Sensible Contract

A Feature-Recognition Based Backward-Traceable and Collaborative Contract Model

Abstract

A typical bitcoin script allows us to have some degrees of control over "whether an amount of money can be used or not" in a single transaction. By adopting PUSH_TX technology, Stateful Contract allows us to extend this control beyond a single transaction, to ensure consecutiveness between a single transaction and its subsequent transactions. Based on the existing technology, we propose Sensible Contract and its corresponding implementation. By verifying the signatures of critical fields from different contracts, a single bitcoin script can now perceive and distinguish different contract types, therefore, it has acquired the ability to detect the forged contracts from untrusted parties and do backward-tracing to its original transaction. On this basis, Sensible Contract provides the capability for multiple contracts from different inputs to identify and collaborate with other contracts within a single transaction. Furthermore, by perceiving, identifying, distinguishing, responding to, and collaborating by recognizing the features of different contracts, any single contract, as a minimal independent action unit, can acquire the ability to establish relevance to all contracts across all transactions on the blockchain. In particular, this capability is inherent in the current bitcoin script system. As you can see from the use cases we provide, this capability can be achieved within the scope of the existing bitcoin script opcodes set.

Introduction

In this document, we begin with a brief history of the bitcoin script and some key changes; then, by examining two demands of real-world projects, we briefly outline the usage limitations of the current bitcoin script system; then we introduce the concept of Sensible Contract, and establish the perceiving and responding ability for bitcoin scripts to capture and verify information from critical fields from other contracts. To verify the effectiveness of the method, as a contrast, we provide two reference implementations at the software level, and we also present several actual use cases at the application level, to show how to use this method to address the needs from the real world.

A brief history about "Smart Contract"

The scripting system in bitcoin allows us to programmatically build certain constraints for the transfer of bitcoins. This high-level programmable capability provides us various advanced techniques with "advanced flexibility". However, for historical reasons, on the one hand, some opcodes (for "Security" reasons) have been discontinued; on the other hand, network load capacity (aka block-size), was kept artificially small by the development team of BTC (to keep it "decentralized"). These factors, objectively, have led to a significant reduction in the ability of the bitcoin script. Some of the advanced technologies that could have been experimented on the bitcoin network have become impossible. These limitations, at the time, led directly to the emergence of Ethereum (as an "improved"

programmable blockchain). By providing a degree of "completeness" that the bitcoin script system could not provide at the time, Ethereum became the first "smart contract" platform based on blockchain technology.

Bitcoin script development in BSV

In February 2020, when the 'Genesis' version of Bitcoin SV was deployed, most of these restrictions were removed from the Bitcoin SV node software, and some of the earlier bitcoin script opcodes were restored. This allows the scripting system to be freed from unnecessary constraints and become flexible again to be used for the development of more advanced constructs. A high-level language development environment called sCrypt is then built, allowing developers to write contracts in the high-level language and compile them into bitcoin scripts that can be verified and executed by the miners, as are typical bitcoin transactions. sCrypt's work unlocks the potential of the bitcoin script. On the one hand, using a high-level language like Solidity to develop can greatly improve development efficiency comparing to assembly-like opcodes; on the other hand, bitcoin's UTXO model makes single contract execution as localized as possible, rather than having to modify the global state serially in the Ethereum contract model. The bitcoin contracts in different utxo are potentially sufficiently parallelizable, regardless of the order in which they are executed, which greatly improves the scalability of the entire system.

OP_PUSH_TX and Stateful Contract

In 2016/2017, nChain researcher Ying Chan proposed a technology called `OP_PUSH_TX` to transfer information of current transaction to subsequent transactions. This method pushes the key information of the current transaction (so-called PreImage) onto the execution stack of the bitcoin script and uses a temporary private key to generate a signature for it. The OP_CHECKSIG can verify whether the signature belongs to the current transaction to ensure the authenticity of the pushed data, to ensure it can be obtained and used in the script.

With the Genesis upgrade, the technology was implemented by the sCrypt team in their smart contract solution and further extended and optimized. By using a separate data segment to store the state, the sCrypt team further implemented the state propagation of the bitcoin script in a series of contract transactions. This Stateful Contract makes it easy for developers to maintain a valid state between bitcoin transactions, and to update the state as needed according to specified business logic. Stateful Contract greatly expands the scope of the bitcoin script.

Contract Limitations in Real-world

However, considering the actual needs of real-world projects, the `OP_PUSH_TX` method and Stateful Contract still leave some outstanding challenges. Two of the most important issues are the **contract backward traceability** and **contract coordination**. Both of them involve how to perceive and distinguish potentially logically-relevant contracts within a single contract context, and perform actions accordingly. These challenges are directly related to the ability of bitcoin scripts to acquire and process the on-chain data, as well as to the usability of bitcoin scripting in real-world engineering.

The Method

In the following paragraphs, we present a detailed analysis of the issues of traceability and collaboration, thus providing a clearer picture of the capability boundaries of the current bitcoin script system. On this basis, we introduce the new method to solve these problems by identifying and responding to contract features, which will lay a foundation for further development of the use of bitcoin scripting.

Problem #1. Contract Backward-Traceability Problem

"The state is propagatable" is not enough to ensure "the legitimacy of the state itself".

For example, state propagation is not enough to maintain an effective UTXO-based token system, which is often referred to as `Layer-1 token`. To keep such a system safe from malicious offending in the real world, it is not only necessary to map the actual token holding onto a given utxo set, but also, crucially, any UTXO that has a token attaching with it, needs to be identified in a constant timeframe, whether it's derived from the original issuer's genesis transaction.

Returning to the terms of the contract, the essence of this requirement is to distinguish, within the executing context of a single contract, the authenticity of the sources of other "literally similar" contracts, that is, to determine whether different contracts are of the same origin (from a single source). For this kind of authentication issue of the contract, we call it "The Backward-Traceability Problem".

Why is it not possible to trace the root using the information already provided in PreImage? In Preimage's 4/5/6 fields, we can get the outpoint, locking script, amount of satoshis of the current input. However, these pieces of information can be forged. An untrusted third party has the ability to forge a valid outpoint, containing the literally consistent locking script and satoshi amounts needed to satisfy the demand. With these pieces of information alone, it is impossible to identify whether the outpoints with the same locking script are from a single root transaction (the 'genesis' transaction) in the current executing context, thus it is impossible to tell whether the current input is a true valid input.

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)

One possible way is to put the full content of the previous transaction of the given outpoint into the unlocking script. Thus, the authenticity of the content can be ensured by calculating the hash of the full

content and verifying if the result is consistent with the TXID. However, this will lead to a gradual expansion of the sizes of subsequent transactions (every unlocking script contains its own data, besides all data from its parent tx), which is unsustainable.

Another possibility would be, to store the information about the transactions to be tracked in a separate external service that would index, identify, and assist in the traceability of new transactions as they occur in mempool. This solution dramatically reduces the work of the L1 contract, but it needs to rely on the intensive external data and behaviors off-chain, to complete the whole function. Since the most critical steps are off-chain, the hybrid system introduces numerous single-point failure possibilities, thus it weakens the overall integrity and does not "solve the bitcoin problem within the bitcoin system".

Problem #2. Contract Coordination Problem

In addition to the Backward-Traceability Problem, there are times when we need contracts to identify and coordinate with each other to synchronize certain operations. Specifically, when completing a transaction with multiple contracts from different inputs, the contract needs to identify and respond to inputs other than those it is processing. In the existing script system, this requirement is also difficult to achieve, we call it "The Contract Coordination Problem".

For example, implementing the requirement that "two tokens of the same type exchange within a single transaction" requires a contract to confirm that the other inputs are of the same type. However, similar to the traceability problem, the existing PreImage lacks sufficient information to verify this. Specifically, PreImage includes the hash of all input outpoints for the current transaction (2. Hashpreveouts), which only indicates the existence of any given outpoint from the inputs of the current transaction, without acquiring and limiting the contents of these outpoints' locking scripts or the amounts of satoshis that correspond to them.

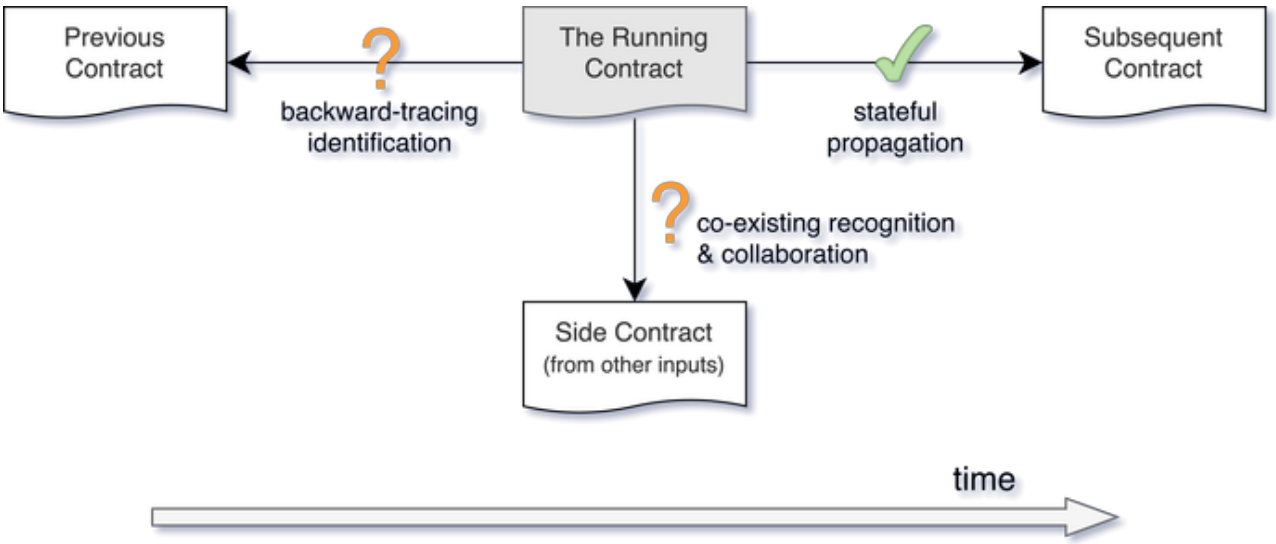
One possible way is to put the full content of the previous transaction of the given outpoint into the unlocking script. Same as above, the locking script can be acquired, provided that its authenticity is assured like above. Again, this will lead to a gradual expansion of the sizes of subsequent transactions, which is also an unsustainable method.

Similarly, indexing and assisting through external services is a hybrid solution, which relies on data and behavior off-chain.

The capability boundary of current script usage

If we look at the contract being executed, we can divide the control of the script into three scopes:

1. Where am I going? (How to impose constraints on subsequent contracts)
2. Who Am I with? (How to find out which contracts are co-existing with the current contract)
3. Where Do I come from? (How to identify the source of the current contract)



By analyzing the above two problems, we can realize the capability boundary of the current contract system: the stateful contract solves the first problem, which is "Where am I going?". It provides us the ability to extend control subsequently over time but does not address the latter two issues. It lacks the ability to do tracing backward through time. It also lacks the ability to perceive and respond to other ongoing contracts within the same context.

The traceability of locking script

So how to deal with these two problems?

Let's go back for a moment to the "gradual expansion" solution. We tried to push the full tx onto the stack when it was unlocked, in order to get specific information of the tx, while ensuring its authenticity. However, much of the information in full tx, such as the ever-expanding unlocking scripts, is useless for our purposes (identification and verification).

In fact, the **exactly same locking script** (`scriptCode`, item 5 in PreImage above) is sufficient to ensure consistent contract behavior. The locking script is the real **contract fingerprint** that needs to be authenticated and can be used for tracing.

With the locking script at hand, we revisit the above three questions:

1. Where am I going?
    - By obtaining and validating the locking script for all output within the current transaction, the output contract type can be determined and controlled (already in the current PreImage structure)
2. Who Am I with?
    - By obtaining and validating the locking scripts for all inputs of the current transaction, the other input contracts adjacent to yourself can be determined and controlled (not in PreImage yet)
3. Where Do I come from?
    - By obtaining and validating the locking scripts for all inputs and outputs of the previous transaction (PreTX), the current contract can now be backward-traced (not in PreImage yet)

In general, as the content carrier of a contract system, the locking script can be used to validate a contract, for traceability and collaboration purposes.

The anti-forgery unspent-quad signature

At this point, naturally, the question arises: if we only consider the content consistency of the locking script, how to achieve anti-forgery?

After all, once the contract is published, anyone can use their own private key to copy the exact same locking script `scriptCode` and the corresponding amount of satoshis. In the case of having only the locking script without the full tx content, we cannot hash and compare TxID to confirm that the suspicious locking script actually belongs to this transaction.

To solve the forging problem, for a given UTXO, we define its `txid`, `index`, `value`, and `scriptCode` as a quaternion, that is, an encapsulated verifiable structure, which we concisely call `u4` (`unspent-quad`). And then if we obtain and hold the two following signatures, we can verify and guarantee the authenticity of the locking script in contract executing context for both cases:

1. Sign `u4` (the signature could be called `sig_u4`):

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

2. Sign `u4` + "txid that spend the `u4`" (the signature could be called `sig_spent_u4`):

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

With the locking script and these two signatures:

1. For any given contract, with its outpoint (txid, index) known, we can verify the signature and then use the utxo's locking script and the amount of satoshis (scriptCode, value), in order to identify the other input locking script in the same transaction, so that the contracts can collaborate with each other in the transaction.
2. For any given contract, with the outpoint (txid, index) it spends (which resides in the previous tx) known, by verifying the signature, we can acquire the specific locking script and the amount of satoshis (scriptCode, value) of any input utxo of the previous tx (PreTX), to make the contract backward-traceable.

Besides gaining the ability to trace and collaborate, we also have gained two noteworthy advantages:

1. The data to be signed is either an utxo or an utxo spent by a given tx. The data is publicly available and easily acquirable. It's standalone with fixed content and limited size.
2. Since the size of the locking script is limited in length, appending the specific locking script required for verification to the unlock script does not cause the tx to continuously inflate.

Constructing a Backward-Traceable Contract Template

If the contract is structured and formulated in the following pattern, we can implement the FT (Fungible Token, corresponding to ERC-20 on Ethereum) and NFT (non-Fungible Token, corresponding to ERC721 on Ethereum).

1. Create one or more contract-embedded utxo(s) from the GenesisTx
2. Anytime 1 utxo is used, it outputs exactly 1 contract-embedded utxo firstly
3. Ensuring every consumed contract can be backward-traced to the GenesisTx

In the absence of `sig_spent_u4`, because the script doesn't know the content of PreTX, it cannot prevent forgeries of such contracts from being mixed-in by third parties, and thus cannot guarantee that all contracts are able to trace back to GenesisTx.

With `sig_spent_u4` at hand, when creating a contract from GenesisTx, we record the first input outpoint of GenesisTx in the locking script (ignore the case that GenesisTx is a coinbase tx). In subsequent use, by getting all current TX output scriptCode from PreImage, we are able to know and limit the first output to be a subsequent stateful contract. At this point, `sig_spent_u4` can be used to verify and acquire all input `u4` of the current input's PreTX.

This Backward-Traceable Contract pattern, derived from a single GenesisTx, is called **Traceable Contract Template** (TCT). By extending the contract template, we can maintain traceability for far more complex asset types.

Constructing a Collaborative Contract Template

If the contract is structured and formulated in the following pattern, we can implement trading BSV/BSV-FT/BSV-NFT in any two pairs within a single TX.

1. Create multiple utxos for this contract
2. Ensuring 2 utxos of this contract being consumed simultaneously, and then 2 utxos of this contract being output

In the absence of the `sig_u4` above, because the contract doesn't know the locking scripts from other inputs, it cannot forcibly demand that 2 utxos with the same type be consumed at the same time.

With the `sig_u4` capability, by acquiring all input outpoints (all preTxID+Index) from PreImage, and acquiring the locking scripts of all outpoints by verifying `sig_u4`, we gathered enough information to enforce that 2 contracts with the same type should be consumed. In the meantime, by getting all of the current TX output scriptCode from PreImage, we can acquire and limit the two contracts being outputted at the same time.

This "always-appear-in-pairs" pattern is called the **Collaborative Contract Template** (CCT). By extending the contract template, more complex and flexible multi-party transactions can be supported.

Using induction to simplify the verification process

We simplify the verification process with an inductive technique, so that the backward-tracing process does not have to take into account the entire transaction history from the original transaction (GenesisTX) to the current transaction, it is simplified to consider only the previous transaction (PreTx) of current TX.

1. The PreTx's first input outpoint matches the first input of GenesisTx (to confirm that PreTx matches the genesis tx)
2. One of the PreTx's input locking scripts matches this contract (to confirm that PreTx originated from a subsequent legitimate transaction)

The logic behind this is that, if PreTx (which owns the current input locking script), is proved to be a verified legitimate transaction, then the legitimacy has been propagated from GenesisTx to PreTx. So, the legitimacy of the current contract either stems from the previous transaction, PreTx, or it stems directly from the original genesis transaction, GenesisTx. It works by either one being satisfied.

Similarly, the validation of collaborative contract templates can also be simplified through the propagation of legitimacy. Check out the contract sources for trivial differences between them.

Implementations

There are two ways to implement the signature and verification process described above.

Implementation 1 - Sensible Contract Signature Service (SCSS)

An alternative approach is to implement the signature function using a separate service.

Notably, in this way, the service can be simplified to a single function: it performs a single fixed action, signing a fixed set of public information. The service does not need to listen to mempool. It only requires the full raw tx content of the tx (and its PreTX) being broadcasted to produce signatures.

It doesn't have to be online.

In this case, we have to trust the validity of the signature generated by the service; the benefit is that this approach does not require modifying bitcoin node software and the subsequent node upgrading.

We have provided a reference implementation on GitHub, with Rabin signature implemented, and the source code implementation has been attached with this document for your reference. As you may see, it's plain and simple.

Implementation 2 - PreImage Upgrading

A simpler way than using an external signature service, is to upgrade the PreImage format and add two additional fields to hold the SHA256 hashed values (32 bytes each, 64 bytes in total).

The current PreImage structure consists of 10 separate fields (see the section "Contract Backward-Traceability Problem" above)

To implement this process, two fields containing `hashPrevoutDatas` need to be added like `2. hashPrevouts`, which is explained as follows:

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

Specifically, the two new fields are described as follows:

1. To implement contract collaboration, `hashPrevoutDatas` of the current TX should be included in PreImage.
2. To implement contract traceability, `hashPrevoutDatas` of the previous TX should be included in PreImage.

In this way, the bitcoin node software needs to be upgraded to use the new PreImage format (containing the two new fields mentioned above).

Compatibility Issue

Some might argue that extending PreImage to achieve new functionality is an "aggressive modification", and that it could be quite difficult to introduce a node upgrade for it. However, after careful study, we believe that, although this will cause backward incompatibility, it by itself is only a software implementation-level improvement. The modified PreImage structure was not a part of the original bitcoin protocol, but was introduced into bitcoin software in 2016 by Johnson Lau and Pieter Wuille via BIP143. The upgrading to PreImage has been done before and does not actually involve changes to the underlying bitcoin protocol. We, therefore, submit this proposal and tend to prefer the second more simple and concise solution between those two.

Use Cases

Alongside the document, we also submitted reference implementations of the following components:

- BSV-NFT contract code implementation based on sensible contract
- BSV-FT contract code implementation based on sensible contract
- Contract implementation of trilateral exchange code between BSV, BSV-FT, and BSV-NFT

These use cases demonstrate the traceability and collaboration capabilities of Sensible Contract.

Miners & SPV Impact

If we could implement Sensible Contract by upgrading PreImage, then the `hashPrevoutDatas` calculation of tx would use existing data only, namely `u4` for each input (i.e., `txid` + `index` + `value` + `scriptCode`), this step would not make it difficult for the miners and SPV clients to do so. The `hashPrevoutDatas` calculation of PreTX transactions requires more additional information, but if the 32-byte `hashPrevoutDatas` is saved as additional information for each UTXO, there is no need for repeated calculation.

The SPV does not need to trust any third party to obtain these utxo-related data from public sources to perform the signature process. The amount of computation required for the signature process is relatively low.

Conclusion

By addressing the issues of traceability and collaboration, the technology enables any single bitcoin script to perceive, identify and distinguish different contracts within its own executing context. This not only makes forged contracts easy to identify, but also eliminates the need for heavy external data service to help to trace and coordinating, or unsustainable contract inflation due to the need for pushing everything in the unlocking script. Furthermore, by establishing correlations between different contract types, Sensible Contract makes it possible for more complex contract instances to work together in the same context. We are seeing plenty of work emerge in this direction.

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