```
BIP: 117
Layer: Consensus (soft fork)
Title: Tail Call Execution Semantics
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-0117
Status: Draft
Type: Standards Track
Created: 2017-08-25
License: CC-BY-SA-4.0
License-Code: MIT
```

## Abstract

BIP16 (Pay to Script Hash)[1] and BIP141 (Segregated Witness)[2] provide mechanisms by which script policy can be revealed at spend time as part of the execution witness. In both cases only a single script can be committed to by the construct. While useful for achieving the goals of these proposals, they still require that all policies be specified within the confine of a single script, regardless of whether the policies are needed at the time of spend.

This BIP, in conjunction with BIP116 (MERKLEBRANCHVERIFY)[3] allows for a script to commit to a practically unbounded number of code pathways, and then reveal the actual code pathway used at spend time. This achieves a form of generalized MAST[4] enabling decomposition of complex branching scripts into a set of non-branching flat execution pathways, committing to the entire set of possible pathways, and then revealing only the path used at spend time.

## Copyright



## Specification

If, at the end of script execution:

- the execution state is non-clean, meaning
  1. the main stack has more than one item on it, or
  2. the main stack has exactly one item and the alt-stack is not empty;
- the top-most element of the main stack evaluates as true when interpreted as a bool; and
- the top-most element is not a single byte or is outside the inclusive range of `0x51` to `0x60`,

then that top-most element of the main stack is popped and interpreted as a serialized script and executed, while the remaining elements of both stacks remain in place as inputs.

If the above conditions hold except for the last one, such that:

- the top-most element *is* a single byte within the inclusive range of `0x51` (`OP_1`, meaning N=2) to `0x60` (`OP_16`, meaning N=17); and
- other than this top-most element there are at least N additional elements on the main stack and alt stack combined,

then the top-most element of the main stack is dropped, and the N=2 (`0x51`) to 17 (`0x60`) further elements are popped from the main stack, continuing from the alt stack if the main stack is exhausted, and concatenated together in reverse order to form a serialized script, which is then executed with the remaining elements of both stacks remaining in place as inputs.

The presence of CHECKSIG or CHECKMULTISIG within the subscript do not count towards the global MAX_BLOCK_SIGOPS_COST limit, and the number of non-push opcodes executed in the subscript is not limited by MAX_OPS_PER_SCRIPT. Execution state, other than the above exceptions, carries over into the subscript, and termination of the subscript terminates execution of the script as a whole. This is known as execution with tail-call semantics.

Only one such tail-call of a subscript is allowed per script execution context, and only from within a segwit redeem script. Alternatively stated, neither evaluation of witness stack nor execution of the scriptPubKey or scriptSig or P2SH redeem script results in tail-call semantics.

## Motivation

BIP16 (Pay to Script Hash)[1] and BIP141 (Segregated Witness)[2] allow delayed revelation of a script's policy until the time of spend. However these approaches are limited in that only a single policy can be committed to in a given transaction output. It is not possible to commit to multiple policies and then choose, at spend time, which to reveal.

BIP116 (MERKLEBRANCHVERIFY)[3] allows multiple data elements to be committed to while only revealing those necessary at the time of spend. The MERKLEBRANCHVERIFY opcode is only able to provide commitments to a preselected set of data values, and does not by itself allow for executing code.

This BIP generalizes the approach of these prior methods by allowing the redeem script to perform any type of computation necessary to place the policy script on the stack. The policy script is then executed from the top of the data stack in a way similar to how BIP16 and BIP141 enable redeem scripts to be executed from the top of the witness stack. In particular, using MERKLEBRANCHVERIFY[3] in the scriptPubKey or redeem script allows selection of the policy script that

contains only the necessary conditions for validation of the spend. This is a form of generalized MAST[4] where a stage of precomputation splits a syntax tree into possible execution pathways, which are then enumerated and hashed into a Merkle tree of policy scripts. At spend time membership in this tree of the provided policy script is proven before execution recurses into the policy script.

## Rationale

This proposal is a soft-fork change to bitcoin's consensus rules because leaving a script that data-wise evaluates as true from its serialized form on the stack as execution terminates would result in the script validation returning true anyway. Giving the subscript a chance to terminate execution is only further constraining the validation rules. The only scripts which would evaluate as false are the empty script, or a script that does nothing more than push empty/zero values to the stack. None of these scripts have any real-world utility, so excluding them to achieve soft-fork compatibility doesn't come with any downsides.

By restricting ourselves to tail-call evaluation instead of a more general EVAL opcode we greatly simplify the implementation. Tail-call semantics means that execution never returns to the calling script's context, and therefore no state needs to be saved or later restored. The implementation is truly as simple as pulling the subscript off the stack, resetting a few state variables, and performing a jump back to the beginning of the script interpreter.

The restriction to allow only one layer of tail-call recursion is admittedly limiting, however the technical challenges to supporting multi-layer tail-call recursion are significant. A new metric would have to be developed to track script resource usage, for which transaction data witness size are only two factors. This new weight would have to be relayed with transactions, used as the basis for fee calculation, validated in-line with transaction execution, and policy decided upon for DoS-banning peers that propagate violating transactions.

However should these problems be overcome, dropping the single recursion constraint is itself a soft-fork for the same reason, applied inductively. Allowing only one layer of tail-call recursion allows us to receive the primary benefit of multi-policy commitments / generalized MAST, while leaving the door open to future generalized tail-call recursion if and when the necessary changes are made to resource accounting and p2p transaction distribution.

The global SIGOP limit and per-script opcode limits do not apply to the policy script because dynamic selection of the policy script makes it not possible for static analysis tools to verify these limits in general, and because performance improvements to libsecp256k1 and Bitcoin Core have made these limits no longer necessary as they once were. The validation costs are still limited by the number of signature operations it is possible to encode within block size limits, and the maximum script size per input is limited to 10,000 + 17*520 = 18,840 bytes.

To allow for this drop of global and per-script limits, tail-call evaluation cannot

be allowed for direct execution of the scriptPubKey, as such scripts are fetched from the UTXO and do not count towards block size limits of the block being validated. Likewise tail-call from P2SH redeem scripts is not supported due to quadratic blow-up vulnerabilities that are fixed in segwit.

## Generalized MAST

When combined with BIP116 (MERKLEBRANCHVERIFY)[3], tail-call semantics allows for generalized MAST capabilities[4]. The script author starts with a full description of the entire contract they want to validate at the time of spend. The possible execution pathways through the script are then enumerated, with conditional branches replaced by a validation of the condition and the branch taken. The list of possible execution pathways is then put into a Merkle tree, with the flattened policy scripts as the leaves of this tree. The final redeem script which funds are sent to is as follows:

```
 redeemScript: <root> 2 MERKLEBRANCHVERIFY 2DROP DROP
witness: <argN> ... <arg1> <policyScript> <proof>
```

Where `policyScript` is the flattened execution pathway, `proof` is the serialized Merkle branch and path that proves the policyScript is drawn from the set used to construct the Merkle tree `root`, and `arg1` through `argN` are the arguments required by `policyScript`. The `2` indicates that a single leaf (`1 << 1`) follows, and the leaf value is not pre-hashed. The `2DROP DROP` is necessary to remove the arguments to MERKLEBRANCHVERIFY from the stack.

The above example was designed for clarity, but actually violates the CLEANSTACK rule of segwit v0 script execution. Unless the CLEANSTACK rule is dropped or modified in a new segwit output version, this would script would have to be modified to use the alt-stack, as follows:

```
 redeemScript: [TOALTSTACK]*N <root> 2 MERKLEBRANCHVERIFY 2DROP DROP
witness: <policyScript> <proof> <arg1> ... <argN>
```

Where `[TOALTSTACK]*N` is the TOALTSTACK opcode repeated N times. This moves `arg1` through `argN` to the alt-stack in reverse order, such that `arg1` is on the top of the alt-stack when execution of `policyScript` begins. The `policyScript` would also have to be modified to fetch its arguments from the alt-stack, of course.

If the total set of policy scripts includes scripts that take a varying number of parameters, that too can be supported, within reasonable limits. The following redeem script allows between 1 and 3 witness arguments in addition to the policy script and Merkle proof:

```
 witness: <policyScript> <proof> <arg1> ... <argN> // N is between 1 and 3
redeemScript: DEPTH TOALTSTACK                    // Save number of witness elements to alt-
TOALTSTACK                          // Save 1st element (required) to alt-stack
DEPTH 2 SUB                         // Calculate number of optional elements, ignoring polic
```

```
DUP IF SWAP TOALTSTACK 1SUB ENDIF   // Save 2nd element (optional) to alt-stack, if it is pr
IF TOALTSTACK ENDIF                 // Save 3rd element (optional) to alt-stack, if it is pr
<root> 2 MERKLEBRANCHVERIFY 2DROP DROP
alt-stack: <N+2> <argN> ... <arg1>
```

Because the number of witness elements is pushed onto the alt-stack, this enables policy scripts to verify the number of arguments passed, even though the size of the alt-stack is not usually accessible to script. The following policy script for use with the above redeem script will only accept 2 witness elements on the alt-stack, preventing witness malleability:

```
policyScript: FROMALTSTACK ...check arg1... FROMALTSTACK ...check&consume arg2/arg1&2... FR
```

The number 4 is expected as that includes the `policyScript` and `proof`.

The verbosity of this example can be prevented by using a uniform number of witness elements as parameters for all policy subscripts, eliminating the conditionals and stack size counts. Future script version upgrades should also consider relaxing CLEANSTACK rules to allow direct pass-through of arguments from the witness/redeem script to the policy script on the main stack.

### Comparison with BIP114

BIP114 (Merkelized Abstract Syntax Tree)[5] specifies an explicit MAST scheme activated by BIP141 script versioning[2]. Unlike BIP114, the scheme proposed by this BIP in conjunction with BIP116 (MERKLEBRANCHVERIFY)[3] implicitly enables MAST constructs using script itself to validate membership of the policy script in the MAST. This has the advantage of requiring vastly fewer consensus code changes, as well as potentially enabling future script-based innovation without requiring any further consensus code changes at all, as the MAST scheme itself is programmable.

Furthermore, by adding MERKLEBRANCHVERIFY and tail-call semantics to all script using the NOP-expansion space, BIP141 style script versioning is not required. This removes a potentially significant hurdle to deployment by making this feature not dependent on resolving outstanding issues over address formats, how script version upgrades should be deployed, and consensus over what other features might go into a v1 upgrade.

## Implementation

An implementation of this BIP, including both consensus code changes and tests are 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 "tailcall" and using bit 3.

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 CLEANSTACK 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)[6].

## Compatibility

The v0 segwit rules prohibit leaving anything on the stack, so for v0 parameters have to be passed on the alt stack for compatibility reasons.

## References

[1] BIP16: Pay to Script Hash

[2] BIP141: Segregated Witness (Consensus Layer)

[3] BIP116: MERKLEBRANCHVERIFY

[4] "An explanation and justification of the tail-call and MBV approach to MAST", Mark Friedenbach, Bitcoin Development Mailing List, 20 September 2017.

[5] BIP114: Merkelized Abstract Syntax Tree

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