费的note的金额减去创建的note的金额的结果。它关联了一个Action statement的实例，可以生成一个zk-SNARK 证明。

每个版本5的交易可以包含一组Action descriptions，版本4的交易不能包含Action descriptions。

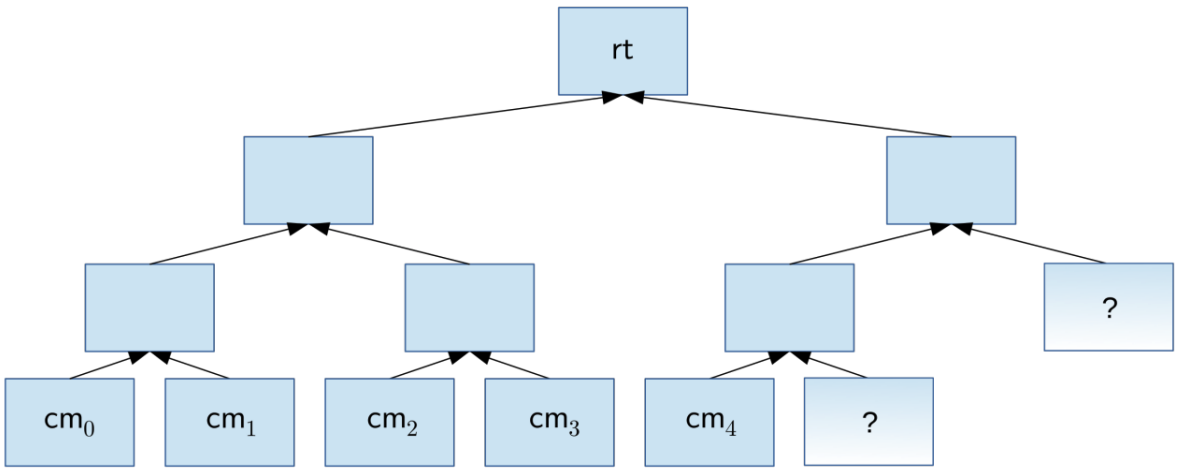
和Sapling中一样，利用Pedersen承诺的同态特性，通过使用Orchard binding signature保证所有value commitment加起来的结果（也是一个承诺），承诺的值和网络上公开金额的变化一样。

Action description中的字段是Spend description和Output description中字段的合并，不过只有一个值承诺（value commitment）。

共识规则：

- 一个交易的Action transfers必须和 $v^{balanceOrchard}$ 值保持一致
- 当有 $anchorOrchard$ 这个字段时，它必须指向一个前面区块的最终treestate。

3.8 Note Commitment树



一个note commitment树是一棵递增的Merkle树，具有固定的深度，被用来存储note commitments，这些commitments由JoinSplit transfer（【Sprout】）或 Spend transfers（【Scapling】）或 Action transfer（【orchard】）产生。和Bitcoin中的UTXO集合一样，这棵树的每个节点用来表示链上存在一笔钱以及可以去花这笔钱。和UTXO集合不同的是，这棵树不能用来防止双花，只能追加节点（译者注：防止双花通过nullifier集合来实现）。

每棵note commitment树的root都和一个treestate关联起来。树上每个节点都关联一个哈希值，树的深度为h，root是第0层，叶子节点共 $2^h - 1$ 个，用 M_i^h 表示第layer h层的第i个叶子节点。Note commitment在叶子层上的下标就是note position。

共识规则：

- 【Sprout】不能增加note commitment使叶子节点数超过 $2^{MerkleDepth^{Sprout}}$
- 【Sapling onward】不能增加note commitment使叶子节点数超过 $2^{MerkleDepth^{Sapling}}$
- 【NU5 onward】不能增加note commitment使叶子节点数超过 $2^{MerkleDepth^{Orchard}}$

3.9 Nullifier集合

每个完整的validator对每个treestate都维护一个nullifier集合，这个集合逻辑上和treestate关联起来。

一个有效的交易里的JoinSplit transfers、Spend transfers和Action transfers在被处理的时候都会公开一个nullifier并插入到新的treestate对应的nullifier集合中。Nullifier在整个区块链上都是唯一的，用来防治双花。

共识规则：不管在一个交易内还是在同一个区块链的多个交易内，nullifier都不能重复。

3.10 区块补贴，资金流， 和创始人奖励

和Bitcoin中一样，Zcash会在一个新的区块生成的时候创造货币，区块补贴（Block subsidy）指区块在生成的时候创造出的货币。

【Pre-Canopy】区块补贴包括矿工的补贴和创始人奖励

【Canopy onward】区块补贴包括矿工补贴和一系列资金流

和Bitcoin一样，矿工也能收到交易费。具体奖励和补贴的计算规则跟区块高度有关。

3.11 Coinbase交易

Coinbase交易指的是只有一个公开输入（其中prevout字段是null）的交易。每个区块的第一笔交易都是coinbase交易。Coinbase交易的作用是收集并且花费矿工补贴以及这个区块里的其他交易支付的交易费（译者注：这里花费的意思是将这笔钱奖励给矿工）。

【Pre-Canopy】coinbase 交易还必须支付Founders奖励

【Canopy onward】coinbase交易还必须支付资金流。

3.12 主网和测试网

Zcash协议主网的token是ZEC，测试网的token是TAZ，最小的单位是zatoshi，1 ZEC = 10^8 zatoshi，1 TAZ = 10^8 zatoshi。

- 主网创世区块：
  `00040fe8ec8471911baa1db1266ea15dd06b4a8a5c453883c000b031973dce08`
- 主网NU5版本激活区块：
  `0000000000d723156d9b65ffcf4984da7a19675ed7e2f06d9e5d5188af087bf8`
- 测试网创世区块：
  `05a60a92d99d85997cce3b87616c089f6124d7342af37106edc76126334a2c38`
- 测试网NU5版本激活区块：
  `0006d75c60b3093d1b671ff7da11c99ea535df9927c02e6ed9eb898605eb7381`

主网和测试网其他参数请阅读原文。

4 抽象协议

抽象对设计一个复杂系统来说非常重要，没有抽象即使直接设计和实现出来的计算机或密码学协议，也是不安全或不可信赖的。抽象的主要目的是让只负责系统一部分组件的人记住较少的信息，专注于理解自己负责的组件和其他组件之间的联系（译者注：不需要关注其他组件的细节），有助于维护和扩展系统，理解系统的安全性，以及向其他人介绍系统。

本书中，我们使用加密社区已经开发的一些抽象来模拟加密原语：Pseudo Random Functions、commitment schemes、signature schemes等。每个抽象原语都有自己的语法（接口）和安全属性。

Zcash扩展了部分语法和安全属性来达到协议需求，例如增加PRF的抗碰撞、扩展签名方案的功能和安全性等。

4.1 抽象加密方案

4.1.1 哈希函数

- $MerkleCRH^{Sprout}$、$MerkleCRH^{Sapling}$、$MerkleCRH^{Orchard}$
- 【Sprout】$hSigCRH$
  - 在“JoinSplit Descriptions”中使用
- 【Sprout】$EquihashGen$
  - 生成Equishhash solver的输入参数
- 【Sapling】CRH^{ivk}
  - “Sapling Key Component”中为隐私支付地址生成incoming view key
  - “Spend statement”中用来确认使用了正确的密钥来花费note
- 【Sapling】$MixingPerdersenHash$
  - “Note Commitment and Nullifiers”阶段生成note的 ρ
  - “Spend statement”中用来确认生成nullifier的 ρ 入参是正确的
- 【Sapling】$DiversifyHash^{Sapling}$、$DiversifyHash^{Orchard}$
  - 在“Sapling Key Components”或“Orchard Key Components”中从diversifier中生成diversified base

4.1.2 Pseudo随机函数

PRF_x 表示一个通过x指定的伪随机函数(Pseudo Random Function，且一个入参为x)。

Sym 表示Symmetric Encryption（对称加密）。

【Sprout】中需要四个独立的 PRF_x：

- PRF^{addr}
  - 在从spending key生成隐私支付地址的过程中也用到
- PRF^{pk}
- PRF^{ρ}
- $PRF^{nfSprout}$

这四个都在“JoinSplit Statement”中使用。

【Sapling】中还需要三个额外的 PRF_x：

- PRF^{expand}
- $PRF^{ockSapling}$

- $PRF^{nfSapling}$

【Orchard】中需要 PRF^{expand} 和两个额外的 PRF_x：

- $PRF^{ockOrchard}$
- $PRF^{nfOrchard}$

4.1.3 Pseudo随机置换

PRP_x 表示x指定的伪随机置换（Pseudo Random Permutation）。

【Orchard】中使用一个 PRP_x：

- PRP^d
  - 在从diversifier key生成diversifiers的时候使用

4.1.4 对称加密

Sym表示一次性授权的对称加密方案（译者注：一次性指对不同消息使用不同密钥进行加密），密钥空间（keyspace）是 $Sym.K$，明文是 $Sym.P$，密文是 $Sym.C$。

- Sym.Encrypt: $Sym.K \times Sym.P \to Sym.C$ 表示加密算法
- Sym.Decrypt: $Sym.K \times Sym.C \to Sym.P \cup \{\perp\}$ 表示解密算法，则
  - $Sym.Decrypt_k(Sym.Encrypt_k(P)) = P$。

4.1.5 密钥协商

KA（Key Agreement）指加密协议的双方通过自己的私钥和对方的公钥生成一个对称密钥。

KA方案中定义了 $KA.Public$、$KA.Private$和$KA.SharedSecret$ 三种类型，分别表示public keys、private keys和shared secrets。

- KA.DerivePublic: $KA.Private \times KA.Public \to KA.Public$ 表示从私钥和一个基点派生出一个新的公钥。
  - KA.DerivePublic可能是KA.Public的真子集。
- KA.Agree: $KA.Private \times KA.Public \to KA.SharedSecret$ 表示协商函数。

4.1.6 密钥推导

KDF（Key Derivation Function）被定义并使用在一个特别的密钥协商方案和一次性授权的对称加密方案中。它使用shared secret（KA生成的）和额外参数派生出一个适用于（对称）加密方案的密钥。

【Sprout】【Sapling】【Orchard】这三个的KDF的输入各不相同。

【Sprout】KDF input:

- $\{1..N^{new}\}$ 中输出的下标
- h_{Sig}

- Shared Diffie-Hellman secret：sharedSecret
- 临时公钥: epk
- 收款人的公开传输密钥： pk_{enc}

【Sapling】KDF input:

- Shared Diffie-Hellman secret：sharedSecret
- 临时公钥: epk

【Orchard】KDF input:

- Shared Diffie-Hellman secret：sharedSecret
- 临时公钥: epk

通过KA协商出一个共享密钥，然后用这个共享密钥和其他参数做对称加密。

4.1.7 签名

一个签名方案定义了：

- Signing keys的类型 $Sig.Private$
- Validating keys的类型 $Sig.Public$
- Messages的类型 $Sig.Message$
- Signature的类型 $Sig.Signature$
- 随机的signing key生成算法 $Sig.GenPrivate$： $() \to Sig.Private$
- 单向的validating key生成算法 $Sig.DerivePublic : Sig.Private \to Sig.Public$
- 随机的签名算法 $Sig.Sign$： $Sig.Private \times Sig.Message \to Sig.Signature$
- 验证算法 $Sig.Validate : Sig.Public \times Sig.Message \times Sig.Signature \to B$
  - B 是0或1

Zcash使用四种签名方案：

- 一种签名方案可以被脚本验证，例如 $OP_CHECKSIG$和$OP_CHECKMULTISIG$，和Bitcoin中一样
- 一种叫 $JoinSplitSig$，用来签名一个交易，且这个交易中至少包含一个JoinSplit description
- 【Sapling onwrad】一种叫 $SpendAuthSig$，用来签名Spend transfers的授权
- 【Sapling onwrad】一种叫 $BindingSig$，Sapling binding signature用来保证Spend transfers和Output transfers的余额，防止在不同交易间被重放攻击，Orchard binding signature也类似。

用在脚本操作中的签名方案是通过ECDSA在一条secp256k1曲线上初始化。JoinSplitSig是在Ed25519曲线上初始化。SpendAuthSig和BindingSig是在RedDSA上初始化；【Sapling】上是Jubjub曲线，【Orchard】上是Pallas曲线。

4.1.7.1 带再随机化密钥的签名

> 本节对【Sapling】有效

一个签名方案（signature scheme）被称为 re-randomizable keys Sig，是指一个签名方案额外定义了：

- 一个随机数（randomizers）类型 $Sig.Random$
- 一个随机数生成器（randomizer generator） $Sig.GenRandom$ ： $() \to Sig.Random$
- 一个签名密钥随机化算法
- 一个验证密钥（validating key）随机化算法
- 一个好的恒等（identity）随机发生器

从而：

- 对任意 α ： $Sig.Random$ ， $Sig.RandomizePrivate_{\alpha}$ 是单向可逆的
- $Sig.RandomizePrivate_{O}$ 是 $Sig.Private$ 上的恒等函数
- 对任意 sk ： $Sig.Private$ ， $Sig.RandomizePrivate(\alpha, sk)$ 是同分布于 $Sig.GenPrivate()$ 上的
- 对任意 sk ： $Sig.Private$、α ： $Sig.Random$ ：
  $Sig.RandomizePublic(alpha, Sig.DerivePublic(sk)) =$
  $Sig.DerivePublic(Sig.RandomizePrivate(\alpha, sk))$

SURK-CMA攻击定义：Strong Unforgeability with Re-randomized Keys under adaptive Chosen Message Attack

对任意 sk ： $SigPrivate$ ，让 O_{sk} 是一个内置这个sk的签名预言机（signing oracle），且具有状态 Q ： $P(Sig.Message \times Sig.Signature)$ 。 Q 初始化为空集合{}，记录了查询的messages和关联的签名。

O_{sk} ： $Sig.Message \times Sig.Random \to Sig.Signature$ 。

（译者注： O_{sk} 作为预言机，内置了随机数 α 和私钥，查询的输入参数是消息message，输出是签名。）

Q ： $\{(Sig.Message, Sig.Signature)\}$ ，初始化为空{}。

第一次查询的时候输入消息m，算出签名 $\sigma = Sig.Sign_{Sig.RandomizePrivate(\alpha, sk)}(m)$ ，

更新 Q 的状态 $Q = Q \cup \{(m, \sigma)\}$

返回 σ ： $Sig.Signature$ 。

对一个随机的密钥 $sk \leftarrow Sig.GenPrivate()$ 来说， $vk = Sig.DerivePublic(sk)$ ，给一个攻击者vk和新的 O_{sk} 的实例（不知道语言机里的私钥和 α ），这个攻击者想找到 (m', σ', α') 使得 $Sig.Validate_{Sig.RandomizePublic(\alpha', vk)}(m', \alpha') = 1$， 且 $(m', \sigma') \notin O_{sk}.Q$ ，是不可能的。

4.1.7.2 签名密钥和验证密钥是单态的签名

> 本节对【Sapling】有效

一个签名方案被称为密钥单态（key monomorphism） Sig，是指一个签名方案需要额外定义：

- Signing keys上有一个阿贝尔群，满足运算op1:
  $Sig.Private \times Sig.Private \to Sig.Private$ 和有恒等的 O_{op1} （单位元）
- Validating keys上有一个阿贝尔群，满足运算op2: $Sig.Public \times Sig.Public \to Sig.Public$ 和有恒等的 O_{op2} （单位元）

$Sig.DerivePublic(sk1 op1 sk2) = Sig.DerivePublic(sk1)\ op2\ Sig.DerivePublic(sk2)$ 对任意sk1、sk2成立。也就是说， $Sig.DerivePublic$ 是一个从签名密钥群到验证密钥群的单态（单射同态）。

给定任意 $sk1$ ，从 $Sig.GenPrivate()$ 中得到 $sk2$ ， $sk1\ op1\ sk2$ 无法和 $Sig.GenPrivate()$ 区分出来。

4.1.8 承诺

一个承诺方案(COMM)指一个函数，给定一个在随机位置生成的承诺陷门（译者注：commitment trapdoor，就是一个随机数），和一个输入input，这个函数能被用来像下面这样在这个input上承诺陷门：

- 没有陷门，就没有关于这个承诺的信息会被揭露，这是隐藏（hiding）
- 给定陷门和input，这个承诺能被通过打开（open）的方式在输入上验证（译者注：只能在原始多项式上打开，不能在其他多项式上），这是绑定（binding）

注意：

- COMM.GenTrapdoor()不需要时均匀分布的
- 两个不同的陷门在同一个输入上承诺值相同，不违背binding特性

【Sprout】只用一个 $NoteCommit^{Sprout}$ 承诺方案。

【Sapling】使用 $NoteCommit^{Sapling}$ 和 $ValueCommit^{Sapling}$ 两个承诺方案。

【Orchard】使用三个承诺方案：

- $NoteCommit^{Orchard}$
- $ValueCommit^{Orchard}$
- $Commit^{ivk}$

4.1.9 代表群

一个Represented Group包括：

- 一个子群的阶 r_G ： N^+ ，必须是质数
- 一个辅助因子参数 h_G ： N^+
- 一个群 G ，阶是，包含一个加法 $+$ ： $G \times G \to G$ 和加法单位元 O_G

- 一个二进制长度参数 l_G ： N
- 一个表示方法 $repr_G$ ： $G \to B^{[l_G]}$ 和一个抽象方法 $abst_G$ ： $B^{[l_G]} \to G \cup \{-\}$
  - 对G所有的元素P： $abst_G(repr_G(P)) = P$

在群G上存在一个子群 $G^{(r)}$ ，阶是 r_G ，这个子群叫代表子裙represented subgroup，包含单位元 O_G 。不包含单位元的阶也是 r_G 的子群，写作 $G^{(r)*}$

4.1.10 坐标提取器

> 本节对【Sapling】有效

一个群G的坐标提取器就是一个函数 $Extract_{G^{(r)}}$ ： $G^{(r)} \to T$ ，T是某个类型。

Extract和repr的区别是Extract不需要有有效可计算的左逆。

（译者注：坐标提取器就是将椭圆曲线上定义的群里的一个点的横坐标提取出来）

4.1.11 群哈希

> 本节对【Sapling】有效

给定一个代表子群 $G^{(r)}$ ，这个子群上的一簇群哈希，写作 $GroupHash^{G^{(r)}}$ ，包括：

- 一个 $GroupHash^{G^{(r)}}.URSType$ 类型，URS: Uniform Random Strings，必须是随机获取的
- 一个表示输入的类型 $GroupHash^{G^{(r)}}.Input$
- 一个函数 $GroupHash^{G^{(r)}}$ ： $GroupHash^{G^{(r)}}.URSType \times GroupHash^{G^{(r)}}.Input \to G^{(r)}$

4.1.12 代表配对

一个represented pairing $PAIR$ 包括：

- 一个群的阶参数 r_{PAIR} ： N^+ ，必须是质数
- 两个代表子群 $PAIR_{1,2}^{(r)}$ ，阶都是 r_{PAIR}
- 一个群 $PAIR_T^{(r)}$ ，具有乘法操作 $PAIR_T^{(r)} \times PAIR_T^{(r)} \to PAIR_T^{(r)}$ ，和单位元 1_{PAIR}
- 上面三个群的生成器generator $P_{PAIR_{1,2,T}}$
- 一个配对函数 $\hat{e}$ ： $PAIR_1^{(r)} \times PAIR_2^{(r)} \to PAIR_T^{(r)}$ ，满足
  - （双线性Bilinearity）对所有的 a, b ： F_r^* ， P ： $PAIR_1^{(r)}$ ， Q ： $PAIR_2^{(r)}$ ，都有： $\hat{e}_{PAIR}([a]P, [b]Q) = \hat{e}_{PAIR}(P, Q)^{(a \cdot b)}$
  - （非退化）不存在一个 P ： $PAIR_1^{(r)*}$ ，对所有的 Q ： $PAIR_2^{(r)}$ ， 都有 $\hat{e}_{PAIR}(P, Q) = 1_{PAIR}$

4.1.13 零知识证明系统

Zcash使用3种证明系统：

- BCTV14，用来在【Sapling】之前证明和验证【Sprout】 JoinSplit statement
- Grouth16，用来在【Sapling】激活后证明JoinSplit statement，以及证明Spend statement和Output statement
- 【NU5 onwrad】Halo2，证明【Orchard】中的Action statement

其余零知识证明相关的讨论请参考原文。

4.2 密钥组件

4.2.1 Sprout密钥组件

在【Sprout】中， spending key a_{sk} 生成方法是随机选择一个长度是 $l_{a_{sk}}$ 的二进制数 $B^{[l_{a_{sk}}]}$ ，其他key生成如下：

- $a_{pk} \ := \ PRF^{addr}_{a_{sk}}(0)$
- $sk_{enc} \ := \ KA^{Sprout}.FormatPrivate(PRF^{addr}_{a_{sk}}(1))$
- $pk_{enc} \ := \ KA^{Sprout}.DerivePublic(sk_{enc}, KA^{Sprout}.Base)$

4.2.2 Sapling密钥组件

在【Sapling】中，spending key $sk: \ B^{l_{sk}}$ 也是随机选择的长度是 l_{sk} 的二进制数。从 sk 中可以生成:

- Spend authorizing key $ask: \ F^*_{r_J}$ ， $ask \ := \ ToScalar^{Sapling}(PRF^{epand}_{sk}[0])$
  - 如果ask是0，选一个新的 sk
- Proof authorizing key $nsk: \ F_{r_J}$ ， $nsk \ := \ ToScalar^{Sapling}(PRF^{epand}_{sk}[1])$
- Outgoing viewing key $ovk: \ B^{Y^{[l_{ovk}/8]}}$ ， $ovk \ := \ ToScalar^{Sapling}(PRF^{epand}_{sk}[2])$

然后生成：

- $ak: \ J^{(r)*} \ := \ SpendAuthSig^{Sapling}.DerivePublic(ask)$
- $nk: \ J^{(r)} \ := \ [nsk]H^{Sapling}$
- $ivk: \ \{0..2^{l^{Sapling}_{ivk}} - 1\} \ := \ CRH^{ivk}(repr_J(ak), repr_J(nk))$
  - 如果ivk是0，选一个新的 sk

【Sapling】中可以高效地创建很多多样化支付地址（diversified payment addresses），使用相同的spending authority key，这些支付地址共享同一个full viewing key和incoming viewing key。

创建一个新的多样化支付地址流程如下：

1. 给定一个incoming viewing key ivk
2. 重复选择一个多样化随机数 $d: \ B^{[l_d]}$ ，直到diversified base $g_d \ = \ DiversifyHash^{Sapling}(d)$ 不是无效值

3. 计算diversified transimission key $pk_d = KA^{Sapling}.DerivePublic(ivk, g_d)$
4. 最终多样化支付地址就是 $(d : B^{[l_d]}, pk_d : KA^{Sapling}.PublicPrimeSubgroup)$

每个spending key都有一个默认的diversified payment address，用户可以直接使用这个默认地址不需要感知diversified address。

4.2.3 Orchard密钥组件

在【Orchard】中Spending key sk 也是通过随机选择长度是 l_{sk} 的二进制数生成的，其他key生成如下：

- Spend authorizing key ask : $F^*_{r_P} \leftarrow ToScalar^{Orchard}(PRF^{expand}_{sk}([6]))$ ，ask可变
  - 如果ask是0，选一个新的 sk
- nullifier deriving key nk : $F_{q_P} = ToBase^{Orchard}(PRF^{expand}_{sk}[7])$
- Commit^{ivk} randomness $rivk$: $F_{r_P} = ToScalar^{Orchard}(PRF^{expand}_{sk}[8])$
- $ak^P = SpendAuthSig^{Orchard}.DerivePublic(ask)$
- 如果 $repr_P(ak^P)$ 的最后一个bit是1，则 $ask \leftarrow -ask$
- Spend validating key ak : $\{0..q_p - 1\} = Extract_P(ak^P)$
- KA^Orchard private key $ivk = Commit^{ivk}_{rivk}(ak, nk)$
  - 如果ivk是0或者无效，选一个新的sk
- 令 $K = I2LEBSP_{l_{sk}}(rivk)$
- 令 $R = PRF^{expand}_K(([0x82]||I2LEOSP_{256}(ak)||I2LEOSP_{256}(nk))$
- the diversifier key dk是R的前 $l_{dk}/8$ 个字节，outgoing viewing key ovk是R剩下的 $l_{dk}/8$ 个字节
- $(ak_{internal}, nk_{internal}, rivk_{internal}) = DeriveInternalFVK^{Orcahrd}(ak, nk, rivk)$
- $ivk_{internal} = Commit^{ivk}_{rivk_{internal}}(ak_{internal}, nk_{internal})$
  - 如果是0，选一个新的sk
- $K_{internal} = I2LEBSP_{l_{sk}}(rivk_{internal})$
- $R_{internal} = PRF^{expand}_{K_{internal}}([0x82]||I2LEOSP_{256}(ak_{internal})||I2LEOSP_{256}(nk_{internal}))$
- $dk_{internal}$ 是 $R_{internal}$ 的前 $l_{dk}/8$ 个字节， $ovk_{internal}$ 是 $R_{internal}$ 剩下的 $l_{dk}/8$ 个字节
- $ak_{internal} = ak$, $nk_{internal} = nk$

创建一个diversified payment address流程如下：

1. 给定 incoming viewing key (dk, ivk)
2. 从 $B^{[l_d]}$ 中选择一个diversifier index：index
   - 要专门选择index，不能随机选择
3. 计算diversifier $d := PRP^d_{dk}(index)$

4. 计算diversified base $g_d := DiversifyHash^{Orchard}(d)$
5. 计算diversified transmission key $pk_d := KA^{Orchard}.DerivePublic(ivk, g_d)$
6. 最终的diversified payment address就是 (d, pk_d)
7. diversifier index是0的地址就是默认的多样化支付地址

4.3 JoinSplit Descriptions

每个交易都有0个或多个JoinSplit descriptions，用来描述JoinSplit transfer。

如果交易里JoinSplit descriptions不是空的，那这个交易里还有一个JoinSplitSig public validating key和签名的编码（encodings）。

一个JoinSplit description里包括：

- v_{pub}^{old}：$\{0..MAX\_MONEY\}$，这个JoinSplit transfer从公开交易资金池里移除的金额
- v_{pub}^{new}：$\{0..MAX\_MONEY\}$，这个JoinSplit transfer从公开交易资金池里转入的金额
- rt^{Sprout}：$B^{[l_{Merkle}^{Sprout}]}$，表示anchor，要么是前面区块的输出treestate，要么是这个交易里前面JoinSplit transfer的输出treestate
- $nf_{1..N^{old}}^{old}$：$B^{[l_{PRF}][N^{old}]}$，输入notes的nullifiers的序列
- $cm_{1..N^{new}}^{new}$，输出notes的note commitments序列
- epk，KA里的公钥，用来生成加密的密钥
- $randomSeed$：$B^{[l_{seed}]}$，每一个JoinSplit description都需要独立随机选择的seed
- $h_{1..N^{old}}$，tags的序列，将一个 h_{sig} 绑定到每个输入notes的 a_{sk} 上
- $\pi_{ZKjoinSplit}$，一个zk proof，证明的statement是原始输入 rt^{Sprout}、$nf_{1..N^{old}}^{old}$、$cm_{1..N^{new}}^{new}$、v_{pub}^{old}、v_{pub}^{new}、h_{sig}、$h_{1..N^{old}}$
  - 【Sapling】之前是BCTV14证明，之后是Grouth16证明
- $C_{1..N^{new}}^{enc}$ encrypted output notes的密文组件序列
- $h_{Sig} = hSigCRH(randomSeed, nf_{1..N^{old}}^{old}, joinSplitPubkey)$

共识规则：

- v_{pub}^{old} 或 v_{pub}^{new} 有一个必须是0
- 【Canopy onward】v_{pub}^{old} 必须是0

4.4 Spend Descriptions

每个交易里有0个或多个Spend descriptions，每个spend description都被一个签名授权，这个签名叫spend authorization signature。

Spend description包括：

- cv： 输 note里的金额的承诺
- $rt^{Sapling}$ 是anchor，前面区块的输出treestate
- nf： 输 note的nullifier
- rk： 验证spendAuthSig的公钥，randomized validating key
- $\pi_{ZKSpend}$: statement (cv, rt, nf, rk) 的证明
- spendAuthSig: 是spend authorization signature
- cv和rt不能是小的阶(small-order)

4.5 Output Descriptions

Output descriptions代表Output transfer，每个交易里都有0个或多个Output descriptions，但没有跟Output descriptions关联的签名。

Output description包括：

- cv： 输出note的value commitment
- cm_u : Extract（cv）的结果,cv是输 note的commitment
- epk： KA的公钥
- C^{enc}： 加密输出note的加密密文
- C^{out}： 用来帮ock（outgoing cipher key）的持有者恢复收款人的多样化传输密钥 pk_d 和临时私 esk，进一步查看整个note明文
- $\pi_{ZKOutput}$： (cv, cm_u, epk) 的证明
- cv和epk不能是小端顺序（small order）

4.6 Action Descriptions

Action transfer被编码成Action description。每个version 5的交易都有一组0个或多个Action descriptions。

每个Action description都被一个签名授权，这个签名被叫做spend authorization signature。

一个Action description包括：

- cv^{net}： 输入note的金额减去输出note的金额的结果的commitment
- $rt^{Orchard}$： $\{0..q_P - 1\}$，anchor，是前面某个区块的输出treestate
- nf： $\{0..q_P - 1\}$ 输入note的nullifier
- rk： 用来验证spendAuthSig的randomized validating key
- spendAuthSig: 一个spend authorization signature

- cm_x：$\{0..q_P-1\}$，对output note的commitment做提取坐标的结果（译者注：commitment本质是椭圆曲线上的点，提取这个点的横坐标）
- epk：KA里的公钥
- C^{enc}: 加密输出note的密文
- C^{out}：ock（outgoing cipher key）的持有者用来恢复收款人的多样化传输密 pk_d 和 临时私钥 esk 的密码学组件，进一步可以查看整个note 明文
- enableSpends：0或1，这个标识位用来开启Action spends中的non-zero-value功能
- enableOutputs：0或1，这个标识位用来开启Action Outputs中的non-zero-value功能
- π：对 $(cv, rt^{Orchard}, nf, rk, cm_x, enableSpends, enableOutputs)$ 的证明

注意：rt、enableSpends和enableOutputs对这个交易里的所有Action transfers都是一样的，这三个field被编码在transaction body里，而不是在Action Description的结构体里，π 是和其他Action证明一起聚和编码在交易的proofsOrchard字段里。

4.7 发送Notes

4.7.1 发送Notes（Sprout）

想要转账Sprout里的隐私金额，发送方必须建一个包含一个或多个JoinSplit descriptions的交易并发送出去：

1. 生成新的JoinSplitSig密钥对:

$joinSplitPrivKey \leftarrow JoinSplitSig.GenPrivate()$

$joinSplitPubKey := JoinSpligSig.DerivePublic(joinSplitprivKey)$

2. 对每个JoinSplit description，发送方随机选择一个randomSeed和input notes
3. 随机选择 ϕ：$B^{[l_\phi^{Sprout}]}$
4. 使用下标index i：$\{1..N^{new}\}$ 计算每一个output note
   a. 选择随机的 $rcm_i \leftarrow NoteCommit^{Sprout}.GenTrapdoor()$
   b. 计算 $\rho_i = PRF_\phi^\rho(i, h_{Sig})$
   c. 计算 $cm_i = NoteCommit_{rcm_i}^{Sprout}(a_{pk}, v_i, \rho_i)$
   d. $np_i = (0x00, v_i, \rho_i, rcm_i, memo_i)$，然后 $np_{1..N^{new}}$ 使用transmitted notes ciphertext $(epk, C_{1..N^{New}}^{enc})$ 被加密成接收方的transmission keys $pk_{enc,1..,N^{new}}$
5. 为了防止信息泄漏，发送方应该把输入notes和输出notes的顺序打乱
6. 所有的JoinSplit descriptions都生成之后，发送方得到了dataToBeSigned数据，并用private JoinSplit signing key对这个数据签名得到 joinSplitSig
7. 包含joinSplitSig的被编码的（endcoded）交易然后被提交到点对点网络上

4.7.2 发送Notes（Sapling）

为了发送Sapling的隐私金额，发送方需要构建一个包含一个或多个Output descriptions的交易。

令 ovk 是Sapling的outgoing viewing key，能够用来解密这次支付，那ovk应该是以下三选一：

- ovk是发送方地址的outgoing viewing key
- ovk是关联了所有支付的“账户”的outgoing viewing key
- 无效值，发送方以后不能解密这次支付，保持支付信息的隐私性

每一个Output description，发送方选择一个value v：$\{0..MAX\_MONEY\}$，和一个目的支付地址 (d, pk_d)，然后执行下面的操作：

- 检查 pk_d 的有效性
- 计算 $g_d = DiversifyHash^{Sapling}(d)$，且 g_d 是有效值
- 选择一个随机数（uniformly random commitment trapdoor）
  $rcv \leftarrow ValueCommit.GenTrapdoor()$
- 如果leadByte
  - 是0x01，下一个区块的高度小于CanopyActiveHeight
    - 选择一个uniformly random ephemeral private key $esk \leftarrow KA.Private$（不为{0}）
    - 选择一个uniformly random commitment trapdoor
      $rcm \leftarrow NoteCommit.GenTrapdoor()$
    - 设置 $rseed = I2LEOSP_{256}(rcm)$
  - 是0x02，下一个区块的高度大于等于CanopyActiveHeight
    - 选择uniformly random $rseed \leftarrow B^{Y^{[32]}}$
    - 生成 $esk = ToScalar(PRF_{rseed}([4]))$
    - 生成 $rcm = ToScalar(PRF_{rseed}([5]))$
- $cv = ValueCommit_{rcv}(v)$
- $cm = NoteCommit_{rcm}(repr(g_d), repr(pk_d), v)$
- $np = (leadByte, d, v, rseed, memo)$
- 利用diversified base g_d 将note明文使用接收方的多样化传输密钥 pk_d 和 ovk（outgoing viewing key）加密，得到传输的note密文 (epk, C^{enc}, C^{out})。这个加密过程中还会用到cv、cmu和esk来生成ock。
- 生成证明 $\pi_{ZKOutput}$
- 返回 $(cv, cm, epk, C^{enc}, C^{out}, \pi_{ZKOutput})$
- 需要打乱Output descriptions的顺序，为了减少信息泄漏。

4.7.3 发送Notes（Orchard）

为了花费Orchard隐私金额，需要构建一个包含一个或多个Action descriptions的交易。

每个Action description，发送方都需要选择金额 v和目的支付地址 (d, pk_d)，然后执行以下操作：

- 检查 pk_d 是有效的
- 计算 $g_d = DiversifyHash(d)$
- 选择uniformy random commiment trapdoor $rcv \leftarrow ValueCommit.GenTrapdoor$
- 选择 $rseed \leftarrow B^{Y^{[32]}}$
- 令 $\rho = nf^{old}$，计算 $\underline{\rho} = I2LEOSP_{256}(\rho)$
- 生成 $esk = ToScalar(PRF_{rseed}([4]||\underline{\rho}))$
  - 如果esk mod r_p == 0，重复上面步骤选择一个新的rseed
- 生成 $rcm = ToScalar(PRF_{rseed}([5]||\underline{\rho}))$
- 生成 $\psi = ToBase(PRF_{rseed}([9]||\underline{\rho}))$
- 令 cv^{net} 作为input note减去output note的value值的commitment，使用rcv
- $cm_x = Extract_P(NoteCommit_{rcm}(repr(g_d), repr(pk_d), v, \rho, \psi))$
  - 如果cm_x是无效值，选择新的rseed
- $np = (leadByte, d, v, rseed, memo)$
- 将np加密成 pk_d 和ovk(outgoing viewing key)，同时生成ock
- 生成证明 π
- 返回 $(cv, cm_x, epk, C^{enc}, C^{out}, \pi)$

如果同一个Action transfer中没有真的Orcahrd note被花费掉，那发送方应该创建一个假的note（dummy note），使用dummy note的nullifier作为 ρ 。为了减少信息泄漏，应该将Action descriptions的顺序打乱。

4.8 Dummy Notes

4.8.1 Dummy Notes（Sprout）

JoinSplit description支持 N^{old} 个输入notes和 N^{new} 个输出notes，需要创建dummy notes来填满未使用的note（译者注：对多2个输入notes和2个输出notes）。

创建一个index是i的dummy【Sprout】 input note 如下：

1. 生成新的uniformly random spending key $a_{sk}^{old} \leftarrow B^{l_{PRF}}$
2. 生成1中 a_{sk} 的 paying key $a_{pk,\ i}$
3. 选择uniformly random $\rho_i^{old} \leftarrow B^{l_{PRF}}$ 和 $rcm_i^{old} \leftarrow NoteCommit.GenTrapdoor()$

4. 计算 $nf_i^{old} = PRF_{a_{sk,i}old}(\rho_i^{old})$
5. 令 $path_i$ 是一个JoinSplit statement的假的auxiliary input（隐私输入）的Merkle path
6. 生成JoinSplit proof，设置 $enforceMerklePath_i$ 为0

一个dummy output note就是一个普通的output note，但是携带的金额是0，隐私支付地址是随机的。

4.8.2 Dummy Notes （Sapling）

Sapling中不需要创建假的input note来填满未使用的notes，但是可以掩盖真实的隐私输入notes数量。

1. 选择随机数 $sk \leftarrow B^{[l_{sk}]}$
2. 生成full viewing key的 ak和nk
3. 使用 sk 生成多样化支付地址 (d, pk_d)
4. 令金额v = 0，pos = 0（译者注：pos是note commitment在Merkle树上的下标）
5. 选择随机数 $rcv \leftarrow ValueCommit.GenTrapdoor()$
6. 选择随机数 $rseed \leftarrow B^{Y^{[32]}}$
7. 推导出 $rcm = ToScalar(PRF_{rseed}^{expand}([5]))$
8. 令 $cv = ValueCommit_{rcv}(v)$
9. 令 $cm = NoteCommit_{rcm}(repr_J(g_d), repr_J(pk_d), v)$
10. 令 $\rho* = repr_J(MixingPedersenHash(cm, pos))$
11. 令 $nk* = repr_J(nk)$
12. 令 $nf = PRF_{nk*}(\rho*)$
13. 构造假的Merkle路径，在之后隐私输入中使用，生成zk proof（因为金额为0所以Merkle路径不会被验证）

Sapling中dummy output note生成和Sprout中一样，携带金额为0和发送到随机的隐私支付地址。

4.8.3 Dummy Notes（Orchard）

Orchard的dummy notes作用也是隐藏真实的隐私输入notes数量，构造方法如下：

1. 选择随机数 $sk \leftarrow B^{[l_{sk}]}$
2. 生成full viewing key $(ak, nk, rivk)$ 和使用 sk 生成多样化支付地址 (d, pk_d)
3. 令金额v = 0
4. 选择随机数 $rseed \leftarrow B^{Y^{[32]}}$
5. 选择随机数 $\rho^P \leftarrow P$ （译者注：椭圆曲线Pallas上的随机点）

6. 令 $\rho = Extract_x(\rho^P)$，$\underline{\rho} = I2LEOSP_{256}(\rho)$
7. 推导出 $rcm = ToScalar(PRF^{expand}_{rseed}([5]||\underline{\rho}))$
8. 令 $\psi = ToBase(PRF^{expand}_{rseed}([9]||\underline{\rho}))$
9. 令 $cm = NoteCommit_{rcm}(repr_P(g_d), repr_P(pk_d), v, \rho, \psi)$
10. 如果 $cm = \perp$，选择新的rseed重复上面过程
11. 令 $nf = DeriveNullifier_{nk}(\rho, \psi, cm)$
12. 构造假的Merkle路径用来生成零知识证明（因为金额是0，假的路径不会被验证）

Orchard中dummy output note生成和Sprout中一样，携带金额为0和发送到随机的隐私支付地址。

4.9 Merkle路径验证

Sprout、Sapling和Orchard中的Merkle路径验证是互相独立的。

Merkle 树layer用h表示，root是layer0，每一层有 $0..2^h - 1$ 共 2^h 个nodes。

M^h_i 表示layer h上index 是 i 的node $i \in [0, 2^h - 1]$，h = MerkleDepth。

叶子的node是按下标依次使用的，当一个note commitment加到树上的时候，就占据一个叶子节点。没使用的叶子节点的hash值是Uncommitted，【Sprout】中很难找到一个note n使得NoteCommitment(n) == Uncommitted，【Sapling】和【Orchard】中不可能产生一个hash值是Uncommitted。

在layer 0 ~ layer MerkleDepth-1 的层上的节点叫中间节点，关联到MerkleCRH的输出上。中间节点的值是从它的两个子节点算出来的：

$M^h_i := MerkleCRH(h, M^{h+1}_{2i}, M^{h+1}_{2i+1})$

- 对【Sapling】来说，Merkle hash值被编码成比特流，但是root节点被编码成 F_{q_J} 上的一个元素，因为它要作为Spend proof的入参。
- Action circuit可以实现为，任意节点的hash value是0，包括root，Merkle path validity check也可以通过。

4.10 SIGHASH交易哈希

Bitcoin和Zcash使用签名和对交易输入的非交互式证明来授权花费资产，但是这些签名和证明容易在其他交易里被重放攻击，所以需要把交易和特定的输入信息绑定，一起哈希成一个SIGHASH交易哈希，用这个哈希来给这次花费授权。同一个交易输入在【Sprout】【Sapling】和【Orchard】上的签名各不相同。

在Zcash中使用的零知识证明算法是可塑的，即攻击者可以在不知道具体输入的情况下，修改旧的证明来生成一个新的有效证明，类似于加密方案里的malleability攻击。

为了在把不同来源的spend authorizations绑定到一起的同时增加灵活性，，Bitcoin和Zcash定义了很多不同类型的SIGHASH（译者注：即根据需要对交易的不同字段进行哈希）。

例如有一个类型是SIGHASH_ALL，这个是对所有Zcash特有的签名字段进行哈希，例如 JoinSplit signatures、spend authorization signatures、Sapling binding signatures等，但不包含transparent input，以及所有scriptSig等非Zcash特有的字段。

在Zcash中，所有的SIGHASH类型都覆盖了Zcash特有的字段：nJoinSplit、vJoinSplit和joinSplitPubkey（以及后续升级带来的新字段），没有覆盖joinSplitSig。

4.11 不可塑性（Sprout）

令 $dataToBeSigned$ 是transaction的哈希值，不包含输入input，用的是SIGHASH_ALL SIGHASH类型。

为了保证一个JoinSplit description跟transparent inputs v_{pub}^{new} 和transparent outputs v_{pub}^{old} 以及其他同一个交易内的JoinSplit description是绑定的，需要给每个交易都生成一个临时的JoinSplitSig密钥对，用其中的私钥给dataToBeSigned签名，公钥被编码放在交易的joinSplitPubKey字段里。

令 h_{Sig}，PRF^{pk} ，对每个JoinSplit description，都计算 $h_i = PRF^{pk}_{a^{old}_{sk,i}}(i, h_{Sig})$ ，得到的 $h_{1..N^{old}}$ 的正确性是通过JoinSplit statement保证的，使得 $a^{old}_{sk,i}$ 的持有者能够使用joinSplitPubKey对应的私钥给这个交易签名授权。

4.12 余额（Sprout）

Bitcoin中，所有输入和输出的金额都是公开的，输出金额不能大于输入金额。一个交易里输入金额减去输出金额的差值会发送给矿工作为区块奖励。

Zcash通过增加JoinSplit transfers扩展了这种模式，每个JoinSplit transfer都可以看成是公开交易资金池的一个输入和一个输出。Sprout上的链资金池余额被定义为区块链上所有交易的 $v_{pub}^{old}(like\ output)$ 字段总和减去 $v_{pub}^{new}(like\ input)$ ，余额不能为负数。

一个JoinSplit description中 v_{pub}^{old} 和 v_{pub}^{new} 必有一个为0（译者注：如果都不为0，可以用 v_{pub}^{old} 减去 v_{pub}^{new} ）。

4.13 余额和Binding签名（Sapling）

Sapling中的余额定义是Spend transfers的金额减去Output transfers的金额， $v^{balanceSapling}$ 。

$v^{balanceSapling}$ 被编码成 $valueBalanceSapling$ 字段，在V4的交易中是明确编码，在v5的交易中如果交易内没有Spend descriptions或者没有Output descriptions， $v^{balanceSapling}$ 就为0。

正的 $v^{balanceSapling}$ 表示从Sapling隐私交易资金池里转钱到公开资金池里，即输入到公开资金池里；负的 $v^{balanceSapling}$ 表示从公开资金池里转钱到Sapling隐私交易资金池，即从公开资金池里输出出来。

Sapling的链资金池的余额是区块链上 $valueBalanceSapling$ 的总和的相反数，且不能为负数。

$v^{balanceSapling}$ 和 Spend descriptions和Output descriptions里的value commitments的一致性靠Sapling里的binging签名保证，这个签名有2个作用：

- 证明Spend transfers的金额减去Output transfers的金额和交易里的 $v^{balanceSapling}$ 是一致的

- 防止Output descriptions被用来在其他交易里重放攻击

假设一个交易有：

- n个Spend descriptions，带有value commitments: $cv_{1..n}^{old}$ ，承诺的value是 $v_{1..n}^{old}$ 和随机数 $rcv_{1..n}^{old}$
- m个Output descriptions，带有value commitments: $cv_{1..m}^{new}$ ，承诺的value是 $v_{1..m}^{new}$ 和随机数 $rcv_{1..m}^{new}$
- Sapling余额金额是 $v^{balanceSapling}$

那么 $v^{balanceSapling} = \sum_{i=1}^{n}(v_i^{old}) - \sum_{j=1}^{m}(v_j^{new})$ ，但是validator不能直接这样验证因为values没有透露出来，被藏在commitments里了。

相反，validator这样计算transaction binding validating key:

- $bvk^{Sapling} = \sum_{i=1}^{n} cv_i^{old} - \sum_{j=1}^{m} cv_j^{new} - ValueCommit_0^{Sapling}(v^{balanceSapling})$
  - bvk这个key没有在交易中存储，必须计算出来
  - 这里的求和和减法不是真的求和和减法，代表一种操作方便写文档

签名的发送方知道 $rcv_{1..n}^{old}$ 和 $rcv_{1..m}^{new}$ ，因此可以计算出对应的signing key：

- $bsk^{Sapling} = \sum_{i=1}^{n}(rcv_i^{old}) - \sum_{j=1}^{m}(rcv_j^{new})$

为了检查实现错误，签名的发送方还需要检查：

- $bvk^{Sapling} = BindingSig^{Sapling}.DerivePublic(bsk^{Sapling})$

validator可以这样检查balance是正确的：

- $BindingSig^{Sapling}.Validate_{bvk^{Sapling}}(Sighash, bindingSigSapling) = 1$
  - SigHash是SIGHASH transaction hash，类型是SIGHASH_ALL，不包含input

4.14 余额和Binding签名（Orchard）

Orchard的Action transfer可以表示一个Spend或一个Output，余额定义是spends的金额减去outputs的金额 $v^{balanceorchard}$ 。

其他和Sapling类似，只是使用不同的密码学组件。

4.15 Spend授权签名 (Sapling and Orchard)

SpendAuthSig指spend authorization signature scheme，被用来证明知道一个输入note的spending key授权支付（译者注：即能花费这个note）。

可以通过Spend statement或Action statement来证明知道spending key，这个SpendAuthSig的作用是在某些设备上，由于内存或算力的限制，不能生成证明以及验证证明，所以需要这个。

这个签名的Validating key 必须在Spend description中揭露出来，为了确保validating key不能关联到隐私支付地址或已经被花费掉的note的spending key上，需要使用带有再随机化密钥的签名方案（signature scheme with re-randomizable keys）。Spend statement或Action statement需要证明这个validating key是签名方使用自己知道的一个随机数来对一个spend authorization address做re-randomization得到的key。因为spend authorization signature是在SIGHASH transaction hash上计算的，所以不能重投放。

4.16 Note Commitments and Nullifiers

当一个交易有JoinSplit description、Spend description和Action description的时候，需要将所有的note commitments加到note commitment树上，所有的nullifiers也都插入到nullifier集合里，这个集合和treestate关联起来（译者注：treestate包含note commitment树和nullifier集合）。

在【Sprout】中，每个note都有一个 ρ 。

在【Sapling】中，每个被确定好位置的note都有一个关联的 ρ ，
$\rho = MixingPedersenHash(cm, pos)$ ，从note commitment和位置计算出来。

- 【Sprout】中，nullifier计算如下：
  - $nullfifier = PRF^{nfSprout}_{a_{sk}}(\rho)$
  - a_{sk} 是note 的 spending key
- 【Sapling】中，nullifier计算如下：
  - $nullifier = PRF^{nfSapling}_{nk*}(\rho*)$
  - $nk*$ 是这个note的nullifier deriving key表示形式： $repr(nullifier\ deriving\ key)$
  - $\rho* = repr_J(\rho)$
- 【Orchard】中，nullifier的计算方法有点复杂，主要通过 nk、ρ、ψ和cm 生成。

4.17 Zk-SNARK Statements

4.17.1 JoinSplit Statement (Sprout)

JoinSplit statement的原始输入(公开的)：

- rt^{Sprout}
- $nf^{old}_{1..N^{old}}$
- $cm^{new}_{1..N^{new}}$
- v^{old}_{pub}
- v^{new}_{pub}
- h_{Sig}
- $h_{1..N^{old}}$

prover还知道一些附加信息（隐私的）：

- $path_{1..N^{old}}$
- $pos_{1..N^{old}}$
- $n^{old}_{1..N^{old}}$
  - $n^{old}_i = (a^{old}_{pk,i}, v^{old}_i, \rho^{old}_i, rcm^{old}_i)$
- $a^{old}_{sk,1..N^{old}}$
- $n^{new}_{1..N^{new}}$
  - $n^{new}_i = (a^{new}_{pk,i}, v^{new}_i, \rho^{new}_i, rcm^{new}_i)$
- ψ
- $enforceMerklePath_{1..N^{old}}$

约束见原文。

4.17.2 Spend Statement (Sapling)

Spend statement的原始输入(公开的)：

- $rt^{Sapling}$
- cv^{old}
- nf^{old}
- rk

prover还知道一些附加信息（隐私的）：

- $path$
- pos
- g_d
- pk_d
- v^{old}
- rcv^{old}
- cm^{old}
- rcm^{old}
- α
- ak
- nsk

约束见原文。

4.17.3 Output Statement (Sapling)

Output statement的原始输入(公开的)：

- cv^{new}
- cm_u
- epk

prover还知道一些附加信息（隐私的）：

- g_d
- $pk*_d$
- v^{new}
- rcv^{new}
- rcm^{new}
- esk

约束见原文。

4.17.4 Action Statement (Orchard)

Action statement的原始输入(公开的)：

- $rt^{Orchard}$
- cv^{net}
- nf^{old}
- rk
- cm_x
- $enableSpends$
- $enableOutputs$

prover还知道一些附加信息（秘密的）：

- $path$
- pos
- g_d^{old}
- pk_d^{old}
- v^{old}
- ρ^{old}
- ψ^{old}

- cm^{old}
- rcm^{old}
- α
- ak^P
- nk
- $rivk$
- g_d^{new}
- pk_d^{new}
- v^{new}
- ψ^{new}
- rcm^{new}
- rcv

约束见文档。

有些参数需要检查是否是canonical编码，防止双花漏洞。

4.18 带内秘密分发 (Sprout)

在Sprout中，发送方需要传递给接收方的秘密是*v*、*ρ*、*rcm*和*memo*这四个值，会用传输密钥 pk_{enc} 来加密它们。接收方需要用他的 ivk （incoming viewing key ）来解密重新构造出原始的note和memo字段内容。一个临时公钥会在一个Split description的 N^{new} 个隐私输出中共享，所有加密的密文都封装成transmitted notes ciphertext的形式。

加密和解密都需要用到：

- Sym： 对称加密方案
- KDF： 密钥推导函数
- KA： 密钥协商方案
- h_{sig}： JoinSplit descriptions中的签名哈希值
  - $h_{sig} = hSigCRH(randomSeed, nf_{1..N^{old}}, joinSplitPubKey)$

4.18.1 加密（Sprout）

- KA : 密钥协商方案（key agreement scheme）
- $pk_{enc,1..N^{new}}$： 传输密钥（transmission keys），每个新创建的note的接收方地址都对应一个 pk_{enc}
- $np_{1..N^{new}}$: Sprout note的明文

加密过程：

- 生成新的 KA^{Sprout} 密钥对 (epk, esk)
- 对每个 $i \in 1..N^{new}$ ：
  - 令 P_i^{enc} 表示 np_i 的raw encoding
  - 令 $sharedSecret_i = KA^{Sprout}.Agree(esk, pk_{enc,i})$
  - 令 $K_i^{enc} = KDF^{Sprout}(i, h_{Sig}, sharedSecret_i, epk, pk_{enc,i})$
  - 令 $C_i^{enc} = Sym.Encrypt_{K_i^{enc}}(P_i^{enc})$
- 最终需要传输的notes密文就是 $(epk, C_{1..N^{new}}^{enc})$ 。

4.18.2 解密（Sprout）

- 令 $ivk = (a_{pk}, sk_{enc})$ 是接收方的incoming viewing key
- 令 pk_{enc} 是对应的从 sk_{enc} 产生的transmission key
- 令 $cm_{1..N^{new}}$ 是output 的note commitment
- 对每个 $i \in 1..N^{new}$ ，接收方解密收到的 $(epk, C_{1..N^{new}}^{enc})$
  - 令 $sharedSecret_i = KA^{Sprout}.Agree(sk_{enc}, epk)$
  - 令 $K_i^{enc} = KDF^{Sprout}(i, h_{Sig}, sharedSecret_i, epk, pk_{enc})$
  - 返回 $DecryptNoteSprout(K_i^{enc}, C_i^{enc}, cm_i, a_{pk})$ ，定义如下：
    - 令 $P_i^{enc} = Sym.Decrypt_{K_i^{enc}}(C_i^{enc})$
      - 如果 P_i^{enc} 无效，直接返回无效值
    - 从 P_i^{enc} 中提取 $np_i = (leadByte, v_i, \rho_i, rcm_i, memo_i)$
    - 令 $n_i = (a_{pk}, v_i, \rho_i, rcm_i)$
    - 如果 $leadByte \neq 0x00$, $NoteCommitment^{Sprout}(n_i) \neq cm_i$ ，返回无效值
    - 返回 $(n_i, memo_i)$

4.19 带内秘密分发（Sapling和Orchard）

Sapling和Orchard中需要传输的秘密是 d, v, rcm, $rseed$和$memo$ 。为了传输这些秘密，会用多样化传输密钥（diversified transmission key） pk_d 来加密它们。收款方用他的 $KA^{Sapling}$ 或$KA^{Orchard}$ 私钥 ivk 来解密。

4.19.1 加密（Sapling和Orchard）

- 令 pk_d ： $KA.PublicPrimeSubgroup$ 是一个note的recipient address的diversified transmission key
- 令 g_d ： $KA.PublicPrimeSubgroup = DiversifyHash(d)$

- 令 esk 是ephemeral private key
- 令 ovk 是note被发送的接收方的shielded payment address的outgoing viewing key，或者是个无效值
- 令 $np = (leadByte, d, v, rseed, memo)$ 是 note plaintext的编码数据
- 令 cv 是 value commitment， cm 是 note commitment

加密过程如下：

- 令 P^{enc} 表示 np 的raw encoding
- 令 $epk = kA.DerivePublic(esk, g_d)$
- 令 $ephemeralKey = LEBS2OSP(repr_G(epk))$
- 令 $sharedSecret = KA.Agree(esk, pk_d)$
- 令 $K^{enc} = KDF(sharedSecret, ephemeralKey)$
- 令 $C^{enc} = Sym.Encrypt_{K^{enc}}(P^{enc})$
- 如果 ovk 是无效值，选择一个随机的ock和op
  - $ock \leftarrow Sym.K$ 和 op
- 如果 ovk 不是无效值
  - 令 $cv = LEBS2OSP(repr_G(cv))$
  - 令 $cm* = LEBS2OSP_{256}(Extract_{G^{(r)}}(cm))$
  - 令 $ock = PRF^{ock}_{ovk}(cv, cm*, ephemeralKey)$
  - 令 $op = LEBS2OSP_{l_G+256}(repr_G(pk_d)||I2LEBSP_{256}(esk))$
- 令 $C^{out} = Sym.Encrypt_{ock}(op)$
- 传输的note密文就是 $(ephemeralKey, C^{enc}, C^{out})$

4.19.2 使用Incoming Viewing Key解密（Sapling and Orchard）

解密过程：

- 令 ivk 是接收方的KA private key
- 令 $(ephemeralKey, C^{enc}, C^{out})$ 是note的密文
- 令 $cm*$ 是 cmu 或者 cmx （分别对应【Sapling】【Orchard】）
  - $cm* = Extract_{G^{(r)}}(cm)$
- 令height是包含这个交易的block的高度

接收方解密note密文中的ephemeralKey和 C^{enc} :

- 令 $epk = abst_G(ephemeralKey)$

    - 如果是无效值，直接返回无效值
- 令 $sharedSecret = KA.Agree(ivk, epk)$
- 令 $K^{enc} = KDF(sharedSecret, ephemeralKey)$
- 令 $P^{enc} = Sym.Decrypt_{K^{enc}}(C^{enc})$
    - 如果是无效值，直接返回无效值
- 从 P^{enc} 提取 $np = (leadByte, d, v, rseed, memo)$
- 【Pre-Canopy】如果 $leadByte \neq 0x01$，返回无效值
- 计算 $\underline{rcm} = rseed$ 或...看文档
- 令 $rcm = LEOS2IP_{256}(\underline{rcm})$
- 令 $pk_d = KA.DerivePublic(ivk, g_d)$
- 计算n
    - 【Sapling】 $n = (d, pk_d, v, rcm)$
    - 【Orchard】 $n = (d, pk_d, v, \rho, \psi, rcm)$
- 计算 $cm' * = NoteCommitment(n)$，并进行检查，失败返回无效值
- 返回 $(n, memo)$

4.19.3 使用Full Viewing Key解密（Sapling和Orchard）

Outgoing viewing key的持有者解密note密文：

- 令 ovk 是outgoing viewing key
- 令height是包含这个交易的区块高度
- 令 $(ehpemeralKey, C^{enc}, C^{out})$ 是传输的note密文
- 令 $cv = cv$, $cm* = cmu|cmx$
- 令 $ock = PRF^{ock}_{ovk}(cv, cm*, ephemeralKey)$
- 令 $op = Sym.Decrypt_{ock}(C^{out})$
    - 如果op无效就直接返回无效值
- 从 op 中提取 $(pk*_d, \underline{esk})$
- 令 $esk = LEOS2IP_{256}(\underline{esk})$, $pk_d = abst_G(pk*_d)$
    - 如果 $esk >= r_G$ 或 pk_d 是无效值，直接返回无效值
    - 【NU5 onwrad】如果 $repr_P(pk_d) \neq pk*_d$，返回无效值
- 令 $sharedSecret = KA.Agree(esk, pk_d)$
- 令 $K^{enc} = KDF(sharedSecret, ephemeralKey)$

- 令 $P^{enc} = Sym.Decrypt_{K^{enc}}(C^{enc})$
  - 如果结果是无效值就直接返回无效值
- 从 P^{enc} 中提取 $np = (leadByte, d, v, rseed, memo)$
- 令 $\underline{rcm} = rseed$
  - 这是最简化结果，具体计算较复杂见文档
- 令 $rcm = LEOS2IP_{256}(\underline{rcm})$ ， $g_d = DiversifyHash(d)$
  - 结果无效就直接返回无效值
- 计算 n :
  - 【Sapling】 $n = (d, pk_d, v, rcm)$
  - 【Orchard】 $n = (d, pk_d, v, \rho, \psi, rcm)$
- 令 $cm'_* = NoteCommitment(n)$
  - 检查 $cm'_* \neq I2LEOSP_{256}(Extract_{G^{(r)}}(cm'_*))$ 或 $repr_G(KA > DerivePublic(esk, g_d)) \neq ephemeralKey$ ，上述情况返回无效值 $\perp$
- 返回 $(n, memo)$

4.20 扫描区块链（Sprout）

$ivk = (a_{pk}, sk_{enc})$ 是跟 a_{sk} 对应的incoming viewing key

pk_{enc} 是关联的传输密钥（transmission key）

- 对每个交易：
  - 对交易里的每个JoinSplit description:
    - 令 $(epk, C^{enc}_{1..N^{new}})$ 是 传输的note密文，对每个密文 $i \in 1..N^{new}$ ：
      - 使用 ivk 解密密文 (epk, C^{enc}_i) 得到 $(n, memo)$
      - 将 $(n, memo)$ 加到ReceivedSet
      - 使用 a_{sk} 计算 n的$nullifier$ ： nf
      - 将 $nf \rightarrow n$ 加到Nullifiermap
    - 令 $nf_{1..N^{old}}$ 是JoinSplit description的nullifiers，对 $i \in 1..N^{old}$ ：
      - 如果 nf_i 出现在NullifierMap中，就将 $NullifierMap(nf_i)$ 加到SpentSet中
- 返回 $(ReceivedSet, SpentSet)$

4.21 扫描区块链（Sapling和Orchard）

在Sapling和Orchard中，扫块只需要 nk和ivk，不需要像Sprout中那样使用spending key a_{sk}

nk和ivk 是从一个full viewing key派生出来的。

使用下面算法来利用 (nk, ivk) 来获取每个发送到shielded payment address的note信息，包括memo字段和note的状态（spent还是unspent）：

- 对每个交易：
  - 对交易内的每个Output description或Action description:
    - 使用 ivk 来解密note密文 (epk, C^{enc}) 得到 $(n, memo)$
    - 将 $(n, memo)$ 加到ReceivedSet中
    - 使用 nk 来计算n的nullifier nf
    - 将 $nf \ \rightarrow \ n$ 加到NullifierMap中。
  - 对Spend description或Action description中的每个nullifier nf :
    - 如果 nf 出现在NullifierMap，就将 $NullifierMap(nf)$ 加到SpentSet中
- 返回 $(ReceivedSet, SpentSet)$

注意：

- 上面的算法不使用 ovk 或者note传输密文 C^{out} ，使用 ovk 解密的好处是可以单独使用 ovk 就能恢复交易里的note明文信息。
- 如果只扫部分区块，可以修改上面的算法来把每个交易里的 C^{out} 解密出来，从而获取在扫描时间内被花费但没扫描到的note的信息。
- 上面的算法扫描不到“out-of-band”发送的notes（译者注：带外即没上链），或者密文错误的notes，只有这些notes的nullifier是已知的才能知道这些notes有没有被花费掉。

5 具体协议

5.1 整数，位序列和字节顺序

- $I2LEBSP_l(x)$: 把x表示为l bit长的二进制数（小端）
  - 0x12345678写入0x0000~0x0004 是 0x78 0x56 0x34 0x12
- $I2LEOSP_l(x)$ ： 把x表示为ceiling(l/8)个字节 （小端）
- $I2BEBSP_l(x)$: 把x表示为l bit长的二进制数（大端）
  - 0x12345678写入0x0000~0x0004 是 0x12 0x34 0x56 0x78
- $LEBS2IP_l(S)$ ： 把二进制数S表示为整数 （小端）
- $LEOS2IP_l(S)$ ： 把l/8个字节的S表示为整数 （小端）
- $BEOS2IP_l(S)$ ： 把l/8个字节的S表示为整数 （大端）

- $LEBS2OSP_l(x)$：把二进制数x右边补0，使得能被8整除，然后每8个bit转化为一个字节顺序连接起来
- $LEOS2BSP_l(S)$：把字节序列S转成二进制数

5.2 位布局图

我们有时会使用位布局图（bit layout diagrams），图中每个box都代表一个位序列（bit sequence）。图的阅读是从左到右，行的顺序是从上到下，跨多行的box没有意义（译者注：没有跨多行的box）。每个box的位长度l显示给定在box中，除非长度能明显看出来。

整个图表示先把这些位序列连接起来，然后每8个比特（从最高有效位到最低有效位）作为一个字节的字节序列，每个字节的最高有效位都在最左边。

5.3 常量

常量定义请参考原文。

5.4 具体的密码学方案

5.4.1 哈希函数

5.4.1.1 SHA-256, SHA-256d, SHA256Compress, 和 SHA-512 哈希函数

- SHA-256：任意字节输入，输出为32个字节
  - $NoteCommitment^{Sprout}$ 中使用了完整的(full) SHA-256 hash函数。
- SHA-256d：执行2次SHA-256，还是输出32个字节
  - 用来哈希block headers
- SHA256Compress：512bit输入，输出是256bit
  - 用来初始化一些伪随机算法PRF和MerkleCRH^{Sprout}
- SHA-512: 任意字节输入，输出64个bytes
  - Ed25519用了SHA-512

5.4.1.2 BLAKE2 哈希函数

Zcash使用BLAKE2的两个变种BLAKE2b和BLAKE2s

- $BLAKE2b-l(p,x)$ 工作在顺序模式（sequential mode），输出是 l/8 字节，p指16字节的个性化字符串，x是任意字节哈希数据。
  - 用来初始化 $hSigCRH$、$EquihashGen$和KDF^{Sprout}
  - 从OverWinter开始，也被用来计算SIGHASH transaction hashes

    - 【Sapling】中用来初始化 PRF^{expand}、$PRF^{ockSapling}$、$KDF^{Sapling}$，以及在RedJubjub签名方案中初始化 $SpendAuthSig^{Sapling}$和$BindingSig^{Sapling}$
- $BLAKE2s-l(p,x)$ 工作在顺序模式（sequential mode），输出是 l/8 字节，p指8字节的个性化字符串，x是任意字节哈希数据。
    - 用来初始化 $PRF^{nfSapling}$、CRH^{ivk}和$GroupHash^{J^{(r)*}}$

5.4.1.3 Merkle树哈希函数

- $MerkleCRH^{Sprout}(layer,left*,right*) := SHA256Compress(left*||right*)$
    - layer对结果没影响
    - left*和right*都是256bit，输出也是256bit
    - SHA256Compress必须是抗碰撞的，且不能有preimage x结果是 $[0]^{256}$
- $MerkleCRH^{Sapling}(layer,left*,right*) = PedersenHash(”Zcash_PH”,l*||left*||right*)$
    - $l* = I2LEBSP_6(MerkleDepth^{Sapling}-1-layer)$
        - l*输出6bit长的二进制数
    - left*、right*和输出都是255bit
- $MerkleCRH^{Orchard}(layer,left,right) = hash$
    - $hash = SinsemillaHash(”z.cash:Orchard-MerkleCRH”,l*||left*||right*)$
        - $l* = I2LEBSP_{10}(MerkleDepth^{Orchard}-1-layer)$
        - $left* = I2LEBSP_{255}(left)$
        - $right* = I2LEBSP_{255}(right)$
        - left*、right*和hash都是{0..q_P - 1}的元素
    - 如果hash是无效值，MerkleCRH = 0
    - SinsemillaHash必须是抗碰撞的，且不能有长度是 10 + 2 * 255 bits的输入使得hash结果是无效值

5.4.1.4 h_{Sig} 哈希函数

- hSigCRH被用来计算JoinSplit description中的 h_{Sig} :

$hSigCRH(randomSeed,nf^{old}_{1..N^{old}},joinSplitPubKey) = BLAKE2b-256(”ZcashComputehSig”,hSigInput)$

    - $hSigInput = 256-bit\ randomSeed\ ||\ 256-bit\ nf^{old}_1\ ||\ ...\ ||\ 256-bit\ nf^{old}_{N^{old}}\ ||\ 256-bit\ joinSplitPubKey$

5.4.1.5 CRH^{ivk} 哈希函数

- CRH^{ivk} 被用来生成incoming viewing key，为了后续生成Sapling的隐私支付地址。

- $CRH^{ivk}(ak*, nk*) = LEOS2IP_{256}(BLAKE2s-256(”Zcashivk”, crhInput)) \ mod \ 2^{251}$
- $crhInput = LEBS2OSP_{256}(ak*) \ || \ LEBS2OSP_{256}(nk*)$
- $CRH^{ivk}(ak*, nk*)$ 必须是对64字节的输入x来说是抗碰撞的

5.4.1.6 $DiversifyHash^{Sapling}$ and $DiversifyHash^{Orchard}$ 哈希函数

- $DiversifyHash^{Sapling}$ 是用来生成diversified base的：
  - $DiversifyHash^{Sapling}(d) = GroupHash_U^{J^{(r)*}}(”Zcash_gd”, LEBS2OSP_{88}(d))$
- $DiversifyHash^{Orchard}$ 是用来生成diversified base
  - $P = GroupHash^P(”z.cash:Orchard-gd”, LEBS2OSP_{l_d}(d))$
  - 如果 $P = O_P$ 单位元
    - $DiversifyHash^{Orchard}(d) = GroupHash^P(”z.cash:Orchard-gd”, ””)$
  - 否则
    - $DiversifyHash^{Orchard}(d) = P$
  - $GroupHash^{J^r*}$ 类似于一个预言机，输入是diversifiers，输出是阶是 r_J 的Jubjub曲线上的点

5.4.1.7 Pedersen 哈希函数

Pedersen哈希函数是利用了Jubjub椭圆曲线上的离散对数问题，在zkSNARK电路中的效率比较高，在Zcash中用于Pedersen承诺和Sapling的Merkle树的增长。

- $PedeersenHash(D, M) = Extract_{J^{(r)}}(PedersenHashToPoint(D, M))$
  - D: domain，8字节的字符串， M： 任意字节的hash的输入数据，输出是255 bit

$PedersenHashToPoint(D, M) \rightarrow J^{(r)}$ 定义如下：

- 将M补0pad到长度是3的倍数得到M'
- 计算 $n = ceiling(len(M')/(3 \cdot c))$
- 将 M' 分割成 n 个块，每个块除了最后一个长度都是3*c bit
- 返回 $\sum_{i=1}^{n}[< M_i >]I(D, i) \rightarrow J^{(r)}$

其中:

- $I(D, i) = FindGroupHash^{J^{(r)*}}(D, i-1)$
  - D 是8字节，i-1 是32 bit的整数
  - 输出是 $J^{(r)*}$ 中的元素（一个点）
- $[< M_i >]$ 定义如下：
  - 令 $k_i = len(M_i) \ / \ 3$

- 将 M_i 分割成3 bit的块： $M_i \ = \ concat(m_1..m_{k_i})$
- 令 $m_j \ = \ [s_0^j, s_1^j, s_2^j]$
- 令 $enc(m_j) \ = \ (1 - 2 \cdot s_2^j) \cdot \ (1 + s_0^j + 2 \cdot \ s_1^j)$ ，结果是一个整数
- 令 $< M_i > \ = \ \sum_{j=1}^{k_i} enc(m_j) \cdot \ 2^{4 \cdot (j-1)}$ ，结果是 $\{-\frac{r_J - 1}{2}..\frac{r_J - 1}{2}\} \setminus \{0\}$ ，是一个分数

- enc和 $[< M_i >]$ 都是单射，证明见论文

5.4.1.8 混合Pedersen 哈希

混合Pedersen哈希（Mixing Pedersen Hash）被用来从 cm 和 pos 中计算 ρ ，它接受的参数是一个Pedersen承诺（ J 上的点）P 和一个输入x，输出也是群 J 上的一个点。

$$MixingPedersenHash(P, x) \ = \ P \ + \ [x]FindGroupHash^{J^{(r)*}}(”Zcash\_J\_”, ””)$$

5.4.1.9 Sinsemilla 哈希函数

Sinsemilla哈希也是基于离散对数问题，定义在Pallas曲线上，提出这个新的哈希是为了让lookups这个技术在Halo2等证明系统实际使用，Sinsemilla Hash在Zcash中用于Sinsemilla 承诺和Orchard的Merkle树的增长。

- $Q(D) \ = \ GroupHash^P(”z.cash : SinsemillaQ”, D)$
  - 输入是任意字节的二进制，输出是 P^* 的一个元素
- $S(j) \ = \ GroupHash^P(”z.cash : SinsemillaS”, I2LEOSP_{32}(j))$
  - 输出是 P^* 的一个元素
- 令k = 10，c是一个最大的整数，且满足 $2^c \ <= \ \frac{r_P - 1}{2}$ ，如c=253
- 在Pallas曲线上定义一个加法，见文档
- $SinsemillaHash(D, M) \ = \ Extract_P(SinsemillaHashToPoint(D, M))$
  - D 是 N字节的二进制数，M是{0..k.c} bit的二进制数
  - 输出是{0..q_p-1}或无效值
- $SinsemillaHashToPoint(D, M)$ 定义如下：
  - 令 $n : \ \{0..c\} \ = \ ceiling(\frac{len(M)}{k})$
  - 令 $m \ = \ pad_n(M)$
  - 令 $Acc \ \leftarrow \ Q(D)$
  - For i form 1 to n:
    - $Acc \ \leftarrow \ (Acc + S(m_i)) \ + \ Acc$
      - 这个加法不是普通的加法，是上面定义的在Pallas曲线上的操作

    - 返回Acc
- Sinsemilla hash也是单射的

其他相关理论见原文。

5.4.1.10 PoseidonHash Function

Poseidon是一个密码学的置换算法，能够在一组有限域的元素上操作，full rounds 是8，partial rounds是56，sponge的容量是一个域元素，rate是2个域元素，输出是一个域元素，如果输入是2个元素，初始的capacity element是 2^{65} 。

PoseidonHash的定义如下：

- 令 $f: F_{q_P} \to F_{q_P}$ 是Poseidon置换，则
- $PoseidonHash(x, y) = f([x, y, 2^{65}])_1$
    - 使用1-based indexing（从1开始）
    - $PoseidonHash: F_{q_P} \times F_{q_P} \to F_{q_P}$

5.4.1.11 Equihash Generator

$EquihashGen_{n,k}$ 是一个特殊的hash函数，能把一个输入和一个下标映射到一个n bits的输出。

- 令 $powtag = 64 - bit"ZcashPow" \,||\, 32 - bit\ n \,||\, 32 - bit$
- 令 $powcount(g) = 32 - bit\ g$
- 令 $EquihashGen_{n,k}(S, i) = T_{h+1..h+n}$
    - $m = floor(\frac{512}{n})$
    - $h = (i - 1\ mod\ m) \cdot n$
    - $T = BLAKE2b - (n \cdot m)(powtag, S||powcount(floor(\frac{i-1}{m})))$
        - T中的下标是1开始的

5.4.2 伪随机函数

- PRF^{addr}、$PRF^{nfSprout}$、PRF^{pk}、PRF^{ρ} 等伪随机函数使用SHA256Compress完成对输入的计算，见文档。
    - SHA256Compress的前4 bit用来区分函数的用途，上面四个PRF里的SHA256Compress的前4 bit各不相同。
- PRF^{expand} 被用来生成Spend authorizaing key ask 和 proof authorizaing key nsk，且使用BLAKE2b哈希
    - $PRF^{expand}_{sk}(t) = BLAKE2b - 512("Zcash\_ExpandSeed", LEBS2OSP_{256}(sk)||t)$

- $PRF^{ockSapling}$ 被用来生成outgoing cipher key ock，用来加密 outgoing ciphertext，使用BLAKE2b
  - $PRF_{ovk}^{ockSapling}(cv, cmu, ephemeralKey) = BLAKE2b-256(\text{"}Zcash\_Derive\_ock\text{"}, ockInput)$
  - $ockInput = LEBS2OSP_{256}(ovk)||32-byte\ cv\ ||\ 32-byte\ cmu\ ||\ 32-byte\ ephemeralKey$
- $PRF^{nfSapling}$ 被用来给一个【Sapling】note 生成nullifier，它使用BLAKE2s
  - $PRF_{nk*}^{nfSapling}(\rho*) = BLAKE2s-256(\text{"}Zcash\_nf\text{"}, LEBS2OSP_{256}(nk*)||LEBS2OSP_{256}(\rho*))$
- $PRF^{ockOrchard}$ 被用来生成【Orchard】的outgoing cipher key ock，来加密outgoing ciphertext，它使用BLAKE2b
  - $PRF_{ovk}(cv, cmx, ephemeralKey) = BLAKE2b-256(\text{"}Zcash\_Orchardock\text{"}, ockInput)$
  - $ockInput = LEBS2OSP_{256}(ovk)\ ||\ 32-byte\ cv\ ||\ 32-byte\ cmx\ ||\ 32-byte\ ephemeralKey$
- $PRF^{nfOrchard}: F_{q_P} \times F_{q_P} \to F_{q_P}$ 被用来生成【Orchard】note的nullifier，它使用PoseidonHash
  - $PRF_{nk}(\rho) = PoseidonHash(nk,\ \rho)$

5.4.3 对称加密

令 $Sym.K$ 是256 bit的二进制数集合， $Sym.P$和$Sym.C$ 都是N个字节的二进制数集合

令一次性授权对称加密方案 $Sym.Encrypt_K(P)$ 是使用AEAD_CHACHA20_POLY1305加密来对明文 $P \in Sym.P$ 进行授权加密，其中256 bit的 $K \in Sym.K$，nonce是 $[0]^{96}$， “associated data”是空的。

同样的令 $Sym.Decrypt_K(C)$ 是用AEAD_CHACHA20_POLY1305解密对密文 $C \in Sym.C$ 进行解密，其中nonce、K和associated data跟加密一样，结果要么是明文要么是无效值。

5.4.4 伪随机置换

$PRP^d: B^{Y^{[l_{dk}/8]}} \times B^{[l_d]} \to B^{[l_d]}$ 被称为伪随机置换，在Zcash中用来生成【Orchard】的隐私支付地址的多样化参数。

- $PRP_K^d(d) = FF1-AES256_K(\text{""}, d)$
  - 输入d是88 bit，输出也是88 bit
- $FF1-AES256_K(tweak, x)$ 是FF1保留格式的AES算法，使用256 bit 的密钥K，其他参数radix = 2，minlen = 88，maxlen = 88，tweak是空字符串，x是88 bit

5.4.5 密钥协议和推导

5.4.5.1 Sprout 密钥协商

KA^{Sprout} 使用Curve25519密钥协议：

- $KA.Public$和$KA.SharedSecret$ 的类型是Curve25519的公钥类型
- $KA.Private$ 的类型是Curve25519的私钥类型（secret keys）
- 令 $Curve25519(\underline{n}, \underline{q})$ 是两个点相乘的结果，这2个点一个是Curve25519的公钥，表示为 $\underline{q}$ ，另一个为Curve25519的私钥，表示为 $\underline{n}$ 。
- 令 $KA.Base := \underline{9}$ 是表示Curve25519 base point的公共字节序列（public byte sequence）
- 令 $clamp_{Curve25519}(\underline{x})$ ，接受一个32字节的参数x，输出一个字节序列，表示Curve25519的私钥。clamp表示对首尾字节的一些bit位进行一些操作，见论文
- 令 $KA.FormatPrivate(x) = clamp_{Curve25519}(x)$
- 令 $KA.DerivePublic(n, q) = Curve25519(n, q)$
- 令 $KA.Agree(n, q) = Curve25519(n, q)$

5.4.5.2 Sprout 密钥推导

- KDF^{Sprout} 表示Key Derivation Function，使用BLAKE2b-256:
  - $KDF^{Sprout}(i, h_{Sig}, sharedSecret_i, epk, pk^{new}_{enc,i}) = BLAKE2b-256(kdftag, kdfinput)$
  - $kdftag = 64-bit\ "ZcashKDF" \ || \ 8-bit\ i-1 \ || \ [0]^{56}$
  - $kdfinput = 256-bit\ h_{Sig} \ || \ 256-bit\ sharedSecret_i \ || \ 256-bit\ epk \ || \ 256-bit\ pk^{new}_{enc,i}$

5.4.5.3 Sapling 密钥协议

Sapling中的KA使用Diffie-Hellman和Jubjub上的乘法来实现:

- 令 J 、 $J^{(r)}$ 、 $J^{(r)*}$ 分别表示定义在Jubjub上的群及其子群
- 定义 $KA.Public = J$
- 定义 $KA.PublicPrimeSubgroup = J^{(r)}$
- 定义 $KA.SharedSecret = J^{(r)}$
- 定义 $KA.Private = F_{r_J}$
- 定义 $KA.DerivePublic(sk, B) = [sk]B$
- 定义 $KA.Agree(sk, P) = [h_J \cdot sk]B$

5.4.5.4 Sapling 密钥推导

Sapling中的KDF使用BLAKE2b-256初始化:

- $KDF(sharedSecret, ephemeralKey) := BLAKE2b-256(\text{“}Zcash\_SaplingKDF\text{”}, kdfinput)$
- $kdfinput = LEBS2OSP_{256}(repr_J(sharedSecret))||ephemeralKey$

5.4.5.5 Orchard 密钥协议

Orchard中的KA使用在Pallas曲线上的Diffie-Hellma来实现：

- 令 P 表示定义在Pallas上的群
- 定义 $KA.Public = P^*$
- 定义 $KA.SharedSecret = P^*$
- 定义 $KA.Private = F^*_{r_P}$
- 定义 $KA.DerivePublic(sk, B) = [sk]B$
- 定义 $KA.Agree(sk, P) = [sk]B$

5.4.5.6 Orchard Key Derivation

Orchard中的KDF使用BLAKE2b-256初始化:

- $KDF(sharedSecret, ephemeralKey) := BLAKE2b-256(\text{“}Zcash\_OrchardKDF\text{”}, kdfinput)$
- $kdfinput = LEBS2OSP_{256}(repr_P(sharedSecret))||ephemeralKey$

5.4.6 Ed25519

> 为方便理解，本节补充一些密码学基础知识

Ed25519是一种签名方案，用来初始化JoinSplitSig。

基础：

- 数字签名：私钥加密，公钥解密
- 数字加密：公钥加密，私钥解密
- RSA：基于大整数的因式分解问题很难，能用于加密和签名
- DSA：基于有限域上的离散对数问题很难，只能用于签名，不能用于加密(除了一些扩展)
- RSA和DSA安全性差不多
- ECC: 椭圆曲线密码，使用椭圆曲线模拟DSA
  - 椭圆曲线：$y^2 = x^3 + ax + b \bmod p \cup \{0\}$，且 $4a^3 + 27b^2 \neq 0$（不包含奇点）的点集
- ECDSA: 椭圆曲线签名算法
- 群：一个集合 + 集合上的二元运算 +
  - 封闭性

    - 结合律
    - 有单位元
    - 有逆元
    - 阿贝尔群：交换律
- 椭圆曲线上的群
    - 集合：曲线上的点
    - 单位元：无穷远点0，P+0=P，P+（-P）= 0
    - 逆元：关于x轴对称
    - 二元运算：P 、Q连线取交点的x轴对称点
    - P+Q = Q+P，符合交换律，所以椭圆曲线上的群是一个阿贝尔群
    - 离散对数问题：知道Q和P，很难找到n使得 Q = nP，反过来知道n和P计算Q = nP比较简单
- EdDSA：比ECDSA更新的数字签名算法，基于爱德华曲线，Schnorr签名变种
    - 采用和ECDSA不同的椭圆曲线和签名算法，可以分为：
    - Ed25519： 基于curve25519曲线
        - 比448速度更快，key更小，应用更广
        - 采用哈希函数代替随机数发生器，对同一消息签名是一样的
    - Ed448：基于curve448曲线

Ed25519

- 蒙哥马利曲线：$By^2 = x^3 + Ax^2 + x$
    - Curve25519曲线: $y^2 = x^3 + 486662x^2 + x \ mod \ p$,
        - p = 2^255 - 19(素数域的阶)
        - 基点 B ： $x = 9$ ，生成一个群
- 爱德华兹曲线: $x^2 + y^2 = 1 + dx^2y^2$,
- 扭曲爱德华兹曲线：$ax^2 + y^2 = 1 + dx^2y^2$， $a, d \neq 0, \ a \neq d$
    - Ed25519曲线： $-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$
        - p = 2^255 - 19(素数域的阶)
        - 基点 B ： $(9, \frac{4}{5})$ ，生成一个群
- 使用SHA-512作为内部哈希函数，签名64字节，公钥32字节，私钥32字节
- Curve25519、Ed25519、X25519分别对应加密、签名和密钥交换算法
- 私钥k是由一个随机数发生器产生，公钥A是曲线上一个点

- $A = a \cdot B$, $a = H(k)$ 经过计算得到
- 签名(R, S)过程和验签过程见文档

5.4.7 RedDSA, RedJubjub, and RedPallas

RedDSA是一个基于Schnorr的签名方案，参考了EdDSA，支持re-randomization和从私钥生成公钥的单态。

RedJubjub和RedPallas都是RedDSA的特定实现：

- RedJubjub：使用BLAKE2b-512哈希和Jubjub曲线
  - 【Sapling】Spend authorization signature方案SpendAuthSig使用了RedJubjub
  - 【Sapling】binding signature方案BindingSig也使用了不带re-randomization的RedJubjub
- RedPallas：使用BLAKE2b-512哈希和Pallas曲线
  - 【Orchard】Spend authorization signature方案SpendAuthSig使用了RedPallas
  - 【Orchard】binding signature方案BindingSig也使用了不带re-randomization的RedPallas

签名相关的讨论请参考原文。

5.4.7.1 Spend Authorization Signature (Sapling and Orchard)

Spend authorization signature scheme $SpendAuthSig^{Sapling}$和$SpendAuthSig^{Orchard}$ 分别使用了带re-randomization的RedJunjub和RedPallas，以及生成器generator
$P_G = FindGroupHash^{\mathcal{J}^{(r)*}}(”Zcash\_G\_”, ””)$ 和
$P_G = GroupHash^{P}(”z.cash:Orchard”, ”G”)$。

5.4.7.2 Binding Signature (Sapling and Orchard)

【Sapling】binding signature scheme $BindingSig^{Sapling}$ 使用了不带re-randomization的RedJubjub，以及generator P_G。

【Orchard】binding signature scheme $BindingSig^{Orchard}$ 使用了不带re-randomization的RedPallas，以及generator P_G。

5.4.8 承诺方案

5.4.8.1 Sprout Note 承诺

使用 rcm、a_{pk}、v和ρ 作为SHA-256的入参，生成 $NoteCommit^{Sprout}_{rcm}$

$NoteCommit^{Sprout}_{rcm}(a_{pk}, v, \rho) = SHA-256(b'10110000'||256-bit a_{pk}||64-bit v||256-bit \rho||256-bit rcm)$

- b'10110000' = 0xB0

5.4.8.2 Windowed Pedersen 承诺

$$WindowedPedersenCommit_r(s) = PedersenHashToPoint(”Zcash_PH”, s) + [r]FindGroupHash^{J^{(r)*}}(”Zcash\_PH”, ”r”)$$

Windowed Pedersen 承诺复用了Pedersen hash，且在Jubjub曲线上增加了一个随机点来实现。

$$NoteCommit^{Scapling}_{rcm}(g*_d, pk*_d, v) = WindowedPedersenCommit_{rcm}([1]^6||I2LEBSP_{64}(v)||g*_d||pk*_d)$$

- 前缀[1]^6是为了区分PedersenHashToPoint函数在MerkleCRH中的使用，里边前缀是{0..MerkleDepth-1}<64
- $Uncommitted^{Sapling}$不在$NoteCommit^{Sapling}$的范围里

5.4.8.3 同态 Pedersen 承诺 (Sapling and Orchard)

WindowedPedersenCommit不支持同态属性，没法在ValueCommit中使用。
HomomorphicPedersenCommit支持同态：

- $HomomorphicPedersenCommit^{Sapling}_{rcv}(D, v) = [v]FindGroupHash^{J^{(r)*}}(D, ”v”) + [rcv]FindGroupHash^{J^{(r)*}}(D, ”r”)$
- $HomomorphicPedersenCommit^{Orchard}_{rcv}(D, v) = [v]FindGroupHash^{P}(D, ”v”) + [rcv]GroupHash^{P}(D, ”r”)$
- $ValueCommit^{Sapling}_{rcv}(v) = HomeomorphicPedersenCommit^{Sapling}_{rcv}(”Zcash\_cv”, v)$
- $ValueCommit^{Orchard}_{rcv}(v) = HomeomorphicPedersenCommit^{Orcahrd}_{rcv}(”z.cash : Orchard - cv”, v)$

5.4.8.4 Sinsemilla 承诺

Sinsemilla承诺通过复用Sinsemilla hash，且在Pallas曲线上增加了一个随机点来实现。

- $SinsemillaCommit_r(D, M) = M' + [r]GroupHash^{P}(D||” - r”, ””)$
  - $M' = SinsemillaHashToPoint(D||” - M”, M)$
- $SinsemillaShortCommit_r(D, M) = Extract_P(SinsemillaCommit_r(D, M))$
- $NoteCommit^{Orchard}_{rcm}(g*_d, pk*_d, v, \rho, \psi) = SinsemillaCommit_{rcm}(”z.cash : Orchard - NoteCommit”, g*_d||pk*_d||I2LEBSP_{64}(v)||I2LEBSP_{l_{base}}(\rho)||I2LEBSP_{l_{base}}(\psi))$
- $Commit^{ivk}_{rivk}(ak, nk) = SinsemillaShortCommit_{rivk}(”z.cash : Orchard - CommitIvk”, I2LEBSP_{b_{base}}(ak)||I2LEBSP_{b_{base}}(nk))$

5.4.9 代表群和Pairings

5.4.9.1 BN-254

BN-254曲线的具体参数定义请参考原文。

5.4.9.2 BLS12-381

BLS12-381曲线的具体参数定义请参考原文。

5.4.9.3 Jubjub

Jubjub曲线的具体参数定义请参考原文。

- $O_J = (0, 1)$ 是曲线上点加法的单位元
- $repr_J : J \to B^{[l_J]}$
- $abst_j : B^{[l_J]} \to J \cup \{\perp\}$
- $J_*^{(r)} = \{repr_J(P) : B^{[l_J]} \mid p \in J^{(r)}\}$

5.4.9.4 Jubjub曲线上的坐标提取

从曲线点中提取x坐标编码成二进制数

- 令 $U((u, v)) = u$, $V((u, v)) = v$
- 令 $Extract_{J^{(r)}} = I2LEBSP_{l_{Merkle}}(U(P))$
- 两个点(0,1)和（0，-1）能组成阶是2的群
- $J^{(r)}$ 的阶是奇素数
- 令 $P = (u, v) \in J^{(r)}$，则 $(u, -v) \notin J^{(r)}$
  - 证明见文档
- U 在 $J^{(r)}$ 上是单射，即 $J^{(r)}$ 上的点的x坐标不重复

5.4.9.5 群哈希到Jubjub曲线

- $GroupHash^{J^{(r)*}}.Input = B^{Y^{[8]}} \times B^{Y^{[N]}}$
  - 8个字节的参数D是个性化字符串，用于区分不同用途
  - N字节的参数M是需要哈希的数据
  - 哈希结果是 $J^{(r)*}$ 上的一个点
- $GroupHash^{J^{(r)*}}.URSType = B^{Y^{[64]}}$
- 具体过程如下：
  - 令 $\underline{H} = BLAKE2s - 256(D, URS||M)$
    - BLAKE2s-256模拟一个随机数预言机
  - 令 $P = abst_J(LEOS2BSP_{256}(\underline{H}))$
    - 如果P无效则直接返回无效值
  - 令 $Q = [h_J]P$
    - 如果 $Q = Q_J$，直接返回无效值
  - 返回Q

5.4.9.6 Pallas and Vesta

Orchard中使用Pallas和Vesta两种椭圆曲线，且任意一个的base field都是另一个的scalar field，它们的代表群分别是P和V。

Pallas用在电路中，Vesta用在证明系统中。

有限域上的椭圆曲线：

- 椭圆曲线 $y^2 = x^3 + ax + b,\ x, y, a, b \in F_q$ 定义在有限域 F_q 上，q是质数
  - 所有运算在模q下进行
  - $4a^3 + 27b^2 \neq 0$: 没有奇点
- Pallas定义在 F_{q_P} 上，阶是 r_P
- Vesta定义在 F_{q_V} 上，阶是 r_V
- 则 $r_p = q_V$ ， $r_V = q_P$

5.4.9.7 Pallas曲线上的坐标提取

- $P = (x, y), Extract_P(P) = X(P)\ mod\ q_p = x$
  - X(P) = P的x坐标，如果P是无穷远点则是0

详细讨论请参考原文。

5.4.9.8 群哈希 Pallas and Vesta曲线

Orchard中使用"Simplified SWU"算法把扮演随机数预言机的哈希转换到椭圆曲线的点上。

详细讨论请参考原文。

5.4.10 零知识证明系统

5.4.10.1 BCTV14

【Sapling】激活之前Zcash使用BCTV14作为zk-SNARKs的证明系统，由于它有一些漏洞会导致Zcash余额不平衡，已经弃用。

5.4.10.2 Groth16

【Sapling】之后Zcash使用Grouth16作为证明系统，在版本4的交易里生成证明（【Sprout】的JoinSplit description和【Sapling】的Spend/Output description）。证明的编码规则见原文。

5.4.10.3 Halo 2

在【Orchard】的版本5的交易里的Action description使用Halo2作为zk-SNARKs的证明系统。Halo2的证明是字节序列，不需要编码。

5.5 Note明文和Memo字段的编码

Zcash中传输的notes被加密之后存储在区块链中。

【Sprout】JoinSplit description中的note明文被加密到各自的transmission keys $pk_{enc,1..N^{new}}^{new}$ 。

- 【Sprout】note 明文 np 包括：
  - $leadByte$： B^Y ，8 bit，0x00
  - v：$\{0..2^{l_{value}}-1\}$，64 bit
  - ρ： $B^{[l_{PRF}]}$ ，256 bit
  - rcm： $NoteCommit.Output$ ，256 bit
  - $memo$： $B^{Y^{512}}$ ，512字节

【Sampling onward】Output/Action description中的note明文被加密到diversified transmission key pk_d 。

- 每个note明文包括：
  - $leadByte$： B^Y ，8 bit
    - 【Canopy】之前是0x01，之后是0x02，【Orchard】一直都是0x02
  - d： $B^{[l_d]}$ ，88 bit
  - v：$\{0..2^{l_{value}}-1\}$，64 bit
  - $rseed$： $B^{Y^{[32]}}$ ，256 bit
  - $memo$： $B^{Y^{512}}$ ，512字节

5.6 地址和密钥的编码

Address和keys都被编码成字节序列，被称为raw encoding。

【Sprout】里的shielded payment address可以被进一步编码为Base58Check。

【Sapling】中的地址和key使用Bench32代替Base58Check。

【Orchard】使用了新的地址格式叫unified payment address，可以编码Orchard address、Saplng address、transparent address等等。Unified payment address和【Orchard】spending keys被编码为Bech32m，而不是Bench32。

5.6.1 公开编码

5.6.1.1 公开地址

公开地址（transparent address）要么是P2SH（Pay to Script Hash）要么是P2PKH（Pay to Public Key Hash）。

- P2SH的raw encoding格式是：
  - $(8-bit\ 0x1C,\ 8-bit 0xBD, 160-bit\ script\ hash)$
    - 主网是0x1C 0xBD，测试网是0x1C 0xBA

    - 20字节表示一个script hash
- P2PKH的raw encoding格式是：
    - $(8-bit\ 0x1C,\ 8-bit 0xB8, 160-bit\ validating\ key\ hash)$
        - 主网是0x1C 0xB8，测试网是0x1D 0x25
        - 20字节表示一个validating key hash，把一个ECDSA key编码压缩后计算SHA-256哈希，再把哈希结果计算RIPEMD-160 哈希
- Zcash还不支持HD钱包地址

5.6.1.2 Transparent Private Keys

和Bitcoin一样。

5.6.2 Sprout 编码

5.6.2.1 Sprout 支付地址

【Sprout】隐私支付地址包含 a_{pk}和pk_{enc} ， a_{pk} 是一个SHA256Compress的输出二进制数， pk_{enc} 是KA.Public 公钥，这两个都是从spending key派生出来。shielded payment address的raw encoding是：

$(8-bit\ 0x16,\ 8-bit 0x9A, 256-bit\ a_{pk}, 256-bit\ pk_{enc})$

5.6.2.2 Sprout Incoming Viewing Keys

【Sprout】incoming viewing key包含 a_{pk}和sk_{enc} ： $KA.Private$ ， a_{pk} 是一个SHA256Compress的输出二进制数，这2个也从spending key派生出来。

【Sprout】Incoming viewing key的raw encoding 为：

$(8-bit\ 0xA8, 8-bit\ 0xAB, 8-bit\ 0xD3, 256-bit\ a_{pk}, 256-bit\ sk_{enc})$

5.6.2.3 Sprout Spending Keys

【Sprout】spending key a_{sk} 是一个252 bit的二进制数，raw encoding 是：

$(8-bit\ 0xAB, 8-bit\ 0x36, [0]^4, 252-bit\ a_{sk})$

5.6.3 Sapling 编码

5.6.3.1 Sapling 支付地址

【Sapling】隐私支付地址包含 d和pk_d ， pk_d 是 $KA^{Sapling}$ 公钥的编码，raw encoding 是：

- $(LEBS2OSP_{88}(d), LEBS2OSP_{256}(repr_J(pk_d)))$
    - pk_d 是 $J^{(r)}$ 中的元素

5.6.3.2 Sapling Incoming Viewing Keys

【Sapling】incoming viewing key只包含一个 ivk：$\{0..2^{l^{ivk}}-1\}$，ivk是KA.Private key，raw encoding 是：

$(256-bit\ ivk)$，左边高位补0

5.6.3.3 Sapling Full Viewing Keys

【Sapling】的full viewing key包含 ak：$J^{(r)*}$，nk：$J^{(r)}$和ovk：$B^{[l_{ovk}/8]}$，ak和nk都是Jubjub曲线上的点，raw encoding是：
$(LEBS2OSP_{256}(repr_J(ak)), LEBS2OSP_{256}(repr_J(nk)), 32-byte\ ovk)$

5.6.3.4 Sapling Spending Keys

【Sapling】spending key只包含 sk：$B^{[l_{sk}]}$，raw encoding: $(LEBS2OSP_{256}(sk))$

5.6.4 Unified and Orchard 编码

5.6.4.1 Unified Payment Addresses and Viewing Keys

Unified payment address格式能够对不同类型的payment address进行编码，用户可以选择任意一种地址格式而不一定是【Orchard】payment address。Unified incoming viewing keys和unified full viewing keys分别用来编码多种类型的incoming viewing keys和full viewing keys。

一个unified payment address包含一个或0个各种类型的地址（目前是3种），优先级列表是：

- Typecode 0x03 - 【Orchard】 Raw Payment Address
- Typecode 0x02 - 【Sapling】 Payment Address
- Typecode 0x01 - Transparent P2SH Address
- typecode 0x00 - transparent P2PKH address
  - Typecode 0x00和0x01 二者只能有一个
- 一个unified payment address列表里至少有一个shielded payment address（0x02、0x03）

用户使用unified payment address 支付的时候，必须使用上面优先级列表里的一个地址（most preferred），例如第一个地址（也可以是其他地址）。

5.6.4.2 Orchard Raw Payment Addresses

【Orchard】shielded payment address包含 d：$B^{[l_d]}$和pk_d：$KA.Public$，pk_d 是KA public key 的编码，raw encoding：$(LEBS2OSP_{88}(d), LEBS2OSP_{256}(repr_P(pk_d)))$。

- 没有【Orchard】 shielded payment address 编码，要用 unified payment address。

5.6.4.3 Orchard Raw Incoming Viewing Keys

【Orchard】incoming viewing key包含一个diversifier key dk 和 ivk：$\{1..q_P-1\}$，ivk 是KA.Private类型，raw encoding：$(dk, I2LEOSP_{256}(ivk))$

- dk : 32字节

- I2LEOSP_256(Ivk): 32字节
- 使用unified incoming viewing key，不能单独给【Orchard】incoming viewing key编码

5.6.4.4 Orchard Raw Full Viewing Keys

【Orchard】full viewing key包含 $ak:\ \{0..q_P-1\}$， $nk:F_{q_P}$，$rivk:\ F_{r_P}$：

- ak是spend validating key，用Extract_P对Pallas曲线上的一个点计算得来（点的横坐标x）
- nk是nullifier deriving key，是 F_{q_P} 有限域上的元素
- rivk是 $Commit^{ivk}\ randomness$ ，是有限域 F_{r_P} 上的元素
- Raw encoding: $(I2LEOSP_{256}(ak), I2LEOSP_{256}(nk), I2LEOSP_{256}(rivk))$
  - 3个都是32字节
- 使用unified full viewing key，不能单独给【Orchard】full viewing key编码

5.6.4.5 Orchard Spending Keys

【Orchard】spending key只包含 $sk:\ B^{l_{sk}}$ ，raw encoding: $(LEBS2OSP_{256}(sk))$，32字节。

- Spending key用Bech32m编码而不是Bech32

5.7 BCTV14 zk-SNARK参数

参数定义请参考原文。

5.8 Groth16 zk-SNARK参数

参数定义请参考原文。

5.7 Randomness Beacon

参数定义请参考原文。