```
BIP: 114
Title: Merkelized Abstract Syntax Tree
Author: Johnson Lau <jl2012@xbt.hk>
Status: Draft
Type: Standards Track
Created: 2016-04-02
```

Table of Contents

Abstract

This BIP defines a new witness program type that uses a Merkle tree to encode mutually exclusive branches in a script. This enables complicated redemption conditions that are currently not possible, improves privacy by hiding unexecuted scripts, and allows inclusion of non-consensus enforced data with very low or no additional cost.

Motivation

Evolution of Bitcoin script system

Bitcoin uses a script system to specify the conditions for redemption of transaction outputs. In its original design, the conditions for redemption are directly recorded in the scriptPubKey by the sender of the funds. This model has several drawbacks, particularly for complicated scripts:

1. It could be difficult for the receiver to specify the conditions;
2. Large scripts take up more UTXO space;
3. The sender will pay for the additional block space;
4. To prevent DoS attack, scripts are limited to 10,000 bytes and 201 op codes;
5. Any unexecuted branches and non-consensus enforced data in the script are visible to the public, consuming block space while damaging privacy.

The BIP16 (Pay-to-script-hash, "P2SH") fixes the first 3 problems by using a fixed-length 20-byte script hash in the scriptPubKey, and moving the responsibility for supplying the script to the redeemer. However, due to the data push size limit in script, a P2SH script may not be bigger than 520 bytes. Also, P2SH still requires the redeemer to publish all unexecuted branches of the script.

The BIP141 defines 2 new types of scripts that support segregated witness. The pay-to-witness-script-hash (P2WSH) is similar to P2SH is many ways. By supplying the script in witness, P2WSH restores the original 10,000 byte script limit. However, it still requires publishing of unexecuted branches.

Merkelized Abstract Syntax Tree

The idea of Merkelized Abstract Syntax Tree (MAST) is to use a Merkle tree to encode branches in a script. When spending, users may provide only the branches they are executing, and hashes that connect the branches to the fixed size Merkel root. This reduces the size of redemption stack from O(n) to O(log n) (n as the number of branches). This enables complicated redemption conditions that is currently not possible due to the script size and opcode limit, improves privacy by hiding unexecuted branches, and allows inclusion of non-consensus enforced data with very low or no additional cost.

Specification

In BIP141, witness programs with a version byte of 1 or larger are considered to be anyone-can-spend scripts. The following new validation rules are applied if the witness program version byte is 1 and the program size is 32 bytes.[1] The witness program is the `MAST Root`.

To redeem an output of this kind, the witness must consist of the following items:

```
Script_stack_1
Script_stack_2
.
.
Script_stack_X (X ≥ 0)
Subscript_1
```

```
Subscript_2
.
.
Subscript_Y (1 ≤ Y ≤ 255)
Position
Path
Metadata (Y|MAST Version)
```

`Metadata` is the last witness item. It is a vector of 1 to 5 bytes. The first byte is an unsigned integer between 1 to 255 denoting the number of `Subscript` (defined hereinafter). The following 0 to 4 byte(s) is an unsigned little-endian integer denoting the `MAST version`. `MAST Version` must be minimally encoded (the most significant byte must not be 0).

`Path` is the second last witness item. It is a serialized Merkle path of the `Script Hash` (defined hereinafter). Size of `Path` must be a multiple of 32 bytes, and not more than 1024 bytes. Each 32 byte word is a double-SHA256 merkle node in the merkle branch connecting to the `Script Root` (defined hereinafter). `Depth` of the tree (0 to 32) is the size of `Path` divided by 32.

`Position` is the third last witness item. It indicates the location of the `Script Hash` in the Merkle tree, with zero indicating the leftmost position. It is an unsigned little-endian integer with not more than 4 bytes. It must be minimally encoded: the value must not be larger than the maximum number of items allowed by the `Depth` of the tree, and the most significant byte must not be 0. For example, if `Depth` is 4, the valid range of `Position` is 0 to 15 (2^4-1).

Depends on the first byte of `Metadata`, there should be 1 to 255 `Subscript` witness item(s) before `Position`.

`Script Hash` is defined as:

```
Script Hash = H(Y|H(Subscript_1)|H(Subscript_2)|...|H(Subscript_Y))
H() = SHA256(SHA256())
```

where `Y` is a 1-byte value denoting number of `Subscript`, followed by the hash of each `Subscript`

`Script Root` is the Merkle root calculated by the `ComputeMerkleRootFromBranch` function, using `Script Hash`, `Path` and `Position`.

`MAST Root` is `H(MAST Version|Script Root)`. The pre-image has a fixed size of 36 bytes: 4 bytes for `MAST Version` (unsigned little-endian integer) and 32 bytes for `Script Root`.

The script evaluation fails if `MAST Root` does not match the witness program.

If the `MAST Root` matches the witness program and `MAST Version` is greater than 0, the script returns a success without further evaluation. `SigOpsCost` is counted as 0. This is reserved for future script upgrades.

If the `MAST Version` is 0, the `Subscript`(s) are serialized to form the final `MAST Script`, beginning with Subscript_1. The unused witness item(s) before the Subscript_1 are used as `Input Stack` to feed to the `MASTScript`. (Similar to P2WSH in BIP141)

The script fails with one of the following conditions:

- `MAST Script` is malformed (i.e. not enough data provided for the last push operation). Individual `Subscript` might be malformed, as long as they are serialized into a valid `MAST Script`
- Size of `MAST Script` is larger than 10,000 bytes
- Size of any one of the `Input Stack` item is larger than 520 bytes
- Number of non-push operations (`nOpCount`) is more than 201. `nOpCount` is the sum of the number of non-push operations in `MAST Script` (counted in the same way as P2WSH `witnessScript`), number of `Subscript` (Y), and `Depth` of the Merkle tree.

The `MAST Script` is then evaluated with the `Input Stack` (with some new or redefined opcodes described in BIPXXX). The evaluation must not fail, and result in an exactly empty stack.

Counting of `SigOpsCost` is based on the `MAST Script`, described in BIPYYY.

Rationale

MAST Structure

This proposal is a restricted case of more general MAST. In a general MAST design, users may freely assign one or more script branches for execution. In this proposal, only one branch is allowed for execution, and users are required to transform a complicated condition into several mutually exclusive branches. For example, if the desired redeem condition is:

```
(A or B) and (C or D or E) and (F or G)
```

In a general MAST design, the 7 branches (A to G) will form a 3-level Merkle tree, plus an "overall condition" describing the relationship of different branches. In redemption, the "overall condition", executed branches (e.g. B, D, F), and Merkle path data will be provided for validation.

In the current proposal, the user has to transform the redeem condition into 12 mutually exclusive branches and form a 4-level Merkle tree, and present only one branch in redemption:

```
A and C and F
B and C and F
A and D and F
.
.
B and E and G
```

One way to implement the general MAST design is using a combination of `OP_EVAL`, `OP_CAT`, and `OP_HASH256`. However, that will suffer from the problems of `OP_EVAL`, including risks of indefinite program loop and inability to do static program analysis. A complicated implementation is required to fix these problems and is difficult to review.

The advantages of the current proposal are:

- `Subscript` are located at a fixed position in the witness stack. This allows static program analysis, such as static `SigOpsCost` counting and early termination of scripts with disabled opcodes.
- If different parties in a contract do not want to expose their scripts to each other, they may provide only `H(Subscript)` and keep the `Subscript` private until redemption.
- If they are willing to share the actual scripts, they may combine them into one `Subscript` for each branch, saving some `nOpCount` and a few bytes of witness space.

The are some disadvantages, but only when the redemption condition is very complicated:

- It may require more branches than a general MAST design (as shown in the previous example) and take more witness space in redemption
- Creation and storage of the MAST structure may take more time and space. However, such additional costs affect only the related parties in the contract but not any other Bitcoin users.

MAST Version

This proposal allows users to indicate the version of scripting language in the witness, which is cheaper than doing that in `scriptPubKey` or `scriptSig`. Undefined versions remain anyone-can-spend and are reserved for future expansions. A new version could be used for relaxing constraints (e.g. the 10,000 bytes size limit of `MAST Script`), adding or redefining opcodes, or even introducing a completely novel scripting system.

nOpCount limit

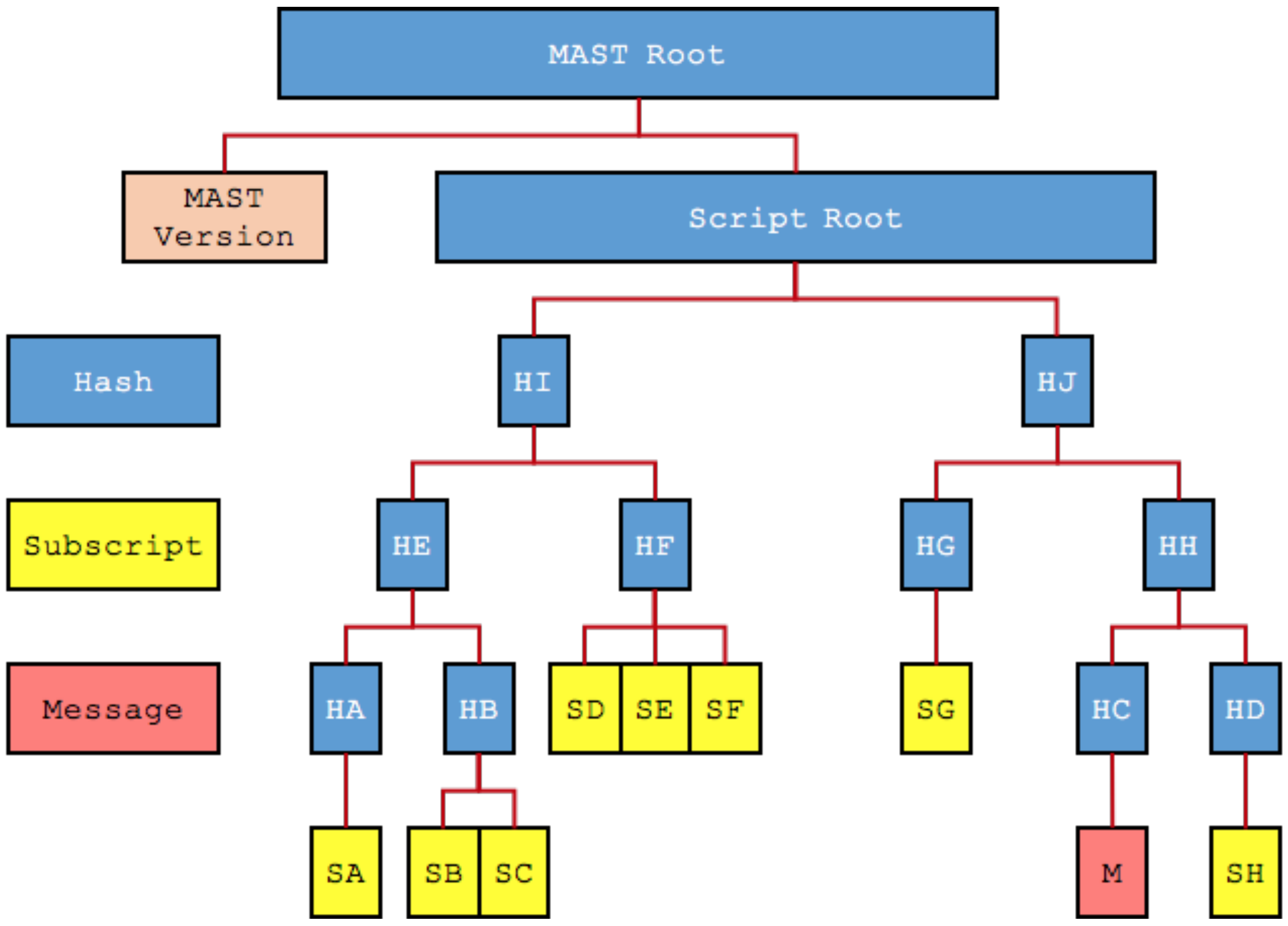
In version 0 MAST, the extra hashing operations in calculating the `MAST Root` are counted towards the 201 `nOpCount` limit to prevent abusive use. This limitation is not applied to undefined `MAST Version` for flexibility, but it is constrained by the 255 `Subscript` and 32 `Depth` limits.

Script evaluation

This proposal requires script evaluation resulting in an empty stack, instead of a single `TRUE` value as in P2WSH. This allows each party in a contract to provide its own `Subscript`, and demonstrate the required `Input Stack` to clean up its own `Subscript`. In this case, order of the `Subscript` is not important since the overall objective is to clean up the stack after evaluation.

Examples

Calculation of MAST Root



```
Subscript:
  SA = 1 EQUALVERIFY (0x5188)
  SB = 2 EQUALVERIFY (0x5288)
  SC = 3 EQUALVERIFY (0x5388)
  SD = 4 EQUALVERIFY (0x5488)
  SE = 5 EQUALVERIFY (0x5588)
  SF = 6 EQUALVERIFY (0x5688)
  SG = 7 EQUALVERIFY (0x5788)
  SH = 8 EQUALVERIFY (0x5888)
  M = RETURN "Hello" (0x6a0548656c6c6f)
Hash:
  HA = H(0x01|H(SA)) = H(0x015acb54166e0db370cd1b05a29120373568dacea2abc3748459ec3da2106e4b4e) = 0xd385d7268ad7
  HB = H(0x02|H(SB)|H(SC)) = 0x7cbfa08e44ea9f4f996873be95d9bffd97d4b91a5af32cc5f64efb8461727cdd
  HF = H(0x03|H(SD)|H(SE)|H(SF)) = 0x4611414355945a7c2fcc62a53a0004821b87e68f93048ffba7a55a3cb1e9783b
  HG = H(0x01|H(SG)) = 0xaa5fbdf58264650eadec33691ba1e7606d0a62f570eea348a465c55bc86ffc10
  HC = H(0x01|H(M)) = 0x70426d480d5b28d93c5be54803681f99abf4e8df4eab4dc87aaa543f0d138159
  HD = H(0x0x|H(SH)) = 0x8482f6c9c3fe90dd4d533b4efedb6a241b95ec9267d1bd5aaaee36d2ce2dd6da
```

```
    HE = H(HA|HB) = 0x049b9f2f94f0a9bdea624e39cd7d6b27a365c6a0545bf0e9d88d86eff4894210
    HH = H(HC|HD) = 0xc709fdc632f370f3367da45378d1cf430c5fda6805e731ad5761c213cf2d276e
    HI = H(HE|HF) = 0xead5e1a1e7e41b77b794f091df9be3f0e9f41d47304eb43dece90688f69843b7
    HJ = H(HG|HH) = 0xd00fc690c4700d0f983f9700740066531ea826b21a4cbc62f80317261723d477
    Script Root = H(HI|HJ) = 0x26d5235d20daf1440a15a248f5b5b4f201392128072c55afa64a26ccc6f56bd9
    MAST Root = H(MAST Version|Script Root) = H(0x0000000026d5235d20daf1440a15a248f5b5b4f201392128072c55afa64a26c
```

The scriptPubKey with native witness program is:

```
  1 <0xb4b706e0c02eab9aba58419eb7ea2a286fb1c01d7406105fc12742bf8a3f97c9>
  (0x5120b4b706e0c02eab9aba58419eb7ea2a286fb1c01d7406105fc12742bf8a3f97c9)
```

To redeem with the `SD|SE|SF` branch, the witness is

```
  Script_stack_1: 0x06
  Script_stack_2: 0x05
  Script_stack_3: 0x04
  Subscript_1:    0x5488
  Subscript_2:    0x5588
  Subscript_3:    0x5688
  Position:       0x01 (HF is the second hash in its level)
  Path (HE|HJ):   0x049b9f2f94f0a9bdea624e39cd7d6b27a365c6a0545bf0e9d88d86eff4894210d00fc690c4700d0f983f970074006
  Metadata:       0x03 (3 Subscript)
```

Imbalance MAST

When constructing a MAST, if the user believes that some of the branches are more likely to be executed, they may put them closer to the `Script Root`. It will save some witness space when the preferred branches are actually executed.

Escrow with Timeout

The following is the "Escrow with Timeout" example in BIP112:

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

Using compressed public key, the size of this script is 150 bytes.

With MAST, this script could be broken down into 2 mutually exclusive branches:[2]

```
    2 <Alice's pubkey> <Bob's pubkey> <Escrow's pubkey> 3 CHECKMULTISIGVERIFY (105 bytes)
    "30d" CHECKSEQUENCEVERIFY <Alice's pubkey> CHECKSIGVERIFY (42 bytes)
```

Since only one branch will be published, it is more difficult for a blockchain analyst to determine the details of the escrow.

Hashed Time-Lock Contract

The following is the "Hashed TIme-Lock Contract" example in BIP112:

```
    HASH160 DUP <R-HASH> EQUAL
    IF
        "24h" CHECKSEQUENCEVERIFY
        2DROP
        <Alice's pubkey>
    ELSE
        <Commit-Revocation-Hash> EQUAL
        NOTIF
            "Timestamp" CHECKLOCKTIMEVERIFY DROP
        ENDIF
        <Bob's pubkey>
    ENDIF
    CHECKSIG
```

To create a MAST Root, it is flattened to 3 mutually exclusive branches:

```
    HASH160 <R-HASH> EQUALVERIFY "24h" CHECKSEQUENCEVERIFY <Alice's pubkey> CHECKSIGVERIFY
    HASH160 <Commit-Revocation-Hash> EQUALVERIFY <Bob's pubkey> CHECKSIGVERIFY
    "Timestamp" CHECKLOCKTIMEVERIFY <Bob's pubkey> CHECKSIGVERIFY
```

which significantly improves readability and reduces the witness size when it is redeemed.

Large multi-signature constructs

The current CHECKMULTISIG supports up to 20 public keys. Although it is possible to extend it beyond 20 keys by using multiple CHECKSIGs and IF/ELSE conditions, the construction could be very complicated and soon use up the 10,000 bytes and 201 `nOpCount` limit.

With MAST, large and complex multi-signature constructs could be flattened to many simple CHECKMULTISIGVERIFY conditions. For example, a 3-of-2000 multi-signature scheme could be expressed as 1,331,334,000 3-of-3 CHECKMULTISIGVERIFY, which forms a 31-level MAST. The scriptPubKey still maintains a fixed size of 34 bytes, and the redemption witness will be very compact, with less than 1,500 bytes.

Commitment of non-consensus enforced data

Currently, committing non-consensus enforced data in the scriptPubKey requires the use of OP_RETURN which occupies additional block space. With MAST, users may commit such data as a branch. Depends on the number of executable branches, inclusion of such a commitment may incur no extra witness space, or 32 bytes at most.

An useful case would be specifying "message-signing keys", which are not valid for spending, but allow users to sign any message without touching the cold storage "funding key".

Backward compatibility

As a soft fork, older software will continue to operate without modification. Non-upgraded nodes, however, will consider MAST programs as anyone-can-spend scripts. Wallets should always be wary of anyone-can-spend scripts and treat them with suspicion.

Deployment

This BIP depends on BIP141 and will be deployed by version-bits BIP9 after BIP141 is enforced. Exact details TBD.

Credits

The idea of MAST originates from Russell O’Connor, Pieter Wuille, and Peter Todd.

Reference Implementation

https://github.com/jl2012/bitcoin/tree/bip114v2 (WIP)

```
//New rules apply if version byte is 1 and witness program size is 32 bytes
if (witversion == 1 && program.size() == 32 && (flags & SCRIPT_VERIFY_MAST)) {
    CHashWriter sRoot(SER_GETHASH, 0);
    CHashWriter sScriptHash(SER_GETHASH, 0);
    uint32_t nMASTVersion = 0;
    size_t stacksize = witness.stack.size();
    if (stacksize < 4)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    std::vector<unsigned char> metadata = witness.stack.back(); // The last witness stack item is metadata
    if (metadata.size() < 1 || metadata.size() > 5)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);

    // The first byte of metadata is the number of subscripts (1 to 255)
    uint32_t nSubscript = static_cast<uint32_t>(metadata[0]);
    if (nSubscript == 0 || stacksize < nSubscript + 3)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    int nOpCount = nSubscript; // Each condition consumes a nOpCount
    sScriptHash << metadata[0];

    // The rest of metadata is MAST version in minimally-coded unsigned little endian int
    if (metadata.back() == 0)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    if (metadata.size() > 1) {
        for (size_t i = 1; i != metadata.size(); ++i)
            nMASTVersion |= static_cast<uint32_t>(metadata[i]) << 8 * (i - 1);
    }

    // Unknown MAST version is non-standard
    if (nMASTVersion > 0 && flags & SCRIPT_VERIFY_DISCOURAGE_UPGRADABLE_WITNESS_PROGRAM)
        return set_error(serror, SCRIPT_ERR_DISCOURAGE_UPGRADABLE_WITNESS_PROGRAM);

    sRoot << nMASTVersion;

    // The second last witness stack item is the pathdata
    // Size of pathdata must be divisible by 32 (0 is allowed)
    // Depth of the Merkle tree is implied by the size of pathdata, and must not be greater than 32
    std::vector<unsigned char> pathdata = witness.stack.at(stacksize - 2);
    if (pathdata.size() & 0x1F)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    unsigned int depth = pathdata.size() >> 5;
    if (depth > 32)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);

    // Each level of Merkle tree consumes a nOpCount
    // Evaluation of version 0 MAST terminates early if there are too many nOpCount
    // Not enforced in unknown MAST version for upgrade flexibility
    nOpCount = nOpCount + depth;
    if (nMASTVersion == 0 && nOpCount > MAX_OPS_PER_SCRIPT)
        return set_error(serror, SCRIPT_ERR_OP_COUNT);

    // path is a vector of 32-byte hashes
    std::vector <uint256> path;
    path.resize(depth);
    for (unsigned int j = 0; j < depth; j++)
        memcpy(path[j].begin(), &pathdata[32 * j], 32);

    // The third last witness stack item is the positiondata
    // Position is in minimally-coded unsigned little endian int
    std::vector<unsigned char> positiondata = witness.stack.at(stacksize - 3);
    if (positiondata.size() > 4)
        return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    uint32_t position = 0;
    if (positiondata.size() > 0) {
        if (positiondata.back() == 0)
            return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
        for (size_t k = 0; k != positiondata.size(); ++k)
            position |= static_cast<uint32_t>(positiondata[k]) << 8 * k;
    }

    // Position value must not exceed the number of leaves at the depth
    if (depth < 32) {
        if (position >= (1U << depth))
            return set_error(serror, SCRIPT_ERR_INVALID_MAST_STACK);
    }

    // Sub-scripts are located before positiondata
    for (size_t i = stacksize - nSubscript - 3; i <= stacksize - 4; i++) {
        CScript subscript(witness.stack.at(i).begin(), witness.stack.at(i).end());

        // Evaluation of version 0 MAST terminates early if script is oversize
        // Not enforced in unknown MAST version for upgrade flexibility
        if (nMASTVersion == 0 && (scriptPubKey.size() + subscript.size()) > MAX_SCRIPT_SIZE)
            return set_error(serror, SCRIPT_ERR_SCRIPT_SIZE);
        uint256 hashSubScript;
        CHash256().Write(&subscript[0], subscript.size()).Finalize(hashSubScript.begin());
        sScriptHash << hashSubScript;
        scriptPubKey = scriptPubKey + subscript; // Final scriptPubKey is a serialization of subscripts
    }
    uint256 hashScript = sScriptHash.GetHash();

    // Calculate MAST Root and compare against witness program
    uint256 rootScript = ComputeMerkleRootFromBranch(hashScript, path, position);
    sRoot << rootScript;
    uint256 rootMAST = sRoot.GetHash();
    if (memcmp(rootMAST.begin(), &program[0], 32))
        return set_error(serror, SCRIPT_ERR_WITNESS_PROGRAM_MISMATCH);

    if (nMASTVersion == 0) {
        stack = std::vector<std::vector<unsigned char> >(witness.stack.begin(), witness.stack.end() - 3 - nSubscr
        for (unsigned int i = 0; i < stack.size(); i++) {
            if (stack.at(i).size() > MAX_SCRIPT_ELEMENT_SIZE)
                return set_error(serror, SCRIPT_ERR_PUSH_SIZE);
        }

        // Script evaluation must not fail, and return an empty stack
        if (!EvalScript(stack, scriptPubKey, flags, checker, SIGVERSION_WITNESS_V1, nOpCount, serror))
            return false;
        if (stack.size() != 0)
            return set_error(serror, SCRIPT_ERR_EVAL_FALSE);
    }

    return set_success(serror);
}
```

Copying from `src/consensus/merkle.cpp`:

```
uint256 ComputeMerkleRootFromBranch(const uint256& leaf, const std::vector<uint256>& vMerkleBranch, uint32_t nInd
    uint256 hash = leaf;
    for (std::vector<uint256>::const_iterator it = vMerkleBranch.begin(); it != vMerkleBranch.end(); ++it) {
        if (nIndex & 1) {
            hash = Hash(BEGIN(*it), END(*it), BEGIN(hash), END(hash));
        } else {
            hash = Hash(BEGIN(hash), END(hash), BEGIN(*it), END(*it));
        }
        nIndex >>= 1;
    }
    return hash;
}
```

References

- BIP141 Segregated Witness (Consensus layer)

Copyright