使比特币交易中的数据具备持续遗传进化能力

张文帅

本文讲述由 一层 Token 方案引发的比特币改进建议。其不仅对 Token 具有重要意义，更对整个比特币网络中的交易（TX）的可遗传进化能力有关键的影响。不夸张的说，比特币网络中，所有需要成长变化的交易数据都将受惠于此改进。

本文提出的改进，是在不影响 UTXO 模型可扩展性的前提下，实现**比特币系统图灵完备能力**的必要一环。比特币的脚本能力已经完备了，其完备涵义的前提是一个假设：我们具有一个无限长的纸带（磁盘与带宽资源）。与此学界理论共识有所区别，我所强调的系统完备是在实用层面，其准确涵义是：**本提议将使比特币脚本的图灵完备能力可以在有限的纸带上具备充分且必要的实用性**。其充分必要的含义有：接近极限的数据资源（存储与带宽）利用率、必不可少的数据资源回收（数据裁剪）能力，本文后续将解释这一点。

为获得以上益处，**仅需要一个优雅、简单、一劳永逸的小改进**。此提议的最小版本可以简短清楚地表示为：

> 将 TXID 的计算公式由
>
> TXID = HASH(HASH(TX-Version:2, UnlockingScript, OtherTxData))
>
> 改为
>
> TXID = HASH(HASH(TX-Version:3, HASH(UnlockingScript), OtherTxData))

本提议仅包括两个部分：a) **将比特币交易中的版本号由 2 改为 3**，b) **将计算 TXID 所依赖的数据序列中 Inputs 的解锁脚本数据 UnlockingScript 改为使用其 Hash 值。**因为我们将 TXID 中的 2 级 Hash 改为了 3 级 Hash，因为此改进可称为 **TXID-3LevelsHash** 改进。因为 TXID 计算过程是一个不可逆的过程，所以此 Hash 计算的单向变化不影响任何其他部分的程序逻辑，不影响矿工使用包括 UnlockingScript 数据在内的 TX 全文做脚本验证，不影响 SPV（简单支付验证）等等。

以下将详细解释上述论点，文中简洁的描述了所有必要的基础知识，部分读者可选择性跳过前面的章节。当遇到名词理解困难时，可以尝试跳转至 **名词解释** 与 **比特币交易数据结构** 处查询（最好预先阅读名词解释部分）。

技术基础知识

我们开始一点点拨开技术的迷雾，理解比特币的一层 Token 智能合约并不困难。

在讲一层 Token 之前，我们需要先首先讲一下比特币的智能合约的核心概念 "**OP_PUSH_TX 技术**"，而在此之前，我们还需要讲一点点比特币的脚本运行原理。

因此，以下倒叙展开，相信我，这些都很简单。说简单并不等同于篇幅短，额是指所需起点与技术基础低，因而反而更长，以使逻辑严密，不需读者脑补缺口。但每一小段都很简单，所需能力仅为注意力的集中。**仅集中注意力一个小时，就可以完成技术的入门，并直达前沿，不再混沌，这会很超值。**

比特币脚本程序的运行方式

当前的比特币脚本，通常是长度很短的一些代码，且不像传统高级程序语言那般具有很复杂的跳转结构，而是顺序的执行，如同一个数学算式一般，并不复杂，反而比一般的计算机程序更简单易读。

以最简单的数学算式：`1+1` 为例，其在比特币脚本中，表达顺序变为 `1 1 +`，我们可以将 `+` 理解为一种最简单的函数，其在比特币脚本术语中称为操作符（OP_CODE）。因为操作符有很多，无法只用单个字符来完整的描述，为方便识别，`+` 在比特币脚本中以 `OP_ADD` 表示。所以，最后的脚本表达式为 `1 1 OP_ADD`。比特币中的操作符（函数）总是对其左侧预定数量的输入数据进行加工处理，然后返回预定数量的数据值。此外，只还需要知道：比特币脚本的运算结合律为从左到右顺序执行操作符（如果将其上下排列，此时则为自下而上顺序执行），最后得到的**数据序列**即为执行结果。本段示例表达式的执行结果自然为单个数字：`2` 。

又例如 `1 1 OP_ADD 3 OP_MUL`，其中 `OP_MUL` 为数学中的算术乘操作（其对左边近邻的 2 个数值做乘法，然后返回一个数值）。其首先执行 `OP_ADD`，得到中间结果：`2 3 OP_MUL`，然后执行 `OP_MUL`，得到最后的执行结果为 `6` 。

再举一个执行结果为数据序列的代表例子：`1 2 OP_DUP`，其中 `OP_DUP` 的含义是复制左侧的单个数据，也就是复制 `2`，得到双份数据，所以其执行结果为 `1 2 2`，是 3 个数值组成的一个序列。

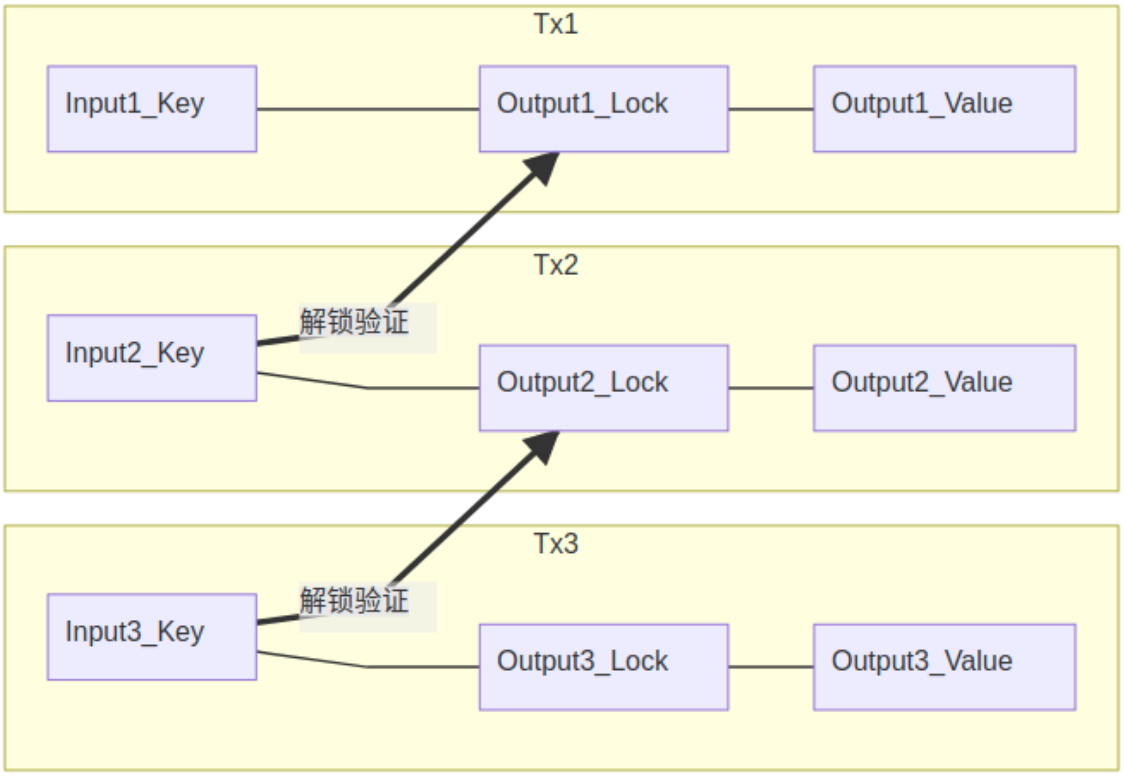
另有许多情况下，比特币脚本中的数据不是一个数值（如 1、2、3 等），而

是具有某字节长度的字符串（可用 16 进制表达），但仍然被视为一个输入数据（也可将其理解为一个很大的正整数），在脚本表达式中，每个输入数据间会用空格隔开。

至此，只要我们在遇到未知 OP_CODE 时，**查找一下其定义**，便基本能看懂比特币的脚本了。

当比特币脚本中的指令执行完毕，所得的结果中最后出现的数据（如上为最右）为非零，则此脚本验证成功（等于 OP_TRUE），此验证相对应的 UTXO (unspent transaction output) 即可以被花费，余额也将转移。因为涉及经济利益，实际中的验证脚本并非如上面的算术计算脚本一般简单。

将整个验证脚本分成两半，右边部分一般为操作算符，可称为 Lock，因为其一般由父交易提供，用于锁定 UTXO 余额，使不被轻易花费；左边的部分一般为数据，可称为钥匙 Key，因为其一般由即将上链的子交易提供，用于解锁 UTXO 余额。如果仅考虑 `1-input-1-output` 的交易，其过程可用下图表示：



P2PK 交易 与 OP_CHECKSIG 操作符

为理解下一章中的 **OP_PUSH_TX** 技术，我们还需要理解比特币中最简单常见的一段代码，其是 P2PK 交易 (pay-to-pubkey transaction) 的核心解锁验证代码：

代码块(1):

```
<Signature 签名>       <PublicKey 公钥> OP_CHECKSIG
└─ ─ ─ ┬ ─ ─ ─┘     └─ ─ ─ ─ ─ ─ ┬ ─ ─ ─ ─ ─ ─ ┘
       Key                       Lock
```

其中，`< ... >` 表达一个字符串输入数据，其意义由 `< >` 中的文字所表达。

本代码中，OP_CHECKSIG 表达一系列计算操作组合成的单一操作符，是比特币系统中自带的最重要的原生操作符之一，其首先对当前的 TX 部分输入与输出数据 <**TxData**> 做 HASH 计算（两次 SHA256 计算，与操作符 OP_HASH256 的功能相同），得到 <**Tx_HashValue**> ：

代码块(2):

```
<Tx_HashValue> = HASH256( <TxData> )
```

然后，读入左侧的两个输入数据 <**Signature 签名**> <**PublicKey 公钥**> ，检查 <**Signature 签名**> 是否是由 <**PublicKey 公钥**> 对应的私钥 <**PrivateKey 私钥**> 对 <**Tx_HashValue**> 进行签名得到的，返回真假判断。 **OP_CHECKSIG 是比特币系统中极为重要的一个操作符** 。

我们可以顺便理解一下比特币支付背后的交易生成、提交与验证原理。首先，将以上 代码块(1) 分成两段： `<Signature 签名>` 与 `<PublicKey 公钥> OP_CHECKSIG`，前一段只有数据没有操作符。然后，将后一段代码放入交易的 Outputs 中（即为一个 UTXO，等同于生成一个未开过的**脚本锁**），并为其分配比特币余额，即完成将比特币发送到 持有私钥 <**PrivateKey 私钥**> （ 其与 <**PublicKey 公钥**> 对应配对） 的人的手中。当接受人花费此余额时，即利用手里的 <**PrivateKey 私钥**> 对 <**Tx_HashValue**> 进行签名得到 <**Signature 签名**>，然后利用此签名作为**钥匙**，与后段代码（**脚本锁**）组合起来，提交给矿工网络去执行验证，矿工们通过执行 **OP_CHECKSIG** 操作符完成认证检查，即完成解锁操作。解锁之后，交易发送者一般会同时构造另一把新锁，如此交易迭代下去。

OP_PUSH_TX 技术： 一层 Token 方案的核心

OP_PUSH_TX 技术在比特币脚本合约中的地位极为重要。利用它可以在构

造比特币交易的时候，**对币的接收者未来发出的后续交易的输入与输出数据均做出限制，实现对后续的一系列子孙交易施加影响（提约束条件）**。这种能力被大多数区块链开发者（包括 BTC 开发人员）认为是不可能的，成为 ETH 链上账户模型出现（被赞赏）的缘由。

前面对 P2PK 交易的介绍中，利用 <**PrivateKey 私钥**> 对 <**Tx_HashValue**> 进行签名获得 <**Signature 签名**> 的过程，是由用户在花费比特币余额时在比特币脚本外执行（为保持 <**PrivateKey 私钥**> 的私密性）。

在 OP_PUSH_TX 技术中，类似的签名操作利用图灵完备的比特币操作符在脚本内完成，此时选择一个可暴露的另一个 <**PrivateKey 私钥 2**> 来做签名，整个签名过程完全爆露在公共的空间中（不影响 OP_PUSH_TX 技术安全性），可以用一个虚构的操作符 OP_Signature 来表示签名计算。完整的过程由以下代码表示：

代码块(3):

```
<TxData(解锁时输入)> OP_HASH256 <PrivateKey 私钥 2> OP_Signature <PublicKey 公钥 2> OP_CHECKSIG
```

这里，OP_HASH256 操作符跟 代码块(2) 中 Hash 函数具有一样的功能（两次 SHA256 计算）。在执行完 OP_HASH256 后，中间结果为：

```
<Tx_HashValue(待验证)> <PrivateKey 私钥 2> OP_Signature <PublicKey 公钥 2> OP_CHECKSIG
```

其中，用 **OP_Signature** 并不是一个比特币脚本中自带的操作符，而是被用于表示使用一系列已有的 OP_CODE 组合而成的一段脚本代码的整体，其计算操作集合为一个单一函数，读入左侧的两个数据，进行椭圆曲线签名运算，计算结果为 `<Signature 签名 2(待验证)>`，此函数库代码由 sCrypt 公司的脚本编译器内部实现。执行之后的中间结果为：

```
<Signature 签名 2(待验证)> <PublicKey 公钥 2> OP_CHECKSIG
```

这时的脚本跟前面解析过的 代码块(1) 是相同，这里最关键的注意点是，**OP_CHECKSIG** 操作符会由矿工执行，读入当前 Tx，得到真实的 <**TxData**> 数据，然后再次计算出 <**TxHashValue**>，如果 **OP_CHECKSIG** 验证通过，则

说明 <**Signature 签名 2(待验证)**> 所签署的数据 <**TxData(解锁时输入)**> 确实是跟当前 Tx 一致的真实的 <**TxData**> 数据。作为对比理解，同样的 OP_CHECKSIG 操作，在 P2PK 交易 代码块(1) 中的目的是验证 Tx 发出者是否拥有正确的<**PrivateKey 私钥**>，而在 OP_PUSH_TX 技术中的目的是验证 Tx 发出者是否在当前交易的输入中插入了正确的包含当前交易输出 <**TxData**> 数据。

因此，通过将 代码块(3) 中除第一个数据 <**TxData(解锁时输入)**> 外的其他数据与代码放入 UTXO 中，就可以要求花费该 UTXO 时，必须要输入正确匹配的 <**TxData(解锁时输入)**> 数据，才能获得验证通过。因 <**TxData(解锁时输入)**> 中包括当前交易的输出数据 <**TxOutputs**>，通过其他比特币的字符串操作符（如：部分提取 OP_SUBSTR、相等验证 OP_EQUAL 等等），即可对当前 Tx 的<**TxOutputs**>添加验证要求。

小结一下，**我们可以构造一个 Tx1，在其中的 UTXO1 中写入 OP_PUSH_TX 相关代码，要求接收者在构造新的 Tx2、花费此 UTXO1 时，输入正确的 <Tx2Data>，同时写入对其 UTXO2 部分数据的各种图灵完备的验证要求，对新生成的 UTXO2 做出限制。该限制条件仍可以被 UTXO3 继承，然后链式的持续遗传下去。**

我们可以把 代码块(3) 中的 <TxData(解锁时输入)> OP_HASH256 <PrivateKey 私钥 2> OP_Signature 代码转换为形式 <TxData(解锁时输入)> <PrivateKey 私钥 2> OP_PUSH_TX。即用封装的单一函数 OP_PUSH_TX 完成 OP_HASH256 与 OP_Signature 的作用，这个 OP_PUSH_TX 也不是比特币脚本自带的操作符，其 OP_ 前缀仅表示改技术具有单一的功能，并可已类比原生操作符的方式在 sCrypt 公司的产品中被调用，sCrypt 编译器会自动完成真实操作符代码的植入工作。

至此，我们解释了 OP_PUSH_TX 技术背后的原理，我们得到了一个非常重要的知识，比特币脚本可是实现对还没现身的未来后续交易 Tx 做出限制，其可做出限制的数据范围即为 代码块(2) 中 Hash 函数的输入数据 <TxData>，此数据被称为交易原像（Preimage），其不是完整的 Tx 数据（至少不能包含签名数据自身），但是却是当前 Tx 中最值得使用签名去保护的有关键意义的数据。

因为 Preimage 表达了 OP_PUSH_TX 技术的可操作数据空间，对认识比特币智能合约的能力边界非常重要，所以至少需要对其有一个简洁清晰的理解。

Preimage 与合约脚本的可操作数据边界

所谓 "巧妇难为无米之炊"，比特币脚本(Script)为巧妇，可输入数据（InputData）为米，即便脚本能力很强（图灵完备），其可操作的数据的信息多少与密度仍是极为重要的独立组成部分。通过本节，可以详细的了解比特币合约脚本的输入数据的边界范围。不想细看的读者可直接跳转至本节后面的示意图与总结的所有可操作数据。

将比特币下基于 UTXO 的智能合约理解为：验证一个钥匙（也可分割为多个）是否与设计好的锁匹配，此过程可表达为验证此函数等式：

公式(1)：

Lock_In_TxPre(<Keys_In_TxNow>) == True

是否成立。其中，TxNow 表示当前正在执行验证与上链的 Tx，TxPre 表示正在花费的 UTXO 所在的前一个 Tx。

一个特别有意义的合约设计基础知识是：其中 <Keys_In_TxNow> 参数的可变空间范围，它决定了智能合约的能力边界，因为脚本锁（Lock_In_TxPre）是可以被任意构造的（源于脚本的图灵完备性）。

Preimage 是 OP_PUSH_TX 技术为 <Keys_In_TxNow> 带来的新的参数空间， 其包含如下内容：

1. nVersion of the transaction (4-byte little endian)
2. hashPrevouts (32-byte hash)
3. hashSequence (32-byte hash)
4. outpoint (32-byte hash + 4-byte little endian)
5. scriptCode of the input (serialized as scripts inside CTxOuts)
6. value of the output spent by this input (8-byte little endian)
7. nSequence of the input (4-byte little endian)
8. hashOutputs (32-byte hash)
9. nLocktime of the transaction (4-byte little endian)
10. sighash type of the signature (4-byte little endian)

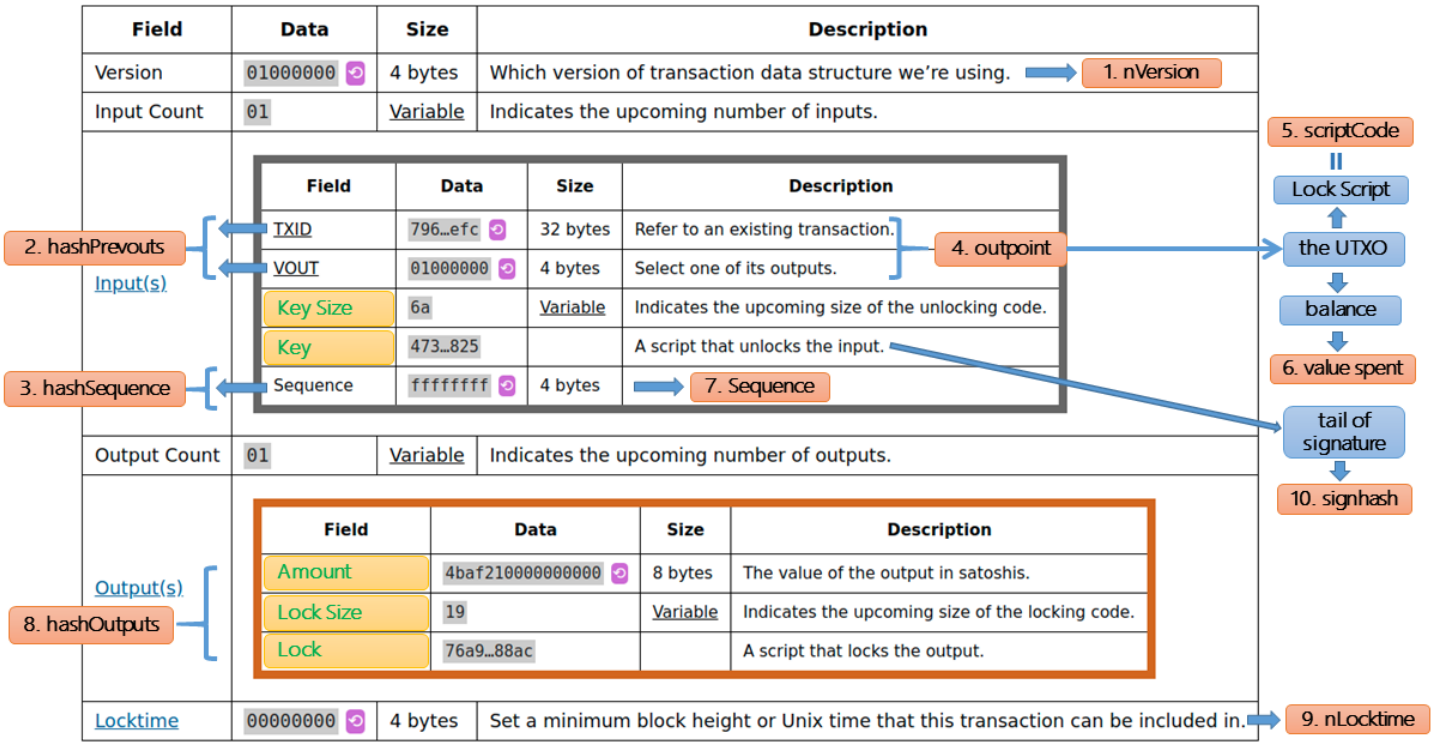
以上参数的涵义原文见于此链接，其中的表述不易理解，本文详细的解释如下：

1. nVersion 为交易的版本号（比特币为交易格式的升级预留了版本号位）。
2. hashPrevouts 一般为所有输入 UTXO 的位置指针的序列化数据的 双 SHA256 Hash 值。即为：SHA256^2(Serialize(**TxPre1 TXID** + **TxPre1_VOUT** + **TxPre2_TXID** + **TxPre2_VOUT** + ...))，其中 **TxPre1_TXID** 为花费的第一个 UTXO 所在的 Tx 的 TXID，**TxPre1_VOUT** 是一个数值，指出要花费的第一个 UTXO 在其 Tx 的 Outputs 中的排序位置（序号）。**TxPre1_TXID** 与 **TxPre1_VOUT** 共同给出了 UTXO1 的全局指针（此处感谢 aaron）。当 TxNow 花费的 UTXO 只有一个时，不存在 **TxPre2** 等。 在少数交易中，其收到 SIGHASH_ANYONECANPAY 的非标准使用而有变化。
3. hashSequence 为所有输入的 nSequence 的序列化数据的 双 SHA256 Hash 值：SHA256^2(Serialize(**TxNow_Input1_nSequence** + **TxNow_Input2_nSequence** + ...))。使用 nSequence 可实现交易在交易池中等待更新、延迟上链的功能。 在少数交易中，其收到 SIGHASH_ANYONECANPAY 的非标准使用而有变化。
4. outpoint 指当前要进行 OP_CHECKSIG 验证的 UTXO 的位置指针的序列化数据，即 Serialize(**TxPreCurrentUnlock_TXID** + **TxPreCurrentUnlock_VOUT**)。因每一个 UTXO 花费时都需要做 OP_CHECKSIG 验证，于是对应每一个 OP_CHECKSIG 操作，都会有一个依赖 **当前需解锁 UTXO** 的不同的 Preimage 。与前述 hashPrevouts 是局部与整体的关系。
5. scriptCode of the input 是当前要打开（执行）的脚本锁，即 **Lock_In_TxPreCurrentUnlock。**某些时候，其因 OP_CODESEPARATOR 的加入而有所变化。
6. value of the output spent by this input: value of the amount of bitcoin spent in this input, 当前要花费的 UTXO 的比特币余额。
7. nSequence of the input： 此项表示当前进行解锁 Unlock 验证的 Input 所对应的 nSequence，可表示为 **TxNow -> InputCurrentUnlock -> nSequence**，与前述 hashSequence 是局部与整体的关系。
8. hashOutputs 一般为当前交易的所有输出的 余额+脚本锁的字节长度+脚本锁 的序列化数据的 双 SHA256 Hash 值，即 SHA256^2(Serialize(**TxNow OUT1 Amount** + **TxNow OUT1 LockSize** + **TxNow OUT1 Lock** + **TxNow OUT2_Amount** + **TxNow_OUT2_LockSize** + **TxNow_OUT2_Lock** + ...))。 某些时候，其会因 sighash

使用了非常用值而有所变化。

9. nLocktime 设定了最小的 block height or Unix time，交易只有在此之后才能被上链。
10. sighash type of the signature： 顾名思义，其表示签名的类型，会在 OP_CHECKSIG 要求输入的签名数据中被给出。其值通常为 ALL，表示对所有输入与所有输出签名。其他取值会导致前述的某些参数的涵义有所变化，本文中不考虑此种情况。

总结一下，下图中给出了在一个 transaction data 中，其中的各部分数据与 Preimage 中的数据的对应关系：



Tx 的数据结构以及与 Preimage 数据中10个分量的对应关系

注意：Preimage 中的第 9 个分量 signhash 只表达了 Key 中紧随签名数据的附加数据，其表达签名的对象，并非签名本身，更与 Key 中的其他数据无关，从基本原理上，本文可以不考虑 signhash 取不同值时的影响。

图中使用了的简称 Lock，其对应业内专有名词： 锁定脚本 / ScriptSig / LockingScript； 图中的简称 Key 对应专有名词：解锁脚本 / ScriptPubKey / UnlockingScript。

可以看到，Preimage 中包含了当前 Tx 中除 Input_Count、Key_Size、Key（signhash 除外）、Output_Count 以外的所有数据。因为在 Preimage 中，数据已经被重新序列化，所以，Preimage 未包含的 Count 或 Size 数据不再具有实用的意义。**需要注意 :参考 公式(1) 中的 Lock_In_TxPre ,当前 Tx_Output 中的 Lock 即是构造的新的 UTXO 锁： Lock In TxNow，其将成为 Lock_In_TxPre 验证函数的输入参数 <Keys_In_TxNow> 的一部分。**

于是，公式(1) 可分解表达为

公式(1)：

`Lock_In_TxPre(< Lock_In_TxNow, TxPre_TXID, TxPre_VOUT, ... >) == True`

这样清晰的表达了前一个 UTXO Lock (Lock_In_TxPre) 可以对花费它形成的新 UTXO Lock (Lock_In_TxNow) 进行约束，一层 spvToken 方案需要这一能力。

此外 ,对任何已知 HASH 值为<HashValue>的数据 ,可以通过在 "脚本锁" 中加入 `OP_HASH256 <HashValue> OP_EQUAL` 代码段来要求在解锁时输入正确的该数据，然后即可继续对数据进行变换与验证。虽然我们在构造一个确定格式的将会持续遗传的合约时，并不知道未来交易的 TXID，但是我们可以通过 OP_PUSH_TX 技术得到当前 TxNow 的输入 UTXO 的 TXID。两者结合 ,可以要求将正确的当前交易 TxNow 的父交易 TxPre 数据输入解锁合约中，而不需要在构造祖先合约时预指定 TxPre_TXID 。如此继续,可要求将正确的爷辈交易 TxPrePre 引入当前交易的合约验证过程中。这类将当前与祖辈的 TX 数据引入到合约解锁过程中的技术，可统称为 PUSH_TX 技术。在只考虑 1-input-1-output 的情况下，整个数据与流程如下：

1. 要求输入数据 Data1
2. 执行计算 TXID 的 Hash 操作,得到 TXID1
3. 跟 Preimage 中的 TxNow_Input_TXID 作比较
4. 结果相等,则输入的 Data1=TxPre
5. 解析 TxPre 数据,得到其中的 TxPre_Input_TXID
6. 要求输入数据 Data2
7. 执行计算 TXID 的 Hash 操作,得到 TXID2

8. 比较 TXID2 与 TxPre_Input_TXID 是否相等

9. 如果相等,则输入的 Data2=TxPrePre

10. 对 TxPre 与 TxPrePre 做其他验证操作,或继续要求输入曾祖辈的交易数据

至此，本文讲述了一层 spvToken 方案所需要的完整的 PUSH_TX 技术，其包括：1）PUSH **当前 Tx** 中除 Key (UnlockingScript) 以外的其他数据到合约中；2）PUSH **祖辈 Tx** 中的完整数据，到 spvToken 合约中。这些链上数据也是所有其他类型的脚本合约的主要可操作数据（巧妇之米）。因为比特币脚本的图灵完备性质，当然可以构造任何可想到的谜语，来要求必须输入其对应的谜底数据，从而引入此谜底数据到合约解锁中。但是，**因为链上的数据的链式遗传特征，其谜底（祖辈交易数据）只有通过如上方式一层层的追溯来得到。**

最后 ,我们考虑一下 ,为何无法通过 OP_CHECKSIG 来引入当前 TX 的 Key 数据呢？因为无法生成一个包含签名自身的签名。那么，此 Key 数据是否影响很大？接下来在 spvToken 的实现原理中，将解答此疑问。

三、一层 spvToken

有了前述的基本知识，后续就非常简单了。但首先，有必要给简单支付验证（SPV）下一个明确的定义，避免不同的内涵被不同的人混淆使用，增加迷惑性。

对 SPV 的明确定义

SPV 的涵义是，交易数据的可靠性由且仅由矿工整体来保证。矿工之间以竞争加合作的方式得到的随区块延续而不断增加的整体工作量（POW）来对交易的有效性做出保证。其他人对交易的信任，即是对矿工整体的信任，任何多余的信任与验证都不是 SPV 的组成部分。实际上，一个交易一旦上链且具有足够的 POW 背书，那么其他生态的参与者不需要再对此交易做验证，甚至包括对自身利益相关的交易做验证（当然，如果只验证自己的数据，且很简单的话，也未尝不可，但此时表示你已经不相信 SPV 了）。

因此，我们引入一个与当前 Token 话题相关的论断，即：如果一个 Token 方案，其要求用户做除 SPV 要求的 MerklePath 验证以外的任何其他真伪验证

工作(不包括查收其 Token 内容)，即便此验证工作是可靠的且不需要引入更多信任方，那么也破坏了 SPV 的原则，其也是不支持 SPV 的。其要么使得后续的接收人需要信任所有之前的接收人的验证，要么后面的用户需要将前面的用户的所有验证重做一遍，要么就需要引入对其他第三方的信任。

需要特别注意的是，即便对区块链原生的比特币的 SPV 验证，也要基于对矿工们历史验证结果的累加信任，并非只需要信任当前的矿工群体诚实性，任何历史的矿工群体的非诚实作为，也会破坏当前的 SPV 验证的可靠性。所以，相信当前矿工及历史矿工的可靠性是 SPV 验证的必要部分，也是绝大部分用户操作的依赖前提，这是系统足够安全并具备大规模扩容两个能力的平衡点。

spvToken 设计原理

当前，我们仅考虑 `1-input-1-output` 的 Non-Fungible spvToken (NFT)。

回顾前面介绍，我们可以要求 PUSH **当前 Tx** 中除 **Key(UnlockingScript)** 以外的其他数据，以及任何一个 **祖辈 Tx** 中的所有交易数据到 spvToken 合约解锁过程中，并将此称为 **PUSH_TX** 技术。基于此，可以实现 spvToken 的两个必要充分条件：

1. 当前交易 TxNow 输出合约，除 "地址 Lock" 中表达接收者地址的数据发生变化外，其中的其他 Locks 必须与父交易 TxPre 的输出合约 Locks 保持一致。
2. 当前合约交易的根交易（指发行 Token 的交易）必须来自发行人地址，或者必须来自确定的 UTXO 交易（此条件可以用于约束合约的总量）。

第一个条件很容易达到，因为我们能 PUSH **当前 Tx** 中除 Key (UnlockingScript) 以外的其他数据，所以当然可以对其中的输出合约做出限制。

要达到第二个条件，是制作 spvToken 的关键部分，因为我们不能无限层级的向上检验祖辈交易。可以很自然的想到，我们 **可以通过迭代的方式来实现它，即 spvToken 合约在执行时只需要检验上一个交易（父辈交易）的诚实性即可。这样，当前交易验证了父交易的诚实性，父交易也已经验证了爷交易的诚实性，如此直达根交易，就会确保所有的祖交易都是诚实的。于是，只基**

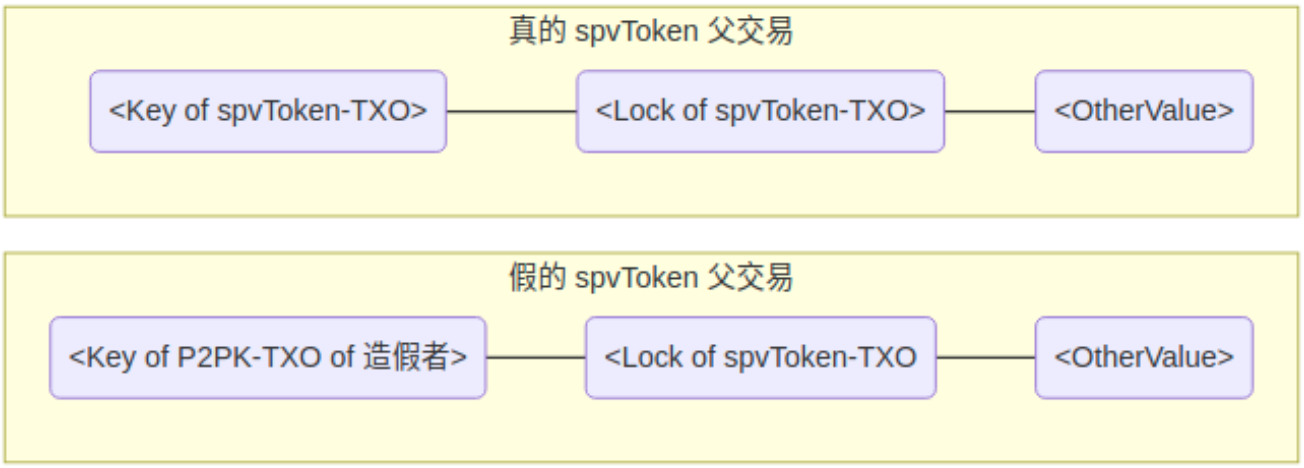
于矿工群体对合约的验证工作，即可实现 spvToken，即可避免任何中间环节的不安全因素。

检验父辈合约交易 TxPre 的诚实性，其包括检验其输入 TxPre_TXI 与输出 TxPre_TXO 两个部分的诚实性。

为何只检查父交易的输出 TXO 不行呢？

原理上，诚实性检查，它不只是形式检查，其目的是：**必须完整的保证父交易是由诚实的人，通过诚实的方式，生成了有效的输出。** 一旦交易链得到遗传并延长，后续不会再次重复此工作，所以需要非常小心的确保检查的完整性：验证输出是检查 Token 内容的有效性，验证输入是检查来源的诚实性。

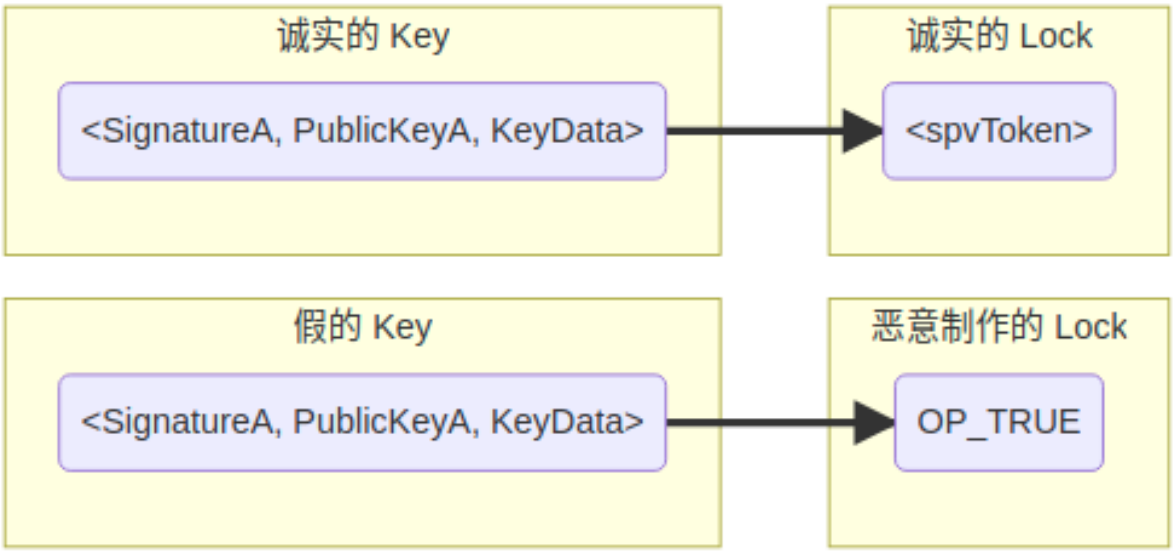
技术上，因为即便输出的 TXO 形式与目标 spvToken 相同，其输入可能可以来源于非发行人的 P2PK 交易，如下图所示：



其中，不同的 Key 解锁不同的前序 UTXO，但都可以生成同样形式的 spvToken-TXO。

那么是否只检查父交易中包含的输入数据（如 Key）即可？

这依然是不足的，因为，即使输入的 TxPre_Key 的内容形式一切如常，其可以是通过构造假的前置交易的 Lock 来实现。下图中使用 OP_True 作为 Lock 可以使得任何 Key 均可以正常解锁（其他的可行方式也是数不胜数的）：



因此，我们必须通过检查 爷辈交易 TxPrePre 的 Lock， 来确保父辈交易 TxPre 中的 Key 是诚实的。这里，我们需要有在心中留下**一个系统的断言（亦或设计方向）：1）任何有意义的对 Key 的检查，都应通过检查前置交易的 Lock 来实现；2）一个有意义的 Lock 一旦被制造时，其 Key 应被约束，在 PUSH_TX 技术中几乎不会（不应）有检查 Key 的必要；3）Key 只在矿工将 TX 上链时（生成 SPV 签证）是必要的组成部分，在其他大多数情况下均是（可以是）一个不必要的负担。** 这段回答了我们在介绍 spvToken 前提出的问题：Key 数据在上链时刻以外，并不重要。

为避免 "鸡生蛋蛋生鸡" 失去源头，spvToken 的合约 Lock 也需要对某些特殊的 TxPrePre 放行。

我们可以不要求 TxPrePre 的 TXO 必须是 spvToken 形式的 Lock，其也可以具有 P2PK 交易形式的 Lock。但是，**应做其他约束：1）其 Lock 中的地址必须是发行人控制的地址，此地址将被遗传到任何子 spvToken 交易，也自然的成为发行人的标记；2）或者 TxPrePre 必须是某固定 TXID， 任何非继承于此 TXID 交易的 spvToken 均是无效的，真正基于一层实现了历史中的染色币概念，此时，spvToken 也不再被发行人控制，严格禁止了任何增发操作。**

至此，我们已经在原理上完成了构造 spvToken 的思维实验，仅考虑 1-input-1-output 交易，其特别适合 NFT 类 Token，可将技术步骤总结如下：

1. 要求 PUSH TxNow_Preimage，并用脚本检查 TxNow_Lock 是否符合 spvToken 标准且参数正确。
2. 要求 PUSH TxPre，并检查 TxPre_Lock 是否符合 spvToken 标准且参数正确。
3. 要求 PUSH TxPrePre，并检查:
   1) 其输出是否对应一个 spvToken Lock，是则验证通过，否则继续
   2) 其输出是否是一个标准比特币交易（如 P2PK、P2PKH），且地址是 **合约内置地址** (如是则证明该 spvToken 确实是有内置地址指明的发行人发行的)。
   3) 判断此 spvToken 是否是不可增发 Token，如是，则验证 TXID(TxPrePre) =?= **合约内置 TXID** ，如相等则验证通过。

其中，每个一个检查失败，都意味着 Token 上链验证的失败，而只要脚本验证通过，则完成上链。一旦上链，则任何人可以进行 SPV 验证，不再需要执行脚本验证工作，所需要的只是将收到的 spvToken 交易中的 **合约内置地址** 与 **合约内置 TXID** 解析出来，然后就知道收到了**哪一种合约**以及**是否是不可增发的合约**。

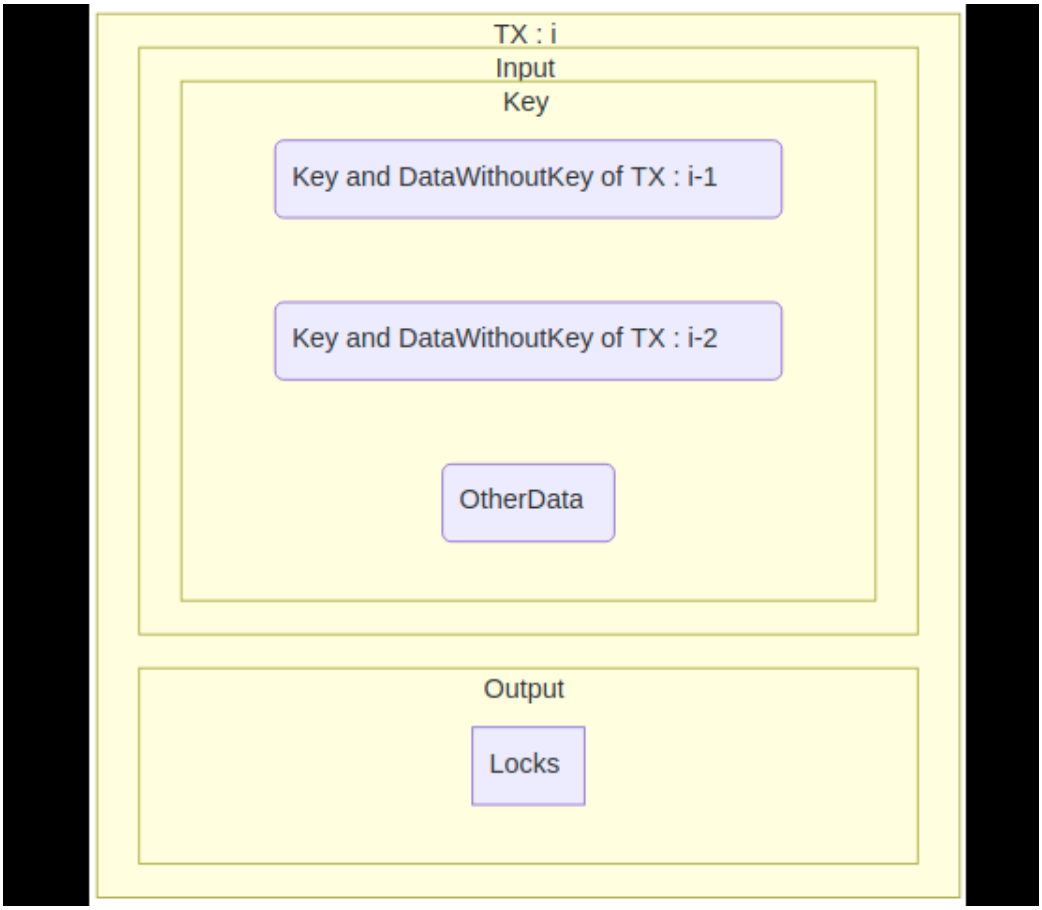
以上构建过程并非被充分优化，但是更加方便的表述了 spvToken 的基本原理。

小思考：为何对原生代币不需要检查一个交易的输入的来源？而只需要交易输出的形式即可？

spvToken 的数据膨胀问题

以上构建的一层合约表现出了强大的功能，但是很不幸，其是无法实用的。

随着 spvToken 交易链的延长，每一个祖辈交易的完整数据，都要被 PUSH 进新的交易的 Input_Key 部分，以满足合约检查中 PUSH_TX 技术的需要，如下图所示：



因此，交易的体积会越大越大，直到手续费昂贵的无法承受。而且，用户在接受的 spvToken 时，也同样面临带宽与存储困难。

四、比特币改进提议

最简改进提议

我们回顾文章开头的最简提议，讨论它如何解决 spvToken 的交易膨胀问题。

观察 spvToken 交易体积暴涨的原因，其重复数据仅在 Key 中继承并遗传。其根源在于，当前的 TXID 的计算是直接对交易的所有输入输出数据做二次 SHA256 HASH 运算得到:

```
TXID = HASH( HASH( TX-Version:1,          Key, OtherTxData )) .
```

将 Key 换成 UnlockingScript，即为文章开头的表示，两者含义相同，前者简写且方便记忆。

因此，当基于 TXID 在合约中引入其 TX 中的部分关键数据时，必须先将完整的交易数据代入，其称为 Key 的一部分，也包含历史交易中的 Key 数据。

我们已在前文中讲过：

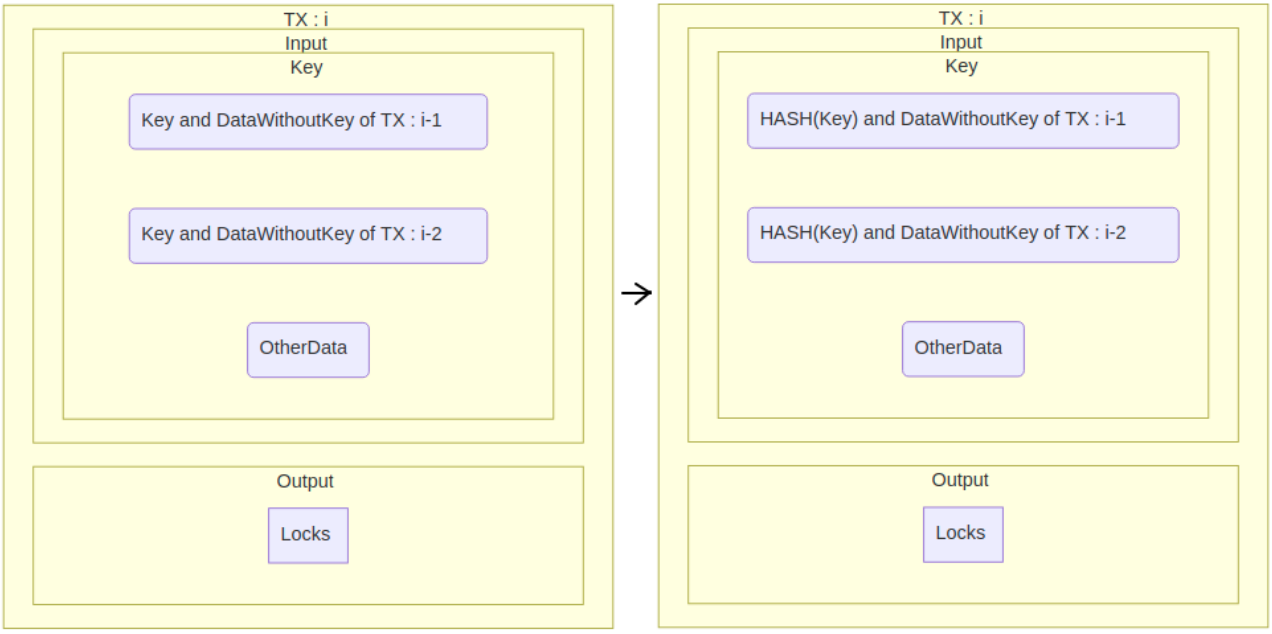
"Key 只在矿工将 TX 上链（生成 SPV 签证）时是必要的组成部分，在其他大多数情况下均是（可以是）一个不必要的负担"

因此，我们需要将 Key 隐藏在历史中，不是抛弃它，不是像隔离见证(SegWit)一般将其排除在区块之外，而是像 Merkle 树一样将其 HASH 化，并放入区块的更远端的数据叶子上，即变为：

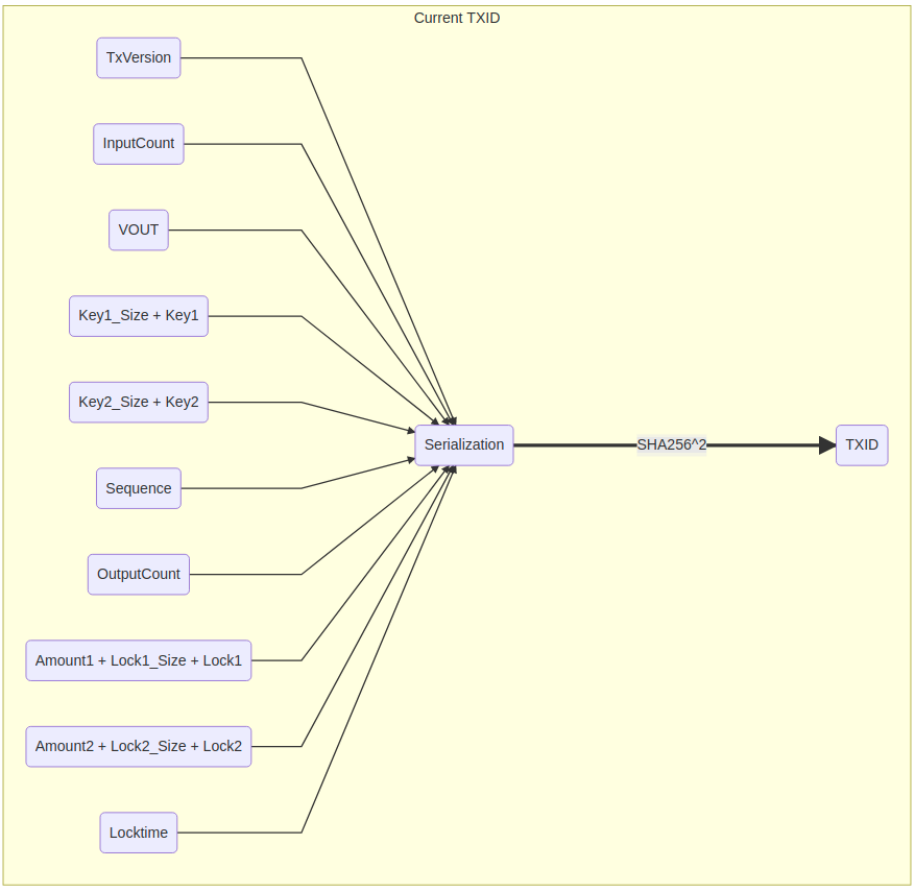
```
TXID = HASH( HASH( TX-Version:2,   HASH(Key), OtherTxData ))
```

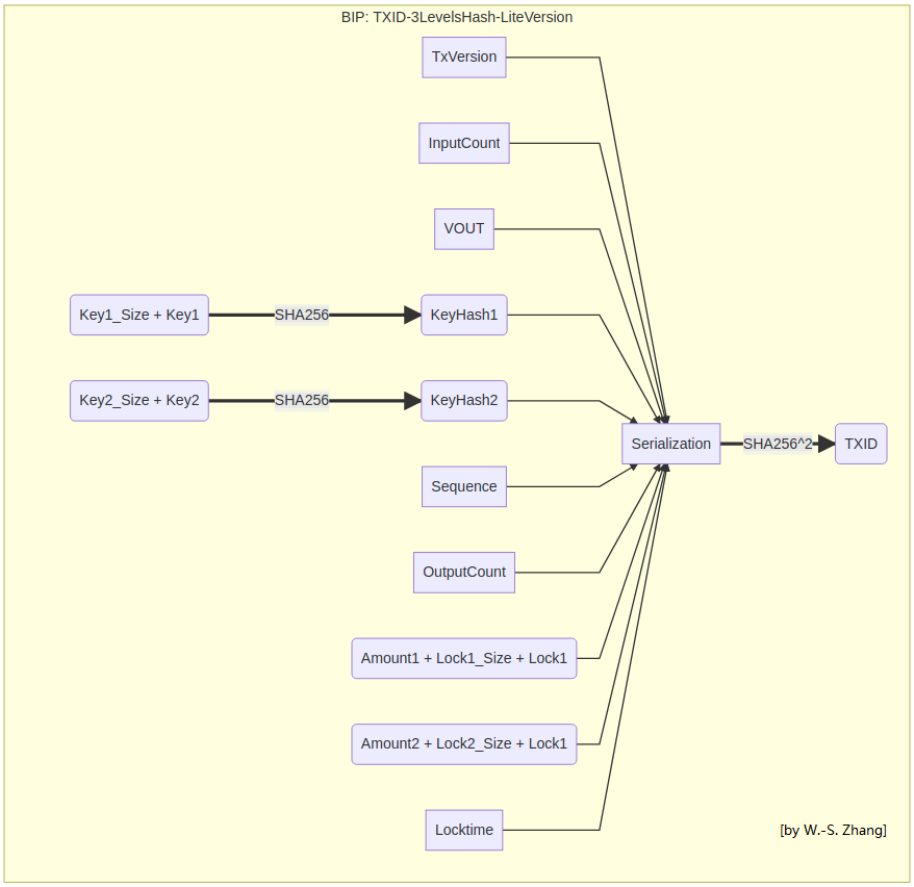
做此改进之后，spvToken 交易中的 TxNow_Key 中不再包含 TxPre_Key 与 TxPrePre_Key，此二 Key 将被具有不变大小的 HASH(Key) 代替。

图示如下：



为更准确的解释提议，下图中使用了一个 2-inputs-2-outputs 的典型交易展示本提议与现有方案的区别。其中，每一个 Key 连同 Key_Size 一同被 Hash 化，再占用原位置与其他交易数据合在一起，计算 TXID。

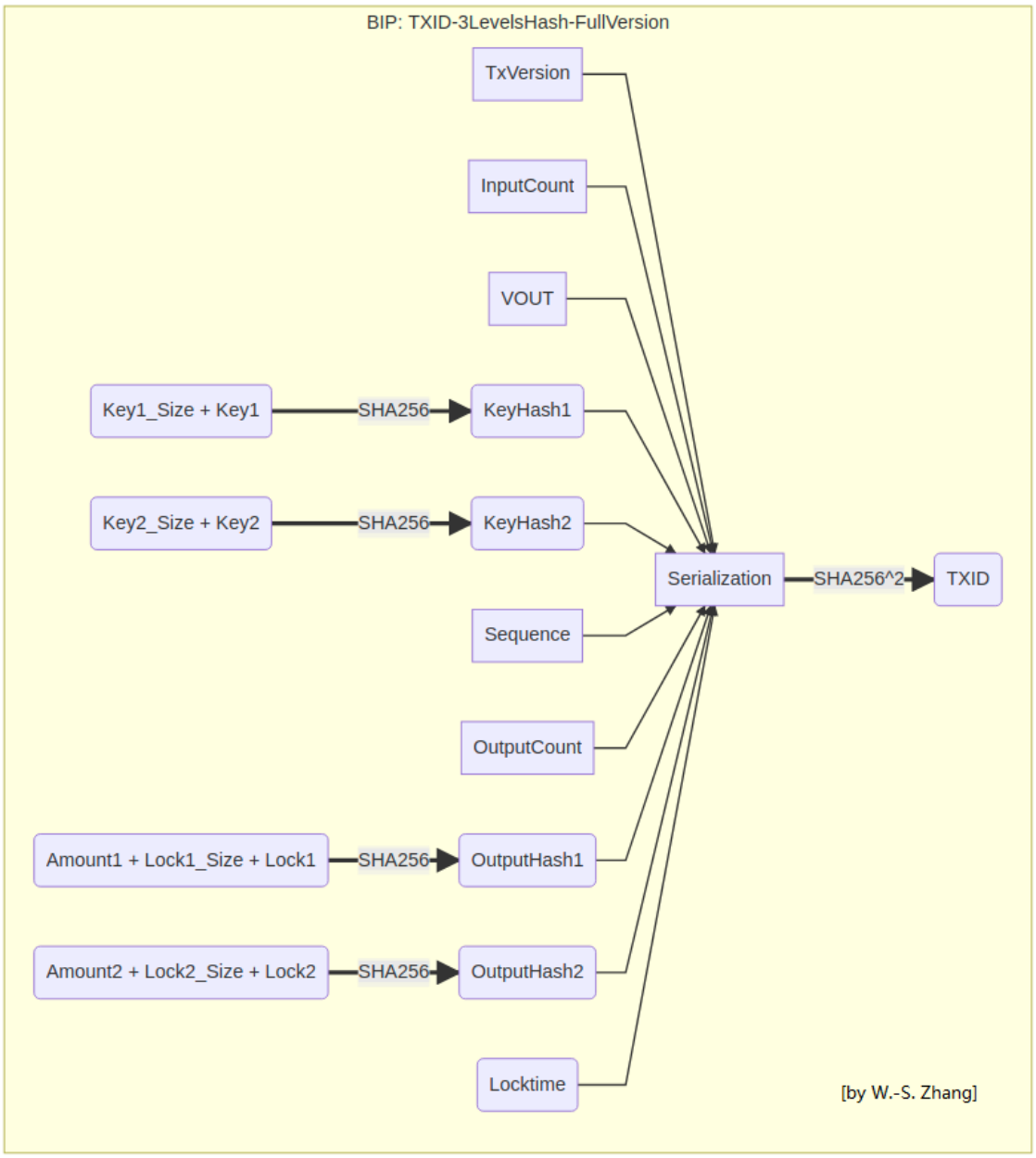

Current TXID
TxVersion
InputCount
VOUT
Key1_Size + Key1
Key2_Size + Key2
Sequence
OutputCount
Amount1 + Lock1_Size + Lock1
Amount2 + Lock2_Size + Lock2
Locktime
Serialization
SHA256^2
TXID


BIP: TXID-3LevelsHash-LiteVersion
TxVersion
InputCount
VOUT
Key1_Size + Key1
SHA256
KeyHash1
Key2_Size + Key2
SHA256
KeyHash2
Sequence
OutputCount
Amount1 + Lock1_Size + Lock1
Amount2 + Lock2_Size + Lock1
Locktime
Serialization
SHA256^2
TXID
[by W.-S. Zhang]

完整改进提议

在考虑 2-inputs-2-outputs 交易，乃至具有更多输出数量的交易下的智能合约的时候，经常出现一种情况：智能合约需要检查的 UTXO 只是其所在 TX 中的其中一个。而且，如果在未来的交易中，用户希望将更巨大的数据的放置在 Outputs 中，那么对不同的 Outputs 做独立不合并的 PUSH_TX，就会比较必要与高效。

因此，我们需要做更多一点改进，将不同的 Outputs 也进行独立的 Hash，最后占用原位置，与其他数据一起计算出 TXID。此完整改进建议可图示如下：



需要特别注意：在 FullVersion 的改进提议中，多出的一类 Hash 计算是对 Amount + Lock_Size + Lock 的 Output 整体进行。假如类似 Key 一般，只对 Lock_Size + Lock 进行 Hash 化，那么矿工可以在不给出 Lock 的情况下，说服其他矿工，其输出余额 Amout 是符合交易费率的，且其 TXID 的计算是可以通过 LockHash 验证的 。此时该交易可以成功上链，其表现如同 P2SH，这将使交易的透明度降低，对合法交易的审计造成了不必要的难题，且对非法交易提供了隐匿保护功能。因此，我们需要将 TXO 的 Amount 也加入到需 Hash 化的数据中，使用户与矿工在展示交易有效性的时候，必须提供包括 Lock 在内的 Output 的所有数据。

改完全版的改进提议，将可以实现极限的数据资源（存储与带宽）利用率。因为除 KeyHash 与 OutputHash 以外的其他 "Serialization 前" 数据均非常小，用户将不仅只需要存储自身相关的交易，还只需要存储相关交易中的相关部分，同样不失可验证性。**使节点具备接近极限的数据资源（存储与带宽）利用率、必不可少的数据资源回收（数据裁剪）能力。**

五、此改进简单与优雅吗？

此改进足够简单：

a. 因为 TXID 计算的不可逆性，TXID 的用途几乎仅在于索引 TX 数据，其最核心的需求是，不同的交易需要产生不同的 TXID。所以 TXID 的计算具有低耦合高独立性 ,只需要在所有需要计算或验证 TXID 的位置使用新的计算逻辑即可，其不会影响任何其他部分的程序逻辑，不影响矿工使用交易数据全文（包括 Key 与 Lock）做脚本验证，不影响 SPV（简单支付验证）等等。

b. 配合升级的 TX Version，来标记新老版本的交易需使用不同的 TXID 计算函数，新老交易可以同时共存，不会产生冲突。

c. 生态应用方也不需要改动任何其他验证单元，也只需要增加一个新版本的 TXID 计算函数即可。

此改进足够优雅：

d. 此改动是延续了 Merkle Path 的数据可裁剪设计，将其应用到 TX 内部，形成完整的数据裁剪需求。特别是 Key 数据一经矿工群体验证上链，其 SPV 将具有足够的安全性，保证 Key 数据即便超级大，也可以安全的移除。

e. 此改动是大区块发展方向的必要组成部分，大区块中必然鼓励大的 TX，大的脚本执行能力。当脚本程序越来越大，其所需求的存储空间（图灵机中的纸带）也会越来越大。如同，现代计算机程序必须做内存垃圾回收，来维持图灵机的实用性。本改进也是在纸带有限的客观条件下，实现对纸带的充分利用，使得比特币脚本的图灵能力得到最大程度的发挥，获得必不可少的资源回收（数据裁剪）能力。

f. 此改进增强或补足了交易数据的遗传与进化能力，为比特币系统打开了类似生物的无限进化可能。类比生物，当二倍体细胞经过减数分裂产生单倍体的配子（精子与卵子）时，一部分染色体被分配到极体中，然后极体通常被降解。设想一下，如果没有不丢弃一些染色体，生物将无法进化，变异与垃圾清理都是必要的能力。对任何需要自我进化的比特币交易，其变异能力由图灵完备的计算机指令来实现，而垃圾清理能力则由本提议来实现，必须在保留（或激活）祖先有用数据的同时，清理（或归档）其中的陈旧无用部分。

此改进的共识可行性：

历史上已发生过变更 TXID 计算方式的改动，本改动应该不会动摇系统的稳定性。

历史上，发生过两次不同交易具有相同 TXID 的情况，block 91722 与 block 91880 的 coinbase 交易 TXID 是相同的，block 91812 与 block 91842 的 coinbase 交易 TXID 也是相同的，反应了 TXID 计算中的不成熟考虑。因此，也相应做过一次针对 TXID 计算的改进(**BIP34**)：

> *BIP 34 required coinbase transactions to include the height of the block the were mining in to their transaction data, so that coinbase transactions could be different.*

在生态还未完全成熟之前（生态的成熟或许也依赖此改进），此改进的 收益与 成本比很大，如果没有严重的漏洞，在充分被节点开发方、社区、矿工群体认识与理解之后，其具有较大的可行性。

六、一劳永逸，真正实现 Set In Stone

总结全文，我再次呼吁全体相关人严肃的考虑本项提议，在没有发现重大漏洞的情况下，推进该提议的实现，使比特币系统中的交易链具有可持续的遗传与进化能力。这些交易链，利用时间（历史与未来）来延伸比特币图灵完备脚本系统的操作空间（无限长图灵纸带）,持续裁剪/凝聚/进化出实用(有用)的数据空间，实现功能上的无限进化，真正具备协议固定的技术能力。

符号与名词解释

本节解释某些必要的专业词汇。此外，为了使表达更加简洁、容易记忆，本文也用了一些自定义的名词简写与命名规则。

< ... >： 表达一个字符串输入数据，其意义由中间的文字所描述。

Tx / TX / tx ： 涵义相同，都表达交易 Transaction 的简称。可跳转至 TX 数据结构查询其包含的数据内容。

LockingScript ： 为锁定脚本，通常也被称为 ScriptPubKey，其中出现 PubKey 是因为：此脚本中通常包含公钥 PublicKey 信息 （用于指定开锁私钥）。[可跳转至 TX 数据结构查询 ScriptPubKey 的记录方法]

UnlockingScript：表示解锁脚本,通常也被称为 ScriptSig,其中出现 Sig 是因为：在最常见的交易中其包含一些签名数据 Signature，即用私钥 PrivateKey 对交易内容做的签名，只有签名正确，交易才能通过矿工验证。[可跳转至 TX 数据结构查询 ScriptSig 的记录方法]

Lock ： 表达 "脚本锁"， 本文中用作锁定脚本 LockingScript 的简称。

Key ： 表达 "脚本锁" 对应的 "钥匙"，是解锁脚本 UnlockingScript 的简称。

Keys ： 表达"解锁钥匙"中的不同组份，可分别对应多个解锁条件的其中之一，表示一个锁需要多个钥匙的情况。

Unlock ： 指用 Key（Keys）解锁 Lock（LockingScript）的过程。

PubKey ： 公钥 PublicKey 的简写，可以用于验证某个签名确实是此 PubKey 对应的私钥（PrivateKey）所签署。

Inputs / Outputs：交易的输入与输出数据，可跳转至 TX 数据结构查询交易的数据结构。

TXO： TX Output，交易输出，其中包括锁定脚本（Lock）与对应的余额（Balance）。

UTXO： Unspent TX Output，当前具有余额，还未花费的交易输出。当在新的 TX 中花费它时，需要在新的 TX 的 inputs 中用指针指定其所在位置。

Outpoint： UTXO 的指针（等于 TXID 加 VOUT）。TXID 指明 UTXO 所在 TX， VOUT 指明其在此 TX 中的 Output 的排序位置（排序计数从 0 开始）。

TXI： TX Iutput，交易输入。其包括 TXID、VOUT、Key_Size、Key、Sequence。

TxNow： 正在执行验证与上链的当前 TX。

TxPre： TxNow 中使用的 UTXO 所在的 TX。每一个输入 UTXO 都对应一个 TxPre，分别记为 TxPre1、TxPre2 … 。

TxPreCurrentUnlock：当前正在执行 Unlock 验证的 UTXO 所在的 TxPre: TxNow -> Current Unlocking UTXO -> TxPre， 仅指代一个 Tx。

TxNow_Input1： 当前 TxNow 的 Inputs 中的第一个 Input 数据：TxNow -> Input1，其包含以下数据： 1） TxPre1 的 TXID 与 UTXO 所在的 VOUT (两者组合起来即为 Outpoint)； 2） 解锁脚本 Key； 3） 前一个数据（Key）的数据大小；4）控制交易是否及时上链的一个整数值 nSequence。

参考文档

(1) https://scryptdoc.readthedocs.io/en/latest/contracts.html#contract-op-push-tx

(2) https://github.com/sCrypt-Inc/boilerplate/blob/dev/contracts/spvToken.scrypt

(3) https://xiaohuiliu.medium.com/peer-to-peer-tokens-6508986d9593

(4) https://blog.csdn.net/weixin_47461167/article/details/108409290

(5) https://wiki.bsv.info/scrypt

(6) https://learnmeabitcoin.com/technical/transaction-data

(7) https://blog.csdn.net/weixin_47461167/article/details/108442784