```
BIP: 112
Layer: Consensus (soft fork)
Title: CHECKSEQUENCEVERIFY
Author: BtcDrak <btcdrak@gmail.com>
        Mark Friedenbach <mark@friedenbach.org>
        Eric Lombrozo <elombrozo@gmail.com>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0112
Status: Final
Type: Standards Track
Created: 2015-08-10
License: PD
```

Abstract

This BIP describes a new opcode (CHECKSEQUENCEVERIFY) for the Bitcoin scripting system that in combination with BIP 68 allows execution pathways of a script to be restricted based on the age of the output being spent.

Summary

CHECKSEQUENCEVERIFY redefines the existing NOP3 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 top item on the stack has the disable flag (1 « 31) unset; and
  - the transaction version is less than 2; or
  - the transaction input sequence number disable flag (1 « 31) is set; or
  - the relative lock-time type is not the same; or
  - the top stack item is greater than the transaction input sequence (when masked according to the BIP68);

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

BIP 68 prevents a non-final transaction from being selected for inclusion in a block until the corresponding input has reached the specified age, as measured in block-height or block-time. By comparing the argument to CHECKSEQUENCEVERIFY against the nSequence field, we indirectly verify a desired minimum age of the the output being spent; until that relative age has been reached any script execution pathway including the CHECKSEQUENCEVERIFY will fail to validate, causing the transaction not to be selected for inclusion in a block.

Motivation

BIP 68 repurposes the transaction nSequence field meaning by giving sequence numbers new consensus-enforced semantics as a relative lock-time. However, there is no way to build Bitcoin scripts to make decisions based on this field.

By making the nSequence field accessible to script, it becomes possible to construct code pathways that only become accessible some minimum time after proof-of-publication. This enables a wide variety of applications in phased protocols such as escrow, payment channels, or bidirectional pegs.

Contracts With Expiration Deadlines

Escrow with Timeout An escrow that times out automatically 30 days after being funded can be established in the following way. Alice, Bob and Escrow create a 2-of-3 address with the following redeemscript.

```
   IF
2 <Alice's pubkey> <Bob's pubkey> <Escrow's pubkey> 3 CHECKMULTISIG
ELSE
"30d" CHECKSEQUENCEVERIFY DROP
<Alice's pubkey> CHECKSIG
ENDIF
```

At any time funds can be spent using signatures from any two of Alice, Bob or the Escrow.

After 30 days Alice can sign alone.

The clock does not start ticking until the payment to the escrow address confirms.

Retroactive Invalidation

In many instances, we would like to create contracts that can be revoked in case of some future event. However, given the immutable nature of the blockchain, it is practically impossible to retroactively invalidate a previous commitment that has already confirmed. The only mechanism we really have for retroactive invalidation is blockchain reorganization which, for fundamental security reasons, is designed to be very hard and very expensive to do.

Despite this limitation, we do have a way to provide something functionally similar to retroactive invalidation while preserving irreversibility of past commitments using CHECKSEQUENCEVERIFY. By constructing scripts with multiple branches of execution where one or more of the branches are delayed we provide a time window in which someone can supply an invalidation condition that allows the output to be spent, effectively invalidating the would-be delayed branch and potentially discouraging another party from broadcasting the transaction in the first place. If the invalidation condition does not occur before the timeout, the delayed branch becomes spendable, honoring the original contract.

Some more specific applications of this idea:

Hash Time-Locked Contracts Hash Time-Locked Contracts (HTLCs) provide a general mechanism for off-chain contract negotiation. An execution pathway can be made to require knowledge of a secret (a hash preimage) that can be presented within an invalidation time window. By sharing the secret it is possible to guarantee to the counterparty that the transaction will never be broadcast since this would allow the counterparty to claim the output immediately while one would have to wait for the time window to pass. If the secret has not been shared, the counterparty will be unable to use the instant pathway and the delayed pathway must be used instead.

Bidirectional Payment Channels Scriptable relative locktime provides a predictable amount of time to respond in the event a counterparty broadcasts a revoked transaction: Absolute locktime necessitates closing the channel and reopen it when getting close to the timeout, whereas with relative locktime, the clock starts ticking the moment the transactions confirms in a block. It also provides a means to know exactly how long to wait (in number of blocks) before funds can be pulled out of the channel in the event of a noncooperative counterparty.

Lightning Network The lightning network extends the bidirectional payment channel idea to allow for payments to be routed over multiple bidirectional payment channel hops.

These channels are based on an anchor transaction that requires a 2-of-2 multisig from Alice and Bob, and a series of revocable commitment transactions that spend the anchor transaction. The commitment transaction splits the funds from the anchor between Alice and Bob and the latest commitment transaction may be published by either party at any time, finalising the channel.

Ideally then, a revoked commitment transaction would never be able to be successfully spent; and the latest commitment transaction would be able to be spent very quickly.

To allow a commitment transaction to be effectively revoked, Alice and Bob have slightly different versions of the latest commitment transaction. In Alice's version, any outputs in the commitment transaction that pay Alice also include a forced delay, and an alternative branch that allows Bob to spend the output if he knows that transaction's revocation code. In Bob's version, payments to Bob are similarly encumbered. When Alice and Bob negotiate new balances and new commitment transactions, they also reveal the old revocation code, thus committing to not relaying the old transaction.

A simple output, paying to Alice might then look like:

```
   HASH160  EQUAL
IF
```

```
<Bob's pubkey>
ELSE
"24h" CHECKSEQUENCEVERIFY DROP
<Alice's pubkey>
ENDIF
CHECKSIG
```

This allows Alice to publish the latest commitment transaction at any time and spend the funds after 24 hours, but also ensures that if Alice relays a revoked transaction, that Bob has 24 hours to claim the funds.

With CHECKLOCKTIMEVERIFY, this would look like:

```
   HASH160  EQUAL
IF
<Bob's pubkey>
ELSE
"2015/12/15" CHECKLOCKTIMEVERIFY DROP
<Alice's pubkey>
ENDIF
CHECKSIG
```

This form of transaction would mean that if the anchor is unspent on 2015/12/16, Alice can use this commitment even if it has been revoked, simply by spending it immediately, giving no time for Bob to claim it.

This means that the channel has a deadline that cannot be pushed back without hitting the blockchain; and also that funds may not be available until the deadline is hit. CHECKSEQUENCEVERIFY allows you to avoid making such a tradeoff.

Hashed Time-Lock Contracts (HTLCs) make this slightly more complicated, since in principle they may pay either Alice or Bob, depending on whether Alice discovers a secret R, or a timeout is reached, but the same principle applies -- the branch paying Alice in Alice's commitment transaction gets a delay, and the entire output can be claimed by the other party if the revocation secret is known. With CHECKSEQUENCEVERIFY, a HTLC payable to Alice might look like the following in Alice's commitment transaction:

```
   HASH160 DUP  EQUAL
IF
"24h" CHECKSEQUENCEVERIFY
2DROP
<Alice's pubkey>
ELSE
EQUAL
NOTIF
"2015/10/20 10:33" CHECKLOCKTIMEVERIFY DROP
ENDIF
<Bob's pubkey>
```

```
ENDIF
CHECKSIG
```

and correspondingly in Bob's commitment transaction:

```
  HASH160 DUP  EQUAL
SWAP  EQUAL ADD
IF
<Alice's pubkey>
ELSE
"2015/10/20 10:33" CHECKLOCKTIMEVERIFY
"24h" CHECKSEQUENCEVERIFY
2DROP
<Bob's pubkey>
ENDIF
CHECKSIG
```

Note that both CHECKSEQUENCEVERIFY and CHECKLOCKTIMEVERIFY are used in the final branch of above to ensure Bob cannot spend the output until after both the timeout is complete and Alice has had time to reveal the revocation secret.

See the Deployable Lightning paper.

2-Way Pegged Sidechains The 2-way pegged sidechain requires a new REORGPROOFVERIFY opcode, the semantics of which are outside the scope of this BIP. CHECKSEQUENCEVERIFY is used to make sure that sufficient time has passed since the return peg was posted to publish a reorg proof:

```
   IF
lockTxHeight  nlocktxOut [] reorgBounty Hash160(<...>)  REORGPROOFVERIFY
ELSE
withdrawLockTime CHECKSEQUENCEVERIFY DROP HASH160 p2shWithdrawDest EQUAL
ENDIF
```

Specification

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

```
/* Below flags apply in the context of BIP 68 */
/* If this flag set, CTxIn::nSequence is NOT interpreted as a
 * relative lock-time. */
static const uint32_t SEQUENCE_LOCKTIME_DISABLE_FLAG = (1 << 31);

/* If CTxIn::nSequence encodes a relative lock-time and this flag
 * is set, the relative lock-time has units of 512 seconds,
 * otherwise it specifies blocks with a granularity of 1. */
static const uint32_t SEQUENCE_LOCKTIME_TYPE_FLAG = (1 << 22);
```

```
/* If CTxIn::nSequence encodes a relative lock-time, this mask is
 * applied to extract that lock-time from the sequence field. */
static const uint32_t SEQUENCE_LOCKTIME_MASK = 0x0000ffff;

case OP_NOP3:
{
    if (!(flags & SCRIPT_VERIFY_CHECKSEQUENCEVERIFY)) {
        // not enabled; treat as a NOP3
        if (flags & SCRIPT_VERIFY_DISCOURAGE_UPGRADABLE_NOPS) {
            return set_error(serror, SCRIPT_ERR_DISCOURAGE_UPGRADABLE_NOPS);
        }
        break;
    }

    if (stack.size() < 1)
       return set_error(serror, SCRIPT_ERR_INVALID_STACK_OPERATION);

    // 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.
    //
    // Thus as a special case we tell CScriptNum to accept up
    // to 5-byte bignums, which are good until 2**39-1, well
    // beyond the 2**32-1 limit of the nSequence field itself.
    const CScriptNum nSequence(stacktop(-1), fRequireMinimal, 5);

    // In the rare event that the argument may be < 0 due to
    // some arithmetic being done first, you can always use
    // 0 MAX CHECKSEQUENCEVERIFY.
    if (nSequence < 0)
        return set_error(serror, SCRIPT_ERR_NEGATIVE_LOCKTIME);

    // To provide for future soft-fork extensibility, if the
    // operand has the disabled lock-time flag set,
    // CHECKSEQUENCEVERIFY behaves as a NOP.
    if ((nSequence & CTxIn::SEQUENCE_LOCKTIME_DISABLE_FLAG) != 0)
        break;

    // Compare the specified sequence number with the input.
    if (!checker.CheckSequence(nSequence))
        return set_error(serror, SCRIPT_ERR_UNSATISFIED_LOCKTIME);

    break;
```

```
}

bool TransactionSignatureChecker::CheckSequence(const CScriptNum& nSequence) const
{
    // Relative lock times are supported by comparing the passed
    // in operand to the sequence number of the input.
    const int64_t txToSequence = (int64_t)txTo->vin[nIn].nSequence;

    // Fail if the transaction's version number is not set high
    // enough to trigger BIP 68 rules.
    if (static_cast<uint32_t>(txTo->nVersion) < 2)
        return false;

    // Sequence numbers with their most significant bit set are not
    // consensus constrained. Testing that the transaction's sequence
    // number do not have this bit set prevents using this property
    // to get around a CHECKSEQUENCEVERIFY check.
    if (txToSequence & CTxIn::SEQUENCE_LOCKTIME_DISABLE_FLAG)
        return false;

    // Mask off any bits that do not have consensus-enforced meaning
    // before doing the integer comparisons
    const uint32_t nLockTimeMask = CTxIn::SEQUENCE_LOCKTIME_TYPE_FLAG | CTxIn::SEQUENCE_LOCK
    const int64_t txToSequenceMasked = txToSequence & nLockTimeMask;
    const CScriptNum nSequenceMasked = nSequence & nLockTimeMask;

    // There are two kinds of nSequence: lock-by-blockheight
    // and lock-by-blocktime, distinguished by whether
    // nSequenceMasked < CTxIn::SEQUENCE_LOCKTIME_TYPE_FLAG.
    //
    // We want to compare apples to apples, so fail the script
    // unless the type of nSequenceMasked being tested is the same as
    // the nSequenceMasked in the transaction.
    if (!(
        (txToSequenceMasked <  CTxIn::SEQUENCE_LOCKTIME_TYPE_FLAG && nSequenceMasked <  CTxI
        (txToSequenceMasked >= CTxIn::SEQUENCE_LOCKTIME_TYPE_FLAG && nSequenceMasked >= CTxI
    ))
        return false;

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

    return true;
}
```

Reference Implementation

A reference implementation is provided by the following pull request:

https://github.com/bitcoin/bitcoin/pull/7524

Deployment

This BIP is to be deployed by "versionbits" BIP9 using bit 0.

For Bitcoin **mainnet**, the BIP9 **starttime** will be midnight 1st May 2016 UTC (Epoch timestamp 1462060800) and BIP9 **timeout** will be midnight 1st May 2017 UTC (Epoch timestamp 1493596800).

For Bitcoin **testnet**, the BIP9 **starttime** will be midnight 1st March 2016 UTC (Epoch timestamp 1456790400) and BIP9 **timeout** will be midnight 1st May 2017 UTC (Epoch timestamp 1493596800).

This BIP must be deployed simultaneously with BIP68 and BIP113 using the same deployment mechanism.

Credits

Mark Friedenbach invented the application of sequence numbers to achieve relative lock-time, and wrote the reference implementation of CHECKSEQUENCEVERIFY.

The reference implementation and this BIP was based heavily on work done by Peter Todd for the closely related BIP 65.

BtcDrak authored this BIP document.

Thanks to Eric Lombrozo and Anthony Towns for contributing example use cases.

References

BIP 9 Versionbits

BIP 68 Relative lock-time through consensus-enforced sequence numbers

BIP 65 OP_CHECKLOCKTIMEVERIFY

BIP 113 Median past block time for time-lock constraints

HTLCs using OP_CHECKSEQUENCEVERIFY/OP_LOCKTIMEVERIFY and revocation hashes

Lightning Network

Deployable Lightning

Scaling Bitcoin to Billions of Transactions Per Day

Softfork deployment considerations

Version bits

Jeremy Spilman Micropayment Channels

Copyright