RECURSIVE SMELTING: A TOKEN LAYER PROTOCOL ON BITCOIN

HARRY BARBER & CHRISTOPHER NOVICK

ABSTRACT. We present a tokenization protocol for the Bitcoin platform wherein a complete chain of custody connecting issuance and spending can be proven using minimal overhead and an open source verifier.

1. Introduction

During Bitcoin's history, there have been several proposed token proposals. In recent months, a new wave of interest has resulted in the creation of many more.

Existing proposals typically fall into four distinct categories:

(1) Requiring a change to the Bitcoin protocol (e.g. GROUP and RootStock)
(2) Requiring an extensive alternative ecosystem (e.g. Counterparty Cash, Omni and Wormhole)
(3) Requiring a central issuer (e.g. Tokeda)
(4) Requiring validation of the chain of prior token transactions reaching back to the genesis transaction (e.g. Coinspark, Colu, EPOBC, OpenAssets, SITO, SLP)

In Recursive Smelting we first construct a token residing in category (4), requiring validation of the chain of prior token transfer back to their genesis. We then supplement each transfer with a (constant sized) proof of the token's history – removing this requirement (4) entirely and replacing it with a fast and secure cryptographically verifiable computation.

Additionally, tokens are exposed to all the possibilities enabled by Bitcoin's scripting language (e.g. multisig, timelocks, etc.).

2. Overview

The Recursive Smelting Protocol enjoys versatility, SPV compatibility and permissionless creation, issuance and transfer of tokens. To allow these properties to coalesce we require the handling of verifiable computations.

Token creation is performed via a genesis transaction containing metadata concerning the token, defining a unique token ID, and delegating minting permissions.

Token minting and issuance is done via a "minting transaction" which mints and issues coins, and again delegates minting permissions.

Token ownership is indicated publicly within the UTXO set indicated by an "assert" output within a valid minting transaction or "ownership transaction". Ownership is transferred, as in Bitcoin, by creation of a transaction with outputs redeemable by the receiving party.

Data concerning token quantity and token ID is passed from primitive transaction to primitive transaction through assert scripts to revoke inputs and conserved as a result of the script structure. Conservation of token ID and quantity are achieved via adherence to user-enforced consensus rules. These rules define the format of primitive transactions (designed to conserve ID and quantity) and define a valid primitive transaction as one in which every input spent is also a valid primitive transaction.

To prevent the need to traverse backwards through *all* prior transactions to the genesis block, we attach a verifiable computation (more specifically a succinct non-interactive argument of knowledge, or SNARK) to each primitive transaction which, when verified, proves that the prior transactions are all themselves valid primitive transactions. Recursive proof techniques are employed to ensure that this method is tractable.

3. Primitive Transactions

The transaction graph of Recursive Smelting can be thought of as a closed subgraph of all Bitcoin transactions[1] and is composed by transactions adhering to one of several primitive transaction formats. Each primitive transaction (excluding the genesis transaction) contains an assert output which, when unspent indicates ownership of a certain quantity of tokens. Validation that the transaction and its predecessors have adhered to the protocol verifies the asserted ownership.

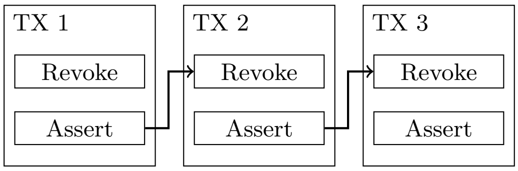
3.1. **Ownership Transaction.**

In the most simple case, ownership transfer can be described as the movement of tokens from Alice's wallet into Bob's wallet. Alice's ownership transaction consists of the following:

- An output asserting Bob's ownership of a given quantity and ID of tokens
- An input revoking the ownership assertion from previous ownership transactions

These ownership transactions are then linked sequentially as shown in the following schematic. Solid arrows denote signature scripts spending public-key scripts.

[1] Absolute closure is not feasible due to miners requiring transaction fees



The format of the assert and revoke scripts are given below where X and Y are any traditional signature and pubkey script.

Alice's Assert PubKey Script

```
⟨ token ID, quantity ⟩ OP_EQUALVERIFY Y
```

Bob's Revoke Signature Script

```
X ⟨ token ID, quantity ⟩
```

The format of the ownership transaction naturally generalizes to multiple inputs and outputs and supports multiple token IDs within a single transaction.

While scripts adhering to the above format ensure that token ID and quantity are preserved from Alice's assert output to Bob's revoke input, they do not however ensure that the token ID and quantity are preserved from input to output within individual ownership transactions. Therefore, the third and final requirement levied onto a valid transaction is that the input and output quantities are preserved within these individual ownership transactions.

To summarize, a basic ownership transaction is a transaction that meets the following criteria:

- Inputs have revoke script format,
- Outputs have assert script format,
- For each token ID the quantity is conserved from inputs to outputs.

3.2. **Genesis Transactions.**

A genesis transaction instantiates a token by dictating the metadata and operational rules of the token and delegating minting duties.

Some examples of meta data could be:

(1) Protocol Version
(2) Token name
(3) Optional central authority address[2]
(4) Optional burn address[3]

This is concatenated and put into an output of the genesis transaction using OP_RETURN.

This metadata instantiates variables used in the token's consensus rules. For example if a burn address is provided, the ownership transaction format should be changed to allow for the burn address to revoke, but never assert ownership.[4]

Minting delegation is accomplished via an output whose pubkey script is of the form:

Minting PubKey Script

```
⟨ token ID ⟩ OP_EQUALVERIFY Y
```

where Y can be any pubkey script with signature script X.

To summarize, a genesis transaction is a transaction that meets the following criteria:

- There is one output specifying token metadata format
- There is one output with minting pubkey script format

3.3. **Minting Transaction.**

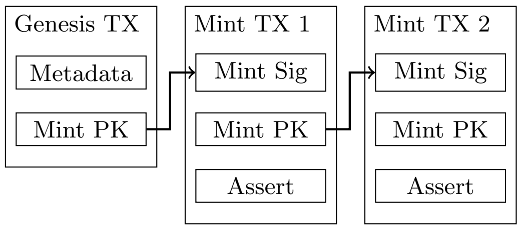
A minting transaction allows one to create additional tokens and, again, delegate minting.

In a similar vein to the ownership transaction format a minting transaction must have assert output scripts (which are then spent by revoke scripts of ownership transactions). The difference being that, unlike ownership transactions, minting transactions do not have revoke input scripts and are therefore not subject to quantity conservation. They are however limited to creation of tokens of the ID given in the script below:

Minting Signature Script

```
X ⟨ token ID ⟩
```

where X is the signature script of the Y given in the "Minting Pubkey Script" of Section **??**. A simple example of such a transaction graph could be:



To summarize, a minting transaction is a transaction that meets the following criteria:

- There is one input with minting signature script format
- There is one (or no) outputs with minting pubkey script format
- There is one or many outputs with assert script format
- The token ID in the assert and minting pubkey scripts must match the token ID in the minting signature script

4. Consensus Rules

In Recursive Smelting the protocol requires users adhere to consensus on what constitutes a valid ownership transaction. The consensus rules can be stated recursively as–a transaction is a valid primitive transaction if:

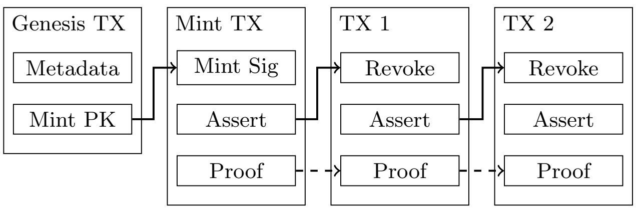
[2]An address with the ability to revoke ownership, e.g. the case of a central authority requiring final say in token ownership
[3]An address which is able to revoke ownership from token holders but cannot assert ownership themselves
[4]At the time of this writing, this feature has not yet been implemented

(1) It adheres to one of the primitive transaction formats,
(2) All inputs with revoke or minting pubkey scripts must spend outputs in a transaction with primitive transaction format.

The recursion defines a traversal back through the transaction tree through ownership transactions, then minting transactions, then the inevitable root: the genesis transaction. At each step of the traversal one checks whether each transaction adheres to the primitive transaction format, and on finding an exception deems every subsequent transaction invalid.

5. Proofs

While possible, it is undesirable to trace back through the transaction graph to the genesis transaction and verify the consensus rules hold as it precludes SPV compatibility. We solve this conundrum by supplementing each primitive transaction with a proof that the revoke and mint signature inputs are spending assert and mint pubkey outputs within valid ownership or minting transactions respectively. In the final incarnation proofs will validate the entire transaction graph by verifying proofs given *only* in the prior primitive transactions by using recursion.



We adopt notation from order theory where $a \lessdot b$ means that transaction b has a (mint signature and revoke) input spending transaction a's (mint pubkey and assert) output. Additionally, $a < b$ means there exists a sequence $a \lessdot \cdots \lessdot b$.

5.1. Non-recursive Proofs.

Acting as an intermediary step, we describe the construction of a non-recursive proof.

Consider a token who's genesis transaction is contained in some block b_n (where n is its block height). Suppose there exists a primitive transaction y in block b_{m+1}, we attach proof P taking the sequence of Merkle roots $\mathcal{M}_n, \ldots, \mathcal{M}_m$ (associated with blocks $b_n, \ldots, b_m$) as public inputs, the transactions $x < y$ and their Merkle proofs as private inputs and showing the following statements to be true: For each $x < y$ we have that

(1) x adheres to the primitive transactions format, and
(2) there exists a Merkle proof with root $\mathcal{M}_i$ where $n \leq i \leq m$ showing that x is contained in block i.

For a user to validate y they then must verify

$$v(P, (\mathcal{M}_n, \ldots, \mathcal{M}_m)) = \text{true}$$

where v is the verification function and that y itself is a primitive transaction.

Advances in cryptography have allowed proofs to be verified succinctly–that is, verification time is independent of the statement proved and takes milliseconds using consumer grade hardware.

While a proof of this type speeds up validation of primitive transactions, its downsides are:

- the time required to prove such a statement may be prohibitive, and
- the owner needs to traverse the transaction tree to collect its terms, making it non-compatible with SPV style wallets

5.2. Recursive Proofs.

To fix the problems described in the previous section we employ recursion.

Again, consider a token whose genesis transaction is contained in some block b_n (where n is its block height). Suppose there exists a primitive transaction y in block b_{m+1}, we attach proof P taking the sequence of Merkle roots $\mathcal{M}_n, \ldots, \mathcal{M}_m$ (associated with blocks $b_n, \ldots, b_m$) as public inputs, the transactions $x \lessdot y$ and their Merkle proofs as private inputs and showing the following statements to be true: For each $x \lessdot y$ we have that

(1) x adheres to the primitive transactions format,
(2) x contains a proof P' adhering to this recursive proof format, and $P'(h_0, \ldots, h_n)$ is true, and
(3) there exists a Merkle proof with root $\mathcal{M}_i$ where $n \leq i \leq m$ showing that x is contained in block i.

Again, for a user to validate y they then must verify

$$v(P, (\mathcal{M}_n, \ldots, \mathcal{M}_m)) = \text{true}$$

where v is the verification function and that y itself is a primitive transaction.

Verification of recursive proofs is fast, and proving is fast and SPV compatible as only x such that $x \lessdot y$ are required terms (rather than a full transaction tree).

6. Applications

6.1. Provable Token Burns.

Provably burning tokens is as easy as creating an assert with Y being OP_RETURN.

6.2. Token Issue Escrow.

Suppose a token issuer wanted to provably release 30% of tokens initially and then the rest at a later date.

This can be accomplished by producing a mint transaction with no mint pubkey script and the following asserts:

- an assert with $0.3 *$ total quantity tokens, and
- an assert with the remaining $0.7 *$ total quantity tokens with Y including a timelock.

6.3. Atomic Swaps: Bitcoin-to-Token.

One follows the exact same atomic swap protocol as before with the token seller conforming to assert/revoke format.

This allows for a decentralized exchange of tokens with bitcoin being the base pair.

6.4. Anyone-Can-Revoke.

In the circumstance that a token is widely traded and known to have value, the possibility to pay the transaction fee in said token arises.

To achieve this one could include an assert with $Y = \langle \text{ true } \rangle$.