TSL1: A Peer-to-Peer Token Protocol for Bitcoin (draft v0.1)

Stephan M. February

stephan@twostack.org

Abstract

A purely peer-to-peer token protocol on bitcoin would allow for the creation of novel tokenisation use-cases. The main benefits of such a token protocol would be lost if a trusted third-party is still required to provide double-spend protection of the tokens. I propose a solution to the double-spend problem for non-native tokens which benefit from the same double-spend protections as bitcoin's native accounting units. Wallets establish direct connections to one another by participating in a peer-to-peer overlay network separate from that of the bitcoin nodes. Tokens are transmitted in an [SPV manner](cite)(Simplified Payment Verification) between wallet peers. Wallets are recommended to simultaneously act as both headers-only SPV nodes within the bitcoin P2P network, and full peering nodes within a P2P overlay network. The P2P overlay network will facilitate direct wallet-to-wallet communications, and the bitcoin network will provide settlement and finality of the token transfers.

1 Introduction

Non-native tokens on bitcoin currently rely on indexing servers. These servers have to exhaustively scan every block that is mined on the network in order to provide double-spend protection for the non-native tokens. With the BitcoinSV network transitioning to Teranode, and with the consequent advent of 1MM+ tps transaction volumes, operating indexing services will become prohibitively expensive.

If an indexer is to continue operating, it will have to levy fees to cover costs. These increased costs will need to be passed on to the end-user, which in turn will limit the prospects of micro transactions for non-native tokens.

Some token protocols have proposed solutions that either suffer from recursive transaction bloating [reference], or which require the need to carry a full history of the token's spending history along with the token's transaction. Both of these approaches are only feasible for a few spending iterations before the computational overhead would become burdensome or downright impracticle.

What is needed is a peer-to-peer token protocol that provides for double-spend protection even in the absense of a token indexing service. Tokens' spending conditions should be directly validated by the bitcoin nodes. There should be no recursive transaction bloating, and the only data that should be stored alongside the token's transaction should be what is needed in order to perform SPV.

In this paper I propose a solution to the double-spending problem for non-native tokens on bitcoin. The solution employs a novel use of bitcoin script along with two transactions to represent a **token transfer**. The first transaction acts as the **token carrier**, and in combination with a second **witness** transaction provides an inductive proof that protects the token against double-spends.

This paper does not itself make specific proposals for either Fungible or Non-Fungible tokens, but instead provides a "scaffolding" framework that shows a method for providing double-spend protection for non-native tokens.

2 Overview of solution

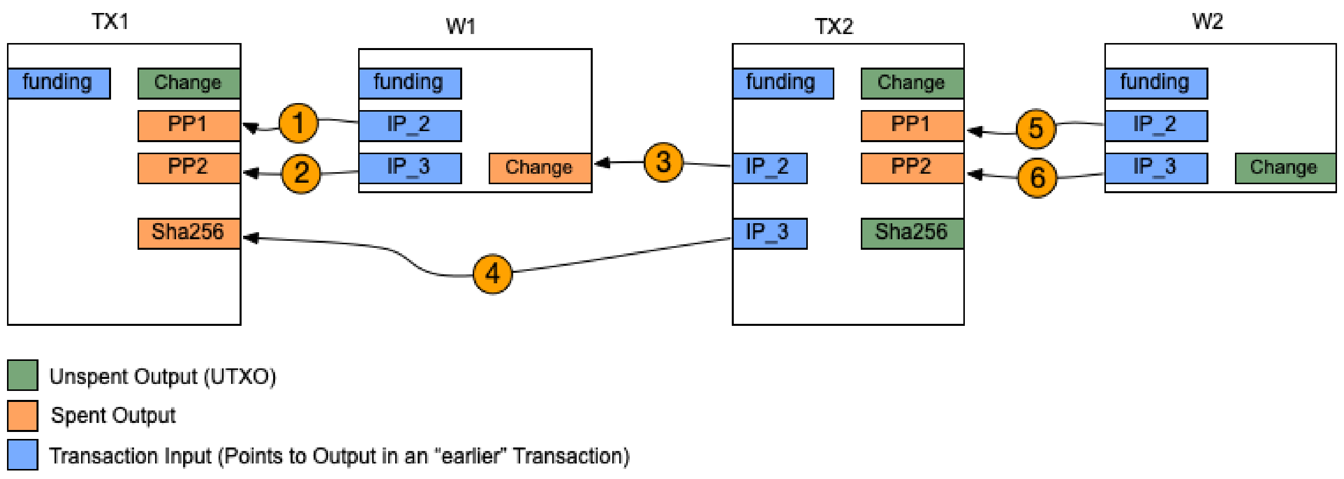

Figure 1: Overview of Token Movement

The proposed solution focuses on being able to provide an inductive proof of token validity. That means that we want the ability to assert within bitcoin script, that :

- consecutive transaction spends have preserved the token properties
- consecutive transaction spends have not been victim of a man-in-the-middle (MiM) attack thereby resulting in a double-spend

The best way to provide an inductive proof, is to have complete copies of both the current transaction, as well as the parent transaction. One can then verify both the preservation of token properties, and also assert that the previous transaction is "whole", i.e. it is not a result of MiM attack. The typical MiM attack results from copying the outputs of a valid token into a "fake" token, and then spending the "fake" token, resulting in an effective double-spend.

Until now inductive proofs have relied on the child transaction directly providing the raw transaction data of the parent via a `PUSH_DATA` within the spending input script (scriptSig) of the child transaction. This approach , while providing double-spend protection, results in recursive transaction bloat, and is therefore impractical.

The TSL1 token protocol works around the problem of recursive bloat by introducing a **Witness** transaction to provide the `PUSH_DATA` of the raw parent transaction data. This approach solves the recursive transaction bloat problem, but then reintroduces the risk of a MiM attack. In the following sections we will show how we are able to successfully mitigate this newly introduced risk.

2.1 Man-in-the-middle attack against the Witness

Consider the diagram wherein `W2[IP_2]` is providing the raw transaction data for `TX1` as input into the `TX2[PP1]` scriptPubKey. The inductive proof only works if we have the assurance that `W1` has performed it's `W1[IP_2]` spend from `TX1[PP1]`, thereby executing the onchain scriptPubKey in `TX1[PP1]` which verifies it's own ancestral spending conditions.

Without steps to mitigate, an attacker would be able to omit the inductive proof in `TX1[PP1]` by simply not spending `W1[IP_2]` from `TX1[PP1]`. There would be no way for the inductive proof step in `TX2[PP1]` to detect this omision, and a successful double-spend would occur.

2.2 Mitigating man-in-the-middle attack against the Witness

The mitigation of the MiM attack starts with the scriptPubKey denoted by `TX1[Sha256]`. The basic idea is that an inductive proof in `TX2[PP1]` can assert that

- `TX2[Sha256]` is the same as `TX1[Sha256]`
- `TX2[IP_3]` spends from `TX1[Sha256]`

TX1[Sha256] now has the responsibility of ensuring that

- `TX2[IP_2]` spends from `W1[Change]`
- `W1[IP_3]` spends from `TX1[PP2]`

Additionally TX1[PP2] is required to assert that

- `W1[IP_2]` spends from `TX1[PP1]`

So as a shorthand, we will have achieved a transitive assurance in `TX1[Sha256]`, via `TX1[PP2]`, that `W1[IP_2]` has correctly spent from `TX1[PP1]`. Combined with the inductive proof that matches all output scripts in `TX2` against output scripts in `TX1`, we would have successfully mitigated the potential MiM attack introduced by our Witness transaction.

2.3 Mitigating transaction bloat from Witness Push

Table 1: Witness Transaction Structure

Version
Number of Inputs in next field
Funding input
IP_1 input
IP_2 input
Number of outputs in next field
Change output
Locktime

Within the previous section, `TX1[Sha256]` requires that `TX2[IP_3]` provide the raw transaction data for `W1` as part of the scriptSig parameters. `TX1[Sha256]` will use this to calculate the transaction id of `W1` as well as to assert that `W1[IP_3]` spends from `TX1[PP2]`.

The astute observer will notice that when we do this, we re-introduce the recursive bloating problem since `W1[IP_2]` provides the raw transaction data for `TX1`'s parent transaction.

We mitigate this latter problem by performing a Partial Sha256 calculation of `W1`. I.e. we run the Sha256 calculation for all of the first part of the `W1` raw transaction, covering transaction headers, `W1[funding]` as well as `W1[IP_2]`, stopping short of covering the last 128 bytes of `W1`. We would have effectively compressed the first part of the raw `W1` transaction into a fixed-length 32 bytes of data.This also has the effect of removing the recursive bloat we would experience as a consequence of `W1[IP_2]` containing the raw transaction data of `TX1`'s parent. Within the remain 128 bytes of data would be all of `W1[IP_3]` and `W1[Change]` (the last input and outputs of `W1`).

Now, `TX2[IP_3]` would push in it's scriptSig ONLY :

- 32 bytes of precomputed Partial Sha256 for `W1`
- Remaining 128 bytes of raw `W1` data
- Sighash PreImage for `TX2` to assist in `OP_PUSH_TX` validation
- Funding input's transactionId to help with verifying witness outpoint spending.

3 Token Movement Explained

Token movement always consists of a Token Transaction **and** a Witness Transaction. The combination of these two transactions represent a token transfer. We will distinguish between two special cases:

1. Token Issuance - Issuing of a new Token
2. Token Transfer - Transferring ownership of a previously issued Token

Please refer to [Figure 1] for the following explanations.

3.1 Token Issuance

3.1.1 TX1 is the token carrier

- `TX1[PP1]` validates initial issuance conditions are met
- `TX1[PP2]` rebuilds SighashPreImage onchain, and does CHECKSIG verification to validate the provided TxId provided by `W1[IP_3]`. Asserts that `W1[IP_2]` spends from `TX1[PP1]`.

3.1.2 W1 is the issuance witness

- `W1[IP_2]` pushes token init params onto stack.
- `W1[IP_3]` pushes TxId of TX1 onto stack.

3.2 Token Transfer

3.2.1 TX2 is the token carrier

- `TX2[IP_2]` spends the W1 change output
- `TX2[IP_3]` pushes `PartialSha256(W1)` onto stack, along with `OP_PUSH_TX` preimage
- `TX1[Sha256]` calculates the TxId of W1, and verifies that `W1[IP_3]` spends from `TX1[PP2]`
- `TX1[PP2]` rebuilds SighashPreImage onchain, and does `CHECKSIG` verification to validate the provided TxId provided by `W1[IP_3]`

3.2.2 W2 is the transfer witness

- `W1[IP_3]` pushes TxId of `TX1` onto stack.
- pushes TX1 onto stack from input `W2[IP_2]`
- `TX2[PP1]` performs Onchain rebuild of `TX2`, along with inductive proof using the `TX1` provided by `W2[IP_2]`
- Combined size of `W2[IP_3]` and `W2[Change]` must be $<$ 128 bytes to limit Sha256 calculation in `TX2[Sha256]` to only two rounds (64 bytes per round)

4 Transaction Structural Details

4.1 Token Transaction

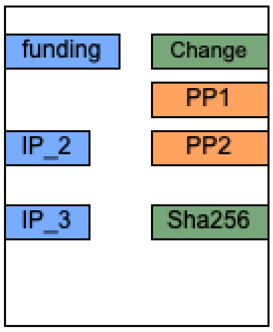

Figure 2: Token Transaction

4.1.1 Token Inputs

1. Funding - `TX[Funding]`

   The Token Transaction is funded using a standard P2PKH output. Take note that token transactions will typically have a size of at least 110kb, and would therefore require at least 110 sats of funding at 1sat/kb fee rates.

2. Spending Witness - `TX[IP_2]`

   The `IP_2` input (scriptSig) is only present during Token Transfer, and not during Token Issuance. This input spends from the first (and only) change output present on the Witness transaction. This would typically consume a 1sat output from the Witness UTXO.

   This is a regular P2PKH scriptSig, with the addition of a small 4-byte identifier to prevent accidental spending of the Witness output by wallets.

   Below is the modified P2PKH scriptSig parameters.

```
public function unlock(
     bytes idBytes,
     Sig signature,
     PubKey publicKey
)
```

Figure 3: Spending the Witness' change output

3. Spending Sha256 - `TX[IP_3]`

   The `IP_3` input (scriptSig) spends from the preceding token's `Sha256` output script (scriptPubKey), and therefore provides all the required input parameters that are needed by that scriptPubKey.

   The sCrypt unlocking script input parameters are :

```
public function unlock(
     SigHashPreimage preImage,
     bytes partialHash,
     bytes witnessPartialPreImage,
     bytes fundingTxId
)
```

Figure 4: Spending the token's witness-verifier

- preImage - The Sighash Preimage of the current transaction
- partialHash - The partial sha256 hash (32 bytes) for the raw witness transaction. See section 2.3
- witnessPartialPreimage - The last 128 bytes the raw witness transaction. See section 2.3
- fundingTxId - The funding input's prevTransactionId reference.

4.1.2 Token Outputs

1. Change - `TX[Change]`

   The Token's change output is a standard P2PKH output.

2. PP1 (Plug Point One) - `TX[PP1]`

   The second output of the Token transaction contains the code that performs the inductive proof checks to ensure proper token propagation, and to help provide double-spend protection for the token itself.

   The locking script template has the following constructor arguments.

   - recipientPKH : The public key hash of the token-holder. This is used in P2PKH-style output unlocking.
   - tokenId : Each token is identified by a unique tokenId. This tokenId will persist for the lifetime of the token, and is unique to each instance of the token.

```
contract PlugPointOne{

Ripemd160 ownerPKH; //the pubkey-hash of token owner.
bytes tokenId; //the token identifier

constructor(Ripemd160 recipientPKH, bytes tokenId){
    this.ownerPKH = recipientPKH;
    this.tokenId = tokenId;
}
.
.
.
```

Figure 5: PP1 locking script template parameters

   The locking script performs three overall functions :

   (a) It rebuilds the full raw image of the current token transaction
   (b) It checks that all output scripts of the current transaction must match their parent counterparts, accounting only for change of ownership
   (c) It checks that the current transaction must have spent from the provided raw parent token transaction
   (d) It checks that the token is P2PKH-locked to the keys of the new token owner.

3. PP2 (Plug Point Two) - `TX[PP2]`

   This output acts a "bridge" that enables the `TX[Sha256]` to perform a "transitive check" to ensure that `TX[PP1]` is actually being spent from a witness transaction.

This locking script therefore has only primary assertion, and that is to verify the outpoints of the Witness transaction. In order to do this, it requires only `OP_PUSH_TX` Sighash Preimage (cite) verification. We cannot require the spending input script in `Witness[IP_2]` to provide a full Sighash Preimage though, as this will violate the 128-byte sha256 optimization requirement. See [Witness Spending PP2].

Instead we this script will perform a full Sighash Preimage rebuild in-script, and then manually trigger `OP_CHECKSIG` to validate it's correctness. Having succeeded in Sighash Preimage rebuild it will proceed to verify the Witness transaction's `PP1` and `PP2` outpoints are spending from the current transaction.

```
contract PlugPointTwo{

   bytes fundingOutpoint;
   Ripemd160 witnessChangePKH;
   int changeAmount;

   constructor(bytes outpoint, Ripemd160 witnessChangePKH, int witnessChangeAmount){
       this.fundingOutpoint = outpoint;
       this.witnessChangePKH = witnessChangePKH;
       this.changeAmount = witnessChangeAmount;
   }
.
.
.
```

Figure 6: PP2 locking script template parameters

4. Sha256 (Witness Verifier) - `TX[Sha256]`

4.2 Witness Transaction

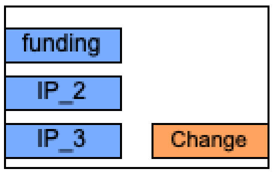

Figure 7: Witness Transaction

4.2.1 Witness Inputs - `TX[Funding]`

1. Funding

   The witness transaction is funded by spending from a P2PKH transaction. This funding is provided by the current token holder.

2. Spending PP1 - `TX[IP_1]`

```
public function transferToken(
    SigHashPreimage preImage,
    bytes           pp2Output,
    PubKey          recipientPubKey,
    Ripemd160       changePkh,
    int             changeAmount,
    Sig             recipientSig,
    bytes           scriptLHS,
    bytes           parentRawTx,
    bytes           padding
    )
```

Figure 8: Token's PP1 input parameters

The unlocking script has the following parameters, which will be supplied via the Witness transaction's second input, i.e. `W1[IP_2]`.

- **preImage** : Sighash preimage of the Witness transaction.
- **pp2Output** :
- **recipientPubKey** : Public Key of the new owner. This is used for P2PKH-style validation.
- **changePkh** :
- **recipientSig** : Signature of new owner needed to unlock this output via P2PKH-style validation.
- **scriptLHS** : Left-hand-side (inputs) of the current transaction
- **parentRawTx** : Raw Tx bytes of the parent token Txn (NOT the parent witness transaction)
- **padding** : a precalculated series of null-bytes. This is done purely for the benefit of the spending Witness transaction, and is used only to ensure that the start of the last 128 bytes of the raw Witness transaction coincides with the start of the `W1[IP_3]` input. This is important for ensuring that our partial sha256 calculation is **always** restricted to only 2 rounds, and that it **ONLY** needs to include `W1[IP_3]`, `W1[Change]` and `W1[locktime]` fields.

3. Spending PP2 - `TX[IP_2]`

   To successfully spend the token's `PP2` output, only one parameter is required, the transaction id of the token transaction. This only takes up 32 bytes of space, and is important in helping to keep the combination of the Witness' last input, and it's change outputs (`Witness[IP_2]` + `Witness[Change]` + `Witness[locktime]`) to within the required 128 byte boundary.

   Recall that the 128 byte limit is established as an optimization to ensure that partial sha256 calculations of the Witness transaction only requires two rounds of in-script sha256 computation.

```
public function unlock(bytes tokenTxId)
```

Figure 9: Token's PP2 input parameters

4.2.2 Witness Output

1. Change = `TX[Change]`

   The Witness transaction only has one change output. The change output is a non-standard P2PKH script, to prevent accidental spending by wallets. In the event that the Witness output is spent in anything other than a Token transfer transaction, the token will be effectively rendered innert, and no further token transfers will be possible.