Radiant Blockchain System Design

Attila Aros
August 11, 2022
radiantblockchain.org

Abstract

The Radiant network is a peer-to-peer digital asset system that enables direct exchange of value without going through a central party. The original Bitcoin[1] protocol provides what is needed to create a peer-to-peer electronic cash system, but lacks the ability to verify transaction histories and therefore cannot be used to validate digital assets. Digital signatures and output constraints provide part of the solution, but the main benefits are lost if a trusted third party is still required to validate digital assets. The Radiant network itself requires minimal structure, and operates similiarly to the Bitcoin network in timestamping transactions into an ongoing hash-based chain of proof-of-work. We introduce two techniques to validate digital assets using a general purpose induction proof system that operates in constant O(1) time and space. The induction proof system makes it possible to efficiently compose outputs in any manner, without compromising the inherent parallelism and scalability characteristics of the UTXO based architecture. Users can leave and rejoin the network at will and be assured of the integrity and authenticity of their digital assets.

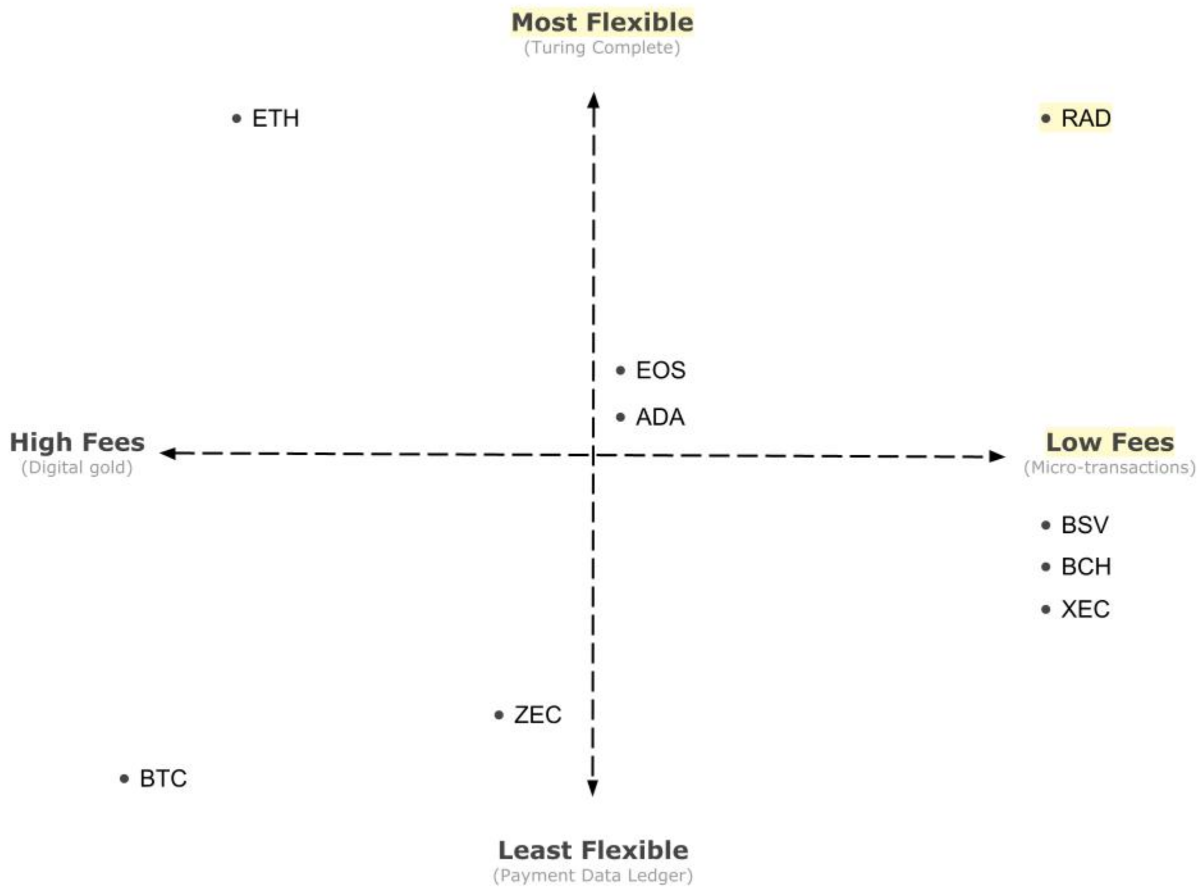
Introduction

Commerce with blockchains and digital ledgers has come to rely on issuers and custodians serving as trusted third parties (sometimes referred to as "bridges", "oracles", "secondary layers") to authenticate digital assets and process electronic payments. While the system works well enough for electronic payment-like transactions, it still suffers from the inherent weaknesses of the trust based model for more advanced usages of the blockchain. The high costs of transactions associated with Ethereum Virtual Machines (EVM) based blockchains is due to the limited block space and the inherent limitations of the account based model of processing.

What is needed is an electronic payment system that can also act as a digital asset management system with the performance characterstics of an unpsent transaction output (UTXO) blockchain architecture, with the flexibility of an account based blockchain. In this paper, we propose a solution to the problem of blockchain scaling using two novel methods which, independently, provide a general induction proof system capable of authenticating digital assets, emulating account based blockchains, while maintaining the performance characteristics of a UTXO based blockchain such as unbounded scale and parallelism.

The original Bitcoin[1] protocol provides what is needed to create a peer-to-peer digital asset system, but lacks the ability to verify transaction histories and as a result cannot authenticate digital assets. Blockchains such as Bitcoin Cash (BCH) and Bitcoin Satoshi Vision (BSV) attempt to authenticate digital assets via trusted third parties called "oracles" which indexes the relevant transactions. Such solutions, however, prevent the possibility of advanced blockchain contracts since a trusted custodian is required. In order to solve the problem of digital asset authenticity, without using central parties, we introduce two novel methods that operate in constant O(1) time and space. The additional programming instructions creates a general purpose induction proof system. Users and applications need only to verify that the latest digital asset transfer is accepted into a block.

Radiant is the first unspent transaction output (UTXO) blockchain that solves the key problems that prevented the development of advanced contracts on other blockchains such as Bitcoin, Cardano, and Dash. This breakthrough design revolutionizes what we imagined to be possible with blockchains; Radiant is a Turing Complete high performance layer one blockchain with no need for secondary layers.



Positioning of Radiant relative to popular blockchains.

Problems

There are three problems which make it impractical to use unspent transaction output (UTXO) blockchains as a general purpose digital ledger. The first problem is the ability to arbitrarily constrain the spend conditions — or forward conditions on all descendant transactions. The second problem is how to efficiently authenticate transaction outputs to ensure they originate from a valid genesis transaction — this is an essential requirement for many programs, especiaully to emulate accounts and create fungible tokens. The third problem is coordination and collaboration between contracts — precise control of message passing between transaction outputs. We will show that all three problems can be solved without compromising performance or the scalability of the UTXO-based blockchain model.

Contract Constraints

The first obstacle to programming with an unspent transaction output (UTXO) blockchain was a misunderstanding of Satoshi Nakamoto's original design and programming codes available in the original Bitcoin protocol. It is not generally acknowledged but the original Bitcoin blockchain had all of the programming codes necessary for Turing Complete [2] smart contracts. The necessary programming codes were removed from the protocol in the BTC upgrades of 2015.

The method to impose constraints on spending conditions is to restore all of the original programming codes from Bitcoin and to provide a method to inspect the current transaction context. There are two ways to inspect the current transaction as a type of introspection. The first way is to push the Signature Hash (known as the "SigHash Preimage") onto the stack and use a temporary private key to generate a signature and then apply the `OP_CHECKSIG` operation to validate that the expected SigHash Preimage for the current transaction is valid. The second way is to provide native introspection programming codes that push the relevant transaction component onto the stack for use in the unlocking script.

The key difference with a UTXO blockchain is there are no loops in the programming codes. However in practice any repetition can be simulated with unrolling the loop operations and replicating the logic for the necessary maximum number of repetitions. In this manner, UTXO blockchains can avoid any concept of "execution time cost" and instead estimate the execution cost by using only the transaction script size. For this reason, it is recommended that UTXO blockchains have a sufficiently large maximum transaction size, such as 2 megabytes or more to be able to accommodate any use cases that may need dozens or hundreds of loop iterations.

Contract Persistent Identity

An electronic coin is defined as a chain of digital signatures. A coin begins at a genesis transaction called a "coinbase" transaction. To transfer a coin, the owner of the unspent output signs the coin with their private key and locks the tokens in a new output which is associated with the public keey of the recipient. At each transaction a new transaction identifier and output index is used, which is globally unique. The concept of a "wallet balance" for a user is the sum total of the nominal token units controlled by the user for the unspent outputs for their corresponding public keys. Each coin in essence is uniquely identified by it's most recent unspent output. There is no inherent concept of an "account" or "coin identity".

In unspent transaction output (UTXO) blockchains the native token unit is the only class — or type and therefore a unique persistent coin identity is not neccesary. It is sufficient to have a different UTXO identity to enumerate the coins that can be spent. However, if we wish to create a different class of tokens, in other words to "color" the native tokens to represent shares, points or any other enumerable type, then we need a way to represent and efficiently validate token class membership. The term "colored coins" have been used to describe an overlay network which mints tokens from a special genesis or minting transaction — similar to the native coin is emitted from a coinbase.

A custom (or colored) digital coin is defined as a chain of digital signatures anchored at a user defined genesis output. Users may mint or create a custom coin issuance by depositing the desired number of native token units at the output and designating it as a coinbase with 36's 0x00 null bytes as the first push data of the otuput. The contract logic is constrained such that the subsequent spend of the output must embed the outpoint (transaction id, output index) of the genesis transaction into the first push data (where the 36 0x00 null bytes were in the genesis transaction) for the entire lifecycle of the colored coin. Following this convention combind with the contract constraints, we can see that this tecnique effectively "colors" the native token and can be identified unamnbiguosly. Additional logic can be added according to the application neeeds such as how the coins are redeemed or returned back to their native token units. For example the issuer can perform the operation or the token holder can "melt" out the native token unit and effectively destroy the color classification. The techniqu of embedding the genesis output forms a globally unique identifier sometimes referred to as asset identifier (or assetId for short) or contract identifier (or contractId for short) that may now be used to identify the coins that belong to that coinc lass. This identity will form the basis of the advanced usages outlined below.

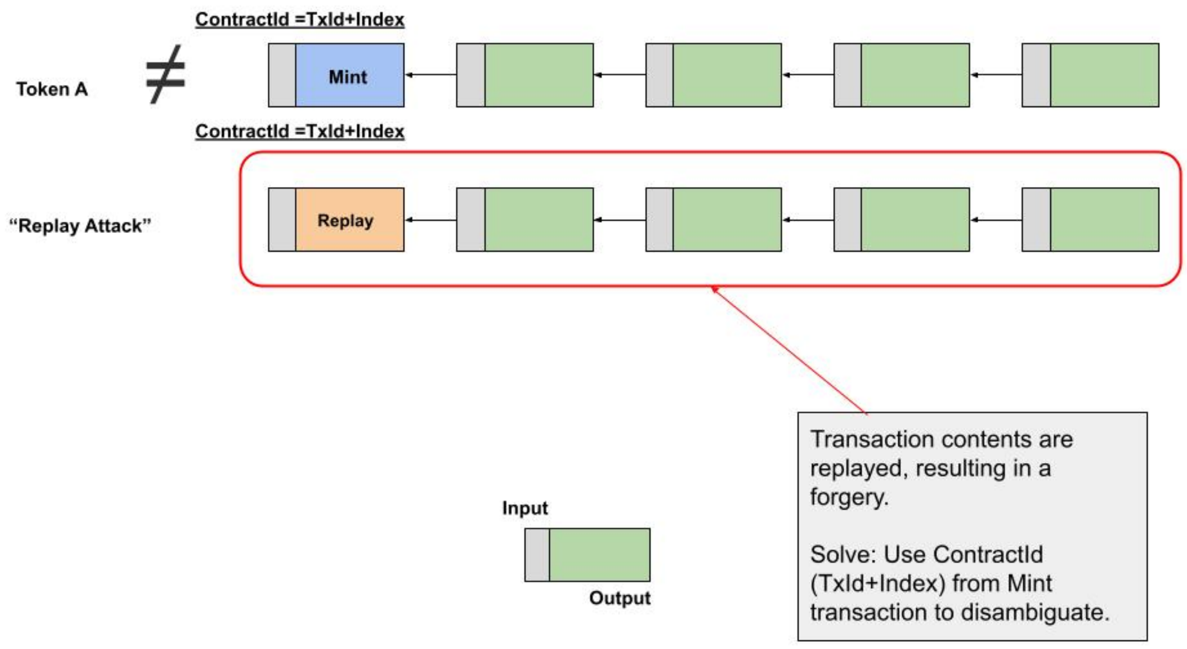

ContractId =TxId+Index
Token A
≠
Mint
ContractId =TxId+Index
"Replay Attack"
Replay
Transaction contents are replayed, resulting in a forgery.
Solve: Use ContractId (TxId+Index) from Mint transaction to disambiguate.
Input
Output

Diagram 1. Token replay attack

There is an outstanding problem however: How can spending transactions ensure that only coins descended from the rightful genesis transaction can be spent and not passed off into spending a forgery that was merely copied?

Contract Traceability & Authenticity

Recall that an unspent transaction output (UTXO) has no persistent identity, but we can give a persistent identity by following the rule that a user may designate some transaction as a genesis minting event, where the outpoint stands in as the assetId or contractId. However, using this convention it is not sufficient because an attacker can copy one of the intermediate transaction spends and begin a new (albiet forgery) chain of signatures to spoof a coin class and pass it off. Any spending transaction are unable to differentiate between a real output that originated as a valid descendent versus the forgery from a false copy. What is required is a way to enforce global uniqueness that is unobtrusive and efficient to verify inside a spending script.

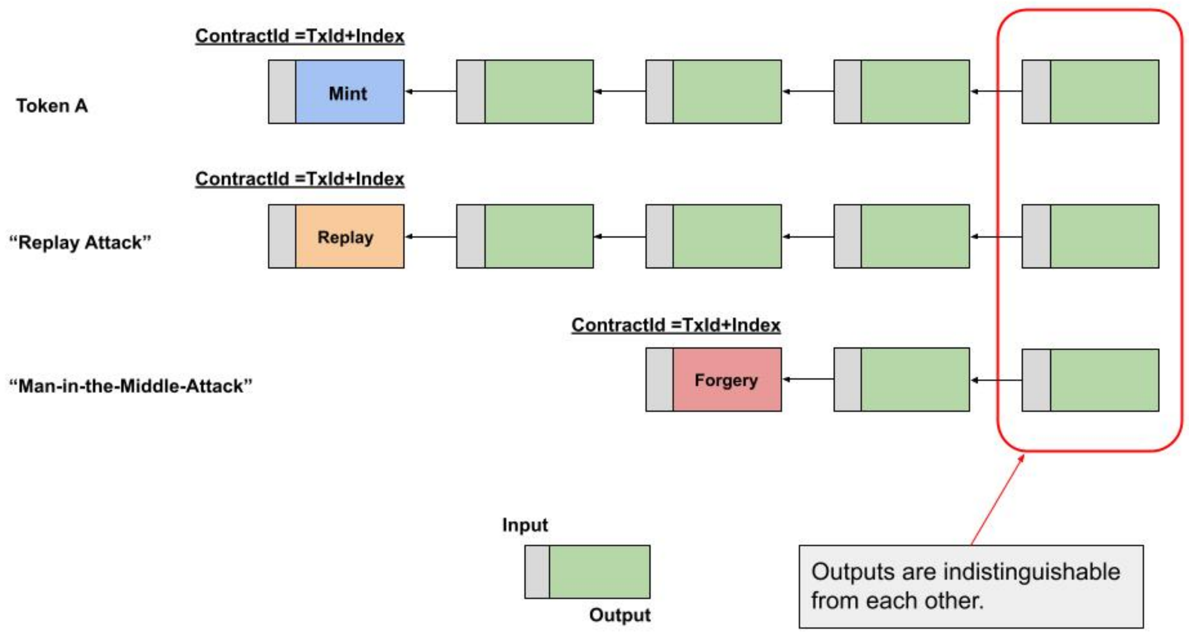


Diagram 2. Token man-in-the-middle forgery attack

OP_PUSHINPUTREF : Push reference

We define the programming operation code (OP code) `OP_PUSHINPUTREF <hash>` is defined as valid accordingly:

1. An `OP_PUSHINPUTREF` may appear only in an output and requires exactly 36 bytes immediately after that is treated as a push onto the stack in interpretor context.

2. The transaction containing an output with a `OP_PUSHINPUTREF` is valid if and only if the provided argument is equal to one of the inputs' outpoints being spent or at least one of the inputs' output locking script bytecode also contains the same `OP_PUSHINPUTREF` argument value.

The only way an `OP_PUSHINPUTREF` can first apppear in an output is if the first occurrence is equal to one of the inputs outpoints being spent. In the case of using the above "Persistent Contract Identity", this corresponds to the transaction that contains the 36 0x00 null bytes signifiying a genesis minting coinbase for a custom (colored) coin class.

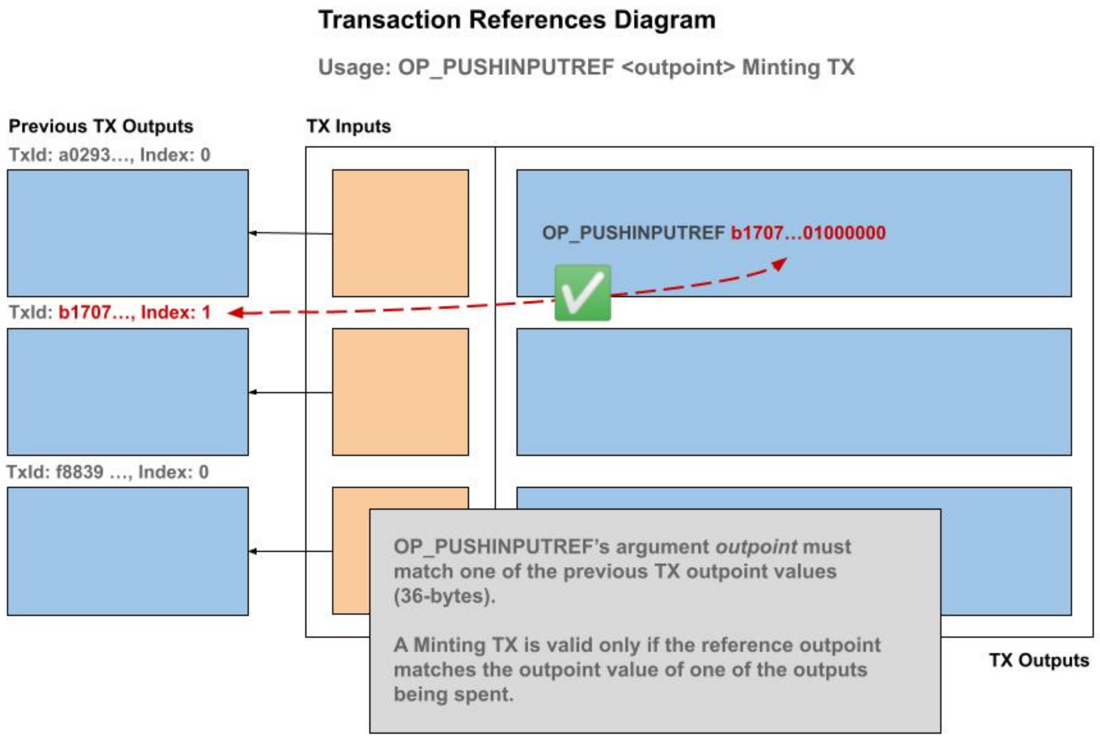

Transaction References Diagram
Usage: OP_PUSHINPUTREF <outpoint> Minting TX
Previous TX Outputs
TX Inputs
TxId: a0293…, Index: 0
OP_PUSHINPUTREF b1707…01000000
TxId: b1707…, Index: 1
TxId: f8839 …, Index: 0
OP_PUSHINPUTREF's argument *outpoint* must match one of the previous TX outpoint values (36-bytes).
A Minting TX is valid only if the reference outpoint matches the outpoint value of one of the outputs being spent.
TX Outputs

Diagram 3. Minting transaction OP_PUSHINPUTREF reference must match outpoint.

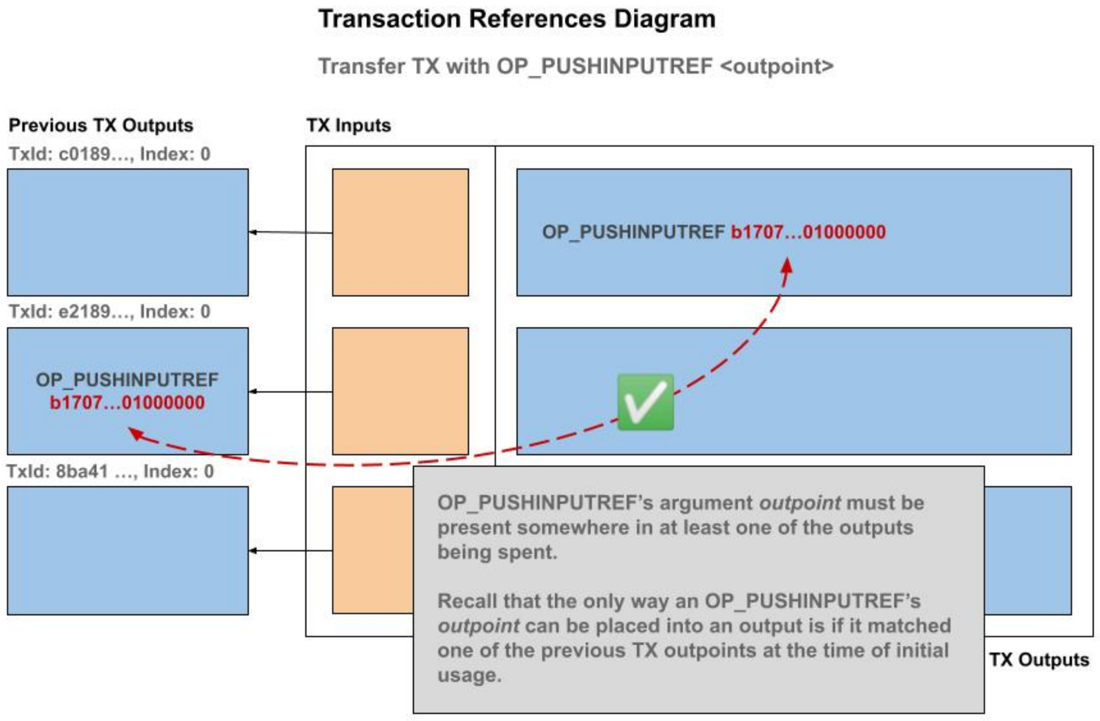

Transaction References Diagram
Transfer TX with OP_PUSHINPUTREF <outpoint>
Previous TX Outputs
TX Inputs
TxId: c0189…, Index: 0
OP_PUSHINPUTREF b1707…01000000
TxId: e2189…, Index: 0
OP_PUSHINPUTREF b1707…01000000
TxId: 8ba41 …, Index: 0
OP_PUSHINPUTREF's argument *outpoint* must be present somewhere in at least one of the outputs being spent.
Recall that the only way an OP_PUSHINPUTREF's *outpoint* can be placed into an output is if it matched one of the previous TX outpoints at the time of initial usage.
TX Outputs

Diagram 4. Transfer transaction OP_PUSHINPUTREF reference must match one of the previous outputs scripts being spent.

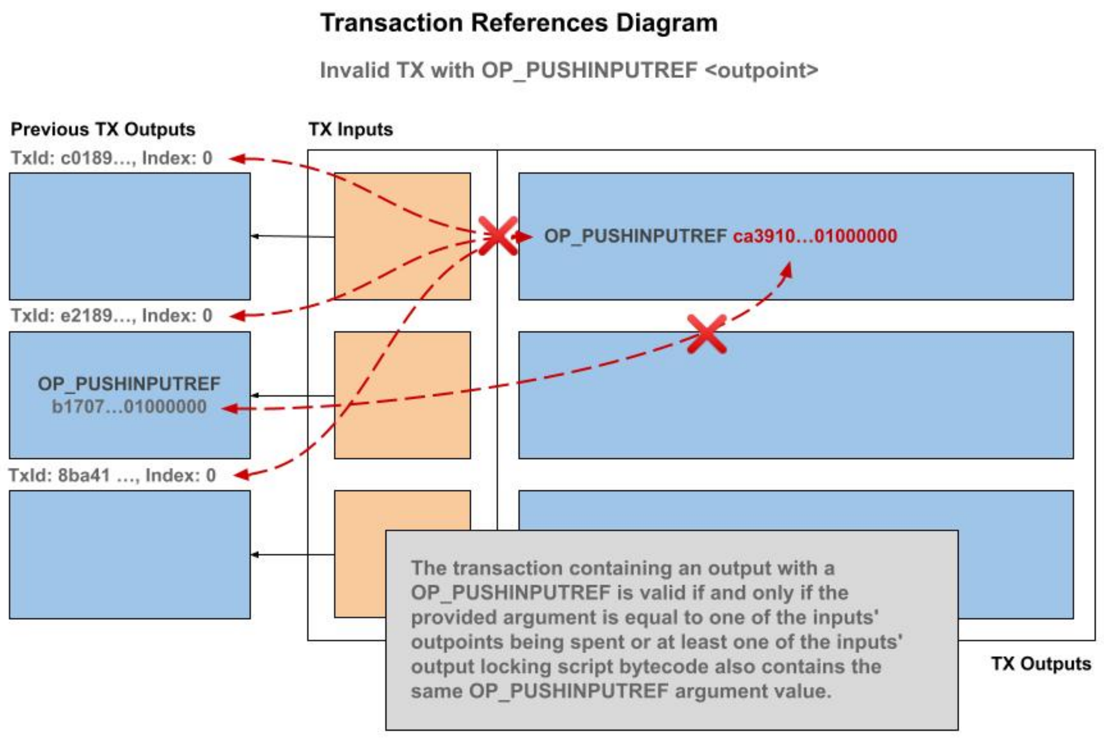

Transaction References Diagram
Invalid TX with OP_PUSHINPUTREF <outpoint>
Previous TX Outputs
TX Inputs
TxId: c0189..., Index: 0
OP_PUSHINPUTREF ca3910...01000000
TxId: e2189..., Index: 0
OP_PUSHINPUTREF b1707...01000000
TxId: 8ba41 ..., Index: 0
The transaction containing an output with a OP_PUSHINPUTREF is valid if and only if the provided argument is equal to one of the inputs' outpoints being spent or at least one of the inputs' output locking script bytecode also contains the same OP_PUSHINPUTREF argument value.
TX Outputs

Diagram 5. Invalid TX with no matching previous output script or outpoint.
We demonstrate that this simple rule is sufficient to form a globally unique identifier, and carries no overhead — as in no extra indexes or lookup tables are required. Only a transaction and it's immediate parent inputs are needed to validate authenticity — all of the data is available to the virtual machine at the time of the unlocking script evaluation and also into accepting the transaction into the mempool and subsequently into a block.

As long as at least one of the input coins has a valid 36 byte hash — either as the outpoint itself (significanty the first genesis chain of the colored coin) or as one of the scripts containing the reference, then the identity exists as a persisted identity. To terminate the lineage, simply omit passing on the reference and that terminates the ability to use that unique identifier in any other UTXO forever.

Although this single OP code is sufficient, there are a handful of additional OP codes that provided flexbility for the programmer and are described next which complement `OP_PUSHINPUTREF` .

OP_REQUIREINPUTREF : Require reference

The `OP_REQUIREINPUTREF` functions identically to `OP_PUSHINPUTREF` except it does not pass on the reference identity to the output in which it appears. This is useful for demanding that at least one input is of a specific coin class — but without passing down the reference immediately.

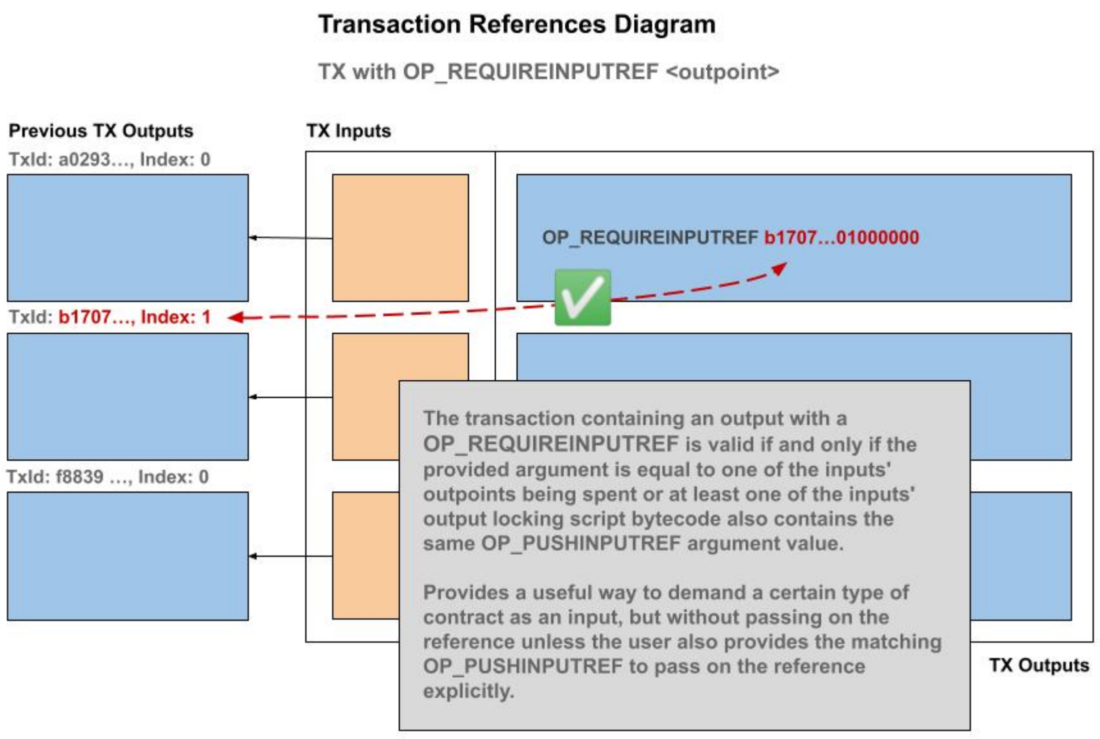

Transaction References Diagram
TX with OP_REQUIREINPUTREF <outpoint>
Previous TX Outputs
TX Inputs
TxId: a0293..., Index: 0
OP_REQUIREINPUTREF b1707...01000000
TxId: b1707..., Index: 1
TxId: f8839 ..., Index: 0
The transaction containing an output with a OP_REQUIREINPUTREF is valid if and only if the provided argument is equal to one of the inputs' outpoints being spent or at least one of the inputs' output locking script bytecode also contains the same OP_PUSHINPUTREF argument value.
Provides a useful way to demand a certain type of contract as an input, but without passing on the reference unless the user also provides the matching OP_PUSHINPUTREF to pass on the reference explicitly.
TX Outputs

Diagram 6. OP_REQUIREINPUTREF functions the same as OP_PUSHINPUTREF but without pushing the reference to the current output.

OP_DISALLOWPUSHINPUTREF : Disallow reference in output

To disallow the use of a `OP_PUSHINPUTREF` in an output, the `OP_DISALLOWPUSHINPUTREF` may be used. This is a useful OP code for smart contracts which leave open the outputs to be used in various contexts, but allows the contract creator to restrict passing down a reference, for example in custom change outputs.

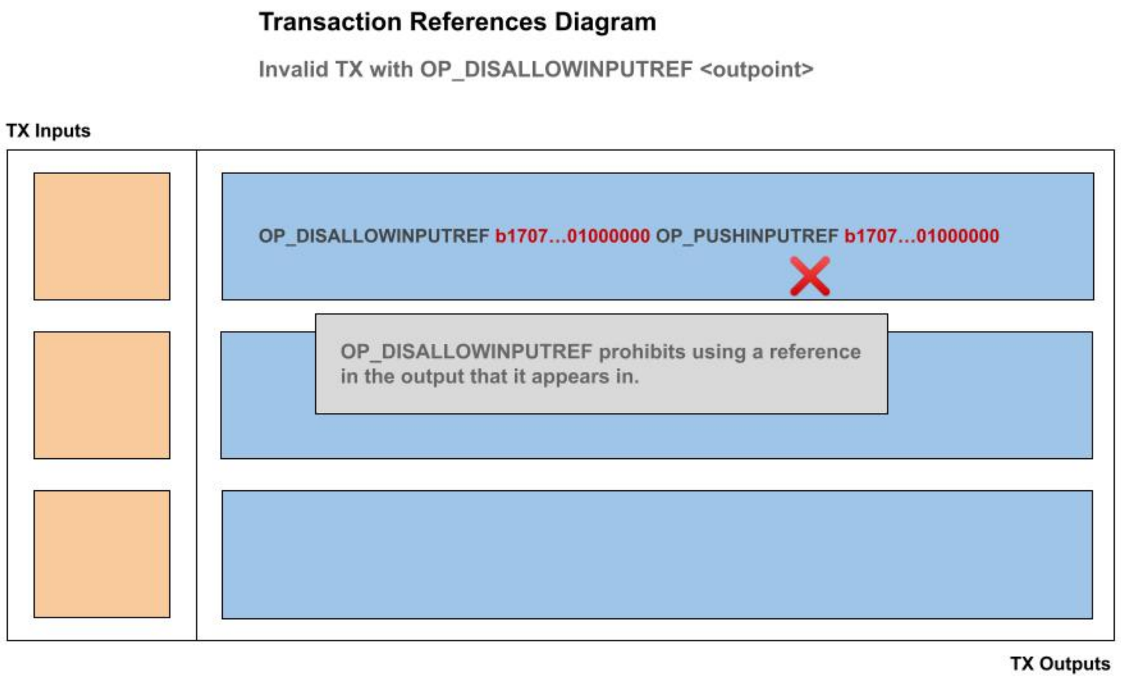

Transaction References Diagram
Invalid TX with OP_DISALLOWINPUTREF <outpoint>
TX Inputs
OP_DISALLOWINPUTREF b1707...01000000 OP_PUSHINPUTREF b1707...01000000
OP_DISALLOWINPUTREF prohibits using a reference in the output that it appears in.
TX Outputs

Diagram 7. OP_DISALLOWPUSHINPUTREF disallows usage of a specific reference.

OP_DISALLOWPUSHINPUTREFSIBLING : Disallow reference in sibling outputs

Similar to `OP_DISALLOWPUSHINPUTREF`, disallow specific outputs in any sibling outputs for the specifi reference. This effectively prohibits using a eference in more than one output and is a way to create a singleton outpoint. By using `OP_DISALLOWPUSHINPUTREFSIBLING` in an output we can create a simple and powerful Non-Fungible Token (NFT) contract which functions with `SIGHASH_SINGLE` signature flag.

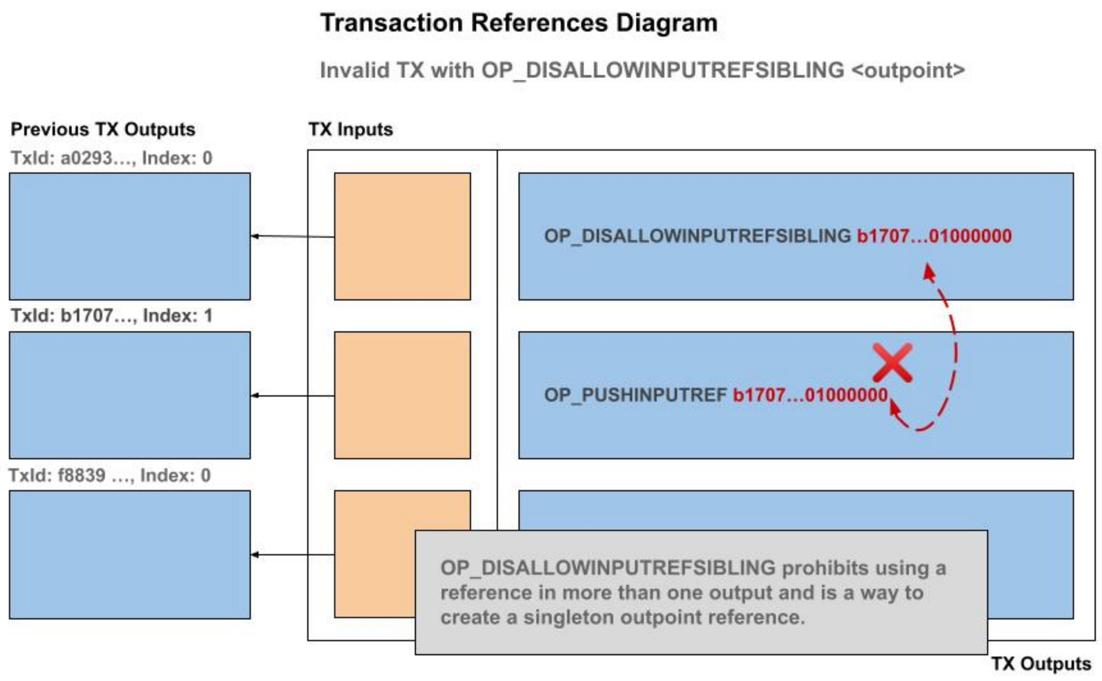

Transaction References Diagram
Invalid TX with OP_DISALLOWINPUTREFSIBLING <outpoint>
Previous TX Outputs
TX Inputs
TxId: a0293..., Index: 0
TxId: b1707..., Index: 1
TxId: f8839 ..., Index: 0
OP_DISALLOWINPUTREFSIBLING b1707...01000000
OP_PUSHINPUTREF b1707...01000000
OP_DISALLOWINPUTREFSIBLING prohibits using a reference in more than one output and is a way to create a singleton outpoint reference.
TX Outputs

Diagram 8. OP_DISALLOWPUSHINPUTREFSIBLING: Disallow usage of a specific reference in all other outputs than the one in which this instruction appears.

OP_UTXODATASUMMARY: Push UTXO data summary

Provides a summary of the contents of an output being spent in the current transaction. Takes the top element of the stack which is the index of the input being spent and then pushes the hash256 of the information about the UTXO being spent: `hash256(<nValue><hash256(scriptPubKey)><numRefs><hash256(sorted_list(pushInputRefs))>)` . During unlocking script evaluation, the relevant data of an UTXO is able to be accessed and incorporated into the logic.

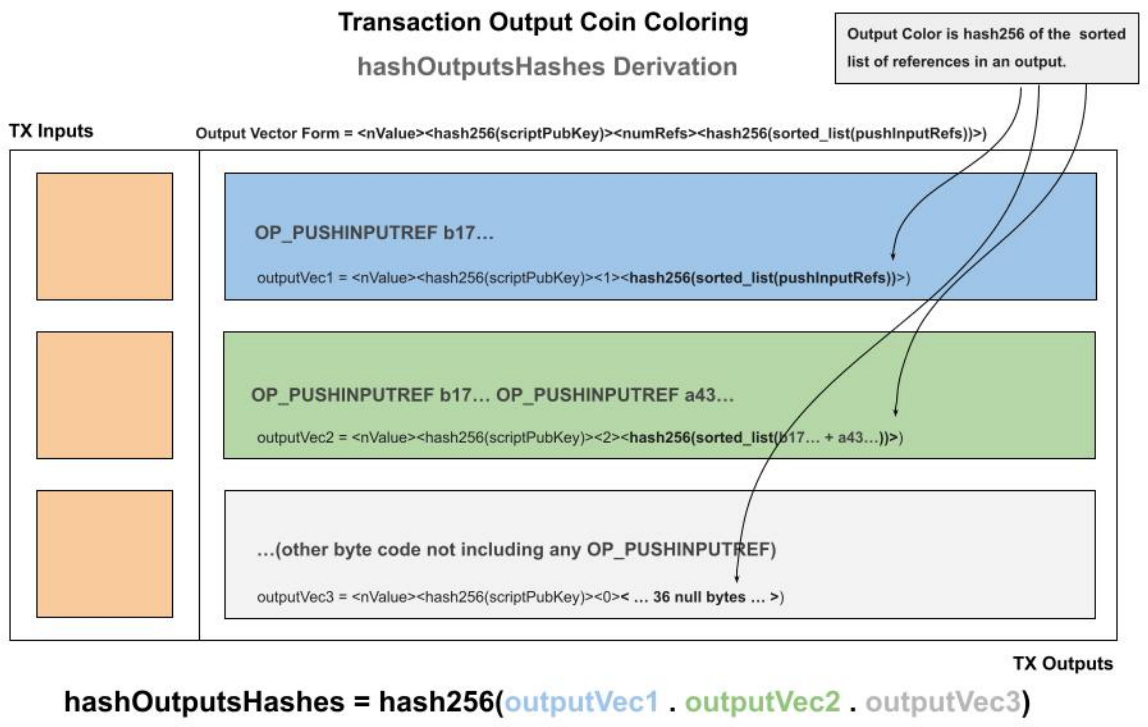

Transaction Output Coin Coloring
hashOutputsHashes Derivation
Output Color is hash256 of the sorted list of references in an output.
TX Inputs
Output Vector Form = <nValue><hash256(scriptPubKey)><numRefs><hash256(sorted_list(pushInputRefs))>)
OP_PUSHINPUTREF b17…
outputVec1 = <nValue><hash256(scriptPubKey)><1><hash256(sorted_list(pushInputRefs))>)
OP_PUSHINPUTREF b17… OP_PUSHINPUTREF a43…
outputVec2 = <nValue><hash256(scriptPubKey)><2><hash256(sorted_list(b17… + a43…))>)
…(other byte code not including any OP_PUSHINPUTREF)
outputVec3 = <nValue><hash256(scriptPubKey)><0>< … 36 null bytes … >)
TX Outputs
hashOutputsHashes = hash256(outputVec1 . outputVec2 . outputVec3)

Diagram 9. OP_UTXODATASUMMARY: Output coloring diagram

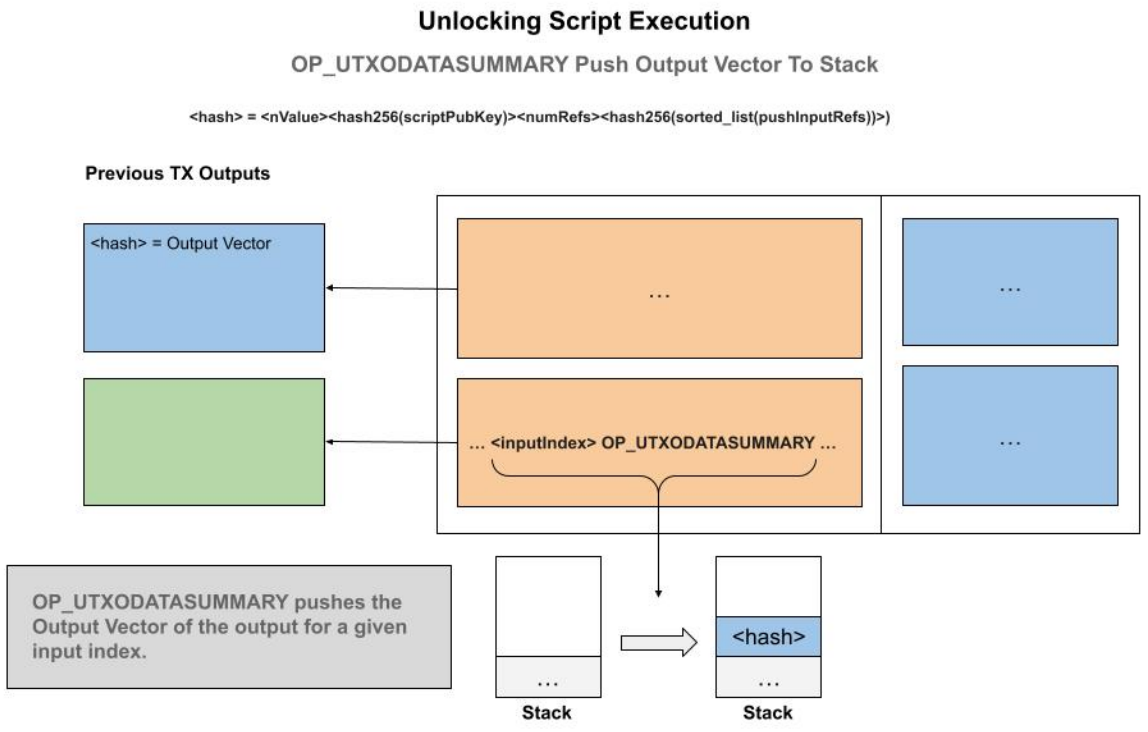

Unlocking Script Execution
OP_UTXODATASUMMARY Push Output Vector To Stack
<hash> = <nValue><hash256(scriptPubKey)><numRefs><hash256(sorted_list(pushInputRefs))>)
Previous TX Outputs
<hash> = Output Vector
…
… <inputIndex> OP_UTXODATASUMMARY …
OP_UTXODATASUMMARY pushes the Output Vector of the output for a given input index.
Stack
<hash>
Stack

Diagram 10. OP_UTXODATASUMMARY: Push to stack the summary of an input being spent.

OP_UTXOREFVALUESUM: Push value sum of UTXO by reference (color)

This programming code accepts a hash256 of the 36 byte reference and pushes onto the stack the sum total of all of the inputs that matches that reference coloring. This is useful for saving data and for quickly assessing the total inputs and the values input to the transaction.

This is very useful for building a compact fungible token accounting system as we shall see below.

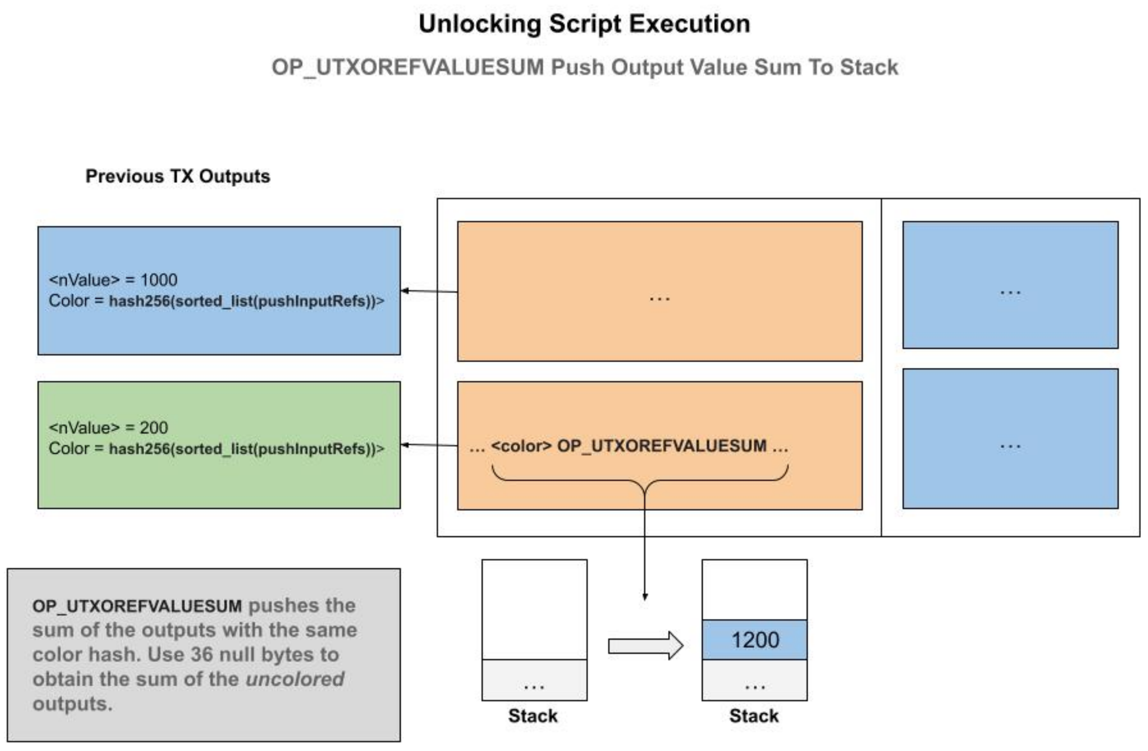

Unlocking Script Execution
OP_UTXOREFVALUESUM Push Output Value Sum To Stack
Previous TX Outputs
<nValue> = 1000
Color = hash256(sorted_list(pushInputRefs))>
<nValue> = 200
Color = hash256(sorted_list(pushInputRefs))>
...
... <color> OP_UTXOREFVALUESUM ...
OP_UTXOREFVALUESUM pushes the sum of the outputs with the same color hash. Use 36 null bytes to obtain the sum of the uncolored outputs.
1200
Stack
Stack

Diagram 11. OP_UTXOREFVALUESUM: Push to stack the total sum of the inputs which match a specific reference hash

Contract Authenticity via Induction

Another method for solving the traceability and aucthenticity problem is to allow the embedding of teh parent transaction into an unlocking script. In this manner, we can perform induction proofs and guarantee that a transaction output originated from a valid genesis minting event.

The general principle in mathematical induction os to prove that some statement P(k) holds for k = 0, k = 1, k = 2... and so on that generally P(k) holds for P(0) and P(k + 1).

In the case of smart contracts, we wish to prove that a given transaction is valid in the base case, as in it descends froma valid parent, the base case P(0), and in the inductive case that the grand parent also satisfies the condition, which is the P(k + 1) step.

With an induction proof it is impossible to forge an intermediate transaction because the grand parent transaction will not be of the required origination.

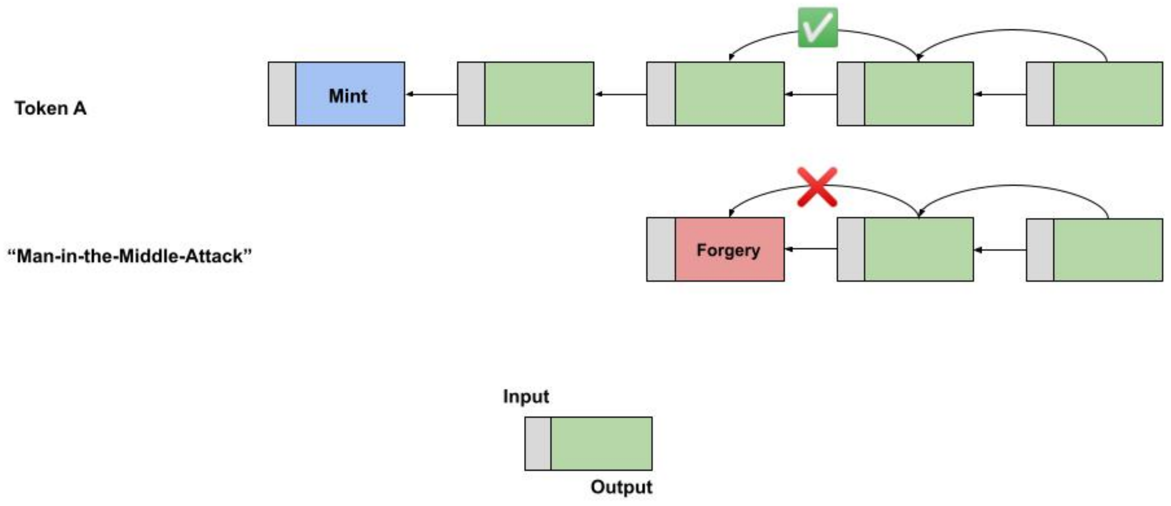


Diagram 12. Verify the parent and grand parent to identify forgery.
This system is not practical however because each time the output is spent a full copy of the parent (and it's parent) transaction must be embedded to calculate the transaction identifier. This leads to a factorial, or exponential, explosion in transaction size. It is not practical since after only about a dozen spends, the transaction size starts to exceed 1 GB and continues growing exponentially.

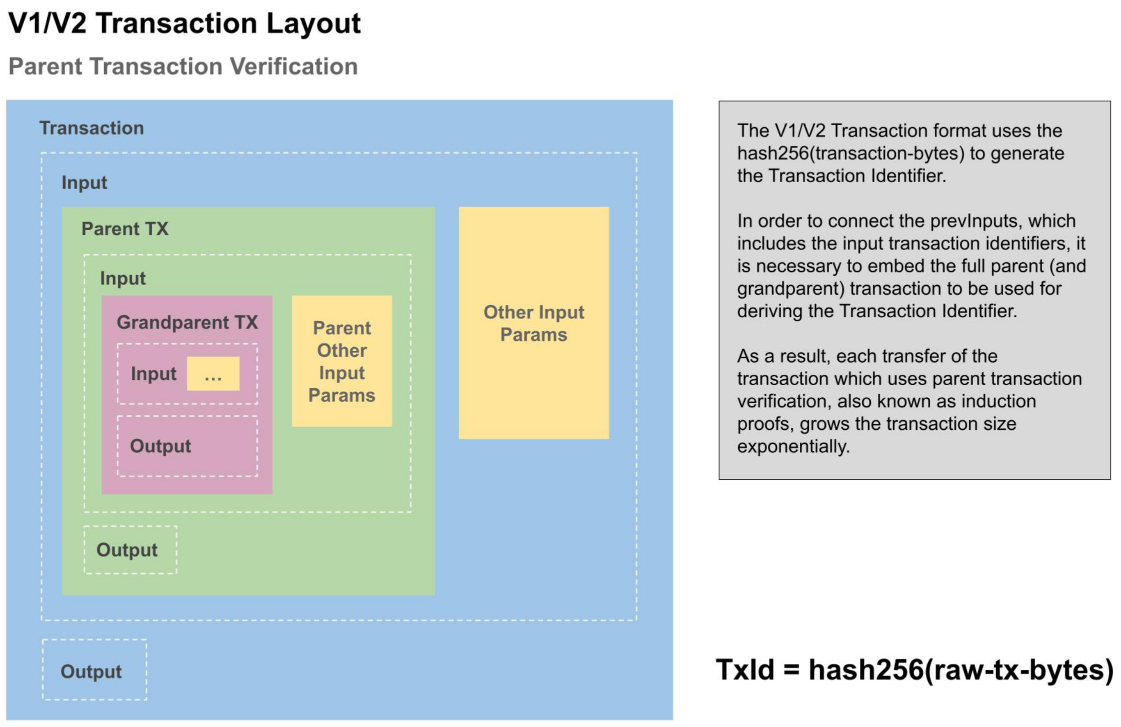

V1/V2 Transaction Layout
Parent Transaction Verification
Transaction
Input
Parent TX
Input
Grandparent TX
Input
...
Output
Parent Other Input Params
Output
Other Input Params
Output
The V1/V2 Transaction format uses the hash256(transaction-bytes) to generate the Transaction Identifier.
In order to connect the prevInputs, which includes the input transaction identifiers, it is necessary to embed the full parent (and grandparent) transaction to be used for deriving the Transaction Identifier.
As a result, each transfer of the transaction which uses parent transaction verification, also known as induction proofs, grows the transaction size exponentially.
TxId = hash256(raw-tx-bytes)

Diagram 13. Transaction Contents with Embedded Parents

To solve this problem of exponential transaction size growth we leverage the "nVersion" field provided by the transaction format. Bitcoin has version 1 and version 2 transactions already and we simply create a version 3 that uses a different transaction identifier generation algorithm instead of hashing the entire bytes of the transaction. The version 3 transaction format is identical except the transaction id is generated from an intermediate fixed size data structure that compresseses the transaction contents into a preimage — that can be embedded in locking scripts to derive the transaction id and avoding the exponential transaction size problem.

Transaction Identifier Version 3

Similar to the Signature Hash algorithm which generates a "Sighash Preimage", we produce a TxId preimage according to the following components and fields of a transaction.

> 1. nVersion of the transaction (4 byte little endian)
> 2. nTotalInputs (4 byte little endian)
> 3. hashPrevoutInputs (32 byte hash)
> 4. hashSequence (32 byte hash)
> 5. nTotalOutputs (4 byte little endian)

> 6. hashOutputHashes (32 byte hash)
> 7. nLocktime of the transaction (4 byte little endian)

Diagram 14. Transaction Identifier Version 3 Preimage

By incrementing the nVersion field, we introduce a way to compress an entire transaction into a fixed size (128 bytes) that can be pushed onto the stack, and hashed to arrive at the Transaction identifier, and therefore solving the problem of exponential size increase from from the embedded parent transactions in the induction proofs.

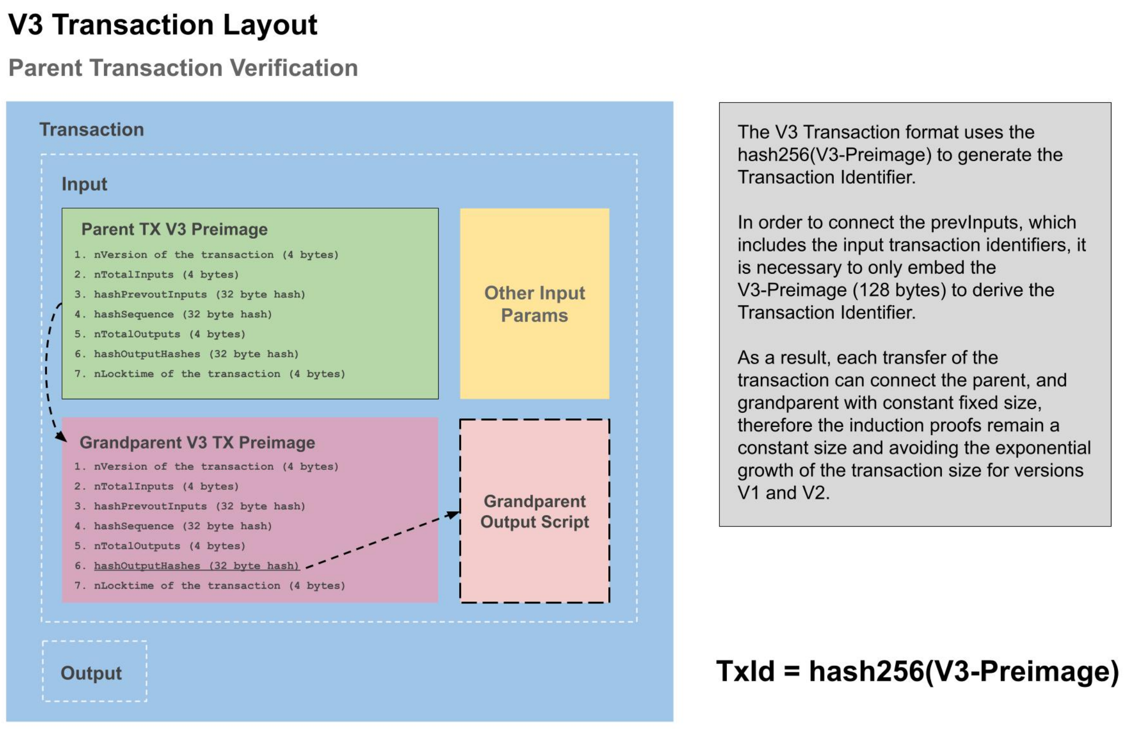

V3 Transaction Layout
Parent Transaction Verification
Transaction
Input
Parent TX V3 Preimage
1. nVersion of the transaction (4 bytes)
2. nTotalInputs (4 bytes)
3. hashPrevoutInputs (32 byte hash)
4. hashSequence (32 byte hash)
5. nTotalOutputs (4 bytes)
6. hashOutputHashes (32 byte hash)
7. nLocktime of the transaction (4 bytes)
Other Input Params
Grandparent V3 TX Preimage
1. nVersion of the transaction (4 bytes)
2. nTotalInputs (4 bytes)
3. hashPrevoutInputs (32 byte hash)
4. hashSequence (32 byte hash)
5. nTotalOutputs (4 bytes)
6. hashOutputHashes (32 byte hash)
7. nLocktime of the transaction (4 bytes)
Grandparent Output Script
Output
The V3 Transaction format uses the hash256(V3-Preimage) to generate the Transaction Identifier.
In order to connect the prevInputs, which includes the input transaction identifiers, it is necessary to only embed the V3-Preimage (128 bytes) to derive the Transaction Identifier.
As a result, each transfer of the transaction can connect the parent, and grandparent with constant fixed size, therefore the induction proofs remain a constant size and avoiding the exponential growth of the transaction size for versions V1 and V2.
TxId = hash256(V3-Preimage)

Diagram 15. Transaction Identifier Version 3 Preimages with Embedded Parents

Notice that this system itself is sufficient to create arbitrary induction proofs and is general purpose. This is a second method in which arbitrary induction proofs may be created in addition to the already discussed `OP_PUSHINPUTREF` technique.

Signature Hash Algorithm Upgrade

Building on the Transaction Id preimage the technique of segmenting the outputs, we can upgrade the default Sighash algorithm with an additional field called `hashOutputsHashes` to make it easier to constrain the outputs and save space and logic.

> 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. hashOutputsHash (32 byte hash)
> 9. hashOutputs (32 byte hash)
> 10. nLocktime of the transaction (4 byte little endian)
> 11. sighash type of the signature (4 byte little endian)

Diagram 16. Radiant Signature Hash Preimage Fields.
This is useful because the other sibling outputs do not need to be included and a hash can be used for the outputs that are not of interest. There is still the color of the push references so that we can assert whether the other outputs contain a valid color, but without requiring the full script to be pushed.

Contract Design Patterns

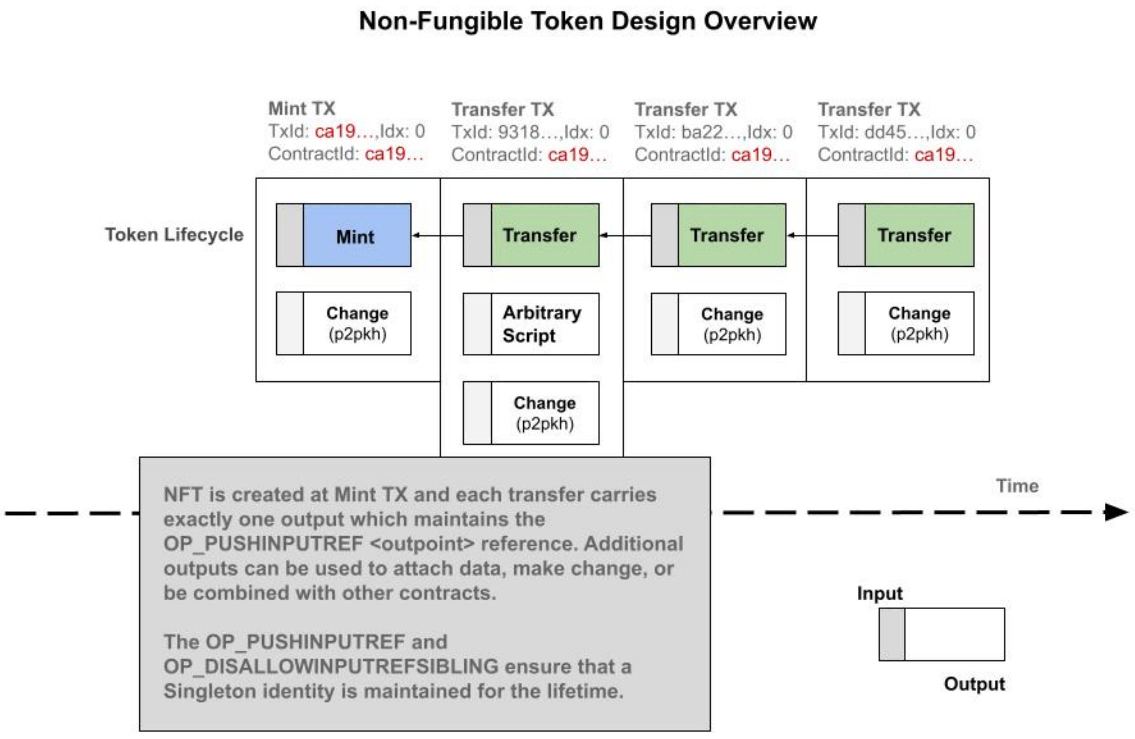
With the `OP_PUSHINPUTREF` and TxId Version 3 constructs, we are in a position to define various contract collaboration design patterns. together these patterns will be used in account emulation, non-fungible tokens (NFTS), fungible tokens (FTs) and other programs.

Non-Fungible Tokens (NFT)

> **Definition:** A "Non-Fungible Token" is a uniquely-identified object in which it’s essential properties are conserved

We present a simple, yet powerful, design pattern called a Non-Fungible Token (NFT). Programms will recognize this by another name called a *Singleton* object. An NFT, or Singleton, guarantees that only one instance of an object can ever exist and is unique identified by a stable persistent identity. Most usages of the term *Non-Fungible Token* have centered around *digital collectibles*, however that need not be the case — the reason for that focus has to do with the high-gas fees on Ethereum and the speculative nature of digital artwork.

In the Radiant blockchain, we use the term *Non-Fungible Tokens* to refer to a uniquely identifable object, or *colored coin*, which maintains some essential properties in addition to being unambiguously traceable through the blockchain. This is a basic building block and design pattern that will appear in more complex contracts, and here we present a simple, yet very powerful, construction below to start.



NFT is created at Mint TX and each transfer carries exactly one output which maintains the OP_PUSHINPUTREF <outpoint> reference. Additional outputs can be used to attach data, make change, or be combined with other contracts.

The OP_PUSHINPUTREF and OP_DISALLOWINPUTREFSIBLING ensure that a Singleton identity is maintained for the lifetime.

Diagram 17. Non-Fungible Token Design Overview.

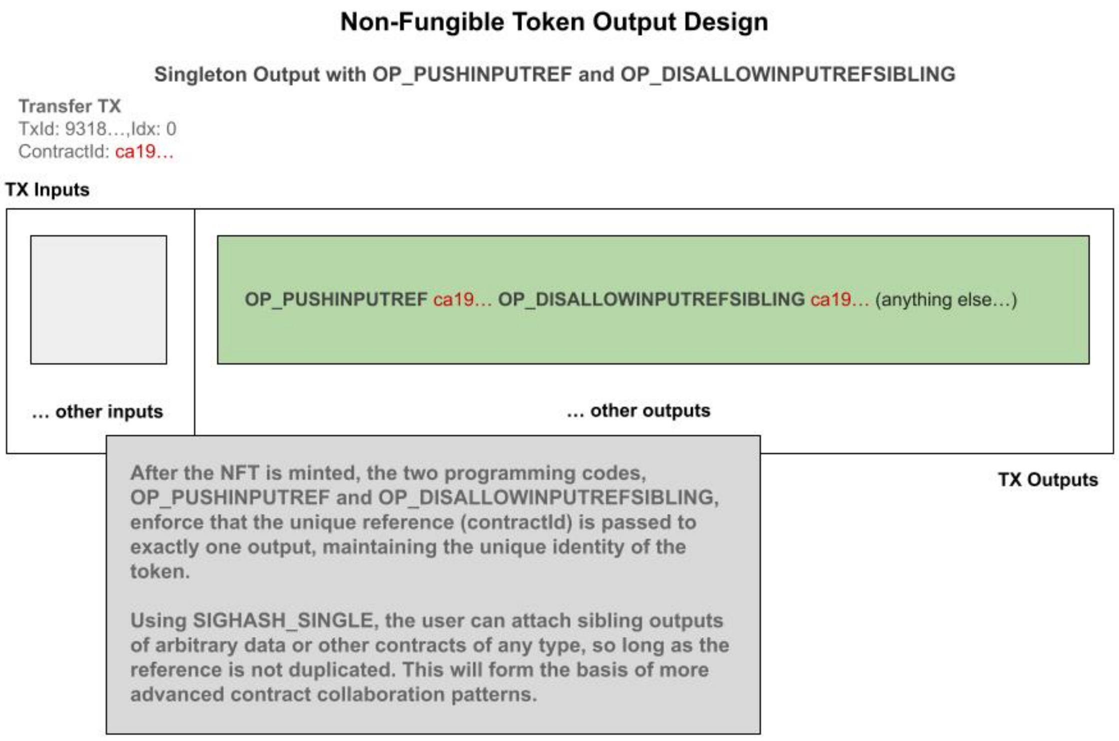

Non-Fungible Token Output Design
Singleton Output with OP_PUSHINPUTREF and OP_DISALLOWINPUTREFSIBLING
Transfer TX
TxId: 9318…,Idx: 0
ContractId: ca19…
TX Inputs
OP_PUSHINPUTREF ca19… OP_DISALLOWINPUTREFSIBLING ca19… (anything else…)
… other inputs
… other outputs
TX Outputs
After the NFT is minted, the two programming codes, OP_PUSHINPUTREF and OP_DISALLOWINPUTREFSIBLING, enforce that the unique reference (contractId) is passed to exactly one output, maintaining the unique identity of the token.
Using SIGHASH_SINGLE, the user can attach sibling outputs of arbitrary data or other contracts of any type, so long as the reference is not duplicated. This will form the basis of more advanced contract collaboration patterns.

Diagram 18. Non-Fungible Token Output Design

Non-Fungible Token Pseudo-code:

```
contract NFT {

    // Asset identifier
    bytes assetId;

    // Current owner is the second push data
    Ripemd160 currentOwnerAddress;

    public function unlock(
        SigHashPreimage txPreimage,
        bytes outputSats,
        bytes newOwnerAddress,
        bool isMelt,
        Sig senderSig,
        PubKey unlockKey
    ) {
        require(hash160(unlockKey) == this.currentOwnerAddress);
        require(checkSig(senderSig, unlockKey));

        // Initial assetId is 36-bytes nulls(0x00 bytes)
        bytes actAssetId = (this.assetId == num2bin(0, 36) ?
            txPreimage[ 68 : 104 ] : this.assetId);

        bytes lockingScript = SigHash.scriptCode(txPreimage);

        // The default usage is to update/transfer
        if (!isMelt) {
            require(
                hash256(
                    outputSats +

                    // Define length of output
                    b'fd' + num2bin(len(lockingScript, 2)) +

                    // OP_PUSHINPUTREF <assetId>
                    b'd0' + actAssetId +

                    // New owner (20 bytes)
                    b'14' + newOwnerAddress +

                    // OP_DISALLOWPUSHINPUTREFSIBLING <assetId>
                    b'd3' + actAssetId +

                    // Get entire locking script after the push vars
                    // 95 = 1+36 + 1+20 + 1+36
                    lockingScript[95 : ]
                )
                ==
                // Compare to HashOuts
                txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8]
            );
        } else {
            // Melt the NFT back and destroy the reference

            // Use OP_DISALLOWPUSHINPUTREF and
            // OP_DISALLOWPUSHINPUTREFSIBLING
            // to prohibit the reference from being passed along

            require(
                hash256(
                    // Hardcode len '4b' is 57 bytes (1 + 1 + 36 + 1 + 36
                    // 'd2' is OP_DISALLOWPUSHINPUTREF
                    // 'd3' is OP_DISALLOWPUSHINPUTREFSIBLING
                    b'00000000000000004b6ad2' + activeAssetId +
                    b'd3' + activeAssetId
                )
                ==
                // Compare to HashOuts
                txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8]
            );
        }
        require(Tx.checkPreimageOpt_(txPreimage));
    }
}
```

Diagram 19. Non-Fungible Token Pseudocode

Accounts

> **Definition:** An "Account" is an object that manages a wallet balance while maintaining an addressable stable unique identifier.

One of the main difficulties in working with a UTXO-based blockchain is there is no protocol level concept of "wallet balance", and instead infrastructure providers and wallet services present a summary balance derived from the total value of all the individual outputs controlled by a key. Account-based blockchains also simplify specific types of problems and contracts, but trade-off performance and privacy to achieve it's aims.

We present a simple design pattern to emulate accounts using one or more outputs which gives the user and developer a stable unique identifier across transactions in the blockchain. It is built using the Non-Fungible Token design pattern, and demonstrates that UTXO-based blockchains are perfecty equipped to emulate accounts with the same level of control, but with much higher performance characteristics.

Recall that in the Non-Fungible Token design pattern, the stable identifier *ContractId* (also sometimes referred to as *AssetId*) is derived from the outpoint of the minting transaction. The same ContractId is used as the public account identifier and can be treated as a wallet balance. We present below pseudo code for a smart contract that implements all the method associated with accounts: deposit, withdraw, changeOwner, and close the account.

```
contract Account {

    bytes assetId;
```

```
Ripemd160 currentOwnerAddress;
bytes disallowAssetIdNotUsed;

static function createSingletonOutput(
    SigHashPreimage txPreimage,
    int amount,
    bytes assetId,
    bytes address
): bool {
    bytes activeAssetId = (assetId == num2bin(0, 36) ?
        txPreimage[ 68 : 104 ] : assetId);
    bytes lockingScript = SigHash.scriptCode(txPreimage);
    require(amount > 0);
    require(
            hash256(
                // Add the deposit amount to the existing balance
                num2bin(SigHash.value(txPreimage) + amount, 8) +

                b'fd' + num2bin(len(lockingScript), 2) +

                // OP_PUSHINPUTREF <assetId>
                b'd0' + activeAssetId +

                // Address/owner (20 bytes)
                b'14' + address +

                // OP_DISALLOWPUSHINPUTREFSIBLING <assetId>
                b'd3' + assetId +

                // Get entire locking script after the push vars
                // 95 = 1+36 + 1+20 + 1+36
                lockingScript[95 : ]
            )
            ==
            txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8] //
    );
    require(Tx.checkPreimageOpt_(txPreimage));
    return true;
}

// Deposit to account
// Anyone can spend this input and deposit funds into
// the account, but only the owner can withdraw funds.
public function deposit(
    SigHashPreimage txPreimage,
    int amount
```

```
) {
    require(amount > 0);

    require(
        Account.createSingletonOutput(
            txPreimage,
            SigHash.value(txPreimage) + amount,
            this.assetId,
            this.currentOwnerAddress
        )
    );
}

// Withdraw from account
// The current owner can withdraw from the account
// via any other outputs.
public function withdraw(
    SigHashPreimage txPreimage,
    int amount,
    Sig senderSig,
    PubKey unlockKey
) {
    require(hash160(unlockKey) == this.currentOwnerAddress);
    require(checkSig(senderSig, unlockKey));

    require(
        Account.createSingletonOutput(
            txPreimage,
            SigHash.value(txPreimage) - amount,
            this.assetId,
            this.currentOwnerAddress
        )
    );
}

// Change the account owner
// The current owner can assign the account to another
// address owner
public function changeOwner(
    SigHashPreimage txPreimage,
    bytes newOwnerAddress,
    Sig senderSig,
    PubKey unlockKey
) {
    require(hash160(unlockKey) == this.currentOwnerAddress);
    require(checkSig(senderSig, unlockKey));
```

```
        require(
            Account.createSingletonOutput(
                txPreimage,
                SigHash.value(txPreimage),
                this.assetId,
                newOwnerAddress
            )
        );
    }

    // Close the account
    // The current owner of the account can permanently close
    // the account and withdraw any tokens via other outputs
    public function close(
        SigHashPreimage txPreimage,
        Sig senderSig,
        PubKey unlockKey
    ) {
        require(hash160(unlockKey) == this.currentOwnerAddress);
        require(checkSig(senderSig, unlockKey));

        bytes activeAssetId = (this.assetId == num2bin(0, 36) ?
            txPreimage[ 68 : 104 ] : this.assetId);

        bytes lockingScript = SigHash.scriptCode(txPreimage);

        // Ensure one of the outputs is unspendable OP_RETURN
        // and uses the OP codes to prohibit passing on the
        // reference.
        // OP_DISALLOWPUSHINPUTREF and
        // OP_DISALLOWPUSHINPUTREFSIBLING which effectively
        // means no output may contain the reference anymore,
        // thereby ending the ability to carry on the assetId
        // anywhere else forever.
        require(
            hash256(
                b'0000000000000000004b6ad2' + activeAssetId +
                // Hardcode len '4b' is 57 bytes (1 + 1 + 36 + 1 + 36)
                b'd3' + activeAssetId
            )
            ==
            // HashOuts
            txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8]
        );
        require(Tx.checkPreimageOpt_(txPreimage));
        }
    }
```

Diagram 20. Account contract pseudocode

Fungible Tokens (FT)

The Fungible Token design pattern allows the same class or type of object to have more than a quantity of one. The fungible tokens can be merged together, with their values summed up into a new output, or an output can be split into two or more outputs where the total sum of the outputs is equal to the input value amount. This design pattern is useful for simulating loyalty points, tokens, and more.

We present the solution:

```
  contract SuperAssetR201 {
      // Do NOT provide a constructor as that will add unnecessary OP_0 OP_
      bytes assetId;                      // Asset identifier
      Ripemd160 currentOwnerAddress;      // Current owner is the second pu
      // Notice that "disallowAssetIdNotUsed" is not used below. The reason
      bytes disallowAssetIdNotUsed;       // Disallow Asset from being used
      static const int MAX_RECEIVE = 6;

      static function buildOutputVector(
          int amount,
          bytes assetId,
          bytes address,
          bytes outputScriptLen,
          bytes lockingScriptCodePart
      ): bytes {
          return
              num2bin(amount, 8) +
              hash256(
                  outputScriptLen +

                  // OP_PUSHINPUTREF <assetId>
                  b'd0' + assetId +

                  // Address/owner (20 bytes)
                  b'14' + address +

                  lockingScriptCodePart
              ) +
            // One color for the output
            b'01000000' +
            hash256(assetId);
    }

    public function mint(SigHashPreimage txPreimage, int amount) {
        require(amount > 0);
        require(this.assetId == num2bin(0, 36));
        bytes lockingScript = SigHash.scriptCode(txPreimage);
        require(
                hash256(
                    num2bin(amount, 8) +

                    b'fd' + num2bin(len(lockingScript), 2) +

                    // OP_PUSHINPUTREF <assetId>
                    b'd0' + txPreimage[68 : 104]+

                    // Address/owner (20 bytes)
                    b'14' + this.currentOwnerAddress +

                    // Get entire locking script after the push vars
                    // 95 = 1+36 + 1+20
                    lockingScript[58 : ]
                )
                ==
                txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8] //
        );
        require(Tx.checkPreimageOpt_(txPreimage));
    }

    public function transfer(SigHashPreimage txPreimage, Ripemd160[6] rec
        require(hash160(unlockKey) == this.currentOwnerAddress);
        require(checkSig(senderSig, unlockKey));

        int expectedRefColorSum = 1337;         // Placeholder for OP_INP
        int actualAccumulatedRefColorSum = 0;   // Used for counting the
        bool break = false;
        bytes expectedOutputVector = b'';
        bytes lockingScript = SigHash.scriptCode(txPreimage);
        // Length of the output script
        bytes outputScriptLen = b'fd' + num2bin(len(lockingScript), 2);
        bytes lockingScriptCodePart = lockingScript[58 : ];
        loop (MAX_RECEIVE) : i {
            if (!break) {
                if (amounts[i] <= 0) {
                    break = true;
                } else {
                    // There is a valid recipient...
                    // Get entire locking script after the push vars
                    // 58 = 1+36 + 1+20
                    expectedOutputVector += SuperAssetR201.buildOutputVec
                    actualAccumulatedRefColorSum += amounts[i];
                }
            }
        }
        require(expectedRefColorSum > 0 && expectedRefColorSum == actualA
        require(
            hash256(expectedOutputVector + otherOutputs)
            ==
            // hashOutputsHashes
            txPreimage[len(txPreimage) - 72 : len(txPreimage) - 40]
        );
    }

    public function melt(SigHashPreimage txPreimage, Sig senderSig, PubKe
        require(hash160(unlockKey) == this.currentOwnerAddress);
        require(checkSig(senderSig, unlockKey));
        // Ensure one of the outputs is unspendable OP_RETURN and uses th
        // OP_DISALLOWPUSHINPUTREF and OP_DISALLOWPUSHINPUTREFSIBLING whi
        // the reference anymore, thereby ending the ability to carry on
        require(
            hash256(
                // Hardcode len '4b' is 57 bytes (1 + 1 + 36 + 1 + 36)
                b'0000000000000004b6ad2' + this.assetId +  b'd3' + this.
            )
            ==
            txPreimage[len(txPreimage) - 40 : len(txPreimage) - 8]
        );
        require(Tx.checkPreimageOpt_(txPreimage));
    }
}
```

Blockchain Network Details

Radiant is a peer-to-peer digital asset system with unbounded scaling as a UTXO-based blockchain with all the flexibility and power of account-based blockchains.

> **Network Name:** Radiant
> **Network Abbreviation:** RAD
> **Mining Algorithm:** SHA512/256 Proof-of-work
> **Block Time:** 5 minutes
> **Initial Block Size:** 128 MB, designed to achieve 10GB+
> **Block Reward Schedule:** 50,000 RAD per block
> **Block Reward Halvening:** 2 years
> **Maximum Supply:** 21,000,000,000 RAD
> **Decimal Places:** 8
> **Launch Date:** 2022-06-21 02:42 UTC

Conclusion

We have proposed a system for digital asset management without relying on trust. We started with the basic blockchain construction of coins made from digital signatures, which provides strong control of ownership. From the needed rules and incentives, we introduced two novel methods for authenticating and tracking digital assets in constant O(1) time and space. Both methods independently provide a general induction proof system which can encode any possible digital asset configuratio. The system is Turing Complete within and across transaction boundaries, with unbounded scale, and never any need for secondary layers. Additionally we have presented three contract design patterns: Non-Fungible Token (NFT), Fungible Token (FT) and Account which emulating account based blockchains, using the UTXO based processing model. Radiant is a breakthrough design which provides the performance and paralellism benefits of an unspent transaction output (UTXO) blockchain, and with the programming sophistication of account-based blockchains, while maintaining ultra low fees and unbounded scale.

References

[1] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System" https://bitcoin.org/bitcoin.pdf, 2009.