```
BIP: 65
Layer: Consensus (soft fork)
Title: OP_CHECKLOCKTIMEVERIFY
Author: Peter Todd <pete@petertodd.org>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0065
Status: Final
Type: Standards Track
Created: 2014-10-01
License: PD
```

Abstract

This BIP describes a new opcode (OP_CHECKLOCKTIMEVERIFY) for the Bitcoin scripting system that allows a transaction output to be made unspendable until some point in the future.

Summary

CHECKLOCKTIMEVERIFY redefines the existing NOP2 opcode. When executed, if any of the following conditions are true, the script interpreter will terminate with an error:

- the stack is empty; or
- the top item on the stack is less than 0; or
- the lock-time type (height vs. timestamp) of the top stack item and the nLockTime field are not the same; or
- the top stack item is greater than the transaction's nLockTime field; or
- the nSequence field of the txin is 0xffffffff;

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

The nLockTime field in a transaction prevents the transaction from being mined until either a certain block height, or block time, has been reached. By comparing the argument to CHECKLOCKTIMEVERIFY against the nLockTime field, we indirectly verify that the desired block height or block time has been reached; until that block height or block time has been reached the transaction output remains unspendable.

Motivation

The nLockTime field in transactions can be used to prove that it is *possible* to spend a transaction output in the future, by constructing a valid transaction spending that output with the nLockTime field set.

However, the nLockTime field can't prove that it is *impossible* to spend a transaction output until some time in the future, as there is no way to know if a valid signature for a different transaction spending that output has been created.

Escrow

If Alice and Bob jointly operate a business they may want to ensure that all funds are kept in 2-of-2 multisig transaction outputs that require the co-operation of both parties to spend. However, they recognise that in exceptional circumstances such as either party getting "hit by a bus" they need a backup plan to retrieve the funds. So they appoint their lawyer, Lenny, to act as a third-party.

With a standard 2-of-3 CHECKMULTISIG at any time Lenny could conspire with either Alice or Bob to steal the funds illegitimately. Equally Lenny may prefer not to have immediate access to the funds to discourage bad actors from attempting to get the secret keys from him by force.

However, with CHECKLOCKTIMEVERIFY the funds can be stored in scriptPubKeys of the form:

```
   IF
<now + 3 months> CHECKLOCKTIMEVERIFY DROP
<Lenny's pubkey> CHECKSIGVERIFY
1
ELSE
2
ENDIF
<Alice's pubkey> <Bob's pubkey> 2 CHECKMULTISIG
```

At any time the funds can be spent with the following scriptSig:

```
   0 <Alice's signature> <Bob's signature> 0
```

After 3 months have passed Lenny and one of either Alice or Bob can spend the funds with the following scriptSig:

```
   0 <Alice/Bob's signature> <Lenny's signature> 1
```

Non-interactive time-locked refunds

There exist a number of protocols where a transaction output is created that requires the co-operation of both parties to spend the output. To ensure the failure of one party does not result in the funds becoming lost, refund transactions are setup in advance using nLockTime. These refund transactions need to be created interactively, and additionally, are currently vulnerable to transaction malleability. CHECKLOCKTIMEVERIFY can be used in these protocols, replacing the interactive setup with a non-interactive setup, and additionally, making transaction malleability a non-issue.

Two-factor wallets Services like GreenAddress store bitcoins with 2-of-2 multisig scriptPubKey's such that one keypair is controlled by the user, and the other keypair is controlled by the service. To spend funds the user uses locally installed wallet software that generates one of the required signatures, and then uses a 2nd-factor authentication method to authorize the service to create the

second SIGHASH_NONE signature that is locked until some time in the future and sends the user that signature for storage. If the user needs to spend their funds and the service is not available, they wait until the nLockTime expires.

The problem is there exist numerous occasions the user will not have a valid signature for some or all of their transaction outputs. With CHECKLOCKTIMEVERIFY rather than creating refund signatures on demand scriptPubKeys of the following form are used instead:

```
   IF
CHECKSIGVERIFY
ELSE
CHECKLOCKTIMEVERIFY DROP
ENDIF
CHECKSIG
```

Now the user is always able to spend their funds without the co-operation of the service by waiting for the expiry time to be reached.

Payment Channels Jeremy Spilman style payment channels first setup a deposit controlled by 2-of-2 multisig, tx1, and then adjust a second transaction, tx2, that spends the output of tx1 to payor and payee. Prior to publishing tx1 a refund transaction is created, tx3, to ensure that should the payee vanish the payor can get their deposit back. The process by which the refund transaction is created is currently vulnerable to transaction malleability attacks, and additionally, requires the payor to store the refund. Using the same scriptPubKey form as in the Two-factor wallets example solves both these issues.

Trustless Payments for Publishing Data

The PayPub protocol makes it possible to pay for information in a trustless way by first proving that an encrypted file contains the desired data, and secondly crafting scriptPubKeys used for payment such that spending them reveals the encryption keys to the data. However the existing implementation has a significant flaw: the publisher can delay the release of the keys indefinitely.

This problem can be solved interactively with the refund transaction technique; with CHECKLOCKTIMEVERIFY the problem can be non-interactively solved using scriptPubKeys of the following form:

```
   IF
HASH160 <Hash160(encryption key)> EQUALVERIFY
CHECKSIG
ELSE
CHECKLOCKTIMEVERIFY DROP
CHECKSIG
ENDIF
```

The buyer of the data is now making a secure offer with an expiry time. If the publisher fails to accept the offer before the expiry time is reached the buyer can cancel the offer by spending the output.

Proving sacrifice to miners' fees

Proving the sacrifice of some limited resource is a common technique in a variety of cryptographic protocols. Proving sacrifices of coins to mining fees has been proposed as a *universal public good* to which the sacrifice could be directed, rather than simply destroying the coins. However doing so is non-trivial, and even the best existing technqiue - announce-commit sacrifices - could encourage mining centralization. CHECKLOCKTIMEVERIFY can be used to create outputs that are provably spendable by anyone (thus to mining fees assuming miners behave optimally and rationally) but only at a time sufficiently far into the future that large miners can't profitably sell the sacrifices at a discount.

Freezing Funds

In addition to using cold storage, hardware wallets, and P2SH multisig outputs to control funds, now funds can be frozen in UTXOs directly on the blockchain. With the following scriptPubKey, nobody will be able to spend the encumbered output until the provided expiry time. This ability to freeze funds reliably may be useful in scenarios where reducing duress or confiscation risk is desired.

```
CHECKLOCKTIMEVERIFY DROP DUP HASH160  EQUALVERIFY CHECKSIG
```

Replacing the nLockTime field entirely

As an aside, note how if the SignatureHash() algorithm could optionally cover part of the scriptSig the signature could require that the scriptSig contain CHECKLOCKTIMEVERIFY opcodes, and additionally, require that they be executed. (the CODESEPARATOR opcode came very close to making this possible in v0.1 of Bitcoin) This per-signature capability could replace the per-transaction nLockTime field entirely as a valid signature would now be the proof that a transaction output *can* be spent.

Detailed Specification

Refer to the reference implementation, reproduced below, for the precise semantics and detailed rationale for those semantics.

```
   case OP_NOP2:
{
// CHECKLOCKTIMEVERIFY
//
// (nLockTime -- nLockTime )

if (!(flags & SCRIPT_VERIFY_CHECKLOCKTIMEVERIFY))
```

```
break; // not enabled; treat as a NOP

if (stack.size() < 1)
return false;

// Note that elsewhere numeric opcodes are limited to
// operands in the range -2**31+1 to 2**31-1, however it is
// legal for opcodes to produce results exceeding that
// range. This limitation is implemented by CScriptNum's
// default 4-byte limit.
//
// If we kept to that limit we'd have a year 2038 problem,
// even though the nLockTime field in transactions
// themselves is uint32 which only becomes meaningless
// after the year 2106.
//
// Thus as a special case we tell CScriptNum to accept up
// to 5-byte bignums, which are good until 2**32-1, the
// same limit as the nLockTime field itself.
const CScriptNum nLockTime(stacktop(-1), 5);

// In the rare event that the argument may be < 0 due to
// some arithmetic being done first, you can always use
// 0 MAX CHECKLOCKTIMEVERIFY.
if (nLockTime < 0)
return false;

// There are two types of nLockTime: lock-by-blockheight
// and lock-by-blocktime, distinguished by whether
// nLockTime < LOCKTIME_THRESHOLD.
//
// We want to compare apples to apples, so fail the script
// unless the type of nLockTime being tested is the same as
// the nLockTime in the transaction.
if (!(
(txTo.nLockTime <  LOCKTIME_THRESHOLD && nLockTime <  LOCKTIME_THRESHOLD) ||
(txTo.nLockTime >= LOCKTIME_THRESHOLD && nLockTime >= LOCKTIME_THRESHOLD)
))
return false;

// Now that we know we're comparing apples-to-apples, the
// comparison is a simple numeric one.
if (nLockTime > (int64_t)txTo.nLockTime)
return false;

// Finally the nLockTime feature can be disabled and thus
```

```
// CHECKLOCKTIMEVERIFY bypassed if every txin has been
// finalized by setting nSequence to maxint. The
// transaction would be allowed into the blockchain, making
// the opcode ineffective.
//
// Testing if this vin is not final is sufficient to
// prevent this condition. Alternatively we could test all
// inputs, but testing just this input minimizes the data
// required to prove correct CHECKLOCKTIMEVERIFY execution.
if (txTo.vin[nIn].IsFinal())
return false;

break;

}
```

https://github.com/petertodd/bitcoin/commit/ab0f54f38e08ee1e50ff72f801680ee84d0f1bf4

Deployment

We reuse the double-threshold IsSuperMajority() switchover mechanism used in BIP66 with the same thresholds, but for nVersion = 4. The new rules are in effect for every block (at height H) with nVersion = 4 and at least 750 out of 1000 blocks preceding it (with heights H-1000..H-1) also have nVersion >= 4. Furthermore, when 950 out of the 1000 blocks preceding a block do have nVersion >= 4, nVersion < 4 blocks become invalid, and all further blocks enforce the new rules.

It should be noted that BIP9 involves permanently setting a high-order bit to 1 which results in nVersion >= all prior IsSuperMajority() soft-forks and thus no bits in nVersion are permanently lost.

SPV Clients

While SPV clients are (currently) unable to validate blocks in general, trusting miners to do validation for them, they are able to validate block headers and thus can validate a subset of the deployment rules. SPV clients should reject nVersion < 4 blocks if 950 out of 1000 preceding blocks have nVersion >= 4 to prevent false confirmations from the remaining 5% of non-upgraded miners when the 95% threshold has been reached.

Credits

Thanks goes to Gregory Maxwell for suggesting that the argument be compared against the per-transaction nLockTime, rather than the current block height and time.

References

PayPub

- https://github.com/unsystem/paypub

Jeremy Spilman Payment Channels

- https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2013-April/002433.html

Implementations

Python / python-bitcoinlib

- https://github.com/petertodd/checklocktimeverify-demos

JavaScript / Node.js / bitcore

- https://github.com/mruddy/bip65-demos

Copyright