```
BIP: 119
Layer: Consensus (soft fork)
Title: CHECKTEMPLATEVERIFY
Author: Jeremy Rubin <j@rubin.io>
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0119
Status: Draft
Type: Standards Track
Created: 2020-01-06
License: BSD-3-Clause
```

Abstract

This BIP proposes a new opcode, OP_CHECKTEMPLATEVERIFY, to be activated as a change to the semantics of OP_NOP4.

The new opcode has applications for transaction congestion control and payment channel instantiation, among others, which are described in the Motivation section of this BIP.

Summary

OP_CHECKTEMPLATEVERIFY uses opcode OP_NOP4 (0xb3) as a soft fork upgrade.

OP_CHECKTEMPLATEVERIFY does the following:

- There is at least one element on the stack, fail otherwise
- The element on the stack is 32 bytes long, NOP otherwise
- The DefaultCheckTemplateVerifyHash of the transaction at the current input index is equal to the element on the stack, fail otherwise

The DefaultCheckTemplateVerifyHash commits to the serialized version, locktime, scriptSigs hash (if any non-null scriptSigs), number of inputs, sequences hash, number of outputs, outputs hash, and currently executing input index.

The recommended standardness rules additionally:

- Reject non-32 byte as SCRIPT_ERR_DISCOURAGE_UPGRADABLE_NOPS.

Motivation

Covenants are restrictions on how a coin may be spent beyond key ownership. This is a general definition based on the legal definition which even simple scripts using CSV would satisfy. Covenants in Bitcoin transactions usually refer to restrictions on where coins can be transferred. Covenants can be useful to construct smart contracts. Covenants have historically been widely considered to be unfit for Bitcoin because they are too complex to implement and risk reducing the fungibility of coins bound by them.

This BIP introduces a simple covenant called a *template* which enables a limited set of highly valuable use cases without significant risk. BIP-119 templates allow for **non-recursive** fully-enumerated covenants with no dynamic state. CTV serves as a replacement for a pre-signed transaction oracle, which eliminates the trust and interactivity requirements. Examples of uses include vaults, non-interactive payment channel creation, congestion controlled batching, efficient to construct discreet log contracts, and payment pools, among many others. For more details on these applications, please see the references.

Detailed Specification

The below code is the main logic for verifying CHECKTEMPLATEVERIFY, described in pythonic pseudocode. The canonical specification for the semantics of OP_CHECKTEMPLATEVERIFY as implemented in C++ in the context of Bitcoin Core can be seen in the reference implementation.

The execution of the opcode is as follows:

```
def execute_bip_119(self):
    # Before soft-fork activation / failed activation
    # continue to treat as NOP4
    if not self.flags.script_verify_default_check_template_verify_hash:
        # Potentially set for node-local policy to discourage premature use
        if self.flags.script_verify_discourage_upgradable_nops:
            return self.errors_with(errors.script_err_discourage_upgradable_nops)
        return self.return_as_nop()

    # CTV always requires at least one stack argument
    if len(self.stack) < 1:
        return self.errors_with(errors.script_err_invalid_stack_operation)

    # CTV only verifies the hash against a 32 byte argument
    if len(self.stack[-1]) == 32:
        # Ensure the precomputed data required for anti-DoS is available,
        # or cache it on first use
        if self.context.precomputed_ctv_data == None:
            self.context.precomputed_ctv_data = self.context.tx.get_default_check_template_p

        # If the hashes do not match, return error
        if stack[-1] != self.context.tx.get_default_check_template_hash(self.context.nIn, se
            return self.errors_with(errors.script_err_template_mismatch)

        return self.return_as_nop()

    # future upgrade can add semantics for this opcode with different length args
    # so discourage use when applicable
    if self.flags.script_verify_discourage_upgradable_nops:
```

```
        return self.errors_with(errors.script_err_discourage_upgradable_nops)
    else:
        return self.return_as_nop()
```

The computation of this hash can be implemented as specified below (where self is the transaction type). Care must be taken that in any validation context, the precomputed data must be initialized to prevent Denial-of-Service attacks. Any implementation *must* cache these parts of the hash computation to avoid quadratic hashing DoS. All variable length computations must be precomputed including hashes of the scriptsigs, sequences, and outputs. See the section "Denial of Service and Validation Costs" below. This is not a performance optimization.

```
def ser_compact_size(l):
    r = b""
    if l < 253:
        # Serialize as unsigned char
        r = struct.pack("B", l)
    elif l < 0x10000:
        # Serialize as unsigned char 253 followed by unsigned 2 byte integer (little endian)
        r = struct.pack("<BH", 253, l)
    elif l < 0x100000000:
        # Serialize as unsigned char 254 followed by unsigned 4 byte integer (little endian)
        r = struct.pack("<BI", 254, l)
    else:
        # Serialize as unsigned char 255 followed by unsigned 8 byte integer (little endian)
        r = struct.pack("<BQ", 255, l)
    return r

def ser_string(s):
    return ser_compact_size(len(s)) + s

class CTxOut:
    def serialize(self):
        r = b""
        # serialize as signed 8 byte integer (little endian)
        r += struct.pack("<q", self.nValue)
        r += ser_string(self.scriptPubKey)
        return r

def get_default_check_template_precomputed_data(self):
    result = {}
    # If there are no scriptSigs we do not need to precompute a hash
    if any(inp.scriptSig for inp in self.vin):
        result["scriptSigs"] = sha256(b"".join(ser_string(inp.scriptSig) for inp in self.vin
    # The same value is also pre-computed for and defined in BIP-341 and can be shared.
    # each nSequence is packed as 4 byte unsigned integer (little endian)
```

```
    result["sequences"] = sha256(b"".join(struct.pack("<I", inp.nSequence) for inp in self.v
    # The same value is also pre-computed for and defined in BIP-341 and can be shared
    # See class CTxOut above for details.
    result["outputs"] = sha256(b"".join(out.serialize() for out in self.vout))
    return result

# parameter precomputed must be passed in for DoS resistance
def get_default_check_template_hash(self, nIn, precomputed = None):
    if precomputed == None:
        precomputed = self.get_default_check_template_precomputed_data()
    r = b""
    # Serialize as 4 byte signed integer (little endian)
    r += struct.pack("<i", self.nVersion)
    # Serialize as 4 byte unsigned integer (little endian)
    r += struct.pack("<I", self.nLockTime)
    # we do not include the hash in the case where there is no
    # scriptSigs
    if "scriptSigs" in precomputed:
        r += precomputed["scriptSigs"]
    # Serialize as 4 byte unsigned integer (little endian)
    r += struct.pack("<I", len(self.vin))
    r += precomputed["sequences"]
    # Serialize as 4 byte unsigned integer (little endian)
    r += struct.pack("<I", len(self.vout))
    r += precomputed["outputs"]
    # Serialize as 4 byte unsigned integer (little endian)
    r += struct.pack("<I", nIn)
    return sha256(r)
```

A PayToBareDefaultCheckTemplateVerifyHash output matches the following template:

```
# Extra-fast test for pay-to-basic-standard-template CScripts:
def is_pay_to_bare_default_check_template_verify_hash(self):
    return len(self) == 34 and self[0] == 0x20 and self[-1] == OP_CHECKTEMPLATEVERIFY
```

Deployment

Deployment could be done via BIP 9 VersionBits deployed through Speedy Trial. The Bitcoin Core reference implementation includes the below parameters, configured to match Speedy Trial, as that is the current activation mechanism implemented in Bitcoin Core. Should another method become favored by the wider Bitcoin comminity, that might be used instead.

The start time and bit in the implementation are currently set to bit 5 and NEVER_ACTIVE/NO_TIMEOUT, but this is subject to change while the BIP is a draft.

For the avoidance of unclarity, the parameters to be determined are:

```
   // Deployment of CTV (BIP 119)
consensus.vDeployments[Consensus::DEPLOYMENT_CHECKTEMPLATEVERIFY].bit = 5;
consensus.vDeployments[Consensus::DEPLOYMENT_CHECKTEMPLATEVERIFY].nStartTime = Consensus::Bl
consensus.vDeployments[Consensus::DEPLOYMENT_CHECKTEMPLATEVERIFY].nTimeout = Consensus::BIP9
consensus.vDeployments[Consensus::DEPLOYMENT_CHECKTEMPLATEVERIFY].min_activation_height = 0;
```

Until BIP-119 reaches ACTIVE state and the SCRIPT_VERIFY_DEFAULT_CHECK_TEMPLATE_VERIFY flag is enforced, node implementations should (are recommended to) execute a NOP4 as SCRIPT_ERR_DISCOURAGE_UPGRADABLE_NOPS (to deny entry to the mempool) for policy and must evaluate as a NOP for consensus (during block validation).

In order to facilitate using CHECKTEMPLATEVERIFY, the common case of a PayToBareDefaultCheckTemplateVerifyHash with no scriptSig data may (is recommended to) be made standard to permit relaying. Future template types may be standardized later as policy changes at the preference of the implementor.

Reference Implementation

A reference implementation and tests are available here in the PR to Bitcoin Core https://github.com/bitcoin/bitcoin/pull/21702.

It is not ideal to link to a PR, as it may be rebased and changed, but it is the best place to find the current implementation and review comments of others. A recent commit hash in that PR including tests and vectors can be found here https://github.com/jeremyrubin/bitcoin/commit/3109df5616796282786706738994a5b97b8a5a38. Once the PR is merged, this BIP should be updated to point to the specific code released.

Test vectors are available in [/bip-0119/vectors the bip-0119/vectors directory] for checking compatibility with the reference implementation and BIP.

Rationale

The goal of CHECKTEMPLATEVERIFY is to be minimal impact on the existing codebase -- in the future, as we become aware of more complex but shown to be safe use cases, new template types can be added.

Below we'll discuss the rules one-by-one:

The DefaultCheckTemplateVerifyHash of the transaction at the current input index matches the top of the stack The set of data committed to is a superset of data which can impact the TXID of the transaction, other than the inputs. This ensures that for a given known input, the TXIDs can also be known ahead of time. Otherwise, CHECKTEMPLATEVERIFY would not be usable for Batched Channel Creation constructions as the redemption TXID could be malleated and pre-signed transactions invalidated, unless the

channels are built using an Eltoo-like protocol. Note that there may be other types of pre-signed contracts that may or may not be able to use Eltoo-like constructs, therefore making TXIDs predictable makes CTV more composable with arbitrary sub-protocols.

Committing to the version and locktime Were these values not committed, it would be possible to delay the spending of an output arbitrarily as well as possible to change the TXID.

Committing these values, rather than restricting them to specific values, is more flexible as it permits users of CHECKTEMPLATEVERIFY to set the version and locktime as they please.

Committing to the ScriptSigs Hash The scriptsig in a segwit transaction must be exactly empty, unless it is a P2SH segwit transaction in which case it must be only the exact redeemscript. P2SH is incompatible (unless the P2SH hash is broken) with CHECKTEMPLATEVERIFY because the template hash must commit to the ScriptSig, which must contain the redeemscript, which is a hash cycle.

To prevent malleability when not using a segwit input, we also commit to the scriptsig. This makes it possible to use a 2 input CHECKTEMPLATEVERIFY with a legacy pre-signed spend, as long as the exact scriptsig for the legacy output is committed. This is more robust than simply disallowing any scriptSig to be set with CHECKTEMPLATEVERIFY.

If no scriptSigs are set in the transaction, there is no purpose in hashing the data or including it in the DefaultCheckTemplateVerifyHash, so we elide it. It is expected to be common that no scriptSigs will be set as segwit mandates that the scriptSig must be empty (to avoid malleability).

We commit to the hash rather than the values themselves as this is already precomputed for each transaction to optimize SIGHASH_ALL signatures.

Committing to the hash additionally makes it simpler to construct DefaultCheck-TemplateVerifyHash safely and unambiguously from script.

Committing to the number of inputs If we allow more than one input to be spent in the transaction then it would be possible for two outputs to request payment to the same set of outputs, resulting in half the intended payments being discarded, the "half-spend" problem.

Furthermore, the restriction on which inputs can be co-spent is critical for payments-channel constructs where a stable TXID is a requirement (updates would need to be signed on all combinations of inputs).

However, there are legitimate use cases for allowing multiple inputs. For example:

Script paths:

```
   Path A: <+24 hours> OP_CHECKSEQUENCEVERIFY OP_CHECKTEMPLATEVERIFY <Pay Alice 1 Bitcoin (1
Path B: OP_CHECKTEMPLATEVERIFY <Pay Bob 2 Bitcoin (2 inputs)>
```

In this case, there are 24 hours for the output to, with the addition of a second output, pay Bob 2 BTC. If 24 hours lapses, then Alice may redeem her 1 BTC from the contract. Both input UTXOs may have the exact same Path B, or only one.

The issue with these constructs is that there are N! orders that the inputs can be ordered in and it's not generally possible to restrict the ordering.

CHECKTEMPLATEVERIFY allows for users to guarantee the exact number of inputs being spent. In general, using CHECKTEMPLATEVERIFY with more than one input is difficult and exposes subtle issues, so multiple inputs should not be used except in specific applications.

In principle, committing to the Sequences Hash (below) implicitly commits to the number of inputs, making this field strictly redundant. However, separately committing to this number makes it easier to construct DefaultCheckTemplateVerifyHash from script.

We treat the number of inputs as a ‘uint32_t‘ because Bitcoin's consensus decoding logic limits vectors to ‘MAX_SIZE=33554432‘ and that is larger than ‘uint16_t‘ and smaller than ‘uint32_t‘. 32 bits is also friendly for manipulation using Bitcoin's current math opcodes, should ‘OP_CAT‘ be added. Note that the max inputs in a block is further restricted by the block size to around 25,000, which would fit into a ‘uint16_t‘, but that is an uneccessary abstraction leak.

Committing to the Sequences Hash If we don't commit to the sequences, then the TXID can be malleated. This also allows us to enforce a relative sequence lock without an OP_CSV. It is insufficient to just pair CHECKTEMPLATEVERIFY with OP_CSV because OP_CSV enforces a minimum nSequence value, not a literal value.

We commit to the hash rather than the values themselves as this is already precomputed for each transaction to optimize SIGHASH_ALL signatures.

Committing to the hash additionally makes it simpler to construct DefaultCheckTemplateVerifyHash safely and unambiguously from script.

Committing to the Number of Outputs In principle, committing to the Outputs Hash (below) implicitly commits to the number of outputs, making this field strictly redundant. However, separately committing to this number makes it easier to construct DefaultCheckTemplateVerifyHash from script.

We treat the number of outputs as a ‘uint32_t‘ because a ‘COutpoint‘ index is a ‘uint32_t‘. Further, Bitcoin's consensus decoding logic limits vectors to ‘MAX_SIZE=33554432‘ and that is larger than ‘uint16_t‘ and smaller than

‘uint32_t‘. 32 bits is also friendly for manipulation using Bitcoin's current math opcodes, should ‘OP_CAT‘ be added.

Committing to the outputs hash This ensures that spending the UTXO is guaranteed to create the exact outputs requested.

We commit to the hash rather than the values themselves as this is already precomputed for each transaction to optimize SIGHASH_ALL signatures.

Committing to the hash additionally makes it simpler to construct DefaultCheck-TemplateVerifyHash safely and unambiguously from script.

Committing to the current input's index Committing to the currently executing input's index is not strictly needed for anti-malleability, however it does restrict the input orderings eliminating a source of malleability for protocol designers.

However, committing to the index eliminates key-reuse vulnerability to the half-spend problem. As CHECKTEMPLATEVERIFY scripts commit to being spent at particular index, reused instances of these scripts cannot be spent at the same index, which implies that they cannot be spent in the same transaction. This makes it safer to design wallet vault contracts without half-spend vulnerabilities.

Committing to the current index doesn't prevent one from expressing a CHECK-TEMPLATEVERIFY which can be spent at multiple indicies. In current script, the CHECKTEMPLATEVERIFY operation can be wrapped in an OP_IF for each index (or Tapscript branches in the future). If OP_CAT or OP_SHA256STREAM are added to Bitcoin, the index may simply be passed in by the witness before hashing.

Committing to Values by Hash Committing to values by hash makes it easier and more efficient to construct a DefaultCheckTemplateVerifyHash from script. Fields which are not intended to be set may be committed to by hash without incurring O(n) overhead to re-hash.

Furthermore, if OP_SHA256STREAM is added in the future, it may be possible to write a script which allows adding a single output to a list of outputs without incurring O(n) overhead by committing to a hash midstate in the script.

Using SHA256 SHA256 is a 32 byte hash which meets Bitcoin's security standards and is available already inside of Bitcoin Script for programmatic creation of template programs.

RIPEMD160, a 20 byte hash, might also be a viable hash in some contexts and has some benefits. For fee efficiency, RIPEMD160 saves 12 bytes. However, RIPEMD160 was not chosen for BIP-119 because it introduces risks around the

verification of programs created by third parties to be subject to a [birthday-attack https://bitcoin.stackexchange.com/questions/54841/birthday-attack-on-p2sh] on transaction preimages.

Using Non-Tagged Hashes The Taproot/Schnorr BIPs use Tagged Hashes ('SHA256(SHA256(tag)||SHA256(tag)||msg)') to prevent taproot leafs, branches, tweaks, and signatures from overlapping in a way that might introduce a security [vulnerability https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-June/016091.html].

OP_CHECKTEMPLATEVERIFY is not subject to this sort of vulnerability as the hashes are effectively tagged externally, that is, by OP_CHECKTEMPLATEVERIFY itself and therefore cannot be confused for another hash.

It would be a conservative design decisison to make it a tagged hash even if there was no obvious benefit and no cost. However, in the future, if OP_CAT were to be introduced to Bitcoin, it would make programs which dynamically build OP_CHECKTEMPLATEVERIFY hashes less space-efficient. Therefore, bare untagged hashes are used in BIP-119.

The Ordering of Fields Strictly speaking, the ordering of fields is insignificant. However, with a carefully selected order, the efficiency of future scripts (e.g., those using a OP_CAT or OP_SHA256STREAM) may be improved (as described in the Future Upgrades section).

In particular, the order is selected in order of least likely to change to most.

1. nVersion
2. nLockTime
3. scriptSig hash (maybe!)
4. input count
5. sequences hash
6. output count
7. outputs hash
8. input index

Several fields are infrequently modified. nVersion should change infrequently. nLockTime should generally be fixed to 0 (in the case of a payment tree, only the *first* lock time is needed to prevent fee-sniping the root). scriptSig hash should generally not be set at all.

Since there are many possible sequences hash for a given input count, the input count comes before the sequences hash.

Since there are many possible outputs hashes for a given out count, the output count comes before the outputs hash.

Since we're generally using a single input to many output design, we're more likely to modify the outputs hash than the inputs hash.

We usually have just a single input on a CHECKTEMPLATEVERIFY script, which would suggest that it does not make sense for input index to be the last field. However, given the desirability of being able to express a "don't care" index easily (e.g., for decentralized kickstarter-type transactions), this value is placed last.

Design Tradeoffs and Risks

Covenants have historically been controversial given their potential for fungibility risks -- coins could be minted which have a permanent restriction on how they may or may not be spent or required to propagate metadata.

In the CHECKTEMPLATEVERIFY approach, the covenants are severely restricted to simple templates. The structure of CHECKTEMPLATEVERIFY template is such that the outputs must be known exactly at the time of construction. Based on a destructuring argument, it is only possible to create templates which expand in a finite number of steps. Thus templated transactions are in theory as safe as transactions which create all the inputs directly in this regard.

Furthermore, templates are restricted to be spendable as a known number of inputs only, preventing unintentional introduction of the 'half spend' problem.

Templates, as restricted as they are, bear some risks.

Denial of Service and Validation Costs CTV is designed to be able to be validated very cheaply without introducing DoS, either by checking a precomputed hash or computing a hash of fixed length arguments (some of which may be cached from more expensive computations).

In particular, CTV requires that clients cache the computation of a hash over all the scriptSigs, sequences, and outputs. Before CTV, the hash of the scriptSigs was not required. CTV also requires that the presence of any non-empty scriptSig be hashed, but this can be handled as a part of the scriptSigs hash.

As such, evaluating a CTV hash during consensus is always O(1) computation when the caches are available. These caches usually must be available due to similar issues in CHECKSIG behavior. Computing the caches is O(T) (the size of the transaction).

An example of a script that could experience an DoS issue without caching is:

```
CTV CTV CTV... CTV
```

Such a script would cause the intepreter to compute hashes (supposing N CTV's) over O(N*T) data. If the scriptSigs non-nullity is not cached, then the O(T) transaction could be scanned over O(N) times as well (although cheaper than

hashing, still a DoS). As such, CTV caches hashes and computations over all variable length fields in a transaction.

For CTV, the Denial-of-Service exposure and validation costs are relatively clear. Implementors must be careful to correctly code CTV to make use of existing caches and cache the (new for CTV) computations over scriptSigs. Other more flexible covenant proposals may have a more difficult time solving DoS issues as more complex computations may be less cacheable and expose issues around quadratic hashing, it is a tradeoff CTV makes in favor of cheap and secure validation at the expense of flexibility. For example, if CTV allowed the hashing only select outputs by a bitmask, caching of all combinations of outputs would not be possible and would cause a quadratic hashing DoS vulnerability.

Permanently Unspendable Outputs The preimage argument passed to CHECKTEMPLATEVERIFY may be unknown or otherwise unsatisfiable. However, requiring knowledge that an address is spendable from is incompatible with sender's ability to spend to any address (especially, OP_RETURN). If a sender needs to know the template can be spent from before sending, they may request a signature of an provably non-transaction challenge string from the leafs of the CHECKTEMPLATEVERIFY tree.

Forwarding Addresses Key-reuse with CHECKTEMPLATEVERIFY may be used as a form of "forwarding address contract". A forwarding address is an address which can automatically execute in a predefined way. For example, a exchange's hot wallet might use an address which can automatically be moved to a cold storage address after a relative timeout.

The issue is that reusing addresses in this way can lead to loss of funds. Suppose one creates an template address which forwards 1 BTC to cold storage. Creating an output to this address with less than 1 BTC will be frozen permanently. Paying more than 1 BTC will lead to the funds in excess of 1BTC to be paid as a large miner fee. CHECKTEMPLATEVERIFY could commit to the exact amount of bitcoin provided by the inputs/amount of fee paid, but as this is a user error and not a malleability issue this is not done. Future soft-forks could introduce opcodes which allow conditionalizing which template or script branches may be used based on inspecting the amount of funds available in a transaction

As a general best practice, it is incumbent on Bitcoin users to not reuse any address unless you are certain that the address is acceptable for the payment attempted. This limitation and risk is not unique to CHECKTEMPLATEVERIFY. For example, atomic swap scripts are single use once the hash is revealed. Future Taproot scripts may contain many logical branches that would be unsafe for being spent to multiple times (e.g., a Hash Time Lock branch should be instantiated with unique hashes each time it is used). Keys which have signed a SIGHASH_ANYPREVOUT transaction can similarly become reuse-unsafe.

Because CHECKTEMPLATEVERIFY commits to the input index currently being spent, reused-keys are guaranteed to execute in separate transactions which reduces the risk of "half-spend" type issues.

NOP-Default and Recommended Standardness Rules If the argument length is not exactly 32, CHECKTEMPLATEVERIFY treats it as a NOP during consensus validation. Implementations are recommended to fail in such circumstances during non-consensus relaying and mempool validation. In particular, making an invalid-length argument a failure aids future soft-forks upgrades to be able to rely on the tighter standard restrictions to safely loosen the restrictions for standardness while tightening them for consensus with the upgrade's rules.

The standardness rules may lead an unscrupulous script developer to accidentally rely on the stricter standardness rules to be enforced during consensus. Should that developer submit a transaction directly to the network relying on standardness rejection, an standardness-invalid but consensus-valid transaction may be caused, leading to a potential loss of funds.

Feature Redundancy CHECKTEMPLATEVERIFY templates are substantially less risky than other covenant systems. If implemented, other covenant systems could make the CHECKTEMPLATEVERIFY's functionality redundant. However, given CHECKTEMPLATEVERIFY's simple semantics and low on chain cost it's likely that it would continue to be favored even if redundant with other capabilities.

More powerful covenants like those proposed by MES16, would also bring some benefits in terms of improving the ability to adjust for things like fees rather than relying on child-pays-for-parent or other mechanisms. However, these features come at substantially increased complexity and room for unintended behavior.

Alternatively, SIGHASH_ANYPREVOUTANYSCRIPT based covenant designs can implement something similar to templates, via a scriptPubKey like:

```
<sig of desired TX with PK and fixed nonce R || SIGHASH_ANYPREVOUTANYSCRIPT  OP_CHECKSIG
```

SIGHASH_ANYPREVOUTANYSCRIPT bears additional technical and implementation risks that may preclude its viability for inclusion in Bitcoin, but the capabilities above are similar to what CHECKTEMPLATEVERIFY offers. The key functional difference between SIGHASH_ANYPREVOUTANYSCRIPT and OP_CHECKTEMPLATEVERIFY is that OP_CHECKTEMPLATEVERIFY restricts the number of additional inputs and precludes dynamically determined change outputs while SIGHASH_ANYPREVOUTANYSCRIPT can be combined with SIGHASH_SINGLE or SIGHASH_ANYONECANPAY. For the additional inputs, OP_CHECKTEMPLATEVERIFY also commits to the scriptsig and sequence, which allows for specifying specific P2SH scripts (or segwit v0 P2SH) which have some use cases. Furthermore, CHECKTEMPLATEVERIFY has benefits in terms of script size (depending on choice of PK, SIGHASH_ANYPREVOUTANYSCRIPT may use about 2x-3x the bytes) and

verification speed, as OP_CHECKTEMPLATEVERIFY requires only hash computation rather than signature operations. This can be significant when constructing large payment trees or programmatic compilations. CHECKTEMPLATEVERIFY also has a feature-wise benefit in that it provides a robust pathway for future template upgrades.

OP_CHECKSIGFROMSTACKVERIFY along with OP_CAT may also be used to emulate CHECKTEMPLATEVERIFY. However such constructions are more complicated to use than CHECKTEMPLATEVERIFY, and encumbers additional verification overhead absent from CHECKTEMPLATEVERIFY. These types of covenants also bear similar potential recursion issues to OP_COV which make it unlikely for inclusion in Bitcoin.

Given the simplicity of this approach to implement and analyze, and the benefits realizable by user applications, CHECKTEMPLATEVERIFY's template based approach is proposed in lieu of more complete covenants system.

Future Upgrades This section describes updates to OP_CHECKTEMPLATEVERIFY that are possible in the future as well as synergies with other possible upgrades.

CHECKTEMPLATEVERIFY Versions OP_CHECKTEMPLATEVERIFY currently only verifies properties of 32 byte arguments. In the future, meaning could be ascribed to other length arguments. For example, a 33-byte argument could just the last byte as a control program. In that case, DefaultCheckTemplateVerifyHash could be computed when the flag byte is set to CTVHASH_ALL. Other programs could be added similar to SIGHASH_TYPEs. For example, CTVHASH_GROUP could read data from the Taproot Annex for compatibility with SIGHASH_GROUP type proposals and allow dynamic malleability of which indexes get hashed for bundling.

Eltoo with OP_CHECKSIGFROMSTACKVERIFY Were both OP_CHECKTEMPLATEVERIFY and OP_CHECKSIGFROMSTACKVERIFY to be added to Bitcoin, it would be possible to implement a variant of Eltoo's floating transactions using the following script:

```
   witness(S+n):  <H(tx with nLockTime S+n paying to program(S+n))>
program(S): OP_CHECKTEMPLATEVERIFY <musig_key(pk_update_a, pk_update_b)> OP_CHECKSIGFROMSTA
```

Compared to SIGHASH_ANYPREVOUTANYSCRIPT, because OP_CHECKTEMPLATEVERIFY does not allow something similar to SIGHASH_ANYONECANPAY or SIGHASH_SINGLE, protocol implementers might elect to sign multiple versions of transactions with CPFP Anchor Outputs or Inputs for paying fees or an alternative such as transaction sponsors might be considered.

OP_AMOUNTVERIFY An opcode which verifies the exact amount that is being spent in the transaction, the amount paid as fees, or made available in a given output could be used to make safer OP_CHECKTEMPLATEVERIFY

addressses. For instance, if the OP_CHECKTEMPLATEVERIFY program P expects exactly S satoshis, sending S-1 satoshis would result in a frozen UTXO and sending S+n satoshis would result in n satoshis being paid to fee. A range check could restrict the program to only apply for expected values and default to a keypath otherwise, e.g.:

```
IF OP_AMOUNTVERIFY  OP_GREATER  CHECKSIG ELSE  OP_CHECKTEMPLATEVERIFY
```

OP_CAT/OP_SHA256STREAM OP_CHECKTEMPLATEVERIFY is (as described in the Ordering of Fields section) efficient for building covenants dynamically should Bitcoin get enhanced string manipulation opcodes.

As an example, the following code checks an input index argument and concatenates it to the template and checks the template matches the transaction.

```
   OP_SIZE 4 OP_EQUALVERIF
<nVersion || nLockTime || input count || sequences hash || output count || outputs hash>
OP_SWAP OP_CAT OP_SHA256 OP_CHECKTEMPLATEVERIFY
```

Backwards Compatibility

OP_CHECKTEMPLATEVERIFY replaces a OP_NOP4 with stricter verification semantics. Therefore, scripts which previously were valid will cease to be valid with this change. Stricter verification semantics for an OP_NOP are a soft fork, so existing software will be fully functional without upgrade except for mining and block validation. Similar soft forks for OP_CHECKSEQUENCEVERIFY and OP_CHECKLOCKTIMEVERIFY (see BIP-0065 and BIP-0112) have similarly changed OP_NOP semantics without introducing compatibility issues.

In contrast to previous forks, OP_CHECKTEMPLATEVERIFY's reference implementation does not allow transactions with spending scripts using it to be accepted to the mempool or relayed under standard policy until the new rule is active. Other implementations are recommended to follow this rule as well, but not required.

Older wallet software will be able to accept spends from OP_CHECKTEMPLATEVERIFY outputs, but will require an upgrade in order to treat PayToBareDefaultCheckTemplateVerifyHash chains with a confirmed ancestor as being "trusted" (i.e., eligible for spending before the transaction is confirmed).

Backports of OP_CHECKTEMPLATEVERIFY can be trivially prepared (see the reference implementation) for older node versions that can be patched but not upgraded to a newer major release.

References

- utxos.org informational site
- Sapio Bitcoin smart contract language

- 27 Blog Posts on building smart contracts with Sapio and CTV, including examples described here.
- Scaling Bitcoin Presentation
- Optech Newsletter Covering OP_CHECKOUTPUTSHASHVERIFY
- Structuring Multi Transaction Contracts in Bitcoin
- Lazuli Notes (ECDSA based N-of-N Signatures for Certified Post-Dated UTXOs)
- Bitcoin Covenants
- CoinCovenants using SCIP signatures, an amusingly bad idea.
- Enhancing Bitcoin Transactions with Covenants
- Simple CTV Vaults
- Python Vaults
- CTV Dramatically Improves DLCs
- Calculus of Covenants
- Payment Pools with CTV
- Channels with CTV
- Congestion Control with CTV
- Building Vaults on Bitcoin

Note on Similar Alternatives

An earlier version of CHECKTEMPLATEVERIFY, CHECKOUTPUTSHASHVERIFY, is withdrawn in favor of CHECKTEMPLATEVERIFY. CHECKOUTPUTSHASHVERIFY did not commit to the version or lock time and was thus insecure.

CHECKTEMPLATEVERIFY could also be implemented as an extension to Taproot, and was proposed this way earlier. However, given that CHECKTEMPLATEVERIFY has no dependency on Taproot, it is preferable to deploy it independently.

CHECKTEMPLATEVERIFY has also been previously referred to as OP_SECURETHEBAG, which is mentioned here to aid in searching and referencing discussion on this BIP.

Copyright