感应合约 (Sensible Contract)

一种基于特征识别的可溯源及协作的合约

Abstract

典型的比特币脚本，允许我们对单次交易中，“单笔资金是否可被使用” 进行一定程度的控制。通过运用 PUSH_TX 技术，有状态的合约允许我们将这种控制能力向后延伸，从而保证单次交易及其后继交易之间的延续性。基于已有技术，我们提出 **感应合约** 的技术方案和对应实现，通过对比不同合约的签名特征，使得单个比特币脚本得以感知并区分不同的合约类型，从而使其获得了 **鉴伪及溯源** 的能力。在此基础上，感应合约提供了同一笔交易内，多个输入上的不同合约相互识别并协作的能力。更进一步，通过对不同合约特征的感知，识别，区分，响应与协作，任何一个单一合约，作为一个最小的独立行动单元，可以获得与整个区块链上所有合约 **建立相关性** 的能力。需要特别指出的是，这种能力是当前比特币脚本所固有的能力。从我们提供的用例可以看出，对这种能力的运用，在现有的比特币脚本操作码集合内即可完成。

Introduction

本文中，我们首先简要介绍比特币脚本的历史，及一些关键的变动；然后，通过考察一般情况下两个典型的需求，我们简要分析了对比特币脚本能力的运用在当前存在怎样的局限性；接着，我们引入感应合约的概念，并尝试赋予比特币脚本在新的维度上获取和处理信息的能力（即所谓的感知及响应能力）。为了验证该方案的有效性，在软件实现层面，我们同时提供了两种参考实现作为对比；在应用层面，我们将呈现若干实际的用例，并籍此说明如何运用此技术去解决来自真实世界的需求。

A brief history about "Smart Contract" (智能合约简史)

存在于比特币系统中的脚本机制，允许我们以可编程的方式，对比特币的转移构造一定的条件。这种可编程的能力，在由 A 到 B 这种常见的普通转账之外，为我们提供了 “更高级的灵活性”。然而，由于历史原因，一方面，一部分操作码（基于“安全”的原因）被停止使用；另一方面，网络承载能力（即区块尺寸）被当时的开发团队人为限制在很小的规模（以保持其所谓“去中心化”的特性）。这些因素，客观上导致了比特币脚本的能力被大幅削弱，一些本来可以在比特币网络上得到实验和应用的高级技术变得无法实现。这些存在于当时的限制，直接促使了以太坊（作为一个“改良版的”可编程区块链）的出现。通过提供（当时比特币脚本所无法提供的）某种程度上的“完备性”，以太坊成为了第一个基于比特币区块链技术的“智能合约”平台。

Bitcoin script development in BSV (BSV上的脚本开发)

在 2020 年 2 月，Bitcoin SV 的 Genesis 版本升级后，上述限制中的大部分被从 BSV 的节点软件中移除，一些早期的比特币脚本操作码也得以恢复。这使得脚本系统脱离了不必要的约束，重新变得灵活，可被用于更多高级构件的开发。在这样的条件下，一套高级语言的开发环境 sCrypt 被构建出来，使得开发人员得以使用高级语言来编写合约，并通过 sCrypt 编译为比特币脚本，在比特币交易处理的过程中，这些脚本和典型的比特币交易一样被矿工验证和执行。sCrypt 的工作，使得比特币脚本的潜力得到了挖掘。

一方面，相较于类似汇编的操作码集合，使用类似 Solidity 这样的高级语言来开发，能极大地提升开发效率；另一方面，比特币的 utxo 模型使得单个合约的执行被尽可能地局部化，与以太坊上合约需要串行地修改全局状态相比，位于不同 utxo 上的比特币合约不关心相互之间的执行顺序，可被充分地并行化，这极大地提高了整个系统的扩展性。

OP_PUSH_TX and Stateful Contract (OP_PUSH_TX和有状态的合约)

2016/2017 年，nChain 的研究员 Ying Chan 提出了一种将比特币交易的相关信息传递到后继交易的技术方案 `OP_PUSH_TX`。这个方案将当前交易的关键信息（即所谓的 PreImage）压入比特币脚本的执行堆栈，利用一个临时私钥为其生成签名，并巧妙地利用了 OP_CHECKSIG 可以验证该签名是否属于当前交易的特性，来确保入栈数据的真实性，使得这些数据在脚本中得以被获取和利用。

在 Genesis 版本升级后，该技术由 sCrypt 团队在其智能合约解决方案中实现，并做了进一步的优化和拓展。通过使用一个独立的数据段来存储状态，sCrypt 团队进一步实现了一系列合约交易中比特币脚本在状态上的延续性。这种有状态的合约 (Stateful Contract) 使得开发者可以很方便地在比特币的前后交易之间维持有效的状态，并按照指定的逻辑在需要时对状态进行更新，大大拓展了比特币脚本的应用范围。

Contract Limitations in Real-World (真实世界中的合约限制)

然而，考虑到真实世界项目中的实际需求，OP_PUSH_TX 和有状态的合约仍然留下了不少悬而未决的挑战。其中，最重要的问题有两个，分别是 **合约溯源问题** 和 **合约协作问题。**这两个问题都涉及到，如何在单一合约上下文内，感知与区分那些潜在的与其逻辑相关的合约，并相应地做出有效的响应。这些挑战能否解决，直接关系到比特币脚本对已有的链上数据的利用和处理能力，决定了比特币合约在工程中的实用性和扩展性。

The Method

接下来我们针对溯源问题和协作问题做出具体的分析，从而得以更清晰地了解到当前比特币脚本的能力边界。在此基础上，我们介绍如何通过对关键特征的识别和响应来解决这些问题，为进一步拓展比特币脚本的运用打下基础。

Problem #1. Contract Backward Traceability Problem (合约溯源问题)

“状态可延续”不足以保证“状态本身的合法性”。

举例而言，仅仅是“状态可延续”，并不足以维系一个有效的基于 utxo 和比特币脚本的 token 系统，也就是我们常说的层一 token 方案。要想使得这样一个系统在真实世界中能够不被恶意地破坏，不仅需要在 utxo 上映射并管理实际的 token 持有情况，更为关键的是，任何一个存有 token 的 utxo，系统都需要能做到快速判断其真伪 (是否为原发行人发行 token 的一部分)。

回到合约的角度，这个需求的本质是，在单一合约的执行上下文内，区分形式上一致的其他合约的来源真实性，也就是确认不同合约是否同源（来自于同一个单一的来源）。对于这一类鉴定合约真伪的问题，我们称其为合约的溯源问题。

为什么无法借助 PreImage 内提供的已有信息来做溯源？在 PreImage 的第 4. 5. 6. 项中，我们可以获取到当前输入的 outpoint，锁定脚本，金额，而这些信息是可被伪造的。一个不可信的第三方，有能力伪造一份存有满足需求的锁定脚本和金额的 outpoint。仅凭这些信息，我们无法在比特币脚本的运行时，

区分相同的锁定脚本是否来自于一个单一来源 (拥有共同的"根交易")，从而也就无法断定当前的输入是否为一个真实有效的输入。

1. nVersion of the transaction (4-byte little endian)
2. hashPrevouts (32-byte hash)
3. hashSequence (32-byte hash)
4. **outpoint** (32-byte **preTxID** + 4-byte little endian **index**)
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)

一种可能的做法是，在解锁脚本中，继续放入该 outpoint 对应的前序交易的完整内容。这样，可以通过计算该内容的 hash 并验证其与 txid 是否一致，来确保其真实性。然而，这样做会导致后续交易的体积像“滚雪球”一样逐渐膨胀，是一个不可持续的方案。

另一种可能的做法是，把待追踪的交易的相关信息保存到独立的外部服务中去，当新的交易发生时，通过该外部服务来索引，识别并辅助溯源工作。这个方案使得合约需要依赖 **链外的数据和行为的辅助** 来完成自身的功能。由于关键的步骤未发生在链上，这个混合系统削弱了合约的完整性，也无法“用比特币来解决比特币的问题”。

Problem #2. Contract Coordination Problem (合约协作问题)

除了合约溯源问题之外，一些时候，我们需要合约之间彼此识别并协调，同步完成某些操作。具体地说，在完成一笔多个输入的交易时，合约需要对自己正在处理的输入之外的其他输入进行识别和做有针对性的响应。在已有机制下，这个需求同样难以实现，我们称其为“**合约协作问题**”。

比如，实现 “两个同类型 token 在单笔交易内交换” 这个需求，需要合约能够确认其它的输入是同类合约。然而，与溯源问题类似，现有的 PreImage 内缺乏足够的信息来验证这一点。具体地说，PreImage 包括当前交易的所有输入 outpoint 的 hash（2. hashPrevouts），只能获取当前交易的输入由哪些 outpoint 组成，但无法获知并限制这些 outpoint 的锁定脚本内容或与之对应的 BSV 数量。

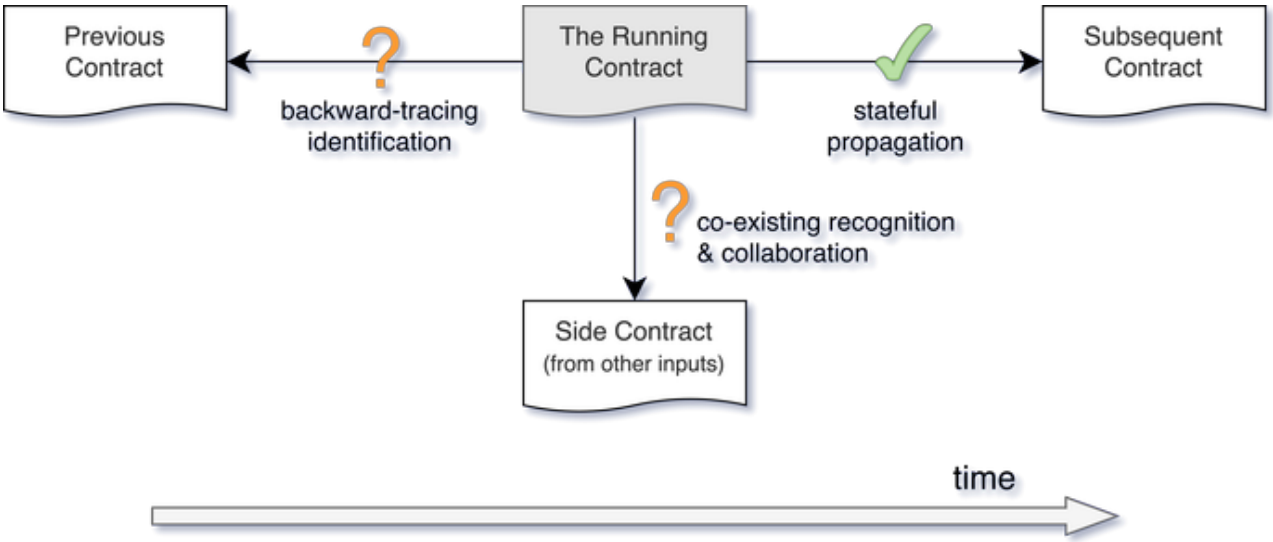
一种可能的做法是，在解锁脚本中继续放入这些 outpoint 所对应的前序交易的完整内容。这样，与上面一样，在确保其真实性的前提下，可以获知 outpoint 对应的锁定脚本。同样的，这样做会导致后续交易的体积逐渐膨胀，同样是一个不可持续的方案。

同样的，通过外部服务来索引并辅助，是一个依赖链外的数据和行为的链下混合技术方案。

The capability boundary of current script usage (当前合约的能力边界)

如果我们以正在执行的脚本合约为观察目标，可以将脚本的控制范围划分为3个层级：

1. 我要往哪里去？（如何对后继合约施加约束）
2. 我是谁/有谁同我一起？（如何获知当前合约与哪些合约共存）
3. 我从哪里来？（如何识别当前合约的出处）



通过对上述两个问题的分析，我们可以认识到当前合约的能力边界：有状态的合约解决了第一个问题，也就是"我要往哪里去"，提供了随着时间的推移向后延伸控制的能力，但没有解决后两个问题，缺乏"**沿着时间线向前逆向识别与回溯**"的能力，也缺乏"**在当下对同一上下文内其他合约的感知与响应**"的能力。

The traceability of locking script (锁定脚本的可追踪性)

那么如何解决这两个问题？

我们先暂时回到上述"会导致交易体积膨胀"的方案中去。我们试图把完整的 TX 在解锁时压入堆栈，目的是为了在可验证其真实性的前提下，获取其内部的信息。然而，完整的 TX 内，大量的信息（如不断膨胀的解锁脚本）其实对于我们的目的（识别与鉴伪）而言是无用的。

事实上，**完全相同的锁定脚本**（scriptCode，即上面 PreImage 中的第5项），就足以保证合约行为的一致性。锁定脚本，才是真正的需要被鉴定的可被用于追踪的 "**合约指纹**"。

带着可被鉴定的锁定脚本，我们重新审视上面的三个疑问：

1. 我要往哪里去？
   - 获得并校验当前交易内的所有输出的锁定脚本，即可判断并控制输出合约类型（当前的 PreImage 中已有）
2. 我是谁/有谁同我一起？
   - 获得并校验当前交易内的所有输入的锁定脚本，即可判断并控制与自己相邻的其他输入合约的类型
3. 我从哪里来？
   - 获得并校验当前输入所在的前序交易 (PreTX) 的所有输入的锁定脚本，以及所有输出锁定脚本，即可判断并控制当前输入的溯源

概括地说，在溯源和协作等场合下，作为合约具体内容的承载物，锁定脚本可被用于确认合约的有效性。

The anti-forgery unspent-quad signature (签名防伪)

这时，很自然的，会产生这样的疑问：如果只考虑锁定脚本的内容是否一致，那如何做到防伪造呢?

毕竟，一旦知道了合约内容，任何人都可以用自己的私钥，仿制完全一致的锁定脚本 `scriptCode` 及对应的比特币金额 `value`。在只有锁定脚本而没有 Tx 完整内容情况下，我们无法直接通过 hash 和比对 TxID 来确认锁定脚本属于此 Tx。

为了解决防伪的问题，对于给定的 utxo，我们定义它的 `txid`, `index`, `value` 和 `scriptCode` 四者作为一个四元组，也就是一个完整的可验证结构体，我们简要地称之为 `u4` (`unspent-quad`)。此时我们获取并保持以下两个的签名，即可在合约中验证并保证锁定脚本在这两种情况下的真实性：

1. 对 `u4` 做签名（可将该签名称为 `sig_u4`）：

```
sig(txid, index, value, scriptCode)                 # sig_u4
```

2. 对 `u4` + "`花费该 u4 的 Tx`" 做签名（可将该签名称为 `sig_spent_u4`）：

```
sig(txid, index, value, scriptCode, spendByTxID)    # sig_spent_u4
```

有了锁定脚本和这两个签名后：

1. 任何合约脚本，已知 outpoint (txid, index)，可通过验证签名来使用此 utxo 的锁定脚本和比特币金额 (scriptCode, value)，便可查看同 Tx 其他输入锁定脚本，使得合约可以彼此协作。
2. 任何合约脚本，已知自己花费的 utxo 的 txid，便可通过验证签名来确信获得：产生该 utxo 的前序交易中的任一输入 utxo 的锁定脚本和比特币金额 (scriptCode, value)，使得合约可以向前追溯。

在获得了溯源及协作能力的同时，我们获得了以下两个值得关注的优势：

1. 这里要处理的签名目标，要么是一个 utxo，要么是 utxo 被某 Tx 花费，这些数据是公开、通用、固定、独立、且有限的。
2. 由于锁定脚本的体积是可控且有限长的，将验证所需的锁定脚本加入到解锁脚本中，并不会引起 Tx 持续膨胀。

Constructing a Backward-Traceable Contract Template (构造溯源合约模板)

如果我们按照下面的方式构造合约，则可以实现 FT (Fungible Token，对应完全功能的 ERC20) 和 NFT (Non-Fungible Token，对应完全功能的 ERC721)。

1. 从 GenesisTx 创建一个或多个此合约的 utxo
2. 要求使用时，必须首先输出 1 个此合约的 utxo
3. 保证每个被消费的合约都能追溯到 GenesisTx

在没有 `sig_spent_u4` 的情况下，因为脚本无法获知 Tx 的当前输入的前置 Tx 内容，不能阻止第三方伪造的此类合约混入使用，从而无法保证所有的合约使用都可以溯源到 GenesisTx。

有了 `sig_spent_u4` 的能力后，当从 GenesisTx 创建合约时，需在锁定脚本中记录 GenesisTx 的第一个输入outpoint（暂不考虑是coinbase的情况）。后续使用时，通过 PreImage 获得当前 Tx 所有输出 scriptCode，即可获知并限制第1个输出是同类合约 (这一点，有状态合约即可满足)。此时，通过 `sig_spent_u4` 可验证并获得当前输入 outpoint 的前置 Tx 相关的所有输入的 `u4`。

这种 “一脉相承”，由单一的 GenesisTx 派生出的可溯源合约模式，我们称其为 **溯源合约模板** TCT (Traceable Contract Template)。在溯源合约模板的基础上扩充，可以保持针对任意复杂的资产类型的可追踪性。

Constructing a Collaborative Contract Template (构造协作合约模板)

如果我们按照下面的方式构造合约，则可以实现定制化的 BSV / BSV-FT / BSV-NFT 在单一 Tx 内任意两两交易。

1. 任意创建多个此合约的 utxo
2. 要求使用时，必须同时消费掉 2 个此合约的 utxo，并输出 2 个此合约的 utxo

在未获取上述 `sig_u4` 的情况下，因为脚本运行时无法获知 Tx 的其他输入锁定脚本，导致无法要求 Tx 同时消费 2 个同类型的 utxo。

有了 `sig_u4` 的能力后，通过 PreImage 获得当前 Tx 所有输入的 outpoint，即获得所有 preTxID+Index，进一步通过 `sig_u4` 验证，获得所有 outpoint 的输出脚本，此时即可有足够的信息判断必须消费2个此类合约。与此同时，通过 PreImage 获得当前 Tx 所有输出 scriptCode，即可获知并限制必须包括2个此类合约输出。

这种两两成对出现的合约模式，我们称其为 **协作合约模板** CCT (Collaborative Contract Template)。在协作合约模板的基础上扩充，可以支持足够复杂，自由和灵活的多方交易。

Using induction to simplify the verification process (使用归纳法来简化验证过程)

这里我们通过一个归纳技巧来化简验证过程，使得溯源过程可以无需考虑从原始交易 (GenesisTX) 直至当前交易的整个交易历史链条，而被简化至仅考虑当前 Tx 的前序交易 (PreTx)。

1. 第一个输入 outpoint 能匹配 GenesisTx 的第一个输入 (确认是否源于最初发行交易)
2. 输入锁定脚本有 1 个能匹配本合约 (确认是否源于后继的合法交易)

这背后的逻辑是，如果当前合约的输入锁定脚本所在的交易，也就是当前交易的前序交易 PreTx，是已经验证的合法交易，那么说明合法性已经由 GenesisTx 传导至该前序交易。所以，当前合约的合法性，要么源于上一笔合法交易 PreTx，要么直接源于起始交易 GenesisTx，二者只要满足其一即可。

以类似的方式，我们可以在一些情况下，通过合法性的传导来简化协作合约模板的验证。这里不再赘述。

Implementations

有两种方式可以被用于实现上述的签名及验证过程。

Implementation 1 - Sensible Contract Signature Service (SCSS)

一种可选的方式是，使用一个独立的最小化签名器来实现签名的功能。

值得注意的是，在这种方式下，该签名器可以简化为仅含一个功能：它仅仅执行固定的单一动作，对一组固定的公开信息进行签名。该签名器无需监听 mempool，只需要调用方提供 utxo 所在的 Tx 被广播的完整原始内容，和花费此 utxo 的 Tx 被广播的完整原始内容，即可对正确性进行验证并签名。

这个签名器甚至无需在线，只要能够为一段消息产生正确的签名即可。

在这种情况下，我们付出的代价是，需要信任该签名器所产生的签名的有效性；获得的好处是，这种方式无需修改比特币节点软件的代码，及引入因此而导致的升级。

我们在 GitHub 上已提供一个 Rabin 签名的签名器实现，此签名器的代码实现已随本文档一同提交，供参考。

Implementation 2 - PreImage Upgrading

比使用外部的签名器更简单的方式，是升级 PreImage 的格式，新增两个 HASH256 字段（每一个 32 字节，共 64 字节）。

当前的 PreImage 由 10 个独立的字段构成 (见上文 “Problem #1. Contract Back-Tracing Problem” 节)

实现该过程，需要在 PreImage 中增加类似 `2. hashPrevouts` 的数据 `hashPrevoutDatas`，解释如下：

```
If the ANYONECANPAY flag is not set, hashPrevoutDatas is the double SHA256 of
the serialization of all input datas.
   Each data is the double SHA256 of (outpoint+value+script);
Otherwise, hashPrevoutDatas is a uint256 of 0x0000......0000.
```

具体新增的两个字段如下：

1. 实现协作能力，需要在 PreImage 中包括当前 Tx 的 `hashPrevoutDatas`。
2. 实现溯源能力，需要在 PreImage 中包括当前 utxo 所在 Tx（也就是前序交易）的 `hashPrevoutDatas`。

使用这种方式，需要升级比特币节点软件，使用新的 PreImage 格式 (包含上述两个 HASH256 字段)。

Compatibility Issue (关于兼容性的说明)

一些人也许会认为，通过扩展 PreImage 来实现新的功能，这种程度的修改属于 “破坏兼容性的修改”，被引入节点升级是比较困难的。但我们经过审慎的研究后认为，这虽然会引起前后版本不兼容，却仅仅是一个软件实现层面的改进。它所修改的 PreImage 结构，并非是比特币原始协议的一部分，而是在 2016 年由 Johnson Lau 和 Pieter Wuille 经由 BIP143 引入比特币的软件实现的。对 PreImage 的内容进行升级，此前已有先例，实际也 **并未涉及对比特币基础协议的修改。**因此我们提交此提议，并在感知合约的两种实现中，更倾向于第二个简洁有力的方案。

Use Cases

在与文档一同提交的文件中，我们同时提交了以下组件的参考实现：

- 基于感应合约的 BSV-NFT 合约代码实现
- 基于感应合约的 BSV-FT 合约代码实现
- BSV，BSV-FT 与 BSV-NFT 三者之间的两两交易代码实现

这些用例展示了感应合约的溯源与协作能力。

Miners & SPV Impact (对矿工和 SPV 实现的影响)

如果我们能够通过升级 PreImage 的方式实现感知合约，那么对当前 Tx 的 `hashPrevoutDatas` 计算，只会用到已有的数据，即每一个输入的 `u4` （也就是 `txid` + `index` + `value` + `scriptCode`），这一步不会给矿工和 SPV 实现带来难度。而当前 Tx 的前序交易的 `hashPrevoutDatas` 计算需要的额外信息较多，但如果将 32 字节的 `hashPrevoutDatas` 作为每一个 utxo 的附属信息保存，则无需重复计算。

SPV 无需信任任何第三方，即可从公开渠道获取这些 utxo 相关数据，进行签名。签名过程所需的计算量是相对较低的。

Conclusion

该技术通过对溯源和协作问题的解决，使得单一的比特币脚本拥有了在区块链的历史交易及当前的上下文中，感知，识别并区分不同合约的能力。这不仅使得伪造的合约可以很容易地被鉴别，也使得针对合约本身的溯源不再需要较重的外部数据支持，或由于需要在脚本中进行追溯而导致不可持续的合约膨胀。 更进一步，通过为不同的合约类型建立相关性，感知合约 (Sensible Contract) 使得更为复杂的多个合约实例在同一上下文中的协同工作变得可能。在这个方向上，目前仍有许多工作可以开展。

References

1. Bitcoin SV wiki. Opcodes used in Bitcoin Script
2. nChain Ltd. Genesis Upgrade Specification
3. sCrypt. OP_PUSH_TX
4. BIP 143 Transaction Signature Verification for Version 0 Witness Program

---

- Author: Jiang Jie, Gu Lu

- Version: v0.2.0
- Date: 2021-03-27