```
BIP: 116
Layer: Consensus (soft fork)
Title: MERKLEBRANCHVERIFY
Author: Mark Friedenbach <mark@friedenbach.org>
        Kalle Alm <kalle.alm@gmail.com>
        BtcDrak <btcdrak@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0116
Status: Draft
Type: Standards Track
Created: 2017-08-25
License: CC-BY-SA-4.0
License-Code: MIT
```

Abstract

A general approach to bitcoin contracts is to fully enumerate the possible spending conditions and then program verification of these conditions into a single script. At redemption, the spending condition used is explicitly selected, e.g. by pushing a value on the witness stack which cascades through a series if if/else constructs.

This approach has significant downsides, such as requiring all program pathways to be visible in the scriptPubKey or redeem script, even those which are not used at validation. This wastes space on the block chain, restricts the size of possible scripts due to push limits, and impacts both privacy and fungibility as details of the contract can often be specific to the user.

This BIP proposes a new soft-fork upgradeable opcode, MERKLEBRANCHVERIFY, which allows script writers to commit to a set of data elements and have one or more of these elements be provided at redemption without having to reveal the entire set. As these data elements can be used to encode policy, such as public keys or validation subscripts, the MERKLEBRANCHVERIFY opcode can be used to overcome these limitations of existing bitcoin script.

Copyright

Specification

MERKLEBRANCHVERIFY redefines the existing NOP4 opcode. When executed, if any of the following conditions are true, the script interpreter will terminate with an error:

1. the stack contains less than three (3) items;
2. the first item on the stack is more than 2 bytes;

3. the first item on the stack, interpreted as an integer, N, is negative or not minimally encoded;
4. the second item on the stack is not exactly 32 bytes;
5. the third item on the stack is not a serialized Merkle tree inclusion proof as specified by BIP98[1] and requiring exactly `floor(N/2)` VERIFY hashes; or
6. the remainder of the stack contains less than `floor(N/2)` additional items, together referred to as the input stack elements.

If the low-order bit of N is clear, `N&1 == 0`, each input stack element is hashed using double-SHA256. Otherwise, each element must be exactly 32 bytes in length and are interpreted as serialized hashes. These are the VERIFY hashes.

If the fast Merkle root computed from the Merkle tree inclusion proof, the third item on the stack, with the VERIFY hashes in the order as presented on the stack, from top to bottom, does not exactly match the second item on the stack, the script interpreter will terminate with an error.

Otherwise, script execution will continue as if a NOP had been executed.

Motivation

Although BIP16 (Pay to Script Hash)[2] and BIP141 (Segregated Witness)[3] both allow the redeem script to be kept out of the scriptPubKey and therefore out of the UTXO set, the entire spending conditions for a coin must nevertheless be revealed when that coin is spent. This includes execution pathways or policy conditions which end up not being needed by the redemption. Not only is it inefficient to require this unnecessary information to be present on the blockchain, albeit in the witness, it also impacts privacy and fungibility as some unused script policies may be identifying. Using a Merkle hash tree to commit to the policy options, and then only forcing revelation of the policy used at redemption minimizes this information leakage.

Using Merkle hash trees to commit to policy allows for considerably more complex contracts than would would otherwise be possible, due to various built-in script size and runtime limitations. With Merkle commitments to policy these size and runtime limitations constrain the complexity of any one policy that can be used rather than the sum of all possible policies.

Rationale

The MERKLEBRANCHVERIFY opcode uses fast Merkle hash trees as specified by BIP98[1] rather than the construct used by Satoshi for committing transactions to the block header as the later has a known vulnerability relating to duplicate entries that introduces a source of malleability to downstream protocols[4]. A source of malleability in Merkle proofs could potentially lead to spend vulnerabilities in protocols that use MERKLEBRANCHVERIFY. For example, a compact 2-of-N policy could be written by using MERKLEBRANCHVERIFY

to prove that two keys are extracted from the same tree, one at a time, then checking the proofs for bitwise equality to make sure the same entry wasn't used twice. With the vulnerable Merkle tree implementation there are privledged positions in unbalanced Merkle trees that allow multiple proofs to be constructed for the same, single entry.

BIP141 (Segregated Witness)[3] provides support for a powerful form of script upgrades called script versioning, which is able to achieve the sort of upgrades which would previously have been hard-forks. If script versioning were used for deployment then MERKLEBRANCHVERIFY could be written to consume its inputs, which would provide a small 2-byte savings for many anticipated use cases. However the more familiar NOP-expansion soft-fork mechanism used by BIP65 (CHECKLOCKTIMEVERIFY)[5] and BIP112 (CHECKSEQUENCEVERIFY)[6] was chosen over script versioning for the following two reasons:

1. **Infrastructure compatibility.** Using soft-fork NOP extensions allows MERKLEBRANCHVERIFY to be used by any existing software able to consume custom scripts, and results in standard P2SH or P2WSH-nested-in-P2SH addresses without the need for BIP143[7] signing code. This allows MERKLEBRANCHVERIFY to be used immediately by services that need it rather than wait on support for script versioning and/or BIP-143[7] signatures in tools and libraries.
2. **Delayed decision on script upgrade protocol.** There are unresolved issues with respect to how script versioning should be used for future script upgrades. There are only 16 available script versions reserved for future use, and so they should be treated as a scarce resource. Additionally, script feature versioning should arguably be specified in the witness and the BIP141 script versioning only be used to specify the structure of the witness, however no such protocol exists as of yet. Using the NOP-expansion space prevents MERKLEBRANCHVERIFY from being stalled due to waiting on script upgrade procedure to be worked out, while making use of expansion space that is already available.

The MERKLEBRANCHVERIFY opcode allows for VERIFY hashes to be presented directly, or calculated from the leaf values using double-SHA256. In most cases the latter approach is expected to be used so that the leaf value(s) can be used for both branch validation and other purposes without any explicit preprocessing. However allowing already-calculated hash values as inputs enables using chained MERKLEBRANCHVERIFY opcodes to verify branches of trees with proofs large enough that they would not fit in the 520 byte script push limitation. As specified, a 30-branch path can be verified by proving the path from the leaf to the 15th interior node as the 'root', then proving that node's hash to be a child of the actual Merkle tree root hash. Validation of a 256-branch path (e.g. a binary prefix tree with a hash value as key) would require 18 chained validations, which would fit within current script limitations.

Applications

1-of-N for large N

Here is a redeem script that allows a coin to be spent by any key from a large set, without linear scaling in script size:

```
 redeemScript:  2 MERKLEBRANCHVERIFY 2DROP DROP CHECKSIG
witness:
```

The redeem script looks very similar to the standard pay-to-pubkey-hash, except instead of showing that the pubkey's hash is the same as the commitment given, we demonstrate that the pubkey is one of potentially many pubkeys included in the Merkle tree committed to in the redeem script. The low-order bit of the first parameter, 2, is clear, meaning that there is one input (`(2>>1) == 1`), the serialized pubkey, and its VERIFY hash needs to be calculated by MERKLEBRANCHVERIFY using double-SHA256.

Honeypots

As described by Pieter Wuille[8] the 1-of-N scheme is particularly useful for constructing honeypots. The desire is to put a large bounty on a server, larger than the value of the server itself so that if the server is compromised it is highly likely that the hacker will claim the bitcoin, thereby revealing the intrusion. However if there are many servers, e.g. 1,000, it becomes excessively expensive to lock up separate bounties for each server. It would be desirable if the same bounty was shared across multiple servers in such a way that the spend would reveal which server was compromised.

This is accomplished by generating 1,000 different keys, building a hash tree of these public keys, and placing each key and associated Merkle path on separate servers. When the honeypot is claimed, the (previous) owner of the coins can tell which server was compromised from the key and path used to claim the funds.

Implementation

An implementation of this BIP, including both consensus code updates and tests is available at the following Github repository:

1

Deployment

This BIP will be deployed by BIP8 (Version bits with lock-in by height)[9] with the name "merklebranchverify" and using bit 2.

For Bitcoin mainnet, the BIP8 startheight will be at height M to be determined and BIP8 timeout activation will occur on height M + 50,400 blocks.

For Bitcoin testnet, the BIP8 startheight will be at height T to be determined and BIP8 timeout activation will occur on height T + 50,400 blocks.

We note that DISCOURAGE_UPGRADABLE_NOPS means that transactions which use this feature are already considered non-standard by the rules of the network, making deployment easier than was the case with, for example, with BIP68 (Relative lock-time using consensus-enforced sequence numbers)[9].

Compatibility

Old clients will consider the OP_MERKLEBRANCHVERIFY as a NOP and ignore it. Proof will not be verified, but the transaction will be accepted.

References

[1] BIP98: Fast Merkle Trees (Consensus layer)

[2] BIP16: Pay to Script Hash

[3] BIP141: Segregated Witness (Consensus layer)

[4] National Vulnerability Database: CVE-2012-2459

[5] BIP65: OP_CHECKLOCKTIMEVERIFY

[6] BIP112: CHECKSEQUENCEVERIFY

[7] BIP143: Transaction Signature Verification for Version 0 Witness Program

[8] Multisig on steroids using tree signatures

[9] BIP68: Relative lock-time using consensus-enforced sequence numbers