BLOCKCHAIN

李康

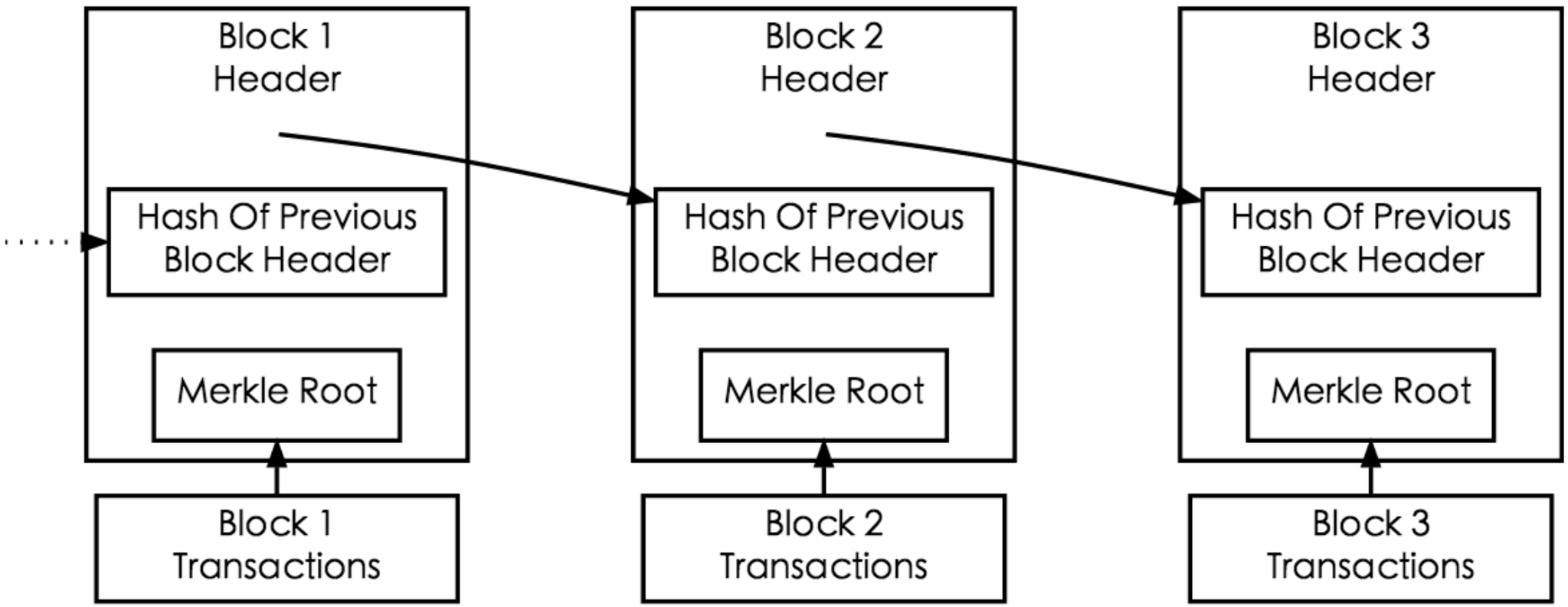
Blockchain

1. Bitcoin
2. Ethereum
3. Bubi
4. Hyperledger fabric
5. Zcash
6. Corda
7. Chain
8. Appendix

Bitcoin

- Blockchain
- UTXO 模型
- Script language based on stack
- POW 共识算法
- Future

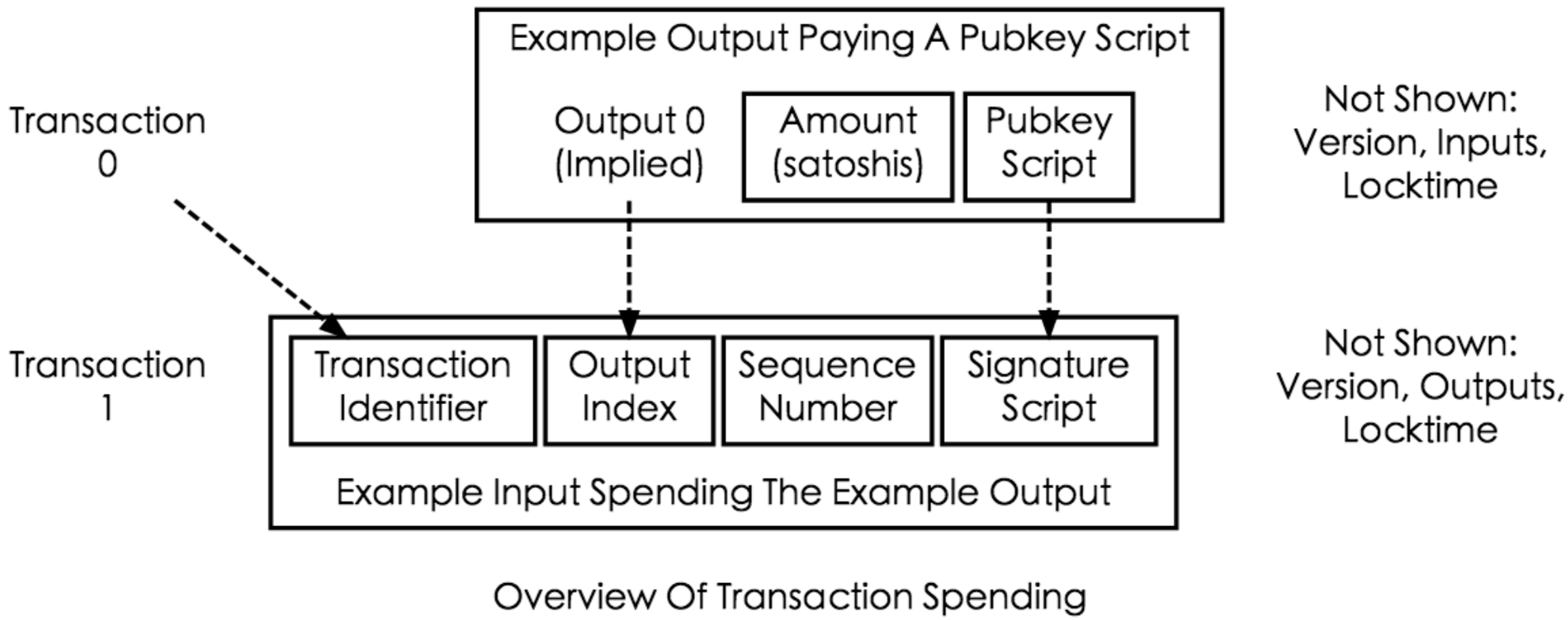
Blockchain



Simplified Bitcoin Block Chain

比特币网络的区块链可能会出现分叉，矿工总是会跟在最长的链后挖矿，这样才可能有钱赚

UTXO 模型

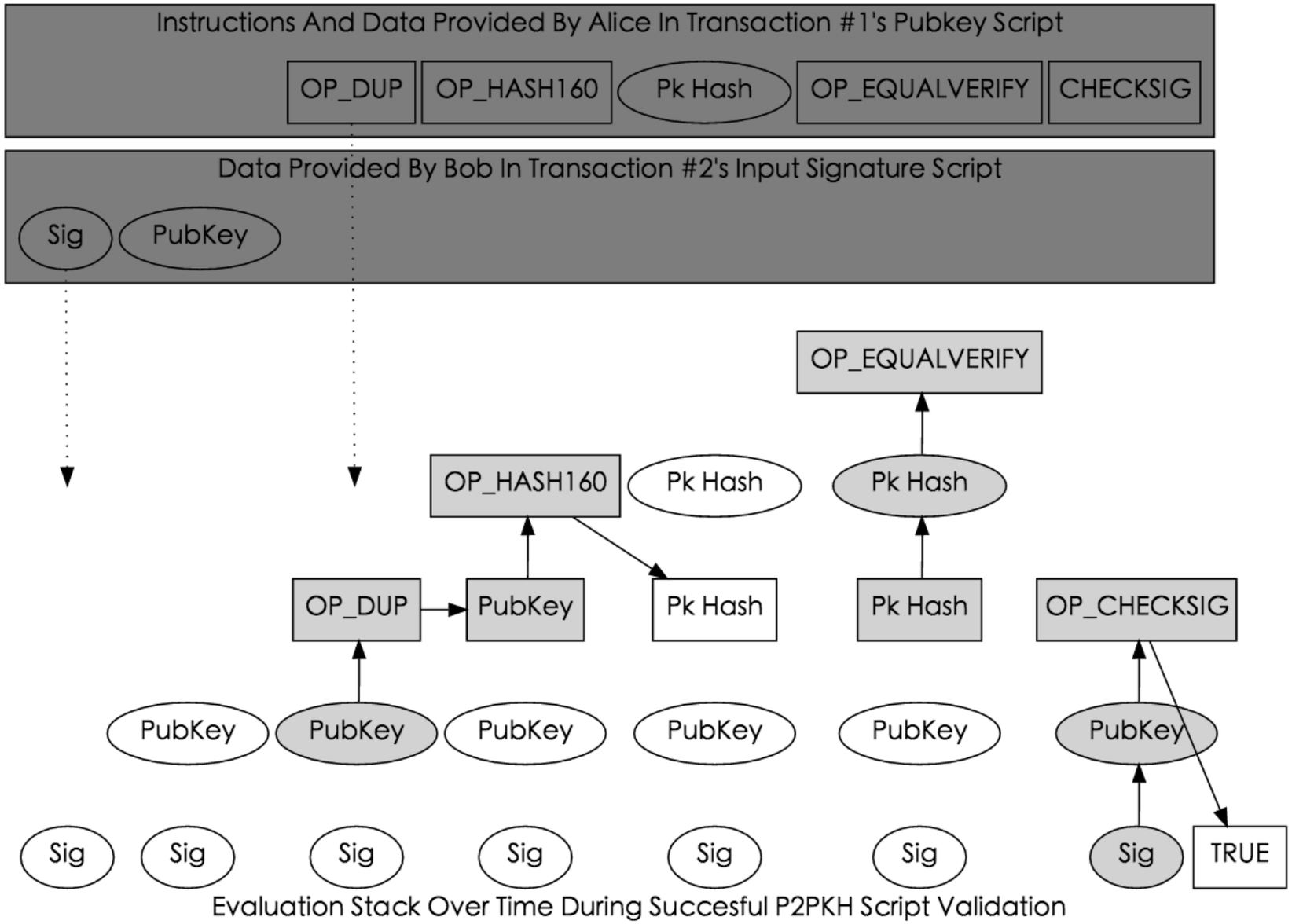


Overview Of Transaction Spending

每一笔交易的输入都会引用先前有效交易的未花费输出，然后创建新的未花费输出。

账户余额就是发往该账户地址的一笔笔未花费输出的累加。

Script language based on stack

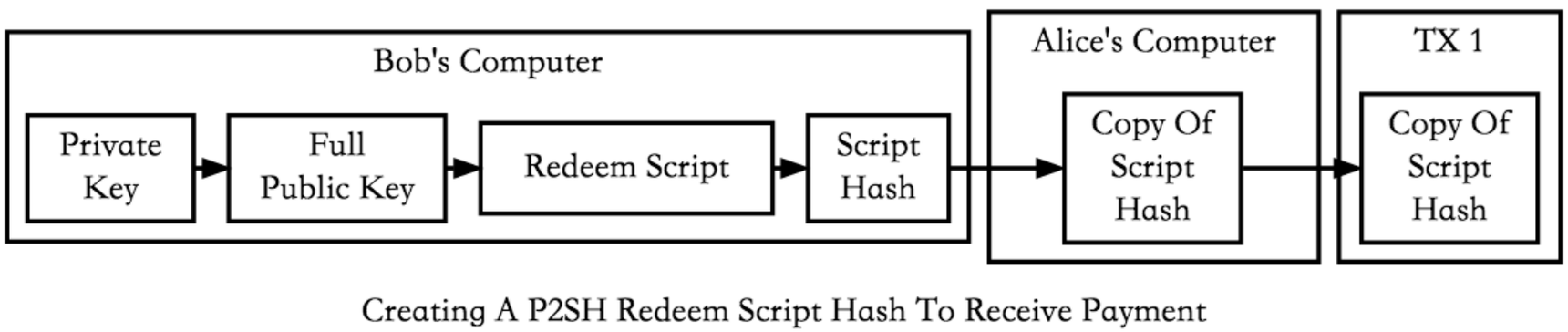


P2PKH Script Validation

P2SH: pay-to-script-hash

- P2SH 交易允许发送者创建一个包含另一脚本哈希值的 pubkey script，其中另一脚本被称为 *redeem script (* 赎回脚本*)*。

- P2SH 基本的工作流如下：



Creating A P2SH Redeem Script Hash To Receive Payment

Bob 按照需要创建一个赎回脚本，对该脚本进行哈希，将得到的哈希值传递给 Alice。Alice 创建一个 P2SH-style 的输出，包含 Bob 的赎回脚本哈希。

当 Bob 花费该笔输出时，他提供 signature script，该解锁脚本中包含：signature、full serialized redeem script。比特币网络会保证解锁脚本中的赎回脚本的哈希值等于先前 Alice 放入输出中的赎回脚本哈希。验证通过后，比特币网络处理该赎回脚本，就像原先的 pubkey script 执行一样，Bob 能够花费该输出，如果赎回脚本没有返回错误。

Multisig

P2SH Multisig 一般用于多方签名交易。在一个 multisig pubkey script 中，称为 m-of-n，其中 n 是提供的公钥数量，m 是匹配公钥的签名最小数量。m 和 n 是操作码 OP_1 到 OP_16。

由于比特币实现中的一个错误，OP_CHECKMULTISIG 会消费栈中 m + 1 个元素，因此在 signature script 中第一个操作码为 OP_0，注意 OP_0 被消费但是不会被使用。

signature script 必须按照 pubkey script 或者 redeem script 中的公钥顺序来提供签名，这是有操作码 OP_CHECKMULTISIG 决定的。

```
Pubkey script: <m> <A pubkey> [B pubkey] [C pubkey...] <n> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> [B sig] [C sig...]
```

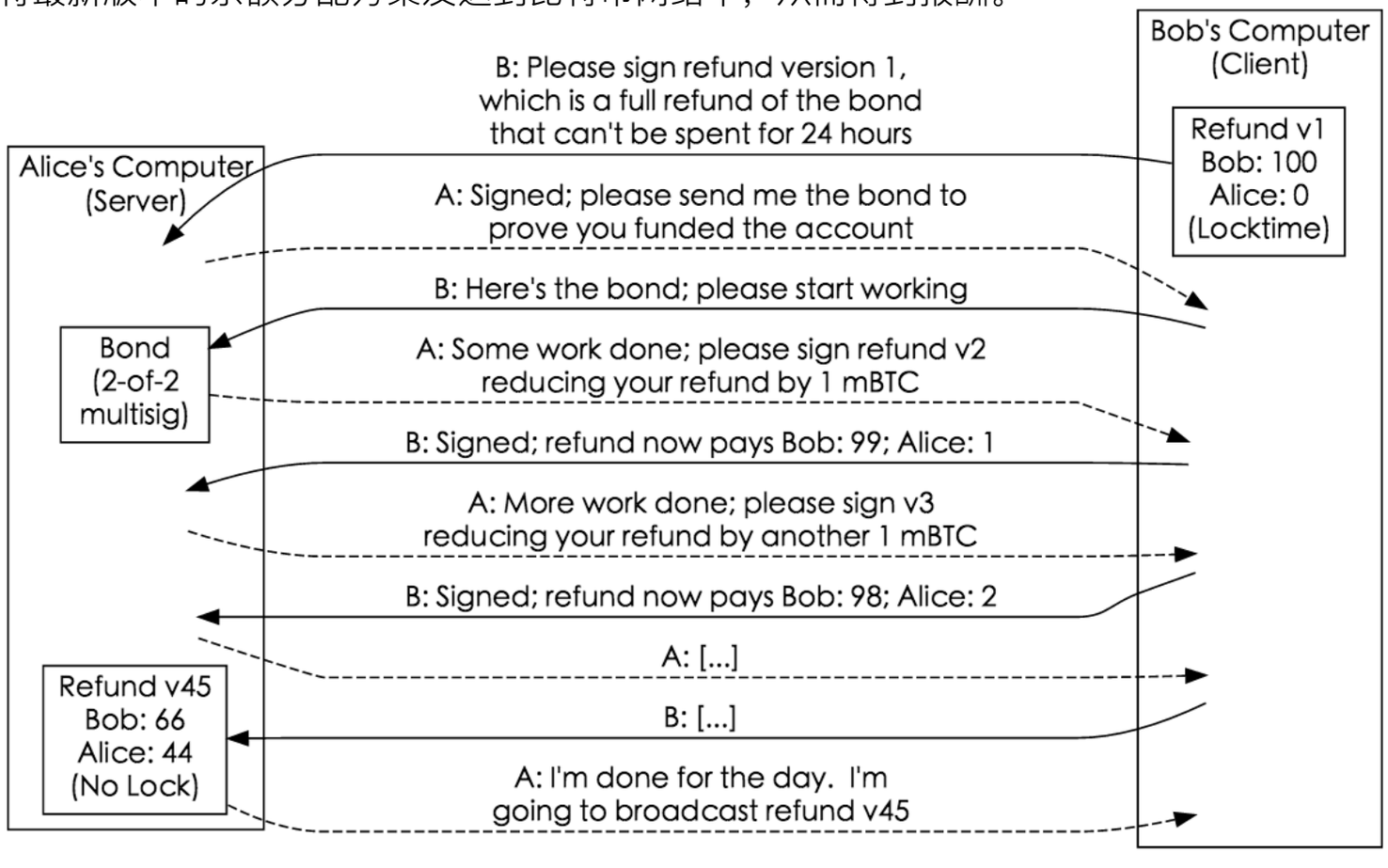
```
Pubkey script: OP_HASH160 <Hash160(redeemScript)> OP_EQUAL
Redeem script: <OP_2> <A pubkey> <B pubkey> <C pubkey> <OP_3> OP_CHECKMULTISIG
Signature script: OP_0 <A sig> <C sig> <redeemScript>
```

Micropayment Channel

Alice 为 Bob 审查稿件，每审查完一个稿件，Alice 希望立即得到报酬，Bob 却不想这样做，原因是大量的小额支付交易会花费他大量的手续费，因此 Alice 建议使用微支付通道。

Bob 索要 Alice 的公钥，创建两笔交易，第一笔交易是将 100m BTC 发送到一个 2-of-2 的多方签名的赎回脚本中，该笔交易被称为 bond tx (纽带交易)。第二笔交易将这 100m BTC 扣除手续费过后如数返回给 Bob，但是这需要 Alice 的签名，且该笔交易需要 24 小时的延迟才能被确认，借助 locktime 字段

Alice 在 24 小时之前，可将最新版本的余额分配方案发送到比特币网络中，从而得到报酬。



Alice broadcasts the bond to the Bitcoin network immediately.
She broadcasts the final version of the refund when she finishes work
or before the locktime. If she fails to broadcast before refund v1's time lock
expires, Bob can broadcast refund v1 to get a full refund.

Bitcoin Micropayment Channels (As Implemented In Bitcoinj)

POW 共识算法

- 矿工监听来自比特币网络中的交易，将交易打包到区块中，此时会导致区块头中 merkle root hash 的改变，因此区块头的哈希也会发生改变。当然矿工也可以不必等待新交易的到来，因为区块头中还包含一个随机值 nonce，通过改变 nonce 值，区块头的哈希值也可以改变。矿工不断地尝试，直到得到的区块头的哈希值小于区块头中的 target threshold nBits。

- 理想地，大概每 10 分钟产出一个区块，由于矿工节点不断地加入与离开，难免会导致全网算力的增加或减少。因此需要进行难度调整。比特币的共识协议，将难度调整的周期定为 每 2016 个块，理想地，2016 个区块的产生需要两周时间，若过去 2016 个区块的时间大于两周，证明算力降低了，需要调低难度值，即增大 nBits；若时间小于两周，反之即可。

Future

- 区块扩容：由于比特币区块的大小有严格限制，导致每个区块容纳的交易笔数不高，使得比特币网络无法承载更多的交易需求。

- 闪电网络：链下建立可信的微支付通道，将最后的结算放入到比特币区块链中，需要比特币隔离见证的支持。

- 隔离见证：将比特币交易输入中的解锁脚本 signature script 拿出来，解决了交易延展性问题(transaction-malleability)。同时可以减小交易的大小，使得在同样的区块大小限制中，允许更多的交易。

- 侧链：将比特币区块链与其它区块链链接起来，实现更大的价值转移。

Ethereum

- Account 模型
- Transaction 模型
- Block 模型
- Smart Contract 与 EVM
- POW 共识算法
- Future

Account 模型

- 以太坊采用 Account 模型，因此以太坊的 World State 可以体现在最新的区块中，而不用像比特币，需要回溯整个历史。

- Account 有两种类型：externalised account 与 contract account。Account 由以下字段组成：(1) **nonce：**在 externalised account 中，代表从该账户地址发送过的交易数量，该数量会出现在交易的字段中，起到防止双花的目的。在 contract account 中，代表由该账户创建过的合约数量；(2) **balance：**该账户拥有的余额；(3) **storageRoot：**Merkle Patricia 树的 256 位根哈希，该树是一种 Merkle 与 Trie 树的混合体，是以太坊独有的概念。storageTree 包含了属于该账户的存储内容 (一个关于 256 位整数值之间的映射) 的编码；(4) **codeHash：**属于该账户的 EVM code 的哈希值。当该账户地址接收到一个 message call 的时候，该 EVM code 会被 EVM 执行。不像账户的其它字段，codeHash 是不可变的，被用来从状态数据库中获得相应的 ECM code。

Transaction 模型

- **nonce :** 与该交易发送地址中的 nonce 相同，256 位；
- **gasPrice :** 交易执行过程需要消耗 gas，每一个单位 gas 的价值为 gasPrice Wei (1 Ether = 10 ^ 18 Wei)，256 位；
- **gasLimit :** 交易执行过程中能够消耗的最大 gas 的个数，为了防止垃圾交易的恶意攻击，gasPrice * gasLimit 数量的 Wei 需要提前在发送者的账户中扣除，256 位；
- **to :** 该 message call 的 160 位接收方地址；对于一笔 “合约创建” 交易，to 为空，160 位；
- **value :** 发送给接收者的 Wei 数量，对于 “合约创建” 交易，作为新创建的合约账户的余额，256 位；
- **v, r, s :** 与该交易签名有关的值，用来确定该交易的发送者，v 为 5 位，r、s 均为 256 位；
- **init :** 账户初始化过程中需要的无限制大小的字节数组，该字段为 “合约创建” 交易所特有。init 是一个 EVM code 片段，执行完成之后会返回另一个 EVM code body，每当接收到一个 message call (要么是一笔交易触发，要么是一个合约代码的内部执行所触发)，body 就会被执行。init 只会在合约账户创建过程中执行一次，随后该代码片段会被丢弃；
- **data :** message call 所需要的输入数据，无限制大小，为 message call 交易所特有。

Block 模型

- **BlockHeader** 由以下结构组成：
- **parentHash :** 父区块头的 Keccak 256 位哈希值；
- **ommersHash :** 该区块的叔区块列表的 Keccak 256 位哈希值；
- **beneficiary :** 该区块所有交易手续费的接收者地址，即成功挖到该区块的矿工地址；
- **stateRoot :** 在区块中所有交易执行完成后，state trie 树的根哈希值，该树的叶子节点是 account，索引的 key 为 account 地址；
- **transactionsRoot :** 该区块中所有交易组成的 Merkle Patricia 树的根哈希值；
- **receiptsRoot :** 该区块中所有交易对应的所有的收据组成的 Merkle Patricia 树的根哈希值；
- **logsBloom :** 布隆过滤器，由 receipts 中每一个 log entry 中的 logger address 与 log topics 形成；

- **difficulty :** 该区块的难度值，由前一个区块的难度值以及时间戳计算得出；
- **number :** 该区块有多少个父区块，创世区块中该值为 0；
- **gasLimit :** 该区块中 gas 消耗的限制数量；
- **gasUsed :** 该区块中所有交易使用的 gas 数量；
- **timestamp :** 一个合理的该区块开始生成的 unix 时间戳；
- **extraData :** 一个任意的与该区块相关联数据的字节数组，大小限制为 32 个字节；
- **mixHash :** 一个 256 位的哈希值，与 nonce 结合来证明在该区块上已消耗了足够多的计算能力；
- **nonce :** 一个 64 位的哈希值，与 mixHash 结合来证明在该区块上已消耗了足够多的计算能力；
- **block** 组成结构 : <header, transaction list, ommer block headers list>

Smart Contract

- 基于以太坊的域名注册系统

```
def register(name, value):
    if !self.storage[name]:
        self.storage[name] = value
```

- 基于以太坊的代币系统

```
def send(to, value):
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

EVM

- EVM 是以太坊开发的基于栈的 Ethereum 虚拟机，智能合约的代码是跑在 EVM 上的。EVM 支持图灵完备，大大扩展了比特币的脚本语言功能，使得开发人员可以编写基于以太坊平台的任意功能的智能合约。来看一段 EVM 可以执行的字节码：

- PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0 CALLDATALOAD SSTORE

- 该字节码程序实现了域名注册系统，任何人都可以发送一个包含 64 个字节 data 的消息，其中 32 个字节当作 key，即域名注册者；另外 32 个字节当作 value，即域名。合约检查 key 是否存在于合约账户的存储中，如若不存在，将其插入合约账户自身的存储中。

- 在执行过程中，一个无限可延伸的字节数组，称为 “memory”，程序计数器，指向当前要执行的指令，称为 “PC”，和一个基于 32 字节的栈被 EVM 所维护使用。开始执行时，PC = 0，内存与栈是空的。现在假设，一个消息被发送，消息包含 123 wei gas 费用与 64 字节的数据，头 32 个字节是数字 54 的编码，后 32 个字节是数字 2020202020 的编码。

- 因此，初始状态是 : <PC : 0, STACK : [], MEM : [], STORAGE : {}> ，位于 0 处的指令是 PUSH1，代表将一个字节的数据压入栈中，并且在 code 中跳跃 2 步，即现在的状态是 <PC : 2, STACK : [0], MEM : [], STORAGE : {}>。位于 2 处的指令是 CALLDATALOAD，代表从当前的栈中弹出一个值 index，装载位于消息 64 个字节数据中 index 位置处的 32 个字节，并且将该数据压入栈中，即现在的状态是 <PC : 3, STACK : [54], MEM : [], STORAGE : {}>

- PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0 CALLDATALOAD SSTORE

- 位于 3 处的 SLOAD 指令，从栈中弹出一个值 index，将合约账户 storage 中索引 index 处的值压入栈中，由于该合约是第一次使用，因此值为0，即现在的状态为 <PC : 4, STACK : [0], MEM : [], STORAGE : {}>。位于 4 处的 NOT 指令从栈中弹出一个值，如果该值为 0 则将 1 压入栈中，反之将 0 压入栈中，即现在的状态为 <PC : 5, STACK : [1], MEM : [], STORAGE : {}>。位于 5 处的指令 PUSH1 执行完成之后，状态变为 <PC : 7, STACK : [1, 9], MEM : [], STORAGE : {}>。位于 6 处的指令 JUMPI 从栈中弹出两个值，a1 : 9，a2 : 1，如果第二个值 a2 为非 0 值，则跳转到 a1 指定的值。如果合约账户的 storage 的索引 54 处的值不为 0，那么 a2 将会变为 0 (由于 NOT 指令)，那么我们就不会跳转到 a1 指定的值，那么接下来的指令就将会是 STOP，从而中止代码的执行。现在的状态是 <PC : 9, STACK : [], MEM : [], STORAGE : {}>。位于 9 处的指令 JUMPDEST 不会对虚拟机状态造成任何影响，仅仅是标记一个有效的跳转地址，PC 变为 10，接下来位于 10 处的 PUSH1 执行完成之后，状态变为 <PC : 12, STACK : [32], MEM : [], STORAGE : {}>。位于 12 处的指令 CALLDATALOAD 执行完成之后，从栈中弹出一个值 index : 32，然后将消息中 64 字节数据中位于索引 32 处之后的 32 个字节压入栈中，现在状态变为 <PC : 13, STACK : [2020202020], MEM : [], STORAGE : {}>。接下来执行 13 处的 PUSH1，状态为 <PC : 15, STACK : [2020202020, 0], MEM : [], STORAGE : {}>。位于 15 处的 CALLDATALOAD，从栈中弹出索引值 0，再次将消息数据中前 32 个字节数据压入栈中，此时状态变为 <PC : 16, STACK : [2020202020, 54], MEM : [], STORAGE : {}>。位于 16 处的 SSTORE 操作从栈中弹出两个值 a1 : 54，a2 : 2020202020，存入合约账户的存储中，即 <PC : 17, STACK : [], MEM : [], STORAGE : {54 : 2020202020}>，位于 17 处没有任何指令，因此 EVM 中止运行。

POW 共识算法

- 以太坊中使用 pow 算法是基于 **Dagger-Hashimoto** 的一个修改版本，称为 **Ethash**。与比特币 pow 类似，是一个不断寻找 nonce 的过程，直到得到的结果小于 difficulty 阈值；与比特币 pow 不同的是，Ethash pow 是特消耗内存的，这样使得它可以抵抗 ASIC 矿机。这意味着计算 pow 需要依赖于 nonce 与 header 的一个固定资源，这个资源 (几个 G 大小) 称为 **DAG**。每 30000 个块 (大概 100 小时，称为一个 epoch) DAG 是完全不同的，需要消耗一定的时间来生成。既然 DAG 只依赖于区块高度，它可以预先被生成，如果不这样做的话，那么矿工就必须等待 DAG 生成之后才能继续挖矿。

- **注意：**在验证 pow 是否有效的时候，不需要 DAG 的参与，因此允许低内存与低 CPU 的客户端对 pow 进行有效性的验证。

- **Mining Rewards：**成功挖到区块的矿工接收一下奖励 : (1) 成功挖到区块的固定奖励 5 Ether；(2) 执行交易过程中，消耗的 gas，从交易发起者账户中扣除，以太坊网络协议发送给矿工；(3) 包含叔区块到区块中的额外奖励，以被包含的叔区块的 1/32 作为额外奖励。注意：最多往上追溯 6 代，有效的叔区块被奖励，以中和网络滞后带来的挖矿奖励分散的影响，7/8 的奖励会发送给叔区块的缔造者，即 4.375 个 Ether。另外，每个块至多允许包含 2 个叔区块，而且叔区块只能被包含一次。

Future

- 从 POW 过渡到 POS Casper 算法，将在 Ethereum Serenity 即 2.0 中实现。

- 实现 sharding 技术，提高区块链目前的 scalability，也将在 Ethereum Serenity 即 2.0 中实现。

Bubi

- Account
- Transaction
- Ledger
- Consensus

Account

- **account_balance :** 代表此账户余额，int64 类型；
- **previous_ledger_seq :** 代表由该账户发出的最后一笔交易所在区块的序列号，int64 类型；
- **previous_tx_hash :** 代表该账户发出的最后一笔交易的哈希值，bytes 类型；
- **account_address :** 代表该账户地址，string 类型；
- **assets :** 代表该账户普通资产，是一个 Asset 类型的数组，可量化；
- **tx_seq :** 代表该账户发出的交易笔数，防止双花，int64 类型；
- **thresholds :** 代表解锁该账户时需达到的阈值，在多方签名时使用，AccountThreshold 类型；
- **signers :** 代表能够解锁该账户的签名者列表，在多方签名时使用，是一个 Signer 类型的数组，其中 Signer 类型由地址与权重组成；
- **metadata :** 元数据，扩展时使用，bytes 类型；
- **asset_unique :** 唯一资产列表，是一个 Asset_unique 类型数组，不可量化。

Transaction

- **source_address :** 该交易的发起方地址，string 类型；
- **source_publicKey :** 该交易发起方的公钥，string 类型；
- **fee :** 该交易发起方需要支付的费用，uint32 类型；
- **sequence_number :** 该交易的序列号，需与该交易发起方中的 tx_seq 对应，int64 类型；
- **close_time_range :** 该交易关闭时间的一个范围，CloseTimeRange 类型；
- **operations :** 定义了该交易具体实施的操作，是一个 Operation 类型的数组；Operation 有创建账户、转账支付、发行资产等操作类型；注意一笔交易里的所有操作是原子的，即 operations 中只要有一个操作失败，整个交易的所有操作作废；
- **metadata :** 元数据，供扩展时使用，bytes 类型。

LedgerHeader

- **parent_hash :** 上一个区块的哈希值，bytes 类型；
- **transaction_tree_hash :** 该区块包含的交易集的哈希值，bytes 类型；
- **account_tree_hash :** 区块链包含的账户组成的 Merkle 树的根哈希值，bytes 类型；
- **ledger_sequence :** 该区块的序列号，创世区块序列号为 0，int64 类型；
- **consensus_value :** 该区块共识出来的值，Value 类型；Value 类型中包含共识出来的交易集哈希值、关闭时间以及可选的 ledger_upgrade，因为有的时候需要对升级过后的 Ledger 进行共识，比如更改最低交易费、Ledger 版本号、账户最低保留费用、元数据最大长度等；
- **hash :** 该区块哈希值，bytes 类型；
- **base_fee :** 需要从本区块包含的所有交易的账户发起者中扣除的最低费用，uint32 类型；
- **base_reserve :** 区块链包含的账户最低的保留费用，注意，这间接反映了创建账户是开销较大，因为需要创建账户交易的发起者来付被创建账户的 base_reserve，unit32 类型；
- **ledger_version :** Ledger 版本号，uint32 类型；
- **tx_count :** 本区块包含的交易个数，int64 类型。

Consensus

- Bubi 目前采取的共识算法是 PBFT，简单介绍一下 PBFT 算法的基本工作流程：

- PBFT 算法可以容忍小于 1/3 恶意节点，小于 1/3 善意故障节点，即在 3f + 1 个节点组成的系统中，可以容忍 f 个拜占庭节点，f 个善意故障节点。

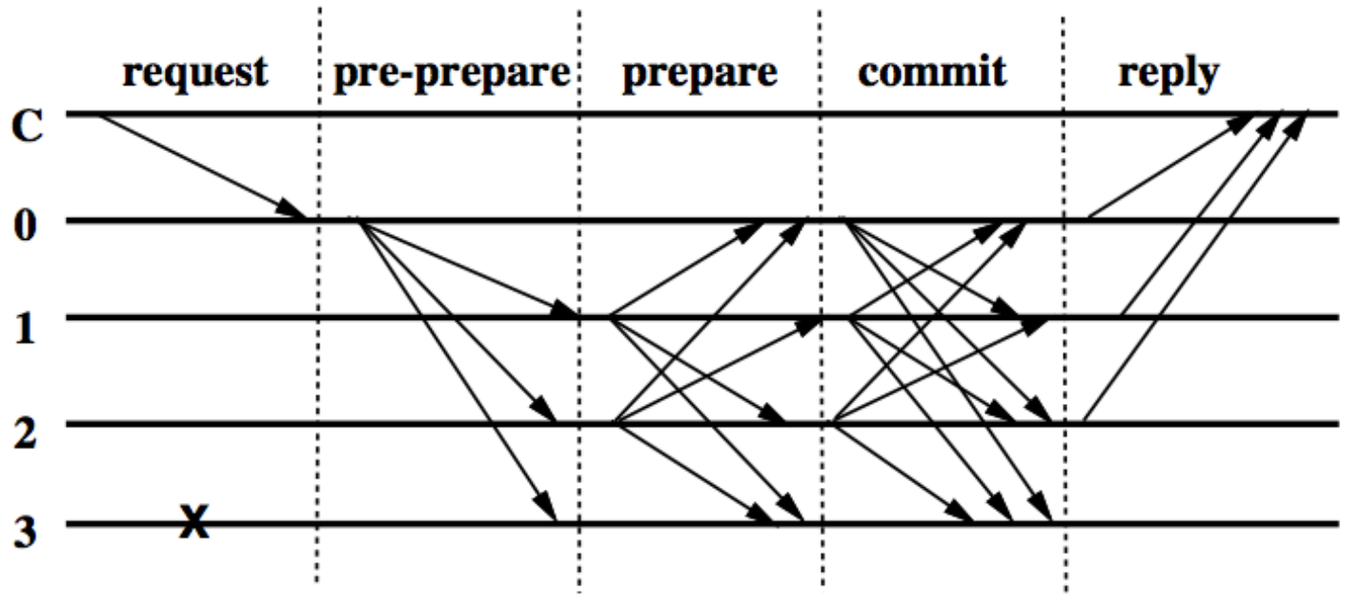


Figure 1: Normal Case Operation

Hyperledger Fabric

- IBM 主导的联盟链开源项目

- 基于 Docker 容器的 chaincode，可使用 go、java、nodejs 来编写，实现了图灵完备的智能合约

- 共识算法可插拔，目前官方实现的共识采用 PBFT 算法，能够容忍少于 1/3 的拜占庭恶意节点个数，即 3f + 1 个共识节点组成的共识网络，允许至多 f 个恶意节点的出现

- 架构设计见 PDF 文档

Zcash

- Zerocoin VS Zerocash VS Zcash

Zerocoin

- Zerocoin 可以被看做成一个去中心化的混合器，它应用了零知识证明来阻止交易图谱的分析。Zerocoin 证明 coins 是有效的，通过以零知识形式地证明 coins 属于一个公开的有效 coins 列表 (维护在区块链中)。但是 Zerocoin 不是一个成熟的匿名货币，它更像是一个去中心化的混合器 (decentralised mix)，用户可以借助 Zerocoin 协议周期性地清洗他们的比特币。日常的交易还是需要以比特币正常交易形式进行，由于以下原因：

- **性能。**赎回 Zerocoin 需要 double-discrete-logarithm 知识证明，该知识证明的大小超过 45KB，需要花费 450ms 时间来验证 (在 128 位的安全水平)。这些证据需要在整个网络中广播，被每一个节点验证，永久地存储在区块链中。所需的花费比比特币中正常交易花费要高出几个数量级，给正常运行的比特币网络带来严重的负担。

- **功能性。**首先，Zerocoin 使用固定面值的 coins：它既不支持精确价值的付款，也不提供一种方式找零。其次，Zerocoin 没有提供用户向另一用户直接以 "zerocoin" 形式付款的机制。最后，尽管 Zerocoin 通过断开一笔交易与它的原始地址的关联提供了匿名性，但是却没有隐藏交易的金额以及其它关于发生在网络中的交易的元数据。

- Zerocoin 扩展了比特币协议，通过创建两种新交易类型：mint 与 spend。一笔 mint 交易允许用户交换一定数量的比特币来发行一个新的 zerocoin。每一个 zerocoin 由基于随机序列号 sn 的数字承诺 cm 组成。随后的某一个时刻，一个用户发行一笔 spend 交易，包含目的实体，序列号 sn，与一个针对 NP 陈述 {"我知道秘密 cm 与 r 满足 (i) cm 能由 sn 与 承诺随机数 r 得出，并且 (ii) cm 在过去某一个时刻被发行 (mint)"} 的非交互性零知识证明。至关重要地，零知识证明没有将 spend 交易与特定的 mint 交易 (目前为止的所有 mint 交易) 相关联。如果证明验证正确，并且 sn 先前没有被使用过，那么该协议就会将一定数量的比特币转到目的地址。Zerocoin 表现的就像是一个去中心化的混合器。

Zerocash

- 我们构建一种去中心化匿名支付模式 (decentralized anonymous payment (DAP) scheme)，是一种去中心化的电子货币模式，允许任意数量的直接匿名转账付款。接下来以 6 个功能逐渐递增的步骤来刻画 DAP 的构建过程 (基于任何基于 ledger 的货币，例如比特币)：

- **Step 1： user anonymity with fixed-value coins。**首先给出一个简化的构建，所有的 coins 有相同的价值，例如 1 BTC。这个构建模式与 Zerocoin 类似，展现如何隐藏一笔转账付款的原始地址。我们使用 zk-SNARKs 和 承诺 (commitment) 模式。令 COMM 为统计学上隐藏地非交互性承诺模式 (例，给定一个随机数 r 与 消息 m，承诺是 cm := $\text{COMM}_r(m)$；随后，可以通过披露 r 与 m 来验证 $\text{COMM}_r(m)$ 是否等于 cm)。在该简化构建中，一个新币 c 按照如下方式发行：用户 u 随机选一个序列号 (serial number) sn 与 trapdoor r，计算出 coin commitment cm := $\text{COMM}_r(sn)$，并且令 c := (r, sn, cm)。一笔相应的发行交易 tx_{Mint}，包含 cm (却既不包含 sn 也不包含 r)，被发送到 ledger 中；只有在用户 u 已经付了 1 BTC 到一个支撑的保管池后，tx_{Mint} 才能够被追加到 ledger中。tx_{Mint} 就像是存款的凭证一样，可以从支撑池中获取它们的价值。

- 随后，令 CMList 为 ledger 中所有 coin commitment 的列表，u 可以发送一笔花费交易 tx_{Spend} 来花费 c，tx_{Spend} 包含 (i) coin 的序列号 sn；(ii) 一个 zk-SNARKs 证明 π 针对于 NP 陈述 “我知道 *r* 使得 $\mathit{COMM}_r(sn)$ 出现在 *CMList* 中”。假设 sn 不曾在 ledger 中出现过，用户 u 就能够赎回 1 BTC，这 1 BTC 用户 u 可以随意支配，保留、转账、发行新币等。如果 sn 曾经出现过，则该笔交易会被认为是双花交易，裁定为无效并且被丢弃。

- 该构建方法实现了用户匿名性，因为证明 π 是零知识的：尽管 sn 被披露，但是没有任何关于 r 的信息。并且寻找花费交易 tx_{Spend} 所对应的发行交易 tx_{Mint} 相当于寻找 f(x) := $\mathit{COMM}_x(sn)$，被认为是不可行的，因此转账付款的起始地址得到了保护。

- **Step 2 : 压缩 CMList。**step 1 提到的 CMList 极大地限制了证明确认算法的扩展性，由于时间和空间上的复杂度。由于 CMList 会随着新币的发行不断增长，并且已被花费的 coin 对应的 commitment 不能从 CMList 中删除，由于零知识提供的匿名性。

- 我们应用一种抵抗冲突的函数 CRH 来避免 CMList 的显示表达形式，即我们在 CMList 上维护一棵基于 CRH 的 Merkle 树，令 rt 代表 Tree(CMList)。众所周知，rt 可以在于树深度成比例的时间与空间复杂度规模内通过插入叶子节点即 coin commitment 来进行更新。因此时间与空间复杂度从原先的 CMList 的长度，降低到对数范围。利用该思想，我们修改上述的 NP 陈述为：”我知道 r 使得 $COMM_r(sn)$ 作为一个叶子出现在一棵根为 rt 基于 CRH 的 Merkle 树中”。这使得 zk-SNARKs 算法可以支撑更多的 coin commitments。

- **Step 3 : 扩展 coins 使得能够支持直接地匿名转账付款。**目前为止，coin commitment cm 是相对于币序列号 sn 的承诺。但是，却带来一个很严重的问题。假设用户 Alice 创建了 coin c，Alice 想将 c 转移给 Bob，Bob 能够花费该 coin 的前提是 Alice 将手中的 r 与 sn 传递给自己，这样一来 Bob 花费 c 的行为就不再匿名，因为 Alice 也知道 sn，Alice 可以监听网络中的花费交易来匹配 sn。除此之外，Alice 也能够花费该 c 除非 Bob 一拿到 Alice 的 r 与 sn 就花费掉 c。所以 Step 1 所提出的简单构建不能满足 DAP 的需要。

- 我们通过修改 coin commitment 的获取方式解决了上述问题，使用伪随机函数来获取序列号与定位付款目标。我们使用了 3 个伪随机函数 (从单个伪随机函数派生出来)。对于一个种子 x，3 个伪随机函数的表现形式为 PRF[x][addr]、PRF[x][pk]、PRF[x][sn]。

- 为了提供付款的目标，我们使用 address 的概念：每一位用户生成一个地址密钥对 (apk, ask)。用户的 coins 需要包含 apk，而且只有知道 ask 的情况下才能花费 coins。ask 是通过随机挑选生成的，令 apk := PRF[x][addr](0)，用户可以生成任意个地址密钥对。

- 接下来，我们重新设计了发行交易来允许更大的功能性。为了发行一个想要的价值 v 的 coin，用户 u 首先挑选了一个随机数 ρ，ρ 是一个决定序列号 sn 的安全值，sn := PRF[x][sn](ρ)。接下来 u 使用两阶段提交 (apk, v, ρ)：(i) 用户 u 计算 k := $COMM_r$(apk || ρ) 对于随机挑选的 r 值；(ii) 用户 u 计算 cm := $COMM_s$(v || k) 对于随机挑选的 s。发行结果是 coin c := (apk, v, ρ, r, s, cm) 和一笔发行交易 tx_{Mint} := (v, k, s, cm)。任何人都可以通过 $COMM_s$(v || k) 来验证 cm 是否是一个价值为 v 的 coin 的承诺，但是却不能分辨出拥有者 apk 与 序列号 sn (从 ρ 得出)，因为这些都统统被隐藏在了 k 中。和以前一样，只有用户 u 花费了等量数值 v 的 BTC，tx_{Min} 才可以被 ledger 所接受。

- pour 交易花费 coins，使用想要花费的 coins 作为输入，生成新鲜的 output coins，output coins 的总价值要等于 input coins 的总价值。假设持有地址密钥对 (ask, apk) 的用户 u 想要花费他的 coins c_old = (apk_old, v_old, ρ_old, r_old, s_old, cm_old)，产生两个 new coins c1_new, c2_new，并且 v_old = v1_new + v2_new，相应的两个目标地址为 apk1_new、apk2_new (可以属于 u 或者任何其它用户)。对于 i ∈ {1, 2}，用户 u 执行下列操作：(I) u 随机挑选能够获得序列号的随机数 ρi_new；(II) u 计算 ki_new := $COMM_{ri\_new}$(apki_new || ρi_new)，对于随机挑选的 ri_new；(III) u 计算 cmi_new := $COMM_{si\_new}$(vi_new || ki_new)，对于随机挑选的 si_new。

- 这就产生了新币 c1_new := (apk1_new, v1_new, ρ1_new, r1_new, s1_new, cm1_new) 与 新币 c2_new := (apk2_new, v2_new, ρ2_new, r2_new, s2_new, cm2_new)。接下来，用户 u 产生一个 zk-SNARK 证明 π_{POUR}，对应于下列我们成为 POUR 的 NP 陈述：

- “
- 给定 Merkle-Tree 根 rt，序列号 sn_old，与 coin commitments cm1_new，cm2_new，我知道 coins c_old，c1_new，c2_new 与地址密钥 ask_old 使得：
- coins 是正确构建的：对于 c_old，下列成立：$k_{old} = COMM_{r_{old}}(apk_{old} \| \rho_{old})$ 与 $cm_{old} = COMM_{s_{old}}(v_{old} \| k_{old})$；c1_new 与 c2_new 也是类似；
- 地址私钥匹配地址公钥，即 $apk_{old} = PRF[ask_{old}][addr](0)$；
- 序列号被正确计算：$sn_{old} = PRF[ask_{old}][sn_{old}](\rho_{old})$；
- coin commitment cm_old 出现在根为 rt 的 Merkle 树的叶子节点中；
- 价值转移相等：v_old = v1_new + v2_new。
- ”
- 即 pour 交易的形式为 $tx_{Pour} := (rt, sn_{old}, cm1_{new}, cm2_{new}, \pi_{POUR})$，被添加到 ledger 中，和之前一样，如若 sn_old 在先前的一笔交易中出现，那么该交易就会认为是双花而被拒绝进入 ledger。

- 如若用户 u 不知道与 apk1_new 相关联的 ask1_new，那么 u 是不能花费 c1_new，因为 u 无法在随后的 pour 操作的见证中提供 ask1_new，即无法生成合格有效的 π_{POUR}。还有就是当拥有 ask1_new 的用户花费 c1_new 的时候，用户 u 是无法追踪的，因为 u 不知道 c1_new 对应的序列号 sn1_new (sn1_new := PRF[ask1_new][sn](ρ1_new))。

- 注意，tx_{Pour} 没有披露任何关于 c_old 的价值是如何分配到 ci_new 的信息，也没有披露 c_old 对应花费的是哪一笔 cm_old 的信息，还没有披露 ci_new 对应的目标地址公钥的信息，因此，转账是以完全匿名的方式进行的。

- 更简单地且不是通用性地，我们设置 Nnew = Nold = 2。当 Nold < 2 的时候，用户可以发行一个价值为 0 的 coin，然后将该 coin 作为 null input；当 Nnew < 2 的时候，用户可以发行一个价值为 0 的 new coin。对于 Nold > 2 或者 Nnew > 2，用户可以构建 log Nold + log Nnew 个2-input/2-output pours。

- **Step 4 : 发送 coins。**假定 apk1_new 是用户 u1 的地址公钥。为了允许 u1 能够花费上述产生的 c1_new，用户 u 必须将 c1_new 中的某些安全值发送给用户 u1。一种方法是用户 u 给 u1 发送一条私密消息，但是依赖于一条可信的私密通道，无疑增加了使用的负担。我们避免了链下的私密通信，将这种能力嵌入到 zerocash 的构建过程。

- 首先，我们修改了地址密钥对的结构。现在每一位用户拥有一个密钥对 (addr_pk, addr_sk)，其中 addr_pk := (apk, pk_enc)，addr_sk := (ask, sk_enc)，(apk, ask) 的生成办法同先前一样，(pk_enc, sk_enc) 是一个 *“key-private encryption scheme”* 密钥对。所谓的 key-private 意思是，密文不能被链接到加密它们的公钥，或与使用同一把公钥加密得到的其它密文联系起来。

- 然后，用户 u 使用 pk_enc1_new 加密明文 (v1_new, ρ1_new, r1_new, s1_new) 得到密文 C1，并且将 C1 加进 tx_{Pour} 中。用户 u1 然后可以找到并且使用 sk_enc1_new 解密 C1 从而还原 (v1_new, ρ1_new, r1_new, s1_new)，注意用户 u1 需扫描 ledger 中的 tx_{Pour} 消息。再次注意，密文 C1 的出现没有泄露任何关于目的地址、转账金额的信息，由于加密方法的 key-private 特性。(用户 u 对 c2_new 进行同样的操作，将密文 C2 加入到 tx_{Pour}。)

- **Step 5：公开的输出。**到目前为止，构建方法允许用户发行、合并、分裂 coins。但是一个用户如何赎回他 coins 的价值，例，将它转换为基础货币 (BTC)？为了达成这个目的，我们修改了 pour 操作以包含一个 *public output*。当花费一个 coin 的时候，用户 u 同时也可以设置一个非负值 v_pub 和一个交易描述 info ∈ {0, 1}*。NP 陈述 POUR 中的余额计算方式相应发生改变：v1_new + v2_new + v_pub = v_old。如此，输入 v_old 中的一部分 v_pub 被公开地声明，并且 v_pub 的目标接收地址也被披露，在 info 里。info 可以被用来具体定义赎回金额的目的地址 (在 Bitcoin 中，是 public key)。v_pub 与 info 被包含在 交易tx_{Pour} 中。(v_pub 是可选的，用户可以将其设置为 0)

- **Step 6 : non-malleability。**为了防止对于交易 tx_{Pour} 的延展性攻击 (例如，通过修改 info 来使 v_public 的输出地址发生改变)，我们修改该 NP 陈述 POUR，并且使用数字签名。具体地，在 pour 操作过程中，用户 u (i) 对于一次性签名模式，随机生成一个密钥对 (pk_sig, sk_sig)；(ii) 计算 h_{Sig} := CRH(pk_sig)；(iii) 计算两个值 h1 := PRF[ask1_old][pk](h_{Sig}) 与 h2 := PRF[ask2_old][pk](h_{Sig})，作用是作为消息验证码将 h_{Sig} 绑定到两个地址的私钥上；(iv) 修改 POUR 来包含 h_{Sig}、h1、h2，并且证明后两者被正确地计算出来；(v) 使用 sk_sig 来对相关的值进行签名，获得签名 σ，与 pk_sig 一起放入 tx_{Pour} 交易中。既然 aski_old 是私密的，并且每一次 pour 交易，h_{Sig} 都会改变，因此 h1 与 h2 是不可预测的。

Setup

- INPUTS: security parameter λ
- OUTPUTS: public parameters pp

1. Construct $C_{\mathtt{POUR}}$ for $\mathtt{POUR}$ at security λ.
2. Compute $(\mathsf{pk}_{\mathtt{POUR}}, \mathsf{vk}_{\mathtt{POUR}}) := \mathsf{KeyGen}(1^\lambda, C_{\mathtt{POUR}})$.
3. Compute $\mathsf{pp}_{\mathsf{enc}} := \mathcal{G}_{\mathsf{enc}}(1^\lambda)$.
4. Compute $\mathsf{pp}_{\mathsf{sig}} := \mathcal{G}_{\mathsf{sig}}(1^\lambda)$.
5. Set $\mathsf{pp} := (\mathsf{pk}_{\mathtt{POUR}}, \mathsf{vk}_{\mathtt{POUR}}, \mathsf{pp}_{\mathsf{enc}}, \mathsf{pp}_{\mathsf{sig}})$.
6. Output pp.

CreateAddress

- INPUTS: public parameters pp
- OUTPUTS: address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$

1. Compute $(\mathsf{pk}_{\mathsf{enc}}, \mathsf{sk}_{\mathsf{enc}}) := \mathcal{K}_{\mathsf{enc}}(\mathsf{pp}_{\mathsf{enc}})$.
2. Randomly sample a $\mathsf{PRF}^{\mathsf{addr}}$ seed a_{sk}.
3. Compute $a_{\mathsf{pk}} = \mathsf{PRF}^{\mathsf{addr}}_{a_{\mathsf{sk}}}(0)$.
4. Set $\mathsf{addr}_{\mathsf{pk}} := (a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
5. Set $\mathsf{addr}_{\mathsf{sk}} := (a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$.
6. Output $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$.

Mint

- INPUTS:
  - public parameters pp
  - coin value $v \in \{0, 1, \ldots, v_{\mathsf{max}}\}$
  - destination address public key $\mathsf{addr}_{\mathsf{pk}}$
- OUTPUTS: coin $\mathbf{c}$ and mint transaction $\mathsf{tx}_{\mathsf{Mint}}$

1. Parse $\mathsf{addr}_{\mathsf{pk}}$ as $(a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
2. Randomly sample a $\mathsf{PRF}^{\mathsf{sn}}$ seed ρ.
3. Randomly sample two COMM trapdoors r, s.
4. Compute $k := \mathsf{COMM}_r(a_{\mathsf{pk}} \| \rho)$.
5. Compute $\mathsf{cm} := \mathsf{COMM}_s(v \| k)$.
6. Set $\mathbf{c} := (\mathsf{addr}_{\mathsf{pk}}, v, \rho, r, s, \mathsf{cm})$.
7. Set $\mathsf{tx}_{\mathsf{Mint}} := (\mathsf{cm}, v, *)$, where $* := (k, s)$.
8. Output $\mathbf{c}$ and $\mathsf{tx}_{\mathsf{Mint}}$.

Pour

- INPUTS:
  - public parameters pp
  - the Merkle root rt
  - old coins $\mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}$
  - old addresses secret keys $\mathsf{addr}_{\mathsf{sk},1}^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},2}^{\mathsf{old}}$
  - path path_1 from commitment $\mathsf{cm}(\mathbf{c}_1^{\mathsf{old}})$ to root rt, path path_2 from commitment $\mathsf{cm}(\mathbf{c}_2^{\mathsf{old}})$ to root rt
  - new values $v_1^{\mathsf{new}}, v_2^{\mathsf{new}}$
  - new addresses public keys $\mathsf{addr}_{\mathsf{pk},1}^{\mathsf{new}}, \mathsf{addr}_{\mathsf{pk},2}^{\mathsf{new}}$
  - public value v_{pub}
  - transaction string info
- OUTPUTS: new coins $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$ and pour transaction $\mathsf{tx}_{\mathsf{Pour}}$

1. For each $i \in \{1, 2\}$:
   (a) Parse $\mathbf{c}_i^{\mathsf{old}}$ as $(\mathsf{addr}_{\mathsf{pk},i}^{\mathsf{old}}, v_i^{\mathsf{old}}, \rho_i^{\mathsf{old}}, r_i^{\mathsf{old}}, s_i^{\mathsf{old}}, \mathsf{cm}_i^{\mathsf{old}})$.
   (b) Parse $\mathsf{addr}_{\mathsf{sk},i}^{\mathsf{old}}$ as $(a_{\mathsf{sk},i}^{\mathsf{old}}, \mathsf{sk}_{\mathsf{enc},i}^{\mathsf{old}})$.
   (c) Compute $\mathsf{sn}_i^{\mathsf{old}} := \mathsf{PRF}^{\mathsf{sn}}_{a_{\mathsf{sk},i}^{\mathsf{old}}}(\rho_i^{\mathsf{old}})$.
   (d) Parse $\mathsf{addr}_{\mathsf{pk},i}^{\mathsf{new}}$ as $(a_{\mathsf{pk},i}^{\mathsf{new}}, \mathsf{pk}_{\mathsf{enc},i}^{\mathsf{new}})$.
   (e) Randomly sample a $\mathsf{PRF}^{\mathsf{sn}}$ seed ρ_i^{new}.
   (f) Randomly sample two COMM trapdoors $r_i^{\mathsf{new}}, s_i^{\mathsf{new}}$.
   (g) Compute $k_i^{\mathsf{new}} := \mathsf{COMM}_{r_i^{\mathsf{new}}}(a_{\mathsf{pk},i}^{\mathsf{new}} \| \rho_i^{\mathsf{new}})$.
   (h) Compute $\mathsf{cm}_i^{\mathsf{new}} := \mathsf{COMM}_{s_i^{\mathsf{new}}}(v_i^{\mathsf{new}} \| k_i^{\mathsf{new}})$.
   (i) Set $\mathbf{c}_i^{\mathsf{new}} := (\mathsf{addr}_{\mathsf{pk},i}^{\mathsf{new}}, v_i^{\mathsf{new}}, \rho_i^{\mathsf{new}}, r_i^{\mathsf{new}}, s_i^{\mathsf{new}}, \mathsf{cm}_i^{\mathsf{new}})$.
   (j) Set $\mathbf{C}_i := \mathcal{E}_{\mathsf{enc}}(\mathsf{pk}_{\mathsf{enc},i}^{\mathsf{new}}, (v_i^{\mathsf{new}}, \rho_i^{\mathsf{new}}, r_i^{\mathsf{new}}, s_i^{\mathsf{new}}))$.
2. Generate $(\mathsf{pk}_{\mathsf{sig}}, \mathsf{sk}_{\mathsf{sig}}) := \mathcal{K}_{\mathsf{sig}}(\mathsf{pp}_{\mathsf{sig}})$.
3. Compute $h_{\mathsf{Sig}} := \mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})$.
4. Compute $h_1 := \mathsf{PRF}^{\mathsf{pk}}_{a_{\mathsf{sk},1}^{\mathsf{old}}}(1 \| h_{\mathsf{Sig}})$ and $h_2 := \mathsf{PRF}^{\mathsf{pk}}_{a_{\mathsf{sk},2}^{\mathsf{old}}}(2 \| h_{\mathsf{Sig}})$.
5. Set $x := (\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, h_{\mathsf{Sig}}, h_1, h_2)$.
6. Set $a := (\mathsf{path}_1, \mathsf{path}_2, \mathbf{c}_1^{\mathsf{old}}, \mathbf{c}_2^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},1}^{\mathsf{old}}, \mathsf{addr}_{\mathsf{sk},2}^{\mathsf{old}}, \mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}})$.
7. Compute $\pi_{\mathtt{POUR}} := \mathsf{Prove}(\mathsf{pk}_{\mathtt{POUR}}, x, a)$.
8. Set $m := (x, \pi_{\mathtt{POUR}}, \mathsf{info}, \mathbf{C}_1, \mathbf{C}_2)$.
9. Compute $\sigma := \mathcal{S}_{\mathsf{sig}}(\mathsf{sk}_{\mathsf{sig}}, m)$.
10. Set $\mathsf{tx}_{\mathsf{Pour}} := (\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, where $* := (\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
11. Output $\mathbf{c}_1^{\mathsf{new}}, \mathbf{c}_2^{\mathsf{new}}$ and $\mathsf{tx}_{\mathsf{Pour}}$.

VerifyTransaction

- INPUTS:
  - public parameters pp
  - a (mint or pour) transaction tx
  - the current ledger L
- OUTPUTS: bit b, equals 1 iff the transaction is valid

1. If given a mint transaction $\mathsf{tx} = \mathsf{tx}_{\mathsf{Mint}}$:
   (a) Parse $\mathsf{tx}_{\mathsf{Mint}}$ as $(\mathsf{cm}, v, *)$, and $*$ as (k, s).
   (b) Set $\mathsf{cm}' := \mathsf{COMM}_s(v \| k)$.
   (c) Output $b := 1$ if $\mathsf{cm} = \mathsf{cm}'$, else output $b := 0$.
2. If given a pour transaction $\mathsf{tx} = \mathsf{tx}_{\mathsf{Pour}}$:
   (a) Parse $\mathsf{tx}_{\mathsf{Pour}}$ as $(\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, and $*$ as $(\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
   (b) If $\mathsf{sn}_1^{\mathsf{old}}$ or $\mathsf{sn}_2^{\mathsf{old}}$ appears on L (or $\mathsf{sn}_1^{\mathsf{old}} = \mathsf{sn}_2^{\mathsf{old}}$), output $b := 0$.
   (c) If the Merkle root rt does not appear on L, output $b := 0$.
   (d) Compute $h_{\mathsf{Sig}} := \mathsf{CRH}(\mathsf{pk}_{\mathsf{sig}})$.
   (e) Set $x := (\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, h_{\mathsf{Sig}}, h_1, h_2)$.
   (f) Set $m := (x, \pi_{\mathtt{POUR}}, \mathsf{info}, \mathbf{C}_1, \mathbf{C}_2)$.
   (g) Compute $b := \mathcal{V}_{\mathsf{sig}}(\mathsf{pk}_{\mathsf{sig}}, m, \sigma)$.
   (h) Compute $b' := \mathsf{Verify}(\mathsf{vk}_{\mathtt{POUR}}, x, \pi_{\mathtt{POUR}})$, and output $b \wedge b'$.

Receive

- INPUTS:
  - public parameters pp
  - recipient address key pair $(\mathsf{addr}_{\mathsf{pk}}, \mathsf{addr}_{\mathsf{sk}})$
  - the current ledger L
- OUTPUTS: set of received coins

1. Parse $\mathsf{addr}_{\mathsf{pk}}$ as $(a_{\mathsf{pk}}, \mathsf{pk}_{\mathsf{enc}})$.
2. Parse $\mathsf{addr}_{\mathsf{sk}}$ as $(a_{\mathsf{sk}}, \mathsf{sk}_{\mathsf{enc}})$.
3. For each Pour transaction $\mathsf{tx}_{\mathsf{Pour}}$ on the ledger:
   (a) Parse $\mathsf{tx}_{\mathsf{Pour}}$ as $(\mathsf{rt}, \mathsf{sn}_1^{\mathsf{old}}, \mathsf{sn}_2^{\mathsf{old}}, \mathsf{cm}_1^{\mathsf{new}}, \mathsf{cm}_2^{\mathsf{new}}, v_{\mathsf{pub}}, \mathsf{info}, *)$, and $*$ as $(\mathsf{pk}_{\mathsf{sig}}, h_1, h_2, \pi_{\mathtt{POUR}}, \mathbf{C}_1, \mathbf{C}_2, \sigma)$.
   (b) For each $i \in \{1, 2\}$:
      i. Compute $(v_i, \rho_i, r_i, s_i) := \mathcal{D}_{\mathsf{enc}}(\mathsf{sk}_{\mathsf{enc}}, \mathbf{C}_i)$.
      ii. If $\mathcal{D}_{\mathsf{enc}}$'s output is not $\perp$, verify that:
         - $\mathsf{cm}_i^{\mathsf{new}}$ equals $\mathsf{COMM}_{s_i}(v_i \| \mathsf{COMM}_{r_i}(a_{\mathsf{pk}} \| \rho_i))$;
         - $\mathsf{sn}_i := \mathsf{PRF}^{\mathsf{sn}}_{a_{\mathsf{sk}}}(\rho_i)$ does not appear on L.
      iii. If both checks succeed, output $\mathbf{c}_i := (\mathsf{addr}_{\mathsf{pk}}, v_i, \rho_i, r_i, s_i, \mathsf{cm}_i^{\mathsf{new}})$.

Zcash

- Zcash 是 Zerocash 协议的一种具体实现，相比于原始论文 Decentralized Anonymous Payment schem Zerocash，Zcash 进行了一些安全问题上的修复，以及一些性能、功能性、术语方面的调整

Corda

- R3 联盟构建的联盟链

- No Block chain

Corda 具有以下新奇特性：

- 可以使用 JVM 字节码定义新交易类型。
- 交易可以在不同的节点上并发执行，无需节点知晓发生在其它节点上发生的交易。
- 节点以认证的 p2p 网络形式组织。所有的交流通信是直向连接的。
- 没有区块链。使用可插拔的 notaries 来解决交易竞争引起的冲突。一个 Corda 网络中可能会包含多个 notaries，它们使用多种多样的不同算法来提供它们的保证。如此，Corda 没有被绑定到任何特定的共识算法。
- 数据基于 need-to-know 共享。当节点向其它节点发送一笔交易时，节点按需提供该笔交易的依赖图谱，但是没有交易全局广播的概念。
- Bytecode-to-bytecode 译码被用来允许复杂、多步的交易构建协议，称为 flows，建模为阻塞码。代码被转换为一个异步状态机，当消息发送和接收的时候，checkpoints 被写入到节点支撑的数据库中。一个节点一次可能会有成千上万个活跃的 flows，并且它们可能会持续数天，跨越节点重启，甚至升级等过程。Flows 可以将进展信息暴露给节点管理员、用户，并且可能会与人和其它节点进行交互。一个 Flow library 被提供给开发者，以复用公共的 Flow 类型，例如 notarisation，membership broadcast 等等。

- 数据模型允许允许任意的对象图谱存储在 ledger 中。这些图谱被称为 states，并且是数据的原子单位。

- 节点由关系型数据库支撑，ledger 中存放的数据可以使用 SQL 语句查询，以及与私有表数据连接，由于状态定义中的 slots，被保留用来 join keys。

- 平台提供了一个富类型系统来代表诸如日期、货币、合法实体以及金融实体，例如现金、保险、合同等等。

- States 能声明一个关系映射，能够使用 SQL 来进行查询。

- 与现有的系统集成从一开始就被考虑进来。Corda 网络支持从其它数据库系统快速导入数据，而且对网络不会造成负担。ledger 上的事件通过内嵌的 JMS 兼容消息代理来通知。

- States 可以声明调度的事件。例如，一个债券状态如果没有及时被支付，可以声明转移到一个默认的状态。

Chain

- 联盟链
- UTXO 模型
- CVM

Appendix

- PBFT 算法简介
- Transaction Malleability
- Replay Attack Protection
- Ciphertext Indistinguishability

Transaction Malleability

- **交易延展性 (Transaction Malleability)**

- 当交易被签名时，签名并没有覆盖交易中所有的数据 (比如位于 txin 中的 scriptSig，由于 scriptSig 中包含公钥和签名数据，不可能对自身自签名)，而交易中所有的数据又会被用来生成交易的哈希值来作为该交易的唯一标示。如此，尽管不常见的，比特币网络中的一个节点能够改变你发送的交易 (通过改变 txin 中的签名)，导致该交易的哈希值发生变化。注意，攻击者仅仅能够改变该哈希值，交易中的 txout 是无法进行改变的，因此比特币最终也许会转入你原本打算的地址中。然而，这确实意味着，例如，在任何情况下，接收一系列未确认交易的链是不安全的，因为未确认交易的哈希值可能会发生变化，而随后的交易中的 txin 会依赖于先前交易的哈希值。即使交易得到了一个确认，也是不安全的，因为区块链可能会被重新调整。此外，客户端必须一直扫描收到的交易，假定一个 txout 是存在的，因为先前创建该笔交易的客户端可能是不安全的 (可能会发两笔同样 txout 的交易)。

- **签名延展性 (Signature Malleability)**

- 延展性的第一种形式就是签名本身。每一个签名恰好有一个 DER-encoded ASN.1 octet representation，但是 OpenSSL 并没有强制，只要一笔签名没有极度的改变，它就是可接受的。此外，对于每一个 ECDSA signature(r,s)，这个 signature(r,-s(mod N)) 是相同消息的一个有效签名。在 363742 区块高度处，BIP66 软分叉强制区块链中所有新交易遵循 DER-encoded ASN.1 标准。仍然需要进一步的努力来关闭 DER 签名其它可能的延展性问题。签名仍然是可以被拥有相应私钥的人改变的。

- **解锁脚本延展性 (scriptSig Malleability)**

- 比特币中使用的签名算法没有签署 scriptSig 中的任何数据。因为对整个 scriptSig 签名是不可能的，scriptSig 包含签名本身。这意味着，附件的数据能被添加到 scriptSig 中，额外的数据会先于所需的签名和公钥压入栈中。类似的 OP_DROP 能被添加，使得最终栈的状态与 scriptPubKey 执行之前的状态相同。阻止 scriptSig 延展性正在被考虑当中。目前，如果交易 txin 的 scriptSig 中若包含除了签名与公钥之外其它数据，则该交易被认为是不标准的，不会被节点转发。最终，这个规则会强制在 scriptPubKey 执行完之后，栈中只剩下一个元素。然而，要做到这样，需要比特币随后的扩展。

- **隔离验证 (segregated witness)** 通过将 scriptSig 从交易的 txin 中分离出来，放入到一个新添加的 witness 字段中，来解决相关延展性问题，由于签名数据从交易中分离出来，使得交易结构尺寸变小，可以使得区块能容纳更多的交易，侧面达到了轻微扩容的目的，有关隔离验证的知识，请 goolge BIP 141-145。