Analysis of Bitcoin Improvement Proposal 340 — Schnorr Signatures

Josh Elbahrawy, James Lovejoy, Anne Ouyang, Justin Perez

May 13, 2020

Abstract

Currently, Bitcoin's transaction throughput is limited by signature verification speed on the least powerful node in the network. BIP340 seeks to standardize Schnorr signatures, which will allow for batch verification of multiple signatures simultaneously, vastly improving transaction verification speed without requiring extra hardware capabilities. This paper will examine the security of properties of Schnorr signatures compared to the currently used ECDSA signatures, and will also analyze the process of implementing Schnorr signatures in three different programming languages—Python, Go, and C—in order to assess the completeness of the BIP340 specification as well as its utility.

Contents

1 Introduction

Since the patent on Schnorr Signatures expired, various members of the Bitcoin development team have been pushing to replace ECDSA with Schnorr for signatures in Bitcoin. In January, a small group of developers published a Bitcoin Improvement Proposal (BIP) that seeks to standardize an algorithm for Schnorr signatures (which have previously not been standardized due to the patent). The project analyzes the BIP and its reference implementation for completeness and security, and also offer recommendations as to how the BIP/code could be improved. This would be useful as for such an important BIP there are relatively few reviewers.

The written specification for Schnorr signatures should fully describe the algorithm. Anyone should be able to produce a new reference implementation that is consistent with others using only the description. In this project, we will implement the signature scheme based on the specification in different languages and compare them with the sample implementation. The goal is to comment on and check the completeness of the specification.

1.1 Permission

BIP-340 is a free software proposal using the 2-clause BSD license. Permission is thus not required to review the proposal or its associated reference implementation. In fact, more reviewers and scrutiny would be welcomed (and is encouraged) by the authors.

2 Background

2.1 ECDSA Signatures

The Elliptic Curve Digital Signature Algorithm (ECDSA) includes the following parameters:

- $CURVE$: an elliptic curve field and the equation used
- G: elliptic curve base point
- n: order of G

- d: private key
- Q: public key

Bitcoin has traditionally used ECDSA signatures over the *secp256k1* curve with SHA256 hashes for authenticating transactions. The *secp256k1* curve defines the curve $y^2 = x^3 + 7$ over the finite field F_p where $p = .2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.
G in compressed form is
$G =$ 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
And the order n is
$n =$ FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141.
In Bitcoin, the private key is a randomly chosen unsigned 256 bit (32 bytes) integer. A compressed public key has 33 bytes, with a prefix and a 32 byte integer x calculated by the elliptic curve.

To sign a message m, the following algorithm is used:

1. Calculate the hash of a message m using a cryptographic hash function: $e = HASH(m)$, and take the L_n leftmost bits where L_n is the bit length of the group order n.
2. Select a cryptographically secure random integer k from $[1, n-1]$
3. Calculate the curve point $(x_1, y_1) = k * G$; if $x_1 \mod n \equiv 0$ go back to step 2 and choose another k.
4. Let $r = x_1 \mod n$ and $s = k^{-1}(e + rd) \mod n$. If $s = 0$, go back to step 2 and choose another k. The signature is the pair (r, s)

To verify the signature (r, s) of a message assuming that all the given parameters are valid:

1. Calculate $e = HASH(m)$, and take the L_n leftmost bits.
2. Calculate $u_1 = es^{-1} \mod n$ and $u_2 = rs^{-1} \mod n$
3. Calculate $(x_1, y_1) = u_1 * G + u_2 * Q$. If (x_1, y_1) is the identity element, the signature is invalid.
4. The signature is valid if and only if $r \equiv x_1 \pmod n$.

2.2 Schnorr Signatures

The standardized 64-byte Schnorr signature algorithm outlined in BIP-340 uses the same elliptic curve (*secp256k1*) as the traditional ECDSA signatures, and it is possible to safely re-purpose existing key generation algorithms in a compatible way. In the Bitcoin specification of Schnorr signatures, the public key Q is 32 bytes, and it can be converted from existing generated public keys by dropping the first byte (the prefix). The secret key d can remain the same. A cryptographic hash function $H : \{0,1\}* \rightarrow \mathbb{Z}_n$ is agreed upon by all users.

To sign a message m:

1. Choose a random nonce k, compute $r = k * G$
2. Compute $s = k + H(Q||r||H(m)) * d$
3. (r, s) is the signature

To verify the signature (r, s) of a message m:

1. Compute $s * G$
2. Compute $r + H(Q||r||H(m)) * Q$
3. The signature is valid if and only if the results from steps 1 and 2 are the same

Compared to ECSDA signatures, Schnorr signatures offer several advantages, including provable security, non-malleability, and linearity.

2.3 Provable security

Schnorr signatures are provably secure while ECDSA signatures are only secure in practice – although there are currently no successful attacks that break ECDSA, a formal proof of its security has not been found.

The Schnorr signature scheme is constructed by applying the Fiat-Shamir heuristic to Schnorr's identification protocol. The Fiat-Shamir heuristic is a method of transforming an interactive proof of knowledge into a non-interactive proof of knowledge. It was proven in a 1998 paper by Pointcheval

and Stern [7] using the forking lemma that the class of signature schemes that are the transformations of a honest-verifier zero-knowledge identification protocols, are existentially unforgeable under chosen message attacks in the random oracle model, assuming the elliptic curve discrete logarithm problem is hard. This result guarantees that as long as the hash function behaves ideally, the only way to break Schnorr signatures is by solving the discrete logarithm problem.

On the other hand, the construction of ECDSA does not result from the Fiat-Shamir heuristic, so the forking lemma cannot be applied to argue for its security. The construction of ECDSA has two independent hash functions. One hash function H is used to hash the message, and there other one F is used to calculate a component of the signature r. The value r is calculated by applying the hash function F to the x coordinate of the point (x_1, y_1). Security proof exists when both hash functions H and F satisfy the random oracle model, but in practice, F is defined to be the modulo operation, and it does not satisfy ROM. On elliptic curves, the modulo function has a huge bias because applying F to a random element in the group does not result in a uniform distribution.

There's currently no rigorous analysis for the unforgeability of the unmodified ECDSA. Some of the current work includes a security proof for the Trusted El Gamal Type Signature Scheme that has more assumptions than the unmodified scheme. This is shown to be strongly unforgeable in the random oracle model. Brown's analysis on the unmodified ECDSA scheme idealizes the conversion hash function F, and formal aspects of his idealization remain unclear, so this does not lead to a rigorous proof on the practical security of ECDSA.

2.4 Non-malleability

A signature system is said to be strongly unforgeable if the signature is existentially unforgeable and, given signatures on some message m, the adversary cannot produce a new signature on m. It has been proven that Schnorr are SUF-CMA, so this implies that it's not possible to alter an existing signature into another valid one, which means that Schnorr is non-malleable.

One problem with ECDSA signatures is that they are inherently malleable. As shown above, ECDSA signatures are in the form of pairs (r, s), and the verification only uses the x-coordinate. So if the signature (r, s) is valid, it is possible to produce another valid signature $(r, -s)$. By plugging this point into the verification expression, we can get a point that is the negation of the original point. Only the y-coordinate is negated and the x-coordinate stays the same, as elliptic curves are symmetric across the x-axis. In practice, Bitcoin Improvement Proposal 62 [8] addresses this problem by restricting the values of s to low values. If the s values are too high, it is replaced with its corresponding low value.

2.5 Linearity

Elliptic curves have the multiplicative property – for two points X, Y and corresponding scalars x, y,

$$(x + y)G = xG + yG = X + Y$$

The construction of Schnorr signatures $s = k + H(Q||r||H(m)) * d$ is also linear, so it has the property of linearity. This property allows signature aggregation where multiple parties can combine their messages and public keys into a single message and public key, resulting into a single signature.

Given n messages m_i and private keys d_i (the corresponding public keys are $Q_i = d_i * G$), one simple way to construct an aggregate signature is the following:

1. Combine all messages by concatenating and taking the hash value: $m = H(m_1||m_2||...||m_n)$
2. Combine all public keys through a summation: $Q = \sum_{i=1}^{n} Q_i$
3. Compute $r = \sum_{i=1}^{n} r_i$, where $r_i = k_i * G$ and k_i is a random nonce for every user.
4. Compute $s = \sum_{i=1}^{n} s_i$ where $s_i = k_i + H(Q||r||m) * d_i$
5. The aggregate signature is (r, s)

The verification process is the same as verifying a non-aggregate signature.

Although this construction is valid, it can be subjected to **Rogue Key Attacks**: Assume there are two parties A and B, and B is an adversary trying to sign any message on their own without A. Let the public-private key pair of A and B be (d_A, Q_A) and (d_B, Q_B) respectively. If B claims that their public key is $Q'_B = Q_B - Q_A$, their aggregated public key will be $Q_A + (Q_B - Q_A) = Q_B$, which means that the adversary B can then sign any message on their own.

MuSig, a multi-signature scheme based on Schnorr signatures is a cryptographic construction to this problem. In this construction, users must pre-commit their public keys before generating signatures, so the aggregated public key in this construction is $Q = \sum_{i=1}^{n} H(L||Q_i) * Q_i$ where $L = H(Q_1||Q_2||...||Q_n)$. The signature generation scheme is then as following:

1. r = $\sum_{i=1}^{n} k_i * G$
2. s = $\sum_{i=1}^{n} k_i + H(Q||r||m) * H(L||Q_i) * d_i$
3. Return (r, s) as the signature

The verification scheme is the same.

This property of linearity is a major motivation for the proposal for Bitcoin to switch to Schnorr signatures, as key aggregation has several benefits

- Currently, if a transaction contains multiple inputs, the ECDSA signatures are verified individually. Key aggregation allows the batch verification method to be used, which can lead to performance improvements.
- Existing signatures take up around 70-72 bytes per transaction while Schnorr signatures only take up a fixed size of 64 bytes. Schnorr signatures take up less space on the blockchain, so it will have smaller transaction sizes and lead to lower fees. Having small transaction sizes can also allow fitting more transactions in a block.
- A verifier of the signature will not be able to distinguish between transactions involving a single signer and transactions involving multiple

signers. Furthermore, a verifier cannot identify the individual parties in an aggregate signature. This leads to greater privacy.

- Taproot is a proposal described in BIP341 whose goals is to minimize the amount of information revealed about the spendability conditions of a transaction. Currently, the smart contract will be revealed entirely if a user wants to spend the funds, so the triggering of one condition will lead to the knowledge of the existence of others. Taproot, on the other hand, only shows the condition triggered. Furthermore, multisigs can hide the existence of a script so that the transactions will look no different than a normal individual to individual transaction to an outside observer. This feature is enabled by Schnorr signatures and can also improve user privacy.

3 Implementations

3.1 Python Improved

3.1.1 Python Environment

The improved Python implementation was implemented in Python 3.6.9, and utilized the following libraries: *hashlib*, *binascii*, *random*, and *ecpy*[1]. The last of these, *ecpy*, provides methods for working with elliptic curves—including *secp256k1*, which is the curve that Bitcoin uses—and also provides objects representing points on curves, with the points themselves possessing useful methods for manipulating them. One particularly useful such method is multiplication of a point with a scalar: a point and a scalar can be multiplied using the $*$ operator the same way two integers are multiplied in Python. Such multiplication is non-trivial in pure Python, but the *ecpy* library makes it quite simple.

3.1.2 Spec Analysis

The BIP specification defines several helper functions and operators before describing how the signing and verification methods work. These helper functions are well defined, explicitly including each function's inputs and outputs

[1] some additional libraries were used, but none of them were strictly required for implementation

as well as clearly describing how the function operates on the inputs. For particularly complex functions, supplemental links are provided in case the implementer is unfamiliar with a certain process.

The specification then describes methods for signing, verification of an individual signature, and verification for a batch of signatures with the same detail that it described the functions and operators with. Each method description is written such that it is essentially pseudo-code: first, the method name and its parameters are given, with the rest of the method described by bullet points. In fact, each bullet point almost directly maps to a line of code assuming that the provided helper functions have also been implemented. For example, one bullet point in the single signature verification method says "Let $R = s \cdot G - e \cdot P$"; this exact representation appears in the implementation essentially unaltered. This is partially due to the simplicity of Python, but should mostly be attributed to the clarity of the specification. Indeed, we were able to produce working implementations of the BIP specification's signing and verification methods—including batch verification—with a *significantly* limited understanding of elliptic curve cryptography.

3.1.3 Performance

While performance is not a primary focus of this work, we felt that at least some basic performance bench-marking was in order since the Schnorr protocol is being proposed largely to reduce signature validation time. Using Python's built-in *time* library, we were able to measure execution times of the BIP's reference implementation of single signature verification against our own, as well as our own single signature verification implementation against our batch verification implementation. Here are some findings:

- Our improved implementation was approximately **13 times faster** than the reference implementation
  Our implementation correctly verified the BIP's test vectors in 0.151 seconds whereas the reference implementation took 2.02 seconds to verify.

- Our batch verification implementation was approximately **2.75 times faster** than our single signature verification implementation
  The batch verification implementation correctly verified a subset of

the BIP's test vectors in 0.0394 seconds whereas the single signature verification implementation took 0.108 seconds.

We feel that these results are promising for the BIP's approval since, in addition to improved security, our *highly unoptimized* implementation achieved significant speedup over the reference implementation—what additional performance gains can the Schnorr protocol achieve with further optimization and implementation in a more performant programming language?

3.2 Go

3.2.1 Go Environment

The Go implementation was implemented in Go 1.13.7 and utilized the following internal libraries: *math/big*, a built-in big integer library, *crypto/sha256*, a built-in implementation of sha256 hashing, and *errors*, in order to give useful error reporting. The implementation also utilized the external libraries, *github.com/ethereum/go-ethereum/crypto/secp256k1*, a Go wrapper of the bitcoin secp256k1 C library, and *golang.org/x/crypto/chacha20* for ChaCha20 encryption which is used in batch verification. The *secp256k1* library mirrors the functionality of the built-in Go library *crypto/elliptic* but uses constant time implementations for the secp256k1 curve operations.

3.2.2 Spec Analysis

Rather than overriding the arithmetic operators for point objects Go provides distinct methods on points for operations. Therefore, the translations from specification to code are not as verbatim as with Python. However, it was still relatively simple to translate the specification to functioning Go code, deriving from its completeness and specificity in defining all necessary functions. Overall, the result was simple code that was relatively performant, as compared to the reference and existing Go implementations of schnorr signatures.

3.2.3 Existing Go Implementation

We also investigated the performance and interface of an existing Go implementation of the BIP, *github.com/hbakhtiyor/schnorr*. The implementations are comparable in terms of performance, the existing one running the sign

and verify tests 10 times in 0.295 seconds, our implementation completing all Sign and Verify tests in 0.312 seconds. On the other hand, their interfaces differ significantly. For example, while our implementation takes variable length slices of bytes as arguments, and returns errors if given the wrong length, the existing implementation takes in fixed size arrays, effectively requiring the correct length beforehand. However, they are somewhat inconsistent in this convention, and somewhat inconsistent with the specification. The Sign function, rather than taking in a byte array for the secret key as the specification and the rest of the implementation suggests, takes in a pointer to an integer. In addition, their verify function takes in a 33 byte array for the public key for some unknown reason rather than a 32 byte one as the specification suggests. Because of our comparable performance, arguably superior interface, and faithfulness to the specification, we believe that our implementation is superior.

3.2.4 Performance

As stated in the previous section, our implementation performed the sign and verify tests 10 times in 0.312 seconds. The reference implementation runs ten times in 22.053 seconds. This is a 70 fold performance increase from the reference implementation.

The batch verify implementation verifies 5 messages in 0.026 seconds, while they verify individually in 0.025 seconds. Surprisingly, as opposed to the improved Python implementation, the batch verify was slower than the individual verification, if only slightly. We would likely see a greater performance increase as the size of the batch grew.

3.3 C

3.3.1 Implementation Details

We also implemented the BIP in C using the OpenSSL cryptography library [4]. The implementation is just under 900 lines of code and uses the BIGNUM, EC_POINT, EC_GROUP and SHA256 components of OpenSSL. Due to time constraints we did not implement batch verification for the C version. Similarly to Go, C does not support operator overloading meaning that the translation from the specification to C requires many more lines of code than the reference Python implementation. Every arithmetic operation

must be implemented as a function call with a heap allocation for the result value, leading to many more intermediate variables being required. Furthermore, unlike Python, C lack built-in support for types wider than 64-bits, requiring the use of an external big number implementation to handle the integers of the size requires for Schnorr signatures. Thankfully, OpenSSL is a mature library offering all of the required functions for implementing BIP340 that the C standard library does not include.

3.3.2 Production Use

While testing for correctness according to the provided test vectors in the specification, our implementation should not be used in production. The code has not been tested against side-channel attacks and makes no effort to clean memory after use or ensure functions run in constant-time irrespective of input values. That being said, if side-channel resistance is not required, the implementation is correct according to the specification. OpenSSL's big number implementation does contain a flag for enabling constant time operations, but we did not experiment with this feature, so it is possible that constant-time operation could be implemented in the future.

3.3.3 Performance

As expected the C implementation is significantly faster than the Python reference implementation. Testing on an AMD FX-9590 CPU core, the Python implementation using Python 3.8 compiled with GCC 5.4.0 completed an average of 3.30 sign operations per second, and 6.50 verify operations. The C implementation on the same hardware, compiled with the same compiler using OpenSSL 1.1.1g and the compiler optimization flag $-O3$, performed 546 sign operations per second, and 1570 verify operations. This represents a $>$ 160X increase in signing performance, and a $>$ 240X increase in verification performance.

While this is clearly a significant improvement, there is still a lot of scope for further performance gains. The code could be restructured to minimize the number of heap memory allocations per operation. Currently every intermediate value requires a new output value to be allocated, which could be reduced to a minimum number of variables. Furthermore, constants such as $\frac{(p-1)}{2}$ used to calculate the Legendre number, or the hash values of tags used in tagged hashes, could be pre-computed rather than recalculated every

time they are used. Finally, the points could be converted to use Jacobian coordinates rather than affine allowing a simplification of the verification equation and a further reduction in the operation count by eliminating the need for modular inverses. The use of Jacobian coordinates is suggested in the specification for performance.

4 Conclusion

Since the BIP specification provides clear security analyses and arguments, as well as a clear and comprehensive guide to implementing the methods required for Schnorr signatures, it is our conclusion that the specification is complete. Schnorr's allowance for batch verification of signatures will allow for significantly faster transaction processing, improving Bitcoin's scalability without stronger hardware requirements, and potentially paving the way for more widespread adoption of Bitcoin. As such, we feel that Bitcoin Core should indeed adopt BIP 340 and standardize the use of Schnorr signatures.

Acknowledgements

We would like to acknowledge the authors of BIP-340 Pieter Wuille, Jonas Nick, and Tim Ruffing for authoring a proposal so well-written, our team of novices was able to mostly comprehend and implement it. We would also like to acknowledge the 6.857 course staff for not only teaching an extremely educational course, but also for making it an interesting and entertaining experience along the way—adapting to the Covid-19 pandemic, no less!

References

[1] BIP-Schnorr Whitepaper

[2] BIP340 Improved Python Implementation

[3] BIP340 Go Implementation

[4] BIP340 C Implementation

[5] Bitcoin Wiki secp256k1

[6] ECDSA Wikipedia

[7] Pointcheval and Stern, Security Arguments for Digital Signatures and Blind Signatures

[8] Bitcoin Improvement Proposal 62

[9] Introduction to Schnorr Signatures

[10] How will Schnorr Signatures Benefit Bitcoin?