```
  BIP: 340
  Title: Schnorr Signatures for secp256k1
  Authors: Pieter Wuille
           Jonas Nick
           Tim Ruffing
  Comments-Summary: No comments yet.
  Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0340
  Status: Deployed
  Type: Specification
  Assigned: 2020-01-19
  License: BSD-2-Clause
  License-Code: BSD-2-Clause OR MIT OR CC0-1.0
  Discussion: 2018-07-06: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-July/016203.html [bitcoin-dev]
Schnorr signatures BIP
```

== Introduction ==

=== Abstract ===

This document proposes a standard for 64-byte Schnorr signatures over the elliptic curve ''secp256k1''.

=== Copyright ===

This document is licensed under the 2-clause BSD license.

=== Motivation ===

Bitcoin has traditionally used [https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm ECDSA] signatures over the [https://www.secg.org/sec2-v2.pdf secp256k1 curve] with [https://en.wikipedia.org/wiki/SHA-2 SHA256] hashes for authenticating transactions. These are [https://www.secg.org/sec1-v2.pdf standardized], but have a number of downsides compared to [http://publikationen.ub.uni-frankfurt.de/opus4/files/4280/schnorr.pdf Schnorr signatures] over the same curve:

- '''Provable security''': Schnorr signatures are provably secure. In more detail, they are ''strongly unforgeable under chosen message attack (SUF-CMA)''Informally, this means that without knowledge of the secret key but given valid signatures of arbitrary messages, it is not possible to come up with further valid signatures. [https://www.di.ens.fr/~pointche/Documents/Papers/2000_joc.pdf in the random oracle model assuming the hardness of the elliptic curve discrete logarithm problem (ECDLP)] and [http://www.neven.org/papers/schnorr.pdf in the generic group model assuming variants of preimage and second preimage resistance of the used hash function]A detailed security proof in the random oracle model, which essentially restates [https://www.di.ens.fr/~pointche/Documents/Papers/2000_joc.pdf the original security proof by Pointcheval and Stern] more explicitly, can be found in [https://eprint.iacr.org/2016/191 a paper by Kiltz, Masny and Pan]. All these security proofs assume a variant of Schnorr signatures that use ''(e,s)'' instead of ''(R,s)'' (see Design above). Since we use a unique encoding of ''R'', there is an efficiently computable bijection that maps ''(R,s)'' to ''(e,s)'', which allows to convert a successful SUF-CMA attacker for the ''(e,s)'' variant to a successful SUF-CMA attacker for the ''(R,s)'' variant (and vice-versa). Furthermore, the proofs consider a variant of Schnorr signatures without key prefixing (see Design above), but it can be verified that the proofs are also correct for the variant with key prefixing. As a result, all the aforementioned security proofs apply to the variant of Schnorr signatures proposed in this document.. In contrast, the [https://nbn-resolving.de/urn:nbn:de:hbz:294-60803 best known results for the provable security of ECDSA] rely on stronger assumptions.
- '''Non-malleability''': The SUF-CMA security of Schnorr signatures implies that they are non-malleable. On the other hand, ECDSA signatures are inherently malleableIf ''(r,s)'' is a valid ECDSA signature for a given message and key, then ''(r,n-s)'' is also valid for the same message and key. If ECDSA is restricted to only permit one of the two variants (as Bitcoin does through a policy rule on the network), it can be [https://nbn-resolving.de/urn:nbn:de:hbz:294-60803 proven] non-malleable under stronger than usual assumptions.; a third party without access to the secret key can alter an existing valid signature for a given public key and message into another signature that is valid for the same key and message. This issue is discussed in [[bip-0062.mediawiki|BIP62]] and [[bip-0146.mediawiki|BIP146]].
- '''Linearity''': Schnorr signatures provide a simple and efficient method that enables multiple collaborating parties to produce a signature that is valid for the sum of their public keys. This is the building block for various

higher-level constructions that improve efficiency and privacy, such as multisignatures and others (see Applications below).

For all these advantages, there are virtually no disadvantages, apart from not being standardized. This document seeks to change that. As we propose a new standard, a number of improvements not specific to Schnorr signatures can be made:

- '''Signature encoding''': Instead of using [https://en.wikipedia.org/wiki/X.690#DER_encoding DER]-encoding for signatures (which are variable size, and up to 72 bytes), we can use a simple fixed 64-byte format.
- '''Public key encoding''': Instead of using [https://www.secg.org/sec1-v2.pdf ''compressed''] 33-byte encodings of elliptic curve points which are common in Bitcoin today, public keys in this proposal are encoded as 32 bytes.
- '''Batch verification''': The specific formulation of ECDSA signatures that is standardized cannot be verified more efficiently in batch compared to individually, unless additional witness data is added. Changing the signature scheme offers an opportunity to address this.
- '''Completely specified''': To be safe for usage in consensus systems, the verification algorithm must be completely specified at the byte level. This guarantees that nobody can construct a signature that is valid to some verifiers but not all. This is traditionally not a requirement for digital signature schemes, and the lack of exact specification for the DER parsing of ECDSA signatures has caused problems for Bitcoin [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-July/009697.html in the past], needing [[bip-0066.mediawiki|BIP66]] to address it. In this document we aim to meet this property by design. For batch verification, which is inherently non-deterministic as the verifier can choose their batches, this property implies that the outcome of verification may only differ from individual verifications with negligible probability, even to an attacker who intentionally tries to make batch- and non-batch verification differ.

By reusing the same curve and hash function as Bitcoin uses for ECDSA, we are able to retain existing mechanisms for choosing secret and public keys, and we avoid introducing new assumptions about the security of elliptic curves and hash functions.

== Description ==

We first build up the algebraic formulation of the signature scheme by going through the design choices. Afterwards, we specify the exact encodings and operations.

=== Design ===

'''Schnorr signature variant''' Elliptic Curve Schnorr signatures for message ''m'' and public key ''P'' generally involve a point ''R'', integers ''e'' and ''s'' picked by the signer, and the base point ''G'' which satisfy ''e = hash(R || m)'' and ''s⋅G = R + e⋅P''. Two formulations exist, depending on whether the signer reveals ''e'' or ''R'':

## Signatures are pairs ''(e, s)'' that satisfy ''e = hash(s⋅G - e⋅P || m)''. This variant avoids minor complexity introduced by the encoding of the point ''R'' in the signature (see paragraphs "Encoding R and public key point P" and "Implicit Y coordinates" further below in this subsection). Moreover, revealing ''e'' instead of ''R'' allows for potentially shorter signatures: Whereas an encoding of ''R'' inherently needs about 32 bytes, the hash ''e'' can be tuned to be shorter than 32 bytes, and [http://www.neven.org/papers/schnorr.pdf a short hash of only 16 bytes suffices to provide SUF-CMA

**security at the target security level of 128 bits]. However, a major drawback of this optimization is that finding collisions in a short hash function is easy. This complicates the implementation of secure signing protocols in scenarios in which a group of mutually distrusting signers work together to produce a single joint signature (see Applications below). In these scenarios, which are not captured by the SUF-CMA model due its assumption of a single honest signer, a promising attack strategy for malicious co-signers is to find a collision in the hash function in order to obtain a valid signature on a message that an honest co-signer did not intend to sign.**

**Signatures are pairs ''(R, s)'' that satisfy ''s • G = R + hash(R || m) • P''. This supports batch verification, as there are no elliptic curve operations inside the hashes. Batch verification enables significant speedups.The speedup that results from batch verification can be demonstrated with the cryptography library [https://github.com/jonasnick/secp256k1/blob/schnorrsig-batch-verify/doc/speedup-batch.md libsecp256k1].**

Since we would like to avoid the fragility that comes with short hashes, the ''e'' variant does not provide significant advantages. We choose the ''R''-option, which supports batch verification.

'''Key prefixing''' Using the verification rule above directly makes Schnorr signatures vulnerable to "related-key attacks" in which a third party can convert a signature ''(R, s)'' for public key ''P'' into a signature ''(R, s + a•hash(R || m))'' for public key ''P + a•G'' and the same message ''m'', for any given additive tweak ''a'' to the signing key. This would render signatures insecure when keys are generated using [[bip-0032.mediawiki#public-parent-key--public-child-key|BIP32's unhardened derivation]] and other methods that rely on additive tweaks to existing keys such as Taproot.

To protect against these attacks, we choose ''key prefixed''A limitation of committing to the public key (rather than to a short hash of it, or not at all) is that it removes the ability for public key recovery or verifying signatures against a short public key hash. These constructions are generally incompatible with batch verification. Schnorr signatures which means that the public key is prefixed to the message in the challenge hash input. This changes the equation to ''s•G = R + hash(R || P || m)•P''. [https://eprint.iacr.org/2015/1135.pdf It can be shown] that key prefixing protects against related-key attacks with additive tweaks. In general, key prefixing increases robustness in multi-user settings, e.g., it seems to

be a requirement for proving multiparty signing protocols (such as MuSig, MuSig2, and FROST) secure (see Applications below).

We note that key prefixing is not strictly necessary for transaction signatures as used in Bitcoin currently, because signed transactions indirectly commit to the public keys already, i.e., ''m'' contains a commitment to ''pk''. However, this indirect commitment should not be relied upon because it may change with proposals such as SIGHASH_NOINPUT ([[bip-0118.mediawiki|BIP118]]), and would render the signature scheme unsuitable for other purposes than signing transactions, e.g., [https://bitcoin.org/en/developer-reference#signmessage signing ordinary messages].

'''Encoding R and public key point P''' There exist several possibilities for encoding elliptic curve points:

## Encoding the full X and Y coordinates of ''P'' and ''R'', resulting in a 64-byte public key and a 96-byte signature.

## Encoding the full X coordinate and one bit of the Y coordinate to determine one of the two possible Y coordinates. This would result in 33-byte public keys and 65-byte signatures.

## Encoding only the X coordinate, resulting in 32-byte public keys and 64-byte signatures.

Using the first option would be slightly more efficient for verification (around 10%), but we prioritize compactness, and therefore choose option 3.

'''Implicit Y coordinates''' In order to support efficient verification and batch verification, the Y coordinate of ''P'' and of ''R'' cannot be ambiguous (every valid X coordinate has two possible Y coordinates). We have a choice between several options for symmetry breaking:

## Implicitly choosing the Y coordinate that is in the lower half.

## Implicitly choosing the Y coordinate that is evenSince ''p'' is odd, negation modulo ''p'' will map even numbers to odd numbers and the other way around. This means that for a valid X coordinate, one of the corresponding Y coordinates will be even, and the other will be odd..

## Implicitly choosing the Y coordinate that is a quadratic residue (i.e. has a square root modulo ''p'').

The second option offers the greatest compatibility with existing key generation systems, where the standard 33-byte compressed public key format consists of a byte indicating the oddness of the Y coordinate, plus the full X coordinate. To avoid gratuitous incompatibilities, we pick that option for ''P'', and thus our X-only public keys become equivalent to a compressed public key that is the X-only key prefixed by the byte 0x02. For consistency, the same is done for ''R''An

earlier version of this draft used the third option instead, based on a belief that this would in general trade signing efficiency for verification efficiency. When using Jacobian coordinates, a common optimization in ECC implementations, it is possible to determine if a Y coordinate is a quadratic residue by computing the Legendre symbol, without converting to affine coordinates first (which needs a modular inversion). As modular inverses and Legendre symbols have similar [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2020-August/018081.html performance] in practice, this trade-off is not worth it..

Despite halving the size of the set of valid public keys, implicit Y coordinates are not a reduction in security. Informally, if a fast algorithm existed to compute the discrete logarithm of an X-only public key, then it could also be used to compute the discrete logarithm of a full public key: apply it to the X coordinate, and then optionally negate the result. This shows that breaking an X-only public key can be at most a small constant term faster than breaking a full one.This can be formalized by a simple reduction that reduces an attack on Schnorr signatures with implicit Y coordinates to an attack to Schnorr signatures with explicit Y coordinates. The reduction works by reencoding public keys and negating the result of the hash function, which is modeled as random oracle, whenever the challenge public key has an explicit Y coordinate that is odd. A proof sketch can be found [https://medium.com/blockstream/reducing-bitcoin-transaction-sizes-with-x-only-pubkeys-f86476af05d7 here]..

'''Tagged Hashes''' Cryptographic hash functions are used for multiple purposes in the specification below and in Bitcoin in general. To make sure hashes used in one context can't be reinterpreted in another one, hash functions can be tweaked with a context-dependent tag name, in such a way that collisions across contexts can be assumed to be infeasible. Such collisions obviously can not be ruled out completely, but only for schemes using tagging with a unique name. As for other schemes collisions are at least less likely with tagging than without.

For example, without tagged hashing a BIP340 signature could also be valid for a signature scheme where the only difference is that the arguments to the hash function are reordered. Worse, if the BIP340 nonce derivation function was copied or independently created, then the nonce could be accidentally reused in the other scheme leaking the secret key.

This proposal suggests to include the tag by prefixing the hashed data with ''SHA256(tag) || SHA256(tag)''. Because this is a 64-byte long context-specific constant and the ''SHA256'' block size is also 64 bytes, optimized implementations are possible (identical to SHA256 itself, but with a modified initial state). Using SHA256 of the tag name itself is reasonably simple and efficient for implementations that don't choose to use the optimization. In general, tags can be arbitrary byte arrays, but are suggested to be textual descriptions in UTF-8 encoding.

'''Final scheme''' As a result, our final scheme ends up using public key ''pk'' which is the X coordinate of a point ''P'' on the curve whose Y coordinate is even and signatures ''(r,s)'' where ''r'' is the X coordinate of a point ''R'' whose Y coordinate is even. The signature satisfies ''s⋅G = R + tagged_hash(r || pk || m)⋅P''.

=== Specification ===

The following conventions are used, with constants as defined for [https://www.secg.org/sec2-v2.pdf secp256k1]. We note that adapting this specification to other elliptic curves is not straightforward and can result in an insecure schemeAmong other pitfalls, using the specification with a curve whose order is not close to the size of the range of the nonce derivation function is insecure..

- Lowercase variables represent integers or byte arrays. ** The constant ''p'' refers to the field size, ''0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F''. ** The constant ''n'' refers to the curve order, ''0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141''.
- Uppercase variables refer to points on the curve with equation ''$y^2 = x^3 + 7$'' over the integers modulo ''p''. ** ''is_infinite(P)'' returns whether or not ''P'' is the point at infinity. ** ''x(P)'' and ''y(P)'' are integers in the range ''0..p-1'' and refer to the X and Y coordinates of a point ''P'' (assuming it is not infinity). ** The constant ''G'' refers to the base point, for which ''x(G) = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798'' and ''y(G) = 0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8''. ** Addition of points refers to the usual [https://en.wikipedia.org/wiki/Elliptic_curve#The_group_law elliptic curve group operation]. ** [https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication Multiplication (⋅) of an integer and a point] refers to the repeated application of the group operation.
- Functions and operations: ** ''||'' refers to byte array concatenation. ** The function ''x[i:j]'', where ''x'' is a byte array and ''i, j ≥ 0'', returns a ''(j - i)''-byte array with a copy of the ''i''-th byte (inclusive) to the ''j''-th byte

(exclusive) of ''x''. ** The function ''bytes(x)'', where ''x'' is an integer, returns the 32-byte encoding of ''x'', most significant byte first. ** The function ''bytes(P)'', where ''P'' is a point, returns ''bytes(x(P))''. ** The function ''int(x)'', where ''x'' is a 32-byte array, returns the 256-bit unsigned integer whose most significant byte first encoding is ''x''. ** The function ''has_even_y(P)'', where ''P'' is a point for which ''not is_infinite(P)'', returns ''y(P) mod 2 = 0''. ** The function ''lift_x(x)'', where ''x'' is a 256-bit unsigned integer, returns the point ''P'' for which ''x(P) = x'' Given a candidate X coordinate ''x'' in the range ''0..p-1'', there exist either exactly two or exactly zero valid Y coordinates. If no valid Y coordinate exists, then ''x'' is not a valid X coordinate either, i.e., no point ''P'' exists for which ''x(P) = x''. The valid Y coordinates for a given candidate ''x'' are the square roots of ''c = $x^3$ + 7 mod p'' and they can be computed as ''y = ±$c^{(p+1)/4}$ mod p'' (see [https://en.wikipedia.org/wiki/Quadratic_residue#Prime_or_prime_power_modulus Quadratic residue]) if they exist, which can be checked by squaring and comparing with ''c''. and ''has_even_y(P)'', or fails if ''x'' is greater than ''p-1'' or no such point exists. The function ''lift_x(x)'' is equivalent to the following pseudocode: *** Fail if ''x ≥ p''. *** Let ''c = $x^3$ + 7 mod p''. *** Let ''y = $c^{(p+1)/4}$ mod p''. *** Fail if ''c ≠ $y^2$ mod p''. *** Return the unique point ''P'' such that ''x(P) = x'' and ''y(P) = y'' if ''y mod 2 = 0'' or ''y(P) = p-y'' otherwise. ** The function ''$\text{hash}_{\text{name}}$(x)'' where ''x'' is a byte array returns the 32-byte hash ''SHA256(SHA256(tag) || SHA256(tag) || x)'', where ''tag'' is the UTF-8 encoding of ''name''.

==== Public Key Generation ====

Input:

- The secret key ''sk'': a 32-byte array, freshly generated uniformly at random

The algorithm ''PubKey(sk)'' is defined as:

- Let ''d' = int(sk)''.
- Fail if ''d' = 0'' or ''d' ≥ n''.
- Return ''bytes(d'⋅G)''.

Note that we use a very different public key format (32 bytes) than the ones used by existing systems (which typically use elliptic curve points as public keys, or 33-byte or 65-byte encodings of them). A side effect is that ''PubKey(sk) = PubKey(bytes(n - int(sk))'', so every public key has two corresponding secret keys.

==== Public Key Conversion ====

As an alternative to generating keys randomly, it is also possible and safe to repurpose existing key generation algorithms for ECDSA in a compatible way. The secret keys constructed by such an algorithm can be used as ''sk'' directly. The public keys constructed by such an algorithm (assuming they use the 33-byte compressed encoding) need to be converted by dropping the first byte. Specifically, [[bip-0032.mediawiki|BIP32]] and schemes built on top of it remain usable.

==== Default Signing ====

Input:

- The secret key ''sk'': a 32-byte array
- The message ''m'': a byte array
- Auxiliary random data ''a'': a 32-byte array

The algorithm ''Sign(sk, m)'' is defined as:

- Let ''d' = int(sk)''
- Fail if ''d' = 0'' or ''d' ≥ n''
- Let ''P = d'⋅G''
- Let ''d = d' '' if ''has_even_y(P)'', otherwise let ''d = n - d' ''.
- Let ''t'' be the byte-wise xor of ''bytes(d)'' and ''$\text{hash}_{\text{BIP0340/aux}}$(a)''The auxiliary random data is hashed (with a unique tag) as a precaution against situations where the randomness may be correlated with the private key itself. It is xored with the private key (rather than combined with it in a hash) to reduce the number of operations exposed to the actual secret key..
- Let ''rand = $\text{hash}_{\text{BIP0340/nonce}}$(t || bytes(P) || m)''Including the [https://moderncrypto.org/mail-archive/curves/2020/001012.html public key as input to the nonce hash] helps ensure the robustness of the

signing algorithm by preventing leakage of the secret key if the calculation of the public key ''P'' is performed incorrectly or maliciously, for example if it is left to the caller for performance reasons..

- Let ''k' = int(rand) mod n''Note that in general, taking a uniformly random 256-bit integer modulo the curve order will produce an unacceptably biased result. However, for the secp256k1 curve, the order is sufficiently close to ''$2^{256}$'' that this bias is not observable (''1 - n / $2^{256}$'' is around ''1.27 * $2^{-128}$'')..
- Fail if ''k' = 0''.
- Let ''R = k'⋅G''.
- Let ''k = k' '' if ''has_even_y(R)'', otherwise let ''k = n - k' ''.
- Let ''e = int($hash_{BIP0340/challenge}$(bytes(R) || bytes(P) || m)) mod n''.
- Let ''sig = bytes(R) || bytes((k + ed) mod n)''.
- If ''Verify(bytes(P), m, sig)'' (see below) returns failure, abortVerifying the signature before leaving the signer prevents random or attacker provoked computation errors. This prevents publishing invalid signatures which may leak information about the secret key. It is recommended, but can be omitted if the computation cost is prohibitive..
- Return the signature ''sig''.

The auxiliary random data should be set to fresh randomness generated at signing time, resulting in what is called a ''synthetic nonce''. Using 32 bytes of randomness is optimal. If obtaining randomness is expensive, 16 random bytes can be padded with 16 null bytes to obtain a 32-byte array. If randomness is not available at all at signing time, a simple counter wide enough to not repeat in practice (e.g., 64 bits or wider) and padded with null bytes to a 32 byte-array can be used, or even the constant array with 32 null bytes. Using any non-repeating value increases protection against [https://moderncrypto.org/mail-archive/curves/2017/000925.html fault injection attacks]. Using unpredictable randomness additionally increases protection against other side-channel attacks, and is '''recommended whenever available'''. Note that while this means the resulting nonce is not deterministic, the randomness is only supplemental to security. The normal security properties (excluding side-channel attacks) do not depend on the quality of the signing-time RNG.

==== Alternative Signing ====

It should be noted that various alternative signing algorithms can be used to produce equally valid signatures. The 32-byte ''rand'' value may be generated in other ways, producing a different but still valid signature (in other words, this is not a ''unique'' signature scheme). '''No matter which method is used to generate the ''rand'' value, the value must be a fresh uniformly random 32-byte string which is not even partially predictable for the attacker.''' For nonces without randomness, this implies that the same inputs must not be presented in another context. This can be most reliably accomplished by not reusing the same private key across different signing schemes. For example, if the ''rand'' value was computed as per RFC6979 and the same secret key is used in deterministic ECDSA with RFC6979, the signatures can leak the secret key through nonce reuse.

'''Nonce exfiltration protection''' It is possible to strengthen the nonce generation algorithm using a second device. In this case, the second device contributes randomness which the actual signer provably incorporates into its nonce. This prevents certain attacks where the signer's device is compromised and intentionally tries to leak the secret key through its nonce selection.

'''Multisignatures''' This signature scheme is compatible with various types of multisignature and threshold schemes such as [https://eprint.iacr.org/2020/1261.pdf MuSig2], where a single public key requires holders of multiple secret keys to participate in signing (see Applications below). '''It is important to note that multisignature signing schemes in general are insecure with the ''rand'' generation from the default signing algorithm above (or any other deterministic method).'''

'''Precomputed public key data''' For many uses, the compressed 33-byte encoding of the public key corresponding to the secret key may already be known, making it easy to evaluate ''has_even_y(P)'' and ''bytes(P)''. As such, having signers supply this directly may be more efficient than recalculating the public key from the secret key. However, if this optimization is used and additionally the signature verification at the end of the signing algorithm is dropped for increased efficiency, signers must ensure the public key is correctly calculated and not taken from untrusted sources.

==== Verification ====

Input:

- The public key ''pk'': a 32-byte array

- The message ''m'': a byte array
- A signature ''sig'': a 64-byte array

The algorithm ''Verify(pk, m, sig)'' is defined as:

- Let ''P = lift_x(int(pk))''; fail if that fails.
- Let ''r = int(sig[0:32])''; fail if ''r ≥ p''.
- Let ''s = int(sig[32:64])''; fail if ''s ≥ n''.
- Let ''e = int($hash_{BIP0340/challenge}$(bytes(r) || bytes(P) || m)) mod n''.
- Let ''R = s⋅G - e⋅P''.
- Fail if ''is_infinite(R)''.
- Fail if ''not has_even_y(R)''.
- Fail if ''x(R) ≠ r''.
- Return success iff no failure occurred before reaching this point.

For every valid secret key ''sk'' and message ''m'', ''Verify(PubKey(sk),m,Sign(sk,m))'' will succeed.

Note that the correctness of verification relies on the fact that ''lift_x'' always returns a point with an even Y coordinate. A hypothetical verification algorithm that treats points as public keys, and takes the point ''P'' directly as input would fail any time a point with odd Y is used. While it is possible to correct for this by negating points with odd Y coordinate before further processing, this would result in a scheme where every (message, signature) pair is valid for two public keys (a type of malleability that exists for ECDSA as well, but we don't wish to retain). We avoid these problems by treating just the X coordinate as public key.

==== Batch Verification ====

Input:

- The number ''u'' of signatures
- The public keys ''$pk_{1..u}$'': ''u'' 32-byte arrays
- The messages ''$m_{1..u}$'': ''u'' byte arrays
- The signatures ''$sig_{1..u}$'': ''u'' 64-byte arrays

The algorithm ''BatchVerify($pk_{1..u}$, $m_{1..u}$, $sig_{1..u}$)'' is defined as:

- Generate ''u-1'' random integers ''$a_{2...u}$'' in the range ''1...n-1''. They are generated deterministically using a [https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator CSPRNG] seeded by a cryptographic hash of all inputs of the algorithm, i.e. ''seed = seed_hash($pk_1$..$pk_u$ || $m_1$..$m_u$ || $sig_1$..$sig_u$ )''. A safe choice is to instantiate ''seed_hash'' with SHA256 and use [https://tools.ietf.org/html/rfc8439 ChaCha20] with key ''seed'' as a CSPRNG to generate 256-bit integers, skipping integers not in the range ''1...n-1''.
- For ''i = 1 .. u'': ** Let ''$P_i$ = lift_x(int($pk_i$))''; fail if it fails. ** Let ''$r_i$ = int($sig_i$[0:32])''; fail if ''$r_i$ ≥ p''. ** Let ''$s_i$ = int($sig_i$[32:64])''; fail if ''$s_i$ ≥ n''. ** Let ''$e_i$ = int($hash_{BIP0340/challenge}$(bytes($r_i$) || bytes($P_i$) || $m_i$)) mod n''. ** Let ''$R_i$ = lift_x($r_i$)''; fail if ''lift_x($r_i$)'' fails.
- Fail if ''$(s_1 + a_2s_2 + ... + a_us_u)⋅G ≠ R_1 + a_2⋅R_2 + ... + a_u⋅R_u + e_1⋅P_1 + (a_2e_2)⋅P_2 + ... + (a_ue_u)⋅P_u$''.
- Return success iff no failure occurred before reaching this point.

If all individual signatures are valid (i.e., ''Verify'' would return success for them), ''BatchVerify'' will always return success. If at least one signature is invalid, ''BatchVerify'' will return success with at most a negligible probability.

=== Usage Considerations ===

==== Messages of Arbitrary Size ====

The signature scheme specified in this BIP accepts byte strings of arbitrary size as input messages.In theory, the message size is restricted due to the fact that SHA256 accepts byte strings only up to size of 2^61-1 bytes. It is understood that implementations may reject messages which are too large in their environment or application context, e.g., messages which exceed predefined buffers or would otherwise cause resource exhaustion.

Earlier revisions of this BIP required messages to be exactly 32 bytes. This restriction puts a burden on callers who typically need to perform pre-hashing of the actual input message by feeding it through SHA256 (or another collision-

resistant cryptographic hash function) to create a 32-byte digest which can be passed to signing or verification (as for example done in [[bip-0341.mediawiki|BIP341]].)

Since pre-hashing may not always be desirable, e.g., when actual messages are shorter than 32 bytes,Another reason to omit pre-hashing is to protect against certain types of cryptanalytic advances against the hash function used for pre-hashing: If pre-hashing is used, an attacker that can find collisions in the pre-hashing function can necessarily forge signatures under chosen-message attacks. If pre-hashing is not used, an attacker that can find collisions in SHA256 (as used inside the signature scheme) may not be able to forge signatures. However, this seeming advantage is mostly irrelevant in the context of Bitcoin, which already relies on collision resistance of SHA256 in other places, e.g., for transaction hashes. the restriction to 32-byte messages has been lifted. We note that pre-hashing is recommended for performance reasons in applications that deal with large messages. If large messages are not pre-hashed, the algorithms of the signature scheme will perform more hashing internally. In particular, the signing algorithm needs two sequential hashing passes over the message, which means that the full message must necessarily be kept in memory during signing, and large messages entail a runtime penalty.Typically, messages of 56 bytes or longer enjoy a performance benefit from pre-hashing, assuming the speed of SHA256 inside the signing algorithm matches that of the pre-hashing done by the calling application.

==== Domain Separation ====

It is good cryptographic practice to use a key pair only for a single purpose. Nevertheless, there may be situations in which it may be desirable to use the same key pair in multiple contexts, i.e., to sign different types of messages within the same application or even messages in entirely different applications (e.g., a secret key may be used to sign Bitcoin transactions as well plain text messages).

As a consequence, applications should ensure that a signed application message intended for one context is never deemed valid in a different context (e.g., a signed plain text message should never be misinterpreted as a signed Bitcoin transaction, because this could cause unintended loss of funds). This is called "domain separation" and it is typically realized by partitioning the message space. Even if key pairs are intended to be used only within a single context, domain separation is a good idea because it makes it easy to add more contexts later.

As a best practice, we recommend applications to use exactly one of the following methods to pre-process application messages before passing it to the signature scheme:

- Either, pre-hash the application message using ''hash$_{name}$'', where ''name'' identifies the context uniquely (e.g., "foo-app/signed-bar"),
- or prefix the actual message with a 33-byte string that identifies the context uniquely (e.g., the UTF-8 encoding of "foo-app/signed-bar", padded with null bytes to 33 bytes).

As the two pre-processing methods yield different message sizes (32 bytes vs. at least 33 bytes), there is no risk of collision between them.

== Applications ==

There are several interesting applications beyond simple signatures. While recent academic papers claim that they are also possible with ECDSA, consensus support for Schnorr signature verification would significantly simplify the constructions.

=== Multisignatures and Threshold Signatures ===

By means of an interactive scheme such as [https://eprint.iacr.org/2020/1261.pdf MuSig2] ([[bip-0327.mediawiki|BIP327]]), participants can aggregate their public keys into a single public key which they can jointly sign for. This allows ''n''-of-''n'' multisignatures which, from a verifier's perspective, are no different from ordinary signatures, giving improved privacy and efficiency versus ''CHECKMULTISIG'' or other means.

Moreover, Schnorr signatures are compatible with [https://en.wikipedia.org/wiki/Distributed_key_generation distributed key generation], which enables interactive threshold signatures schemes, e.g., the schemes by [http://cacr.uwaterloo.ca/techreports/2001/corr2001-13.ps Stinson and Strobl (2001)], by [https://link.springer.com/content/pdf/10.1007/s00145-006-0347-3.pdf Gennaro, Jarecki, Krawczyk, and Rabin (2007)], or the [https://eprint.iacr.org/2020/852.pdf FROST] scheme including its variants such as [https://eprint.iacr.org/2023/899.pdf FROST3]. These protocols make it possible to realize ''k''-of-''n'' threshold

signatures, which ensure that any subset of size ''k'' of the set of ''n'' signers can sign but no subset of size less than ''k'' can produce a valid Schnorr signature.

=== Adaptor Signatures ===

[https://download.wpsoftware.net/bitcoin/wizardry/mw-slides/2018-05-18-l2/slides.pdf Adaptor signatures] can be produced by a signer by offsetting his public nonce ''R'' with a known point ''T = t⋅G'', but not offsetting the signature's ''s'' value. A correct signature (or partial signature, as individual signers' contributions to a multisignature are called) on the same message with same nonce will then be equal to the adaptor signature offset by ''t'', meaning that learning ''t'' is equivalent to learning a correct signature. This can be used to enable atomic swaps or even [https://eprint.iacr.org/2018/472 general payment channels] in which the atomicity of disjoint transactions is ensured using the signatures themselves, rather than Bitcoin script support. The resulting transactions will appear to verifiers to be no different from ordinary single-signer transactions, except perhaps for the inclusion of locktime refund logic.

Adaptor signatures, beyond the efficiency and privacy benefits of encoding script semantics into constant-sized signatures, have additional benefits over traditional hash-based payment channels. Specifically, the secret values ''t'' may be reblinded between hops, allowing long chains of transactions to be made atomic while even the participants cannot identify which transactions are part of the chain. Also, because the secret values are chosen at signing time, rather than key generation time, existing outputs may be repurposed for different applications without recourse to the blockchain, even multiple times.

=== Blind Signatures ===

A blind signature protocol is an interactive protocol that enables a signer to sign a message at the behest of another party without learning any information about the signed message or the signature. Schnorr signatures admit a very [http://publikationen.ub.uni-frankfurt.de/files/4292/schnorr.blind_sigs_attack.2001.pdf simple blind signature scheme] which is however insecure because it's vulnerable to [https://www.iacr.org/archive/crypto2002/24420288/24420288.pdf Wagner's attack]. Known mitigations are to let the signer abort a signing session with a certain probability, which can be [https://eprint.iacr.org/2019/877 proven secure under non-standard cryptographic assumptions], or [https://eprint.iacr.org/2022/1676.pdf to use zero-knowledge proofs].

Blind Schnorr signatures could for example be used in [https://github.com/ElementsProject/scriptless-scripts/blob/master/md/partially-blind-swap.md Partially Blind Atomic Swaps], a construction to enable transferring of coins, mediated by an untrusted escrow agent, without connecting the transactors in the public blockchain transaction graph.

== Test Vectors and Reference Code ==

For development and testing purposes, we provide a [[bip-0340/test-vectors.csv|collection of test vectors in CSV format]], a naive, highly inefficient, and non-constant time [[bip-0340/reference.py|pure Python 3.7 reference implementation of the signing and verification algorithm]] as well as the [[bip-0340/test-vectors.py|script used to generate the test vectors]] under the BSD-2-Clause License, or the MIT License, or CC0 1.0, at your choice. The reference implementation is for demonstration purposes only and not to be used in production environments.

== Changelog ==

To help implementers understand updates to this BIP, we keep a list of substantial changes.

- 2022-08: Fix function signature of lift_x in reference code
- 2023-04: Allow messages of arbitrary size
- 2024-05: Update "Applications" section with more recent references
- 2025-04: Change license of test vectors and code

== Footnotes ==

== Acknowledgements ==

This document is the result of many discussions around Schnorr based signatures over the years, and had input from Johnson Lau, Greg Maxwell, Andrew Poelstra, Rusty Russell, and Anthony Towns. The authors further wish to thank all those who provided valuable feedback and reviews, including the participants of the [https://github.com/ajtowns/taproot-review structured reviews].

```
  BIP: 341
  Layer: Consensus (soft fork)
  Title: Taproot: SegWit version 1 spending rules
  Authors: Pieter Wuille
           Jonas Nick
           Anthony Towns
  Comments-Summary: No comments yet.
  Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0341
  Status: Deployed
  Type: Specification
  Assigned: 2020-01-19
  License: BSD-3-Clause
  Discussion: 2019-05-06: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-May/016914.html [bitcoin-dev]
Taproot proposal
              2019-10-09: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-October/017378.html [bitcoin-
dev] Taproot updates
  Requires: 340
```

==Introduction==

===Abstract===

This document proposes a new SegWit version 1 output type, with spending rules based on Taproot, Schnorr signatures, and Merkle branches.

===Copyright===

This document is licensed under the 3-clause BSD license.

===Motivation===

This proposal aims to improve privacy, efficiency, and flexibility of Bitcoin's scripting capabilities without adding new security assumptions'''What does not adding security assumptions mean?''' Unforgeability of signatures is a necessary requirement to prevent theft. At least when treating script execution as a digital signature scheme itself, unforgeability can be [https://github.com/apoelstra/taproot proven] in the Random Oracle Model assuming the Discrete Logarithm problem is hard. A [https://nbn-resolving.de/urn:nbn:de:hbz:294-60803 proof] for unforgeability of ECDSA in the current script system needs non-standard assumptions on top of that. Note that it is hard in general to model exactly what security for script means, as it depends on the policies and protocols used by wallet software.. Specifically, it seeks to minimize how much information about the spendability conditions of a transaction output is revealed on chain at creation or spending time and to add a number of upgrade mechanisms, while fixing a few minor but long-standing issues.

==Design==

A number of related ideas for improving Bitcoin's scripting capabilities have been previously proposed: Schnorr signatures ([[bip-0340.mediawiki|BIP340]]), Merkle branches ("MAST", [[bip-0114.mediawiki|BIP114]], [[bip-0117.mediawiki|BIP117]]), new sighash modes ([[bip-0118.mediawiki|BIP118]]), new opcodes like CHECKSIGFROMSTACK, [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-January/015614.html Taproot], [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-February/015700.html Graftroot], [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-July/016249.html G'root], and [https://bitcointalk.org/index.php?topic=1377298.0 cross-input aggregation].

Combining all these ideas in a single proposal would be an extensive change, be hard to review, and likely miss new discoveries that otherwise could have been made along the way. Not all are equally mature as well. For example, cross-input aggregation [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-March/015838.html interacts] in complex ways with upgrade mechanisms, and solutions to that are still [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-October/016461.html in flux]. On the other hand, separating them all into independent upgrades would reduce the efficiency and privacy gains to be had, and wallet and service providers may not be inclined to go through many incremental updates. Therefore, we're faced with a tradeoff between functionality and scope creep. In this design we strike a balance by focusing on the structural script improvements offered by Taproot and Merkle branches, as well as changes necessary to make them usable and efficient. For things like sighashes and opcodes we include fixes for known problems, but exclude new features that can be added independently with no downsides.

As a result we choose this combination of technologies:

- '''Merkle branches''' let us only reveal the actually executed part of the script to the blockchain, as opposed to all possible ways a script can be executed. Among the various known mechanisms for implementing this, one where the Merkle tree becomes part of the script's structure directly maximizes the space savings, so that approach is chosen.
- '''Taproot''' on top of that lets us merge the traditionally separate pay-to-pubkey and pay-to-scripthash policies, making all outputs spendable by either a key or (optionally) a script, and indistinguishable from each other. As long as the key-based spending path is used for spending, it is not revealed whether a script path was permitted as well, resulting in space savings and an increase in scripting privacy at spending time.
- Taproot's advantages become apparent under the assumption that most applications involve outputs that could be spent by all parties agreeing. That's where '''Schnorr''' signatures come in, as they permit [https://eprint.iacr.org/2018/068 key aggregation]: a public key can be constructed from multiple participant public keys, and which requires cooperation between all participants to sign for. Such multi-party public keys and signatures are indistinguishable from their single-party equivalents. This means that with taproot most applications can use the key-based spending path, which is both efficient and private. This can be generalized to arbitrary M-of-N policies, as Schnorr signatures support threshold signing, at the cost of more complex setup protocols.
- As Schnorr signatures also permit '''batch validation''', allowing multiple signatures to be validated together more efficiently than validating each one independently, we make sure all parts of the design are compatible with this.
- Where unused bits appear as a result of the above changes, they are reserved for mechanisms for '''future extensions'''. As a result, every script in the Merkle tree has an associated version such that new script versions can be introduced with a soft fork while remaining compatible with BIP 341. Additionally, future soft forks can make use of the currently unused `annex` in the witness (see [[bip-0341.mediawiki#rationale|Rationale]]).
- While the core semantics of the '''signature hashing algorithm''' are not changed, a number of improvements are included in this proposal. The new signature hashing algorithm fixes the verification capabilities of offline signing devices by including amount and scriptPubKey in the signature message, avoids unnecessary hashing, uses '''tagged hashes''' and defines a default sighash byte.
- The '''public key is directly included in the output''' in contrast to typical earlier constructions which store a hash of the public key or script in the output. This has the same cost for senders and is more space efficient overall if the key-based spending path is taken. '''Why is the public key directly included in the output?''' While typical earlier constructions store a hash of a script or a public key in the output, this is rather wasteful when a public key is always involved. To guarantee batch verifiability, the public key must be known to every verifier, and thus only revealing its hash as an output would imply adding an additional 32 bytes to the witness. Furthermore, to maintain [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2016-January/012198.html 128-bit collision security] for outputs, a 256-bit hash would be required anyway, which is comparable in size (and thus in cost for senders) to revealing the public key directly. While the usage of public key hashes is often said to protect against ECDLP breaks or quantum computers, this protection is very weak at best: transactions are not protected while being confirmed, and a very [https://web.archive.org/web/20220531184542/https://twitter.com/pwuille/status/1108085284862713856 large portion] of the currency's supply is not under such protection regardless. Actual resistance to such systems can be introduced by relying on different cryptographic assumptions, but this proposal focuses on improvements that do not change the security model.

Informally, the resulting design is as follows: a new witness version is added (version 1), whose programs consist of 32-byte encodings of points ''Q''. ''Q'' is computed as ''P + hash(P||m)G'' for a public key ''P'', and the root ''m'' of a Merkle tree whose leaves consist of a version number and a script. These outputs can be spent directly by providing a signature for ''Q'', or indirectly by revealing ''P'', the script and leaf version, inputs that satisfy the script, and a Merkle path that proves ''Q'' committed to that leaf. All hashes in this construction (the hash for computing ''Q'' from ''P'', the hashes inside the Merkle tree's inner nodes, and the signature hashes used) are tagged to guarantee domain separation.

== Specification ==

This section specifies the Taproot consensus rules. Validity is defined by exclusion: a block or transaction is valid if no condition exists that marks it failed.

The notation below follows that of [[bip-0340.mediawiki#design|BIP340]]. This includes the ''$\text{hash}_{\text{tag}}(x)$'' notation to refer to ''SHA256(SHA256(tag) || SHA256(tag) || x)''. To the best of the authors' knowledge, no existing use of SHA256

in Bitcoin feeds it a message that starts with two single SHA256 outputs, making collisions between ''$hash_{tag}$'' with other hashes extremely unlikely.

=== Script validation rules ===

A Taproot output is a native SegWit output (see [[bip-0141.mediawiki|BIP141]]) with version number 1, and a 32-byte witness program. The following rules only apply when such an output is being spent. Any other outputs, including version 1 outputs with lengths other than 32 bytes, or P2SH-wrapped version 1 outputs'''Why is P2SH-wrapping not supported?''' Using P2SH-wrapped outputs only provides 80-bit collision security due to the use of a 160-bit hash. This is considered low, and becomes a security risk whenever the output includes data from more than a single party (public keys, hashes, ...)., remain unencumbered.

- Let ''q'' be the 32-byte array containing the witness program (the second push in the scriptPubKey) which represents a public key according to [[bip-0340.mediawiki#design|BIP340]].
- Fail if the witness stack has 0 elements.
- If there are at least two witness elements, and the first byte of the last element is 0x50'''Why is the first byte of the annex `0x50` ?''' The `0x50` is chosen as it could not be confused with a valid P2WPKH or P2WSH spending. As the control block's initial byte's lowest bit is used to indicate the parity of the public key's Y coordinate, each leaf version needs an even byte value and the immediately following odd byte value that are both not yet used in P2WPKH or P2WSH spending. To indicate the annex, only an "unpaired" available byte is necessary like `0x50` . This choice maximizes the available options for future script versions., this last element is called ''annex'' ''a'''''What is the purpose of the annex?''' The annex is a reserved space for future extensions, such as indicating the validation costs of computationally expensive new opcodes in a way that is recognizable without knowing the scriptPubKey of the output being spent. Until the meaning of this field is defined by another softfork, users SHOULD NOT include `annex` in transactions, or it may lead to PERMANENT FUND LOSS. and is removed from the witness stack. The annex (or the lack of thereof) is always covered by the signature and contributes to transaction weight, but is otherwise ignored during taproot validation.
- If there is exactly one element left in the witness stack, key path spending is used: ** The single witness stack element is interpreted as the signature and must be valid (see the next section) for the public key ''q'' (see the next subsection).
- If there are at least two witness elements left, script path spending is used: ** Call the second-to-last stack element ''s'', the script. ** The last stack element is called the control block ''c'', and must have length ''33 + 32m'', for a value of ''m'' that is an integer between 0 and 128'''Why is the Merkle path length limited to 128?''' The optimally space-efficient Merkle tree can be constructed based on the probabilities of the scripts in the leaves, using the Huffman algorithm. This algorithm will construct branches with lengths approximately equal to ''$\log_2(1/\text{probability})$'', but to have branches longer than 128 you would need to have scripts with an execution chance below 1 in ''$2^{128}$''. As that is our security bound, scripts that truly have such a low chance can probably be removed entirely., inclusive. Fail if it does not have such a length. ** Let ''p = c[1:33]'' and let ''P = lift_x(int(p))'' where ''lift_x'' and ''[:]'' are defined as in [[bip-0340.mediawiki#design|BIP340]]. Fail if this point is not on the curve. ** Let ''v = c[0] & 0xfe'' and call it the ''leaf version'''''What constraints are there on the leaf version?''' First, the leaf version cannot be odd as ''c[0] & 0xfe'' will always be even, and cannot be ''0x50'' as that would result in ambiguity with the annex. In addition, in order to support some forms of static analysis that rely on being able to identify script spends without access to the output being spent, it is recommended to avoid using any leaf versions that would conflict with a valid first byte of either a valid P2WPKH pubkey or a valid P2WSH script (that is, both ''v'' and ''v | 1'' should be an undefined, invalid or disabled opcode or an opcode that is not valid as the first opcode). The values that comply to this rule are the 32 even values between ''0xc0'' and ''0xfe'' and also ''0x66'', ''0x7e'', ''0x80'', ''0x84'', ''0x96'', ''0x98'', ''0xba'', ''0xbc'', ''0xbe''. Note also that this constraint implies that leaf versions should be shared amongst different witness versions, as knowing the witness version requires access to the output being spent.. ** Let ''$k_0 = \text{hash}_{\text{TapLeaf}}(v \,||\, \text{compact\_size(size of s)} \,||\, s)$''; also call it the ''tapleaf hash''. ** For ''j'' in ''[0,1,...,m-1]'': *** Let ''$e_j$ = c[33+32j:65+32j]''. *** Let ''$k_{j+1}$ depend on whether ''$k_j < e_j$'' (lexicographically)'''Why are child elements sorted before hashing in the Merkle tree?''' By doing so, it is not necessary to reveal the left/right directions along with the hashes in revealed Merkle branches. This is possible because we do not actually care about the position of specific scripts in the tree; only that they are actually committed to.: **** If ''$k_j < e_j$'': ''$k_{j+1} = \text{hash}_{\text{TapBranch}}(k_j \,||\, e_j)$'''''Why not use a more efficient hash construction for inner Merkle nodes?''' The chosen construction does require two invocations of the SHA256 compression functions, one of which can be avoided in theory (see [[bip-

0098.mediawiki|BIP98]]). However, it seems preferable to stick to constructions that can be implemented using standard cryptographic primitives, both for implementation simplicity and analyzability. If necessary, a significant part of the second compression function can be optimized out by [https://github.com/bitcoin/bitcoin/pull/13191 specialization] for 64-byte inputs.. **** If ''$k_j \geq e_j$'': ''$k_{j+1} = \text{hash}_{\text{TapBranch}}(e_j \| k_j)$''. ** Let ''$t = \text{hash}_{\text{TapTweak}}(p \| k_m)$''. ** If ''$t \geq$ 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141'' (order of secp256k1), fail. ** Let ''$Q = P + \text{int}(t)G$''. ** If ''$q \neq x(Q)$'' or ''$c[0] \& 1 \neq y(Q) \bmod 2$'', fail'''Why is it necessary to reveal a bit in a script path spend and check that it matches the parity of the Y coordinate of ''Q''?''' The parity of the Y coordinate is necessary to lift the X coordinate ''q'' to a unique point. While this is not strictly necessary for verifying the taproot commitment as described above, it is necessary to allow batch verification. Alternatively, ''Q'' could be forced to have an even Y coordinate, but that would require retrying with different internal public keys (or different messages) until ''Q'' has that property. There is no downside to adding the parity bit because otherwise the control block bit would be unused.. ** Execute the script, according to the applicable script rules'''What are the applicable script rules in script path spends?''' [[bip-0342.mediawiki|BIP342]] specifies validity rules that apply for leaf version 0xc0, but future proposals can introduce rules for other leaf versions., using the witness stack elements excluding the script ''s'', the control block ''c'', and the annex ''a'' if present, as initial stack. This implies that for the future leaf versions (non-''0xC0'') the execution must succeed.'''Why we need to success on future leaf version validation''' This is required to enable future leaf versions as soft forks.

''q'' is referred to as ''taproot output key'' and ''p'' as ''taproot internal key''.

=== Signature validation rules ===

We first define a reusable common signature message calculation function, followed by the actual signature validation as it's used in key path spending.

==== Common signature message ====

The function ''SigMsg(hash_type, ext_flag)'' computes the common portion of the message being signed as a byte array. It is implicitly also a function of the spending transaction and the outputs it spends, but these are not listed to keep notation simple.

The parameter ''hash_type'' is an 8-bit unsigned value. The `SIGHASH` encodings from the legacy script system are reused, including `SIGHASH_ALL`, `SIGHASH_NONE`, `SIGHASH_SINGLE`, and `SIGHASH_ANYONECANPAY`. We define a new ''hashtype'' `SIGHASH_DEFAULT` (value ''0x00'') which results in signing over the whole transaction just as for `SIGHASH_ALL`. The following restrictions apply, which cause validation failure if violated:

- Using any undefined ''hash_type'' (not ''0x00'', ''0x01'', ''0x02'', ''0x03'', ''0x81'', ''0x82'', or ''0x83'''''Why reject unknown ''hash_type'' values?''' By doing so, it is easier to reason about the worst case amount of signature hashing an implementation with adequate caching must perform.).
- Using `SIGHASH_SINGLE` without a "corresponding output" (an output with the same index as the input being verified).

The parameter ''ext_flag'' is an integer in range 0-127, and is used for indicating (in the message) that extensions are appended to the output of ''SigMsg()'''''What extensions use the ''ext_flag'' mechanism?''' [[bip-0342.mediawiki#common-signature-message-extension|BIP342]] reuses the same common signature message algorithm, but adds BIP342-specific data at the end, which is indicated using ''ext_flag = 1''..

If the parameters take acceptable values, the message is the concatenation of the following data, in order (with byte size of each item listed in parentheses). Numerical values in 2, 4, or 8-byte are encoded in little-endian.

- Control: ** ''hash_type'' (1).
- Transaction data: ** ''nVersion'' (4): the ''nVersion'' of the transaction. ** ''nLockTime'' (4): the ''nLockTime'' of the transaction. ** If the ''hash_type & 0x80'' does not equal `SIGHASH_ANYONECANPAY`: *** ''sha_prevouts'' (32): the SHA256 of the serialization of all input outpoints. *** ''sha_amounts'' (32): the SHA256 of the serialization of all input amounts. *** ''sha_scriptpubkeys'' (32): the SHA256 of all spent outputs' ''scriptPubKeys'', serialized as script inside `CTxOut`. *** ''sha_sequences'' (32): the SHA256 of the serialization of all input ''nSequence''. ** If ''hash_type & 3'' does not equal `SIGHASH_NONE` or `SIGHASH_SINGLE`: *** ''sha_outputs'' (32): the SHA256 of the serialization of all outputs in `CTxOut` format.

- Data about this input: ** ''spend_type'' (1): equal to ''(ext_flag * 2) + annex_present'', where ''annex_present'' is 0 if no annex is present, or 1 otherwise (the original witness stack has two or more witness elements, and the first byte of the last element is ''0x50'') ** If ''hash_type & 0x80'' equals `SIGHASH_ANYONECANPAY` : *** ''outpoint'' (36): the `COutPoint` of this input (32-byte hash + 4-byte little-endian). *** ''amount'' (8): value of the previous output spent by this input. *** ''scriptPubKey'' (35): ''scriptPubKey'' of the previous output spent by this input, serialized as script inside `CTxOut` . Its size is always 35 bytes. *** ''nSequence'' (4): ''nSequence'' of this input. ** If ''hash_type & 0x80'' does not equal `SIGHASH_ANYONECANPAY` : *** ''input_index'' (4): index of this input in the transaction input vector. Index of the first input is 0. ** If an annex is present (the lowest bit of ''spend_type'' is set): *** ''sha_annex'' (32): the SHA256 of ''(compact_size(size of annex) || annex)'', where ''annex'' includes the mandatory ''0x50'' prefix.
- Data about this output: ** If ''hash_type & 3'' equals `SIGHASH_SINGLE` : *** ''sha_single_output'' (32): the SHA256 of the corresponding output in `CTxOut` format.

The total length of ''SigMsg()'' is at most ''206'' bytes'''What is the output length of ''SigMsg()''?''' The total length of ''SigMsg()'' can be computed using the following formula: ''174 - is_anyonecanpay * 49 - is_none * 32 + has_annex * 32''.. Note that this does not include the size of sub-hashes such as ''sha_prevouts'', which may be cached across signatures of the same transaction.

In summary, the semantics of the [[bip-0143.mediawiki|BIP143]] sighash types remain unchanged, except the following:

## The way and order of serialization is changed.'''Why is the serialization in the signature message changed?''' Hashes that go into the signature message and the message itself are now computed with a single SHA256 invocation instead of double SHA256. There is no expected security improvement by doubling SHA256 because this only protects against length-extension attacks against SHA256 which are not a concern for signature messages because there is no secret data. Therefore doubling SHA256 is a waste of resources. The message computation now follows a logical order with transaction level data first, then input data and output data. This allows to efficiently cache the transaction part of the message across different inputs using the SHA256 midstate. Additionally, sub-hashes can be skipped when calculating the message (for example `sha_prevouts` if `SIGHASH_ANYONECANPAY` is set) instead of setting them to zero and then hashing them as in BIP143. Despite that, collisions are made impossible by committing to the length of

the data (implicit in ''hash_type'' and ''spend_type'') before the variable length data.

The signature message commits to the ''scriptPubKey'' of the spent output and if the `SIGHASH_ANYONECANPAY` flag is not set, the message commits to the ''scriptPubKey''s of ''all'' outputs spent by the transaction. '''Why does the signature message commit to the ''scriptPubKey''?''' This prevents lying to offline signing devices about output being spent, even when the actually executed script (''scriptCode'' in BIP143) is correct. This means it's possible to compactly prove to a hardware wallet what (unused) execution paths existed. Moreover, committing to all spent ''scriptPubKey''s helps offline signing devices to determine the subset that belong to its own wallet. This is useful in [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2020-April/017801.html automated coinjoins]..

If the `SIGHASH_ANYONECANPAY` flag is not set, the message commits to the amounts of ''all'' transaction inputs.'''Why does the signature message commit to the amounts of all transaction inputs?''' This eliminates the possibility to lie to offline signing devices about the fee of a transaction.

The signature message commits to all input ''nSequence'' if `SIGHASH_NONE` or `SIGHASH_SINGLE` are set (unless `SIGHASH_ANYONECANPAY` is set as well).'''Why does the signature message commit to all input ''nSequence'' if `SIGHASH_SINGLE` or `SIGHASH_NONE` are set?''' Because

## setting them already makes the message commit to the `prevouts` part of all transaction inputs, it is not useful to treat the ''nSequence'' any different. Moreover, this change makes ''nSequence'' consistent with the view that `SIGHASH_SINGLE` and `SIGHASH_NONE` only modify the signature message with respect to transaction outputs and not inputs.

## The signature message includes commitments to the taproot-specific data ''spend_type'' and ''annex'' (if present).

==== Taproot key path spending signature validation ====

A Taproot signature is a 64-byte Schnorr signature, as defined in [[bip-0340.mediawiki|BIP340]], with the sighash byte appended in the usual Bitcoin fashion. This sighash byte is optional. If omitted, the resulting signatures are 64 bytes, and a SIGHASH_DEFAULT mode is implied.

To validate a signature ''sig'' with public key ''q'':

- If the ''sig'' is 64 bytes long, return ''Verify(q, $\text{hash}_{\text{TapSighash}}$(0x00 || SigMsg(0x00, 0)), sig)'''''Why is the input to ''$\text{hash}_{\text{TapSighash}}$'' prefixed with 0x00?''' This prefix is called the sighash epoch, and allows reusing the ''$\text{hash}_{\text{TapSighash}}$'' tagged hash in future signature algorithms that make invasive changes to how hashing is performed (as opposed to the ''ext_flag'' mechanism that is used for incremental extensions). An alternative is having them use a different tag, but supporting a growing number of tags may become undesirable., where ''Verify'' is defined in [[bip-0340.mediawiki#design|BIP340]].
- If the ''sig'' is 65 bytes long, return ''sig[64] ≠ 0x00'''Why can the `hash_type` not be `0x00` in 65-byte signatures?''' Permitting that would enable malleating (by third parties, including miners) 64-byte signatures into 65-byte ones, resulting in a different `wtxid` and a different fee rate than the creator intended. and Verify(q, $\text{hash}_{\text{TapSighash}}$(0x00 || SigMsg(sig[64], 0)), sig[0:64])''.
- Otherwise, fail'''Why permit two signature lengths?''' By making the most common type of `hash_type` implicit, a byte can often be saved..

== Constructing and spending Taproot outputs ==

This section discusses how to construct and spend Taproot outputs. It only affects wallet software that chooses to implement receiving and spending, and is not consensus critical in any way.

Conceptually, every Taproot output corresponds to a combination of a single public key condition (the internal key), and zero or more general conditions encoded in scripts organized in a tree. Satisfying any of these conditions is sufficient to spend the output.

'''Initial steps''' The first step is determining what the internal key and the organization of the rest of the scripts should be. The specifics are likely application dependent, but here are some general guidelines:

- When deciding between scripts with conditionals ( `OP_IF` etc.) and splitting them up into multiple scripts (each corresponding to one execution path through the original script), it is generally preferable to pick the latter.
- When a single condition requires signatures with multiple keys, key aggregation techniques like MuSig can be used to combine them into a single key. The details are out of scope for this document, but note that this may complicate the signing procedure.

- If one or more of the spending conditions consist of just a single key (after aggregation), the most likely one should be made the internal key. If no such condition exists, it may be worthwhile adding one that consists of an aggregation of all keys participating in all scripts combined; effectively adding an "everyone agrees" branch. If that is inacceptable, pick as internal key a "Nothing Up My Sleeve" (NUMS) point, i.e., a point with unknown discrete logarithm. One example of such a point is ''H = lift_x(0x50929b74c1a04954b78b4b6035e97a5e078a5a0f28ec96d547bfee9ace803ac0)'' which is [https://github.com/ElementsProject/secp256k1-zkp/blob/11af7015de624b010424273be3d91f117f172c82/src/modules/rangeproof/main_impl.h#L16 constructed] by taking the hash of the standard uncompressed encoding of the [https://www.secg.org/sec2-v2.pdf secp256k1] base point ''G'' as X coordinate. In order to avoid leaking the information that key path spending is not possible it is recommended to pick a fresh integer ''r'' in the range ''0...n-1'' uniformly at random and use ''H + rG'' as internal key. It is possible to prove that this internal key does not have a known discrete logarithm with respect to ''G'' by revealing ''r'' to a verifier who can then reconstruct how the internal key was created.
- If the spending conditions do not require a script path, the output key should commit to an unspendable script path instead of having no script path. This can be achieved by computing the output key point as ''Q = P + int($hash_{TapTweak}$(bytes(P)))G''. '''Why should the output key always have a taproot commitment, even if there is no script path?''' If the taproot output key is an aggregate of keys, there is the possibility for a malicious party to add a script path without being noticed by the other parties. This allows to bypass the multiparty policy and to steal the coin. MuSig key aggregation does not have this issue because it already causes the internal key to be randomized.

The attack works as follows: Assume Alice and Mallory want to aggregate their keys into a taproot output key without a script path. In order to prevent key cancellation and related attacks they use [https://eprint.iacr.org/2018/483.pdf MSDL-pop] instead of MuSig. The MSDL-pop protocol requires all parties to provide a proof of possession of their corresponding secret key and the aggregated key is just the sum of the individual keys. After Mallory receives Alice's key ''A'', Mallory creates ''M = $M_0$ + int(t)G'' where ''$M_0$'' is Mallory's original key and ''t'' allows a script path spend with internal key ''P = A + $M_0$'' and a script that only contains Mallory's key. Mallory sends a proof of possession of ''M'' to Alice and both parties compute output key ''Q = A + M = P + int(t)G''. Alice will not be able to notice the script path, but Mallory can unilaterally spend any coin with output key ''Q''.

- The remaining scripts should be organized into the leaves of a binary tree. This can be a balanced tree if each of the conditions these scripts correspond to are equally likely. If probabilities for each condition are known, consider constructing the tree as a Huffman tree.

'''Computing the output script''' Once the spending conditions are split into an internal key `internal_pubkey` and a binary tree whose leaves are (leaf_version, script) tuples, the output script can be computed using the Python3 algorithms below. These algorithms take advantage of helper functions from the [[bip-0340/reference.py|BIP340 reference code]] for integer conversion, point multiplication, and tagged hashes.

First, we define `taproot_tweak_pubkey` for 32-byte [[bip-0340.mediawiki|BIP340]] public key arrays. The function returns a bit indicating the tweaked public key's Y coordinate as well as the public key byte array. The parity bit will be required for spending the output with a script path. In order to allow spending with the key path, we define `taproot_tweak_seckey` to compute the secret key for a tweaked public key. For any byte string `h` it holds that `taproot_tweak_pubkey(pubkey_gen(seckey), h)[1] == pubkey_gen(taproot_tweak_seckey(seckey, h))`.

Note that because tweaks are applied to 32-byte public keys, `taproot_tweak_seckey` may need to negate the secret key before applying the tweak.
def taproot_tweak_pubkey(pubkey, h): t = int_from_bytes(tagged_hash("TapTweak", pubkey + h)) if t >= SECP256K1_ORDER: raise ValueError P = lift_x(int_from_bytes(pubkey)) if P is None: raise ValueError Q = point_add(P, point_mul(G, t)) return 0 if has_even_y(Q) else 1, bytes_from_int(x(Q))

def taproot_tweak_seckey(seckey0, h): seckey0 = int_from_bytes(seckey0) P = point_mul(G, seckey0) seckey = seckey0 if has_even_y(P) else SECP256K1_ORDER - seckey0 t = int_from_bytes(tagged_hash("TapTweak", bytes_from_int(x(P)) + h)) if t >= SECP256K1_ORDER: raise ValueError return bytes_from_int((seckey + t) % SECP256K1_ORDER)

The following function, `taproot_output_script`, returns a byte array with the scriptPubKey (see [[bip-0141.mediawiki|BIP141]]). `ser_script` refers to a function that prefixes its input with a CompactSize-encoded length.

def taproot_tree_helper(script_tree): if isinstance(script_tree, tuple): leaf_version, script = script_tree h = tagged_hash("TapLeaf", bytes([leaf_version]) + ser_script(script)) return ([((leaf_version, script), bytes())], h) left, left_h = taproot_tree_helper(script_tree[0]) right, right_h = taproot_tree_helper(script_tree[1]) ret = [(l, c + right_h) for l, c in left] + [(l, c + left_h) for l, c in right] if right_h < left_h: left_h, right_h = right_h, left_h return (ret, tagged_hash("TapBranch", left_h + right_h))

def taproot_output_script(internal_pubkey, script_tree): """Given a internal public key and a tree of scripts, compute the output script. script_tree is either: - a (leaf_version, script) tuple (leaf_version is 0xc0 for [[bip-0342.mediawiki|BIP342]] scripts) - a list of two elements, each with the same structure as script_tree itself - None """ if script_tree is None: h = bytes() else: _, h = taproot_tree_helper(script_tree) _, output_pubkey = taproot_tweak_pubkey(internal_pubkey, h) return bytes([0x51, 0x20]) + output_pubkey

[[File:bip-0341/tree.png|frame|This diagram shows the hashing structure to obtain the tweak from an internal key ''P'' and a Merkle tree consisting of 5 script leaves. ''A'', ''B'', ''C'' and ''E'' are ''TapLeaf'' hashes similar to ''D'' and ''AB'' is a ''TapBranch'' hash. Note that when ''CDE'' is computed ''E'' is hashed first because ''E'' is less than ''CD''.]]

To spend this output using script ''D'', the control block would contain the following data in this order:

```
<control byte with leaf version and parity bit> <internal key p> <C> <E> <AB>
```

The TapTweak would then be computed as described [[bip-0341.mediawiki#script-validation-rules|above]] like so:

D = tagged_hash("TapLeaf", bytes([leaf_version]) + ser_script(script)) CD = tagged_hash("TapBranch", C + D) CDE = tagged_hash("TapBranch", E + CD) ABCDE = tagged_hash("TapBranch", AB + CDE) TapTweak = tagged_hash("TapTweak", p + ABCDE)

'''Spending using the key path''' A Taproot output can be spent with the secret key corresponding to the `internal_pubkey`. To do so, a witness stack consists of a single element: a [[bip-0340.mediawiki|BIP340]] signature on the signature hash as defined above, with the secret key tweaked by the same `h` as in the above snippet. See the code below:

def taproot_sign_key(script_tree, internal_seckey, hash_type, bip340_aux_rand): if script_tree is None: h = bytes() else: _, h = taproot_tree_helper(script_tree) output_seckey = taproot_tweak_seckey(internal_seckey, h) sig = schnorr_sign(sighash(hash_type), output_seckey, bip340_aux_rand) if hash_type != 0: sig += bytes([hash_type]) return [sig]

This function returns the witness stack necessary and a `sighash` function to compute the signature hash as defined above (for simplicity, the snippet above ignores passing information like the transaction, the input position, ... to the sighashing code).

'''Spending using one of the scripts''' A Taproot output can be spent by satisfying any of the scripts used in its construction. To do so, a witness stack consisting of the script's inputs, plus the script itself and the control block are necessary. See the code below:

def taproot_sign_script(internal_pubkey, script_tree, script_num, inputs): info, h = taproot_tree_helper(script_tree) (leaf_version, script), path = info[script_num] output_pubkey_y_parity, _ = taproot_tweak_pubkey(internal_pubkey, h) pubkey_data = bytes([output_pubkey_y_parity + leaf_version]) + internal_pubkey return inputs + [script, pubkey_data + path]

== Security ==

Taproot improves the privacy of Bitcoin because instead of revealing all possible conditions for spending an output, only the satisfied spending condition has to be published. Ideally, outputs are spent using the key path which prevents observers from learning the spending conditions of a coin. A key path spend could be a "normal" payment from a single- or multi-signature wallet or the cooperative settlement of hidden multiparty contract.

A script path spend leaks that there is a script path and that the key path was not applicable - for example because the involved parties failed to reach agreement. Moreover, the depth of a script in the Merkle root leaks information including the minimum depth of the tree, which suggests specific wallet software that created the output and helps clustering. Therefore, the privacy of script spends can be improved by deviating from the optimal tree determined by the probability distribution over the leaves.

Just like other existing output types, taproot outputs should never reuse keys, for privacy reasons. This does not only apply to the particular leaf that was used to spend an output but to all leaves committed to in the output. If leaves were reused, it could happen that spending a different output would reuse the same Merkle branches in the Merkle proof. Using fresh keys implies that taproot output construction does not need to take special measures to randomizing leaf positions because they are already randomized due to the branch-sorting Merkle tree construction used in taproot. This does not avoid leaking information through the leaf depth and therefore only applies to balanced (sub-) trees. In addition, every leaf should have a set of keys distinct from every other leaf. The reason for this is to increase leaf entropy and prevent an observer from learning an undisclosed script using brute-force search.

== Test vectors ==

Test vectors for wallet operation (scriptPubKey computation, key path spending, control block construction) can be found [[bip-0341/wallet-test-vectors.json|here]]. It consists of two sets of vectors.

- The first "scriptPubKey" tests concern computing the scriptPubKey and (mainnet) BIP350 address given an internal public key, and a script tree. The script tree is encoded as `null` to represent no scripts, a JSON object to represent a leaf node, or a 2-element array to represent an inner node. The control blocks needed for script path spending are also provided for each of the script leaves.
- The second "keyPathSpending" tests consists of a list of test cases, each of which provides an unsigned transaction and the UTXOs it spends. For each of its BIP341 inputs, the internal private key and the Merkle root it was derived from is given, as well as the expected witness to spend it. All signatures are created with an all-zero (0x0000...0000) BIP340 auxiliary randomness array.
- In all cases, hexadecimal values represent byte arrays, not numbers. In particular, that means that provided hash values have the hex digits corresponding to the first bytes first. This differs from the convention used for txids and block hashes, where the hex strings represent numbers, resulting in a reversed order.

Validation test vectors used in the [https://github.com/bitcoin/bitcoin/blob/3820090bd619ac85ab35eff376c03136fe4a9f04/src/test/script_tests.cpp#L1718 Bitcoin Core unit test framework] can be found [https://github.com/bitcoin-core/qa-assets/blob/main/unit_test_data/script_assets_test.json?raw=true here].

== Rationale ==

== Deployment ==

This BIP is deployed concurrently with [[bip-0342.mediawiki|BIP342]].

For Bitcoin signet, these BIPs are always active.

For Bitcoin mainnet and testnet3, these BIPs are deployed by "version bits" with the name "taproot" and bit 2, using [[bip-0009.mediawiki|BIP9]] modified to use a lower threshold, with an additional ''min_activation_height'' parameter and replacing the state transition logic for the DEFINED, STARTED and LOCKED_IN states as follows:

```
case DEFINED:
    if (GetMedianTimePast(block.parent) >= starttime) {
        return STARTED;
    }
    return DEFINED;

case STARTED:
    int count = 0;
    walk = block;
    for (i = 0; i < 2016; i++) {
        walk = walk.parent;
        if ((walk.nVersion & 0xE0000000) == 0x20000000 && ((walk.nVersion >> bit) & 1) == 1) {
            count++;
        }
    }
    if (count >= threshold) {
        return LOCKED_IN;
    } else if (GetMedianTimePast(block.parent) >= timeout) {
```

```
        return FAILED;
    }
    return STARTED;

case LOCKED_IN:
    if (block.nHeight < min_activation_height) {
        return LOCKED_IN;
    }
    return ACTIVE;
```

For Bitcoin mainnet, the starttime is epoch timestamp 1619222400 (midnight 24 April 2021 UTC), timeout is epoch timestamp 1628640000 (midnight 11 August 2021 UTC), the threshold is 1815 blocks (90%) instead of 1916 blocks (95%), and the min_activation_height is block 709632. The deployment did activate at height 709632 on Bitcoin mainnet.

For Bitcoin testnet3, the starttime is epoch timestamp 1619222400 (midnight 24 April 2021 UTC), timeout is epoch timestamp 1628640000 (midnight 11 August 2021 UTC), the threshold is 1512 blocks (75%), and the min_activation_height is block 0. The deployment did activate at height 2011968 on Bitcoin testnet3.

== Backwards compatibility == As a soft fork, older software will continue to operate without modification. Non-upgraded nodes, however, will consider all SegWit version 1 witness programs as anyone-can-spend scripts. They are strongly encouraged to upgrade in order to fully validate the new programs.

Non-upgraded wallets can receive and send bitcoin from non-upgraded and upgraded wallets using SegWit version 0 programs, traditional pay-to-pubkey-hash, etc. Depending on the implementation non-upgraded wallets may be able to send to Segwit version 1 programs if they support sending to [[bip-0350.mediawiki|BIP350]] Bech32m addresses.

== Acknowledgements ==

This document is the result of discussions around script and signature improvements with many people, and had direct contributions from Greg Maxwell and others. It further builds on top of earlier published proposals such as Taproot by Greg Maxwell, and Merkle branch constructions by Russell O'Connor, Johnson Lau, and Mark Friedenbach.

The authors wish the thank Arik Sosman for suggesting to sort Merkle node children before hashes, removing the need to transfer the position in the tree, as well as all those who provided valuable feedback and reviews, including the participants of the [https://github.com/ajtowns/taproot-review structured reviews].

```
  BIP: 342
  Layer: Consensus (soft fork)
  Title: Validation of Taproot Scripts
  Authors: Pieter Wuille
           Jonas Nick
           Anthony Towns
  Status: Deployed
  Type: Specification
  Assigned: 2020-01-19
  License: BSD-3-Clause
  Discussion: 2019-05-06: https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2019-May/016914.html [bitcoin-dev]
Taproot proposal
  Requires: 340, 341
```

==Introduction==

===Abstract===

This document specifies the semantics of the initial scripting system under [[bip-0341.mediawiki|BIP341]].

===Copyright===



===Motivation===

[[bip-0341.mediawiki|BIP341]] proposes improvements to just the script structure, but some of its goals are incompatible with the semantics of certain opcodes within the scripting language itself. While it is possible to deal with these in separate optional improvements, their impact is not guaranteed unless they are addressed simultaneously with [[bip-0341.mediawiki|BIP341]] itself.

Specifically, the goal is making '''Schnorr signatures''', '''batch validation''', and '''signature hash''' improvements available to spends that use the script system as well.

==Design==

In order to achieve these goals, signature opcodes `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` are modified to verify Schnorr signatures as specified in [[bip-0340.mediawiki|BIP340]] and to use a signature message algorithm based on the common message calculation in [[bip-0341.mediawiki|BIP341]]. The tapscript signature message also simplifies `OP_CODESEPARATOR` handling and makes it more efficient.

The inefficient `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` opcodes are disabled. Instead, a new opcode `OP_CHECKSIGADD` is introduced to allow creating the same multisignature policies in a batch-verifiable way. Tapscript uses a new, simpler signature opcode limit fixing complicated interactions with transaction weight. Furthermore, a potential malleability vector is eliminated by requiring MINIMALIF.

Tapscript can be upgraded through soft forks by defining unknown key types, for example to add new `hash_types` or signature algorithms. Additionally, the new tapscript `OP_SUCCESS` opcodes allow introducing new opcodes more cleanly than through `OP_NOP`.

==Specification==

The rules below only apply when validating a transaction input for which all of the conditions below are true:

- The transaction input is a '''segregated witness spend''' (i.e., the scriptPubKey contains a witness program as defined in [[bip-0141.mediawiki|BIP141]]).
- It is a '''taproot spend''' as defined in [[bip-0341.mediawiki#design|BIP341]] (i.e., the witness version is 1, the witness program is 32 bytes, and it is not P2SH wrapped).
- It is a '''script path spend''' as defined in [[bip-0341.mediawiki#design|BIP341]] (i.e., after removing the optional annex from the witness stack, two or more stack elements remain).
- The leaf version is ''0xc0'' (i.e. the first byte of the last witness element after removing the optional annex is ''0xc0'' or ''0xc1''), marking it as a '''tapscript spend'''.

Validation of such inputs must be equivalent to performing the following steps in the specified order.

# If the input is invalid due to BIP141 or BIP341, fail.

# The script as defined in BIP341 (i.e., the penultimate witness stack element after removing the optional annex) is called the '''tapscript''' and is decoded into opcodes, one by one:

### If any opcode numbered ''80, 98, 126-129, 131-134, 137-138, 141-142, 149-153, 187-254'' is encountered, validation succeeds (none of the rules below apply). This is true even if later bytes in the tapscript would fail to decode otherwise. These opcodes are renamed to `OP_SUCCESS80`, ..., `OP_SUCCESS254`, and collectively known as `OP_SUCCESSx`. `OP_SUCCESSx` is a mechanism to upgrade the Script system. Using an `OP_SUCCESSx` before its meaning is defined by a softfork is insecure and leads to fund loss. The

**inclusion of `OP_SUCCESSx` in a script will pass it unconditionally. It precedes any script execution rules to avoid the difficulties in specifying various edge cases, for example: `OP_SUCCESSx` in a script with an input stack larger than 1000 elements, `OP_SUCCESSx` after too many signature opcodes, or even scripts with conditionals lacking `OP_ENDIF`. The mere existence of an `OP_SUCCESSx` anywhere in the script will guarantee a pass for all such cases. `OP_SUCCESSx` are similar to the `OP_RETURN` in very early bitcoin versions (v0.1 up to and including v0.3.5). The original `OP_RETURN` terminates script execution immediately, and return pass or fail based on the top stack element at the moment of termination. This was one of a major design flaws in the original bitcoin protocol as it permitted unconditional third party theft by placing an `OP_RETURN` in `scriptSig`. This is not a concern in the present proposal since it is not possible for a third party to inject an `OP_SUCCESSx` to the validation process, as the `OP_SUCCESSx` is part of the script (and thus committed to by the taproot output), implying the consent of the coin owner. `OP_SUCCESSx` can be used for a variety of upgrade possibilities:**

- An `OP_SUCCESSx` could be turned into a functional opcode through a softfork. Unlike `OP_NOPx`-derived opcodes which only have read-only access to the stack, `OP_SUCCESSx` may also write to the stack. Any rule changes to an `OP_SUCCESSx`-containing script may only turn a valid script into an invalid one, and this is always achievable with softforks.
- Since `OP_SUCCESSx` precedes size check of initial stack and push opcodes, an `OP_SUCCESSx`-derived opcode requiring stack elements bigger than 520 bytes may uplift the limit in a softfork.
- `OP_SUCCESSx` may also redefine the behavior of existing opcodes so they could work together with the new opcode. For example, if an `OP_SUCCESSx`-derived opcode works with 64-bit integers, it may also allow the existing arithmetic opcodes in the ''same script'' to do the same.
- Given that `OP_SUCCESSx` even causes potentially unparseable scripts to pass, it can be used to introduce multi-byte opcodes, or even a completely new scripting language when prefixed with a specific `OP_SUCCESSx` opcode..

**If any push opcode fails to decode because it would extend past the end of the tapscript, fail.**

## If the '''initial stack''' as defined in BIP341 (i.e., the witness stack after removing both the optional annex and the two last stack elements after that) violates any resource limits (stack size, and size of the elements in the stack; see "Resource Limits" below), fail. Note that this check can be bypassed using `OP_SUCCESSx`.

# The tapscript is executed according to the rules in the following section, with the initial stack as input.

### If execution fails for any reason, fail.

### If the execution results in anything but exactly one element on the stack which evaluates to true with `CastToBool()`, fail.

# If this step is reached without encountering a failure, validation succeeds.

===Script execution===

The execution rules for tapscript are based on those for P2WSH according to BIP141, including the `OP_CHECKLOCKTIMEVERIFY` and `OP_CHECKSEQUENCEVERIFY` opcodes defined in [[bip-0065.mediawiki|BIP65]] and [[bip-0112.mediawiki|BIP112]], but with the following modifications:

- '''Disabled script opcodes''' The following script opcodes are disabled in tapscript: `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` '''Why are `OP_CHECKMULTISIG` and `OP_CHECKMULTISIGVERIFY` disabled, and not turned into OP_SUCCESSx?''' This is a precaution to make sure people who accidentally keep using `OP_CHECKMULTISIG` in Tapscript notice a problem immediately. It also avoids the complication of script disassemblers needing to become context-dependent.. The disabled opcodes behave in the same way as `OP_RETURN`, by failing and terminating the script immediately when executed, and being ignored when found in unexecuted branch of the script.
- '''Consensus-enforced MINIMALIF''' The MINIMALIF rules, which are only a standardness rule in P2WSH, are consensus enforced in tapscript. This means that the input argument to the `OP_IF` and `OP_NOTIF` opcodes must be either exactly 0 (the empty vector) or exactly 1 (the one-byte vector with value 1)'''Why make MINIMALIF consensus?''' This makes it considerably easier to write non-malleable scripts that take branch information from the stack..
- '''OP_SUCCESSx opcodes''' As listed above, some opcodes are renamed to `OP_SUCCESSx`, and make the script unconditionally valid.
- '''Signature opcodes'''. The `OP_CHECKSIG` and `OP_CHECKSIGVERIFY` are modified to operate on Schnorr public keys and signatures (see [[bip-0340.mediawiki|BIP340]]) instead of ECDSA, and a new opcode `OP_CHECKSIGADD` is added. ** The opcode 186 ( `0xba` ) is named as `OP_CHECKSIGADD`. ''' `OP_CHECKSIGADD` ''' This opcode is added to compensate for the loss of `OP_CHECKMULTISIG`-like opcodes, which are incompatible with batch verification. `OP_CHECKSIGADD` is functionally equivalent to `OP_ROT OP_SWAP OP_CHECKSIG OP_ADD`, but only takes 1 byte. All `CScriptNum`-related behaviours of `OP_ADD` are also applicable to `OP_CHECKSIGADD`.'''Alternatives to `CHECKMULTISIG` ''' There are multiple ways of implementing a threshold ''k''-of-''n'' policy using Taproot and Tapscript:
- '''Using a single `OP_CHECKSIGADD`-based script''' A `CHECKMULTISIG` script `m ... n CHECKMULTISIG` with witness `0 ...` can be rewritten as script `CHECKSIG CHECKSIGADD ... CHECKSIGADD m NUMEQUAL` with witness `...`. Every witness element `w_i` is either a signature corresponding to `pubkey_i` or an empty vector. A similar `CHECKMULTISIGVERIFY` script can be translated to BIP342 by replacing `NUMEQUAL` with `NUMEQUALVERIFY`. This approach has very similar characteristics to the existing `OP_CHECKMULTISIG`-based scripts.
- '''Using a ''k''-of-''k'' script for every combination''' A ''k''-of-''n'' policy can be implemented by splitting the script into several leaves of the Merkle tree, each implementing a ''k''-of-''k'' policy using `CHECKSIGVERIFY ... CHECKSIGVERIFY CHECKSIG`. This may be preferable for privacy reasons over the previous approach, as it only exposes the participating public keys, but it is only more cost effective for small values of ''k'' (1-of-''n'' for any ''n'', 2-of-''n'' for ''n ≥ 6'', 3-of-''n'' for ''n ≥ 9'', ...). Furthermore, the signatures here commit to the branch

used, which means signers need to be aware of which other signers will be participating, or produce signatures for each of the tree leaves.
- '''Using an aggregated public key for every combination''' Instead of building a tree where every leaf consists of ''k'' public keys, it is possible instead build a tree where every leaf contains a single ''aggregate'' of those ''k'' keys using [https://eprint.iacr.org/2018/068 MuSig]. This approach is far more efficient, but does require a 3-round interactive signing protocol to jointly produce the (single) signature.
- '''Native Schnorr threshold signatures''' Multisig policies can also be realized with [http://cacr.uwaterloo.ca/techreports/2001/corr2001-13.ps threshold signatures] using verifiable secret sharing. This results in outputs and inputs that are indistinguishable from single-key payments, but at the cost of needing an interactive protocol (and associated backup procedures) before determining the address to send to.

===Rules for signature opcodes===

The following rules apply to `OP_CHECKSIG`, `OP_CHECKSIGVERIFY`, and `OP_CHECKSIGADD`.

- For `OP_CHECKSIGVERIFY` and `OP_CHECKSIG`, the public key (top element) and a signature (second to top element) are popped from the stack. ** If fewer than 2 elements are on the stack, the script MUST fail and terminate immediately.
- For `OP_CHECKSIGADD`, the public key (top element), a `CScriptNum` `n` (second to top element), and a signature (third to top element) are popped from the stack. ** If fewer than 3 elements are on the stack, the script MUST fail and terminate immediately. ** If `n` is larger than 4 bytes, the script MUST fail and terminate immediately.
- If the public key size is zero, the script MUST fail and terminate immediately.
- If the public key size is 32 bytes, it is considered to be a public key as described in BIP340: ** If the signature is not the empty vector, the signature is validated against the public key (see the next subsection). Validation failure in this case immediately terminates script execution with failure.
- If the public key size is not zero and not 32 bytes, the public key is of an ''unknown public key type'''''Unknown public key types''' allow adding new signature validation rules through softforks. A softfork could add actual signature validation which either passes or makes the script fail and terminate immediately. This way, new `SIGHASH` modes can be added, as well as [https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2018-December/016549.html NOINPUT-tagged public keys] and a public key constant which is replaced by the taproot internal key for signature validation. and no actual signature verification is applied. During script execution of signature opcodes they behave exactly as known public key types except that signature validation is considered to be successful.
- If the script did not fail and terminate before this step, regardless of the public key type: ** If the signature is the empty vector: *** For `OP_CHECKSIGVERIFY`, the script MUST fail and terminate immediately. *** For `OP_CHECKSIG`, an empty vector is pushed onto the stack, and execution continues with the next opcode. *** For `OP_CHECKSIGADD`, a `CScriptNum` with value `n` is pushed onto the stack, and execution continues with the next opcode. ** If the signature is not the empty vector, the opcode is counted towards the sigops budget (see further). *** For `OP_CHECKSIGVERIFY`, execution continues without any further changes to the stack. *** For `OP_CHECKSIG`, a 1-byte value `0x01` is pushed onto the stack. *** For `OP_CHECKSIGADD`, a `CScriptNum` with value of `n + 1` is pushed onto the stack.

===Common Signature Message Extension===

We define the tapscript message extension ''ext'' to [[bip-0341.mediawiki#common-signature-message|BIP341 Common Signature Message]], indicated by ''ext_flag = 1'':

- ''tapleaf_hash'' (32): the tapleaf hash as defined in [[bip-0341.mediawiki#design|BIP341]]
- ''key_version'' (1): a constant value ''0x00'' representing the current version of public keys in the tapscript signature opcode execution.
- ''codesep_pos'' (4): the opcode position of the last executed `OP_CODESEPARATOR` before the currently executed signature opcode, with the value in little endian (or ''0xffffffff'' if none executed). The first opcode in a script has a position of 0. A multi-byte push opcode is counted as one opcode, regardless of the size of data being pushed. Opcodes in parsed but unexecuted branches count towards this value as well.

===Signature validation===

To validate a signature ''sig'' with public key ''p'':

- Compute the tapscript message extension ''ext'' described above.
- If the ''sig'' is 64 bytes long, return ''Verify(p, $\text{hash}_{\text{TapSighash}}$(0x00 || SigMsg(0x00, 1) || ext), sig)'', where ''Verify'' is defined in [[bip-0340.mediawiki#design|BIP340]].
- If the ''sig'' is 65 bytes long, return ''sig[64] ≠ 0x00 and Verify(p, $\text{hash}_{\text{TapSighash}}$(0x00 || SigMsg(sig[64], 1) || ext), sig[0:64])''.
- Otherwise, fail.

In summary, the semantics of signature validation is identical to BIP340, except the following:

**The signature message includes the tapscript-specific data ''key_version''.'''Why does the signature message commit to the ''key_version''?''' This is for future extensions that define unknown public key types, making sure signatures can't be moved from one key type to another.**

**The signature message commits to the executed script through the ''tapleaf_hash'' which includes the leaf version and script instead of ''scriptCode''. This implies that this commitment is unaffected by `OP_CODESEPARATOR`.**

**The signature message includes the opcode position of the last executed `OP_CODESEPARATOR`.'''Why does the signature message include the position of the last executed `OP_CODESEPARATOR`?''' This allows continuing to use `OP_CODESEPARATOR` to sign the executed path of the script. Because the `codeseparator_position` is the last input to the hash, the SHA256 midstate can be efficiently cached for multiple `OP_CODESEPARATOR`s in a single script. In contrast, the BIP143 handling of `OP_CODESEPARATOR` is to commit to the executed script only from the last executed `OP_CODESEPARATOR` onwards which requires unnecessary rehashing of the script. It should be noted that the one known `OP_CODESEPARATOR` use case of saving a second public key**

# push in a script by sharing the first one between two code branches can be most likely expressed even cheaper by moving each branch into a separate taproot leaf.

===Resource limits===

In addition to changing the semantics of a number of opcodes, there are also some changes to the resource limitations:

- '''Script size limit''' The maximum script size of 10000 bytes does not apply. Their size is only implicitly bounded by the block weight limit.'''Why is a limit on script size no longer needed?''' Since there is no `scriptCode` directly included in the signature hash (only indirectly through a precomputable tapleaf hash), the CPU time spent on a signature check is no longer proportional to the size of the script being executed.
- '''Non-push opcodes limit''' The maximum non-push opcodes limit of 201 per script does not apply.'''Why is a limit on the number of opcodes no longer needed?''' An opcode limit only helps to the extent that it can prevent data structures from growing unboundedly during execution (both because of memory usage, and because of time that may grow in proportion to the size of those structures). The size of stack and altstack is already independently limited. By using O(1) logic for `OP_IF`, `OP_NOTIF`, `OP_ELSE`, and `OP_ENDIF` as suggested [https://bitslog.com/2017/04/17/new-quadratic-delays-in-bitcoin-scripts/ here] and implemented [https://github.com/bitcoin/bitcoin/pull/16902 here], the only other instance can be avoided as well.
- '''Sigops limit''' The sigops in tapscripts do not count towards the block-wide limit of 80000 (weighted). Instead, there is a per-script sigops ''budget''. The budget equals 50 + the total serialized size in bytes of the transaction input's witness (including the `CompactSize` prefix). Executing a signature opcode (`OP_CHECKSIG`, `OP_CHECKSIGVERIFY`, or `OP_CHECKSIGADD`) with a non-empty signature decrements the budget by 50. If that brings the budget below zero, the script fails immediately. Signature opcodes with unknown public key type and non-empty signature are also counted.'''The tapscript sigop limit''' The signature opcode limit protects against scripts which are slow to verify due to excessively many signature operations. In tapscript the number of signature opcodes does not count towards the BIP141 or legacy sigop limit. The old sigop limit makes transaction selection in block construction unnecessarily difficult because it is a second constraint in addition to weight. Instead, the number of tapscript signature opcodes is limited by witness weight. Additionally, the limit applies to the transaction input instead of the block and only actually executed signature opcodes are counted. Tapscript execution allows one signature opcode per 50 witness weight units plus one free signature opcode.'''Parameter choice of the sigop limit''' Regular witnesses are unaffected by the limit as their weight is composed of public key and (`SIGHASH_ALL`) signature pairs with ''33 + 65'' weight units each (which includes a 1 weight unit `CompactSize` tag). This is also the case if public keys are reused in the script because a signature's weight alone is 65 or 66 weight units. However, the limit increases the fees of abnormal scripts with duplicate signatures (and public keys) by requiring additional weight. The weight per sigop factor 50 corresponds to the ratio of BIP141 block limits: 4 mega weight units divided by 80,000 sigops. The "free" signature opcode permitted by the limit exists to account for the weight of the non-witness parts of the transaction input.'''Why are only signature opcodes counted toward the budget, and not for example hashing opcodes or other expensive operations?''' It turns out that the CPU cost per witness byte for verification of a script consisting of the maximum density of signature checking opcodes (taking the 50 WU/sigop limit into account) is already very close to that of scripts packed with other opcodes, including hashing opcodes (taking the 520 byte stack element limit into account) and `OP_ROLL` (taking the 1000 stack element limit into account). That said, the construction is very flexible, and allows adding new signature opcodes like `CHECKSIGFROMSTACK` to count towards the limit through a soft fork. Even if in the future new opcodes are introduced which change normal script cost there is no need to stuff the witness with meaningless data. Instead, the taproot annex can be used to add weight to the witness without increasing the actual witness size..
- '''Stack + altstack element count limit''' The existing limit of 1000 elements in the stack and altstack together after every executed opcode remains. It is extended to also apply to the size of initial stack.
- '''Stack element size limit''' The existing limit of maximum 520 bytes per stack element remains, both in the initial stack and in push opcodes.

==Rationale==

==Deployment==

This proposal is deployed identically to Taproot ([[bip-0341.mediawiki|BIP341]]).

==Examples==

The Taproot ([[bip-0341.mediawiki|BIP341]]) test vectors also contain examples for Tapscript execution.

==Acknowledgements==

This document is the result of many discussions and contains contributions by a number of people. The authors wish to thank all those who provided valuable feedback and reviews, including the participants of the [https://github.com/ajtowns/taproot-review structured reviews].