BITCOIN IMPROVEMENT PROPOSAL 340 · SPECIFICATION · 19 JANUARY 2020

Schnorr Signatures *for secp256k1*

Pieter Wuille · Jonas Nick · Tim Ruffing

pieter.wuille@gmail.com · jonasd.nick@gmail.com · crypto@timruffing.de

340

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.

BSD 2-Clause License. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Motivation

Bitcoin has traditionally used ECDSA signatures over the secp256k1 curve with SHA256 hashes for authenticating transactions. These are standardized, but have a number of downsides compared to 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)*[1] in the random oracle model assuming the hardness of the elliptic curve discrete logarithm problem (ECDLP) and in the generic group model assuming variants of preimage and second preimage resistance of the used hash function[2]. In contrast, the 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 malleable[3]; 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 BIP62 and 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 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 *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 in the past, needing 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 $s \cdot G = R + e \cdot P$. Two formulations exist, depending on whether the signer reveals e or R:

1. Signatures are pairs *(e, s)* that satisfy $e = hash(s \cdot G - e \cdot P \| m)$. 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 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.
2. Signatures are pairs *(R, s)* that satisfy $s \cdot G = R + hash(R \| m) \cdot P$. This supports batch verification, as there are no elliptic curve operations inside the hashes. Batch verification enables significant speedups.[4]

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 $(R, s + a \cdot hash(R \| m))$ for public key $P + a \cdot G$ 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 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*[5] Schnorr signatures which means that the public key is prefixed to the message in the challenge hash input. This changes the equation to $s \cdot G = R + hash(R \| P \| m) \cdot P$. 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 (BIP118), and would render the signature scheme unsuitable for other purposes than signing transactions, e.g., signing ordinary messages.

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

1. Encoding the full X and Y coordinates of P and R, resulting in a 64-byte public key and a 96-byte signature.
2. 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.
3. 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:

1. Implicitly choosing the Y coordinate that is in the lower half.
2. Implicitly choosing the Y coordinate that is even[6].
3. 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[7].

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.[8].

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 $s \cdot G = R + tagged_hash(r \| pk \| m) \cdot P$.

Specification

The following conventions are used, with constants as defined for secp256k1. We note that adapting this specification to other elliptic curves is not straightforward and can result in an insecure scheme[9].

- Lowercase variables represent integers or byte arrays.
  - The constant p refers to the field size, *0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F*.
  - 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 elliptic curve group operation.
  - Multiplication $(\cdot)$ 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 $i, j \geq 0$, 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 $x(P) = x$[10] 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 $x \geq p$.
    - Let $c = x^3 + 7 \bmod p$.
    - Let $y = c^{(p+1)/4} \bmod p$.
    - Fail if $c \neq y^2 \bmod p$.
    - Return the unique point P such that $x(P) = x$ and $y(P) = y$ if $y \bmod 2 = 0$ or $y(P) = p-y$ otherwise.
  - The function $hash_{name}(x)$ 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 $d' \geq n$.
- 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, 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 $d' \geq n$
- Let $P = d' \cdot G$
- Let $d = d'$ if *has_even_y(P)*, otherwise let $d = n - d'$.
- Let t be the byte-wise xor of *bytes(d)* and $hash_{BIP0340/aux}(a)$[11].
- Let $rand = hash_{BIP0340/nonce}(t \| bytes(P) \| m)$[12].
- Let $k' = int(rand) \bmod n$[13].
- Fail if $k' = 0$.
- Let $R = k' \cdot G$.
- Let $k = k'$ if *has_even_y(R)*, otherwise let $k = n - k'$.
- Let $e = int(hash_{BIP0340/challenge}(bytes(R) \| bytes(P) \| m)) \bmod n$.
- Let $sig = bytes(R) \| bytes((k + ed) \bmod n)$.
- If *Verify(bytes(P), m, sig)* (see below) returns failure, abort[14].
- 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 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 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 $r \geq p$.
- Let $s = int(sig[32:64])$; fail if $s \geq n$.
- Let $e = int(hash_{BIP0340/challenge}(bytes(r) \| bytes(P) \| m)) \bmod n$.
- Let $R = s \cdot G - e \cdot P$.
- Fail if *is_infinite(R)*.
- Fail if *not has_even_y(R)*.
- Fail if $x(R) \neq r$.
- 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 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 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 $r_i \geq p$.
  - Let $s_i = int(sig_i[32:64])$; fail if $s_i \geq n$.
  - Let $e_i = int(hash_{BIP0340/challenge}(bytes(r_i) \| bytes(P_i) \| m_i)) \bmod n$.
  - Let $R_i = lift_x(r_i)$; fail if $lift_x(r_i)$ fails.
- Fail if $(s_1 + a_2 s_2 + ... + a_u s_u) \cdot G \neq R_1 + a_2 \cdot R_2 + ... + a_u \cdot R_u + e_1 \cdot P_1 + (a_2 e_2) \cdot P_2 + ... + (a_u e_u) \cdot P_u$.
- 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.[15] 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 BIP341.)

Since pre-hashing may not always be desirable, e.g., when actual messages are shorter than 32 bytes,[16] 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.[17]

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 MuSig2 (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 distributed key generation, which enables interactive threshold signatures schemes, e.g., the schemes by Stinson and Strobl (2001), by Gennaro, Jarecki, Krawczyk, and Rabin (2007), or the FROST scheme including its variants such as 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

Adaptor signatures can be produced by a signer by offsetting his public nonce R with a known point $T = t \cdot G$, 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 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 simple blind signature scheme which is however insecure because it's vulnerable to Wagner's attack. Known mitigations are to let the signer abort a signing session with a certain probability, which can be proven secure under non-standard cryptographic assumptions, or to use zero-knowledge proofs.

Blind Schnorr signatures could for example be used in 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 collection of test vectors in CSV format, a naive, highly inefficient, and non-constant time pure Python 3.7 reference implementation of the signing and verification algorithm as well as the 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

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 structured reviews.

Footnotes

1 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.

2 A detailed security proof in the random oracle model, which essentially restates the original security proof by Pointcheval and Stern more explicitly, can be found in 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.

3 If *(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 proven non-malleable under stronger than usual assumptions.

4 The speedup that results from batch verification can be demonstrated with the cryptography library libsecp256k1.

5 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.

6 Since 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.

7 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 performance in practice, this trade-off is not worth it.

8 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 here.

9 Among 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.

10 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 $c = x^3 + 7 \bmod p$ and they can be computed as $y = \pm c^{(p+1)/4} \bmod p$ (see Quadratic residue) if they exist, which can be checked by squaring and comparing with c.

11 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.

12 Including the 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.

13 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 $1.27 * 2^{-128}$).

14 Verifying 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.

15 In theory, the message size is restricted due to the fact that SHA256 accepts byte strings only up to size of 2^{64}-1 bytes.

16 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.

17 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.

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