```
BIP: 140
Layer: Consensus (soft fork)
Title: Normalized TXID
Author: Christian Decker <decker.christian@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0140
Status: Rejected
Type: Standards Track
Created: 2015-10-14
License: PD
```

Abstract

This BIP describes the use of normalized transaction IDs (NTXIDs) in order to eliminate transaction malleability, both in the third-party modification scenario as well as the participant modification scenario. The transaction ID is normalized by removing the signature scripts from transactions before computing its hash. The normalized transaction hashes are then used during the signature creation and signature verification of dependent transactions.

Motivation

Transaction malleability refers to the fact that transactions can be modified, either by one of the signers by re-signing the transaction or a third-party by modifying the signature representation. This is a problem since any modification to the serialized representation also changes the hash of the transaction, which is used by spending transaction to reference the funds that are being transferred. If a transaction is modified and later confirmed by ending up in the blockchain all transactions that depended on the original transaction are no longer valid, and thus orphaned.

BIPs 62[1] and 66[2] alleviate the problem of third-party modification by defining a canonical representation of the signatures. However, checking the canonical representation is complex and may not eliminate all sources of third-party malleability. Furthermore, these BIPs do not address modifications by one of the signers, i.e., re-signing the transaction, because signers can produce any number of signatures due to the random parameter in ECDSA.

This proposal eliminates malleability by removing the malleable signatures from the hash used to reference the outputs spent by a transaction. The new hash used to reference an output is called the *normalized transaction ID*. The integrity of all data that is used to reference the output is guaranteed by the signature itself, and any modification that would change the normalized transaction ID would also invalidate the signature itself.

[1] BIP 62 - Dealing with malleability
[2] BIP 66 - Strict DER signatures

Besides eliminating transaction malleability as a source of problems it also allows the use of transaction templates. Transaction templates simplify higher level protocols and allows new uses. They allow an unsigned template transaction to be used as a basis for a sequence of transaction and only once the sequence matches the signers' expectations they provide the necessary signatures for the template to become valid, thus opting in to the sequence.

Specification

The use of normalized transaction IDs is introduced as a softfork. The specification is divided into three parts:

- Computation of the normalized transaction ID
- Introduction of a new extensible signature verification opcode to enable softfork deployment
- Changes to the UTXO tracking to enable normalized transaction ID lookup

Normalized Transaction ID computation

In order to calculate the normalized transaction ID, the signature script is stripped from each input of the transaction of non-coinbase transactions and each input is normalized. Stripping the signature script is achieved by setting the script's length to 0 and removing the `uchar[]` array from the `TxIn`.[3] Inputs are then normalized by replacing the hash of each previous transaction with its normalized version if available, i.e., the normalized hash of the previous transaction that created the output being spent in the current transaction. Version 1 transactions do not have a normalized transaction ID hence the non-normalized transaction ID is used for input normalization.

The normalized transaction ID is then computed as the double `SHA 256` hash of the normalized transaction matching the existing transaction ID computation. The normalized transaction ID remains unchanged even if the signatures of the transaction are replaced/malleated and describe a class of semantically identical transactions. In the following we use *transaction instance ID* to refer to the transaction ID computed on the transaction including signatures. Normalized transaction IDs for coinbase transactions are computed with the signature script in the coinbase input, in order to avoid hash collisions.

OP_CHECKSIGEX

This BIP introduces a new opcode `OP_CHECKSIGEX` which replaces `OP_NOP4`. `OP_CHECKSIGEX` subsumes `OP_CHECKSIGVERIFY` and `OP_CHECKMULTISIGVERIFY`, and extends them by accepting a new `VERSION` parameter. The version parameter is a single integer pushed onto the stack before invoking `OP_CHECKSIGEX` and is used to group and evolve future versions of signature checking opcodes.

[3] Protocol Specification: TX

When executed `OP_CHECKSIGEX` pops the version from the stack and then performs the signature check according to the specified version. If the verifying client does not support the specified version, i.e., the version was defined after the release of the client, the client must treat the `OP_CHECKSIGEX` as an `OP_NOP`.

Version 1 The first version of `OP_CHECKSIGEX` (`VERSION=1`) implements normalized transaction IDs and uses Schnorr signatures instead of the current ECDSA signatures.

Version 1 introduces the following new standard script format:

```
m {pubkey}...{pubkey} n v OP_CHECKSIGEX
```

with matching scriptSig format:

```
{signature}...{signature}
```

This is the standard *m-of-n* script defined in BIP 11 with an additional version parameter `v` and the new opcode. Singlesig transactions are encoded as *1-of-1* transactions.

The existing `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` have a bug[4] that pops one argument too many from the stack. This bug is not reproduced in the implementation of OP_CHECKSIGEX, so the canonical solution of pushing a dummy value onto the stack is not necessary.

The normalization is achieved by normalizing the transaction before computing the signaturehash, i.e., the hash that is signed. The transaction must be normalized by replacing all transaction IDs in the inputs by their normalized variants and stripping the signature scripts. The normalized transsction IDs are computed as described in the previous section. This normalization step is performed both when creating the signatures as well as when checking the signatures.

Tracking Normalized Transaction IDs

The transaction version is bumped to 2. The new version signals to clients receiving the transaction that they should track the normalized transaction ID along with the transaction instance ID in the unspent transaction output (UTXO) set. Upon receiving a version 2 transaction the client computes the normalized transaction ID, annotates the outputs with it, and adds them into the UTXO set indexed by the transaction instance ID as before. Transactions continue using the transaction instance ID to reference the outputs, but while checking the signature they may get normalized. All network messages continue to use the transaction instance ID to reference the transaction, specifically `inv`, `getdata`, `tx` and `block` messages still use transaction instance IDs, not the normalized transaction IDs.

[4] Developer Documentation - Multisig

Outputs created by version 1 transactions are not annotated with the normalized transaction ID, and when normalizing the hashes in transaction inputs referencing version 1 outputs are not modified.

Rationale

Normalization

Normalized transaction IDs are provably non-malleable since no data is included in the signaturehash whose integrity is not also proven in the signature, thus any modification causing the hash to change will also invalidate the signature. Normalized transactions are secure as they still use cryptographic hashes over all the semantic information of the transaction, i.e., the inputs, outputs and metadata, thus it is still computationally infeasible to cause a hash collision between transactions.

There are a number of advantages to using normalized transaction IDs:

- Like BIP 62 and BIP 66 it solves the problem of third-parties picking transactions out of the network, modifying them and reinjecting them.
- *m-of-n* multisig outputs are often used in higher level protocols[5][[http://lightning.network/lightning-network-paper.pdf%7CThe Bitcoin Lightning Network:

Scalable Off-Chain Instant Payments]]

in which several parties sign a transaction. Without normalized transaction IDs it is trivial for one party to re-sign a transaction, hence changing the transaction hash and invalidating any transaction built on top of its outputs. Normalized transaction IDs force the ID not to change, even if a party replaces its signature.

- Many higher level protocols build structures of transactions on top of multisig outputs that are not completely signed. This is currently not possible without one party holding a fully signed transaction and then calculating the ID. It is desirable to be able to build successive transactions without one party collecting all signatures, and thus possibly lock in funds unilaterally. Normalized transaction IDs allow the use of transaction templates, i.e., completely unsigned transactions upon which further transactions can be built, and only once every party is assured the structure matches its expectations it signs the template, thus validating the template.

The only occurrence in which transactions can still be modified unilaterally is in the case `SIGHASH_NONE`, `SIGHASH_SINGLE` or `SIGHASH_ANYONECANPAY` is used. This however is not problematic since in these cases the creator of the transaction explicitly allows modification.

In case of a transaction becoming invalid due to one of the inputs being malleated it is necessary to modify the spending transaction to reference the modified

[5] A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels

transaction ID. However, the signatures, which only use the normalized IDs, remain valid as long as the semantics of the funding transaction remain unchanged. An observer in the network may fix the transaction and reinject a corrected version.

Using version 2 for transactions is an explicit opt-in to the normalized ID tracking and a simple upgrade for existing clients. It avoids having to reprocess the entire blockchain and computing the normalized transaction IDs for existing outputs in the UTXO. This would be further complicated by having to recursively compute normalized transaction IDs down to the coinbase transactions which created the coins.

Tracking the normalized transaction IDs in the UTXO requires the storage of an additional hash per transaction whose outputs are not completely spent, which at 7,000,000 transactions with unspent outputs amounts to 224MB additional storage on disk.

The coinbase transactions have been checked for hash-collisions and no collisions were found except for the coinbase transactions in blocks at heights 91842 and 91880, which are known to be identical[6], and motivated the introduction of BIP 34.[7] Since coinbase transactions are invalid if transmitted outside of a block it is not possible to modify them on the fly and since they only mature after being included for a long time in the blockchain they are considered safe.

OP_CHECKSIGEX

The new opcode `OP_CHECKSIGEX` was introduced in order to allow the use of normalized transaction IDs as a softfork and in order to keep the number of `OP_NOP`s needed to a bare minimum, while enabling future soft-fork updates to the signing algorithms.

The additional argument containing the version can be pushed on the stack using a single byte up to version 16 (`OP_1` - `OP_16`), resulting in one byte overhead for this script type. Using the standard multisig format also for 1-of-1 transactions add an additional 2 bytes, however it also removes the bug requiring a dummy push, resulting in a single byte overhead. Furthermore, using Schnorr signatures instead of ECDSA brings a number of improvements that reduce the size of transactions (*m-of-m* is the same size as *1-of-1*) and increase verification speed (batch signature validation by summing up keys and signatures). The code is already in bitcoin/secp256k1 and can be merged in. We limited the description of this BIP to re-using BIP 11 style *m-of-n* scripts to keep it short, however Schnorr also allows a number of more complex applications which we defer to future BIPs.

Version 0 was intentionally skipped in order to guarantee that the top-most element before `OP_CHECKSIGEX` is non-zero. This is necessary to guarantee that

[6] BIP 30 - Duplicate transactions
[7] Block v2, Height in Coinbase

non-upgraded clients, which interpret `OP_CHECKSIGEX` as `OP_NOP4`, do not end up with a zero value on top of the stack after execution, which would be interpreted as script failure.

Impact

This is a softfork which replaces `OP_NOP4` with the new implementation of `OP_CHECKSIGEX`, as such the impact on the network is minimal. Wallets that do not implement this opcode will not be able to verify the validity of the scripts, however if transactions using `OP_CHECKSIGEX` are included in blocks they will accept them and track the inputs correctly. This is guaranteed since the transaction inputs still use the non-normalized transaction ID to reference the outputs to be claimed, hence non-upgraded wallets can still lookup the outputs and mark them as spent. Furthermore, clients that do not implement this BIP are unable to identify outputs using this script as their own, however upgrading and rescanning the blockchain will make them available.

See also

- BIP 62: Dealing with malleability
- BIP 66: Strict DER Signatures

References

Copyright