Multi-signatures For Cryptocurrencies

BY

Paria IGHANIAN
Damian TABACZYŃSKI

SUPERVISOR
Dr. Rajeev Anand SAHU

February 2, 2022

Abstract

All transactions on the blockchain are accompanied by digital signatures, which allow proving rights of ownership. Bitcoin uses the ECDSA (Elliptic Curve Digital Signature Algorithm) signature scheme which was chosen by Satoshi Nakamoto during the design. Schnorr is currently being discussed in the community as a superior algorithm to ECDSA. This report presents a new Schnorr-based multi-signature scheme called MuSig which makes use of key aggregation and is provably secure in the plain public-key model. As an example, we explain how this new multi-signature scheme would improve both performance and privacy in Bitcoin.

Introduction

The Schnorr multi-signature technology provides a fairly simple and neat solution of combining digital signatures and keys where multiple signers with their own public and private keys sign one message. This means that instead of generating an individual signature for each new bitcoin transaction, signatories can only use one signature. This single signature can be verified by anyone who also knows the message and the public keys of the signers.

1. Digital Signature

In order to better understand what a digital signature scheme is, consider the situation in which Alice wants to sign a document that she is sending to Bob, who wants assurance that the document came from Alice and has not been altered. Just like a handwritten signature that binds a person's identity to a document, a digital signature is unique to each signer which is indeed a mathematical technique used to verify the authenticity and integrity of a message or digital document. A digital signature should provide three important properties which are authentication, non-repudiation and integrity. Authentication is the process of validating that a message was actually created and sent by the claimed sender. Non repudiation does not allow the sender to deny any association with the signed content. Integrity refers to whether data is received in the same state as when it was sent.

As mentioned above, a digital signature works by proving that the digital message or document has not changed since it was signed. The digital signature does this by generating a unique hash of the message or document and encrypting it with the sender's private key. In this model, Alice (the signer) wants to digitally sign a document. First she generates two keys, private key and public key, which are linked to each other. Next a cryptographic hash is generated for the document. That cryptographic hash is encrypted by her private key, appended to the document and sent to Bob along with her public key. Then Bob generates its own hash of the document and decrypts Alice’s hash by her public key. He compares the hash he generates to Alice’s decrypted hash. If they match, it proves that the data hasn't changed since it was signed and Alice is the real sender because the value of a hash is unique to the hashed data

and any change in the data will result in a different value. If the two hashes don't match, the data has either been tampered with in some way or the signature was created with a private key that doesn't correspond to the public key of Alice, which is an authentication problem. The reason for encrypting the hash instead of the entire message or document is that a hash function can convert an arbitrary input into a fixed-length value, which is usually much shorter and saves time as it is much faster than signing.

Now we can look more formally at the signature scheme. A signature scheme is a set of four algorithms (Setup, KeyGen, Sign, Ver) where:

- A central authority, on input the security parameter λ, runs the algorithm Setup to produce the global information params.
- The signer runs the key generation algorithm to generate a key pair: **KeyGen(pp) → (pk, sk)**
- The signer runs the signing algorithm on message m and private key sk, resulting in a signature **σ**: **Sign(sk, m, pp) → σ**
- A verifier can check the validity of the signature running the verification algorithm: **Ver(pk, m, σ, pp) → {0, 1}**. If the result is equal to 1 then the signature is valid, otherwise it is not.

2. Schnorr Signature

Schnorr signature is a digital signature, secure under discrete logarithm problem and known for its simplicity, was invented by Claus-Peter Schnorr back in the 1980s. He implemented his digital signature much simpler than other contemporary algorithms. However, Schnorr's signature algorithm was not widely used until decades later due to his patent. The main difference between Schnorr signature and Bitcoin current signature (ECDSA) is that Schnorr signature is Linear, while ECDSA signature is not.

In the Schnorr signature scheme, we start with a cyclic group $\mathbb{G}$ = <g> of prime order p and pick a random secret key sk $\in \mathbb{Z}_p$. To sign a message m, we pick r $\in \mathbb{Z}_p$ randomly and compute R = g^r, c = H (X, R, m), s = r + cx mod p. Then the final signature is σ =(R,s). In a formal way:

- Setup. A cyclic group $\mathbb{G}$ of prime order p, a generator g of $\mathbb{G}$, a hash function H
- Key generation. A private/public key pair (x, X) where $X = g^x$, $x \in \{0, \ldots, p - 1\}$
- Signature. σ =(R,s) where R = g^r for r $\in \mathbb{Z}_p$, s = r + cx, c = H (X, R, m)
- Verification. $g^s = RX^c$

3. Multi-signature

Multisignature, introduced by Itakura and Nakamura, allows a group of signers to generate a joint signature σ on a common message m such that a verifier ensures that each member of the group took part in signing. As an example, a multisignature transaction on Bitcoin requires more than one key to authorize a transaction. Multisig transactions are also known as M-of-N transactions, where M is the required number of signatures or keys and N is the total number of signatures or keys involved in the transaction. A simple way to change a standard signature scheme into a multi-signature scheme is to have each signer produce a stand-alone signature with their private key, and to then concatenate all individual signatures. However, this approach is not useful, and multisignature can only offer advantages over a standard signature if its size is independent of the number of signers.

4. Naive Schnorr Multi-signature

Assume that n signers want to cosign a message m. Here is a naive way to design a Schnorr multi-signature scheme:

- Setup. A cyclic group $\mathbb{G}$ of prime order p, a generator g of $\mathbb{G}$, a hash function H
- Key generation. A private/public key pair $(\boldsymbol{x}, \boldsymbol{X})$ where $\boldsymbol{X} = g^{x}$, $\boldsymbol{x} \in \{0, \ldots, p - 1\}$
- Signature. $\sigma = (R, s)$ where $s = \Sigma^{n}_{i=1} s_i \bmod p$

  $L = \{ \boldsymbol{X}_1 = g^{x_1}, \ldots, \boldsymbol{X}_n = g^{x} \}$ the set of all signer's public keys, $\boldsymbol{X} = \boldsymbol{\Pi}^{n}_{i=1} \boldsymbol{X}_i$ the product of individual public keys. Each signer randomly generates $R_i = g^{r}$ and sends to cosigners then each of them computes $R = \boldsymbol{\Pi}^{n}_{i=1} R_i$, $c = H(\boldsymbol{X}, R, m)$, partial signature $s_i = r_i + c\boldsymbol{x}_i$
- Verification. The validity of the joint signature can be assessed by $g^{s} = R\boldsymbol{X}^{c}$

But this scheme is not secure and suffers from the cancellation problem. Consider a simple scenario when two signers, namely Alice and Bob, want to sign a common message by this scheme. Here, Alice knows Bob's public key and nonce ahead of time, since she has been waiting for him to reveal them. Now Alice sets her public key and public nonce as follows $\boldsymbol{X}'_a = \boldsymbol{X}_a - \boldsymbol{X}_b$, $R'_a = R_a - R_b$. Note that Alice does not know the private keys for these faked values. After applying the aggregation scheme, we have $s_{agg} = s_a$ which means Alice can create this signature herself. This scenario is known as the key cancellation attack.

5. Rogue Key Attack

Rogue Key attack is a significant concern when using multi-signature schemes. In this case, corrupted signers manipulate the public keys computed as functions of the public keys of honest users, enabling them to easily produce forgeries for the set of public keys despite not knowing the associated secret keys. A general method to prevent rogue-key attacks is to

require users to prove knowledge or possession of the secret key during public key registration with a certification authority.

6. Bellare and Neven Multi-signature

The Bellare-Neven scheme prevents rogue key attack by having a distinct challenge c_i for each signer in its partial signature.

- Setup. A cyclic group $\mathbb{G}$ of prime order p, a generator g of $\mathbb{G}$, two hash functions H, H'
- Key generation. A private/public key pair $(\boldsymbol{x}, \boldsymbol{X})$ where $\boldsymbol{X} = g^{x}$, $\boldsymbol{x} \in \{0, \ldots, p - 1\}$
- Signature. $\sigma = (R, s)$ where $s_i = r_i + c_i\boldsymbol{x}_i$, $c_i = H(\langle L \rangle, \boldsymbol{X}_i, R, m)$
  <L> is some unique encoding of the multiset of public keys $L = \{ \boldsymbol{X}_1 = g^{x_1}, \ldots, \boldsymbol{X}_n = g^{x} \}$
  Each signer randomly generates $R_i = g^{r}$ and sends $t_i = H'(R_i)$ to cosigners then each of them checks $t_i = H'(R_i)$ for received R_i and computes $R = \boldsymbol{\Pi}^{n}_{i=1} R_i$
- Verification. $g^{s} = R\boldsymbol{\Pi}^{n}_{i=1}\boldsymbol{X}_i^{ci}$ for a signature (R,s) on message m for public keys L

BN multi-signature scheme is based on the Schnorr scheme and is proven secure in the plain public-key model under the Discrete Logarithm assumptions, modeling H and H' as random oracles. However, key aggregation is no longer possible since the entire list of public keys is required for verification.

7. Maxwell Multi-signature

This scheme is a variant of the BN scheme which is also based on Schnorr multi-signature.

- Setup. A cyclic group $\mathbb{G}$ of prime order p, a generator g of $\mathbb{G}$, three hash functions H_{com}, H_{agg}, H_{sig}
- Key generation. A private/public key pair $(\boldsymbol{x}, \boldsymbol{X})$ where $\boldsymbol{X} = g^{x}$, $\boldsymbol{x} \in \{0, \ldots, p - 1\}$
- Signature. $\sigma = (R, s)$ where $s_i = r_i + ca_i\boldsymbol{x}_i$, $c = H_{agg}(\langle L \rangle, \boldsymbol{X}_i)$. $H_{sig}(\boldsymbol{X}, R, m)$
  $a_i = H_{agg}(\langle L \rangle, \boldsymbol{X}_i)$, $\boldsymbol{X} = \boldsymbol{\Pi}_{i=1}^{n} \boldsymbol{X}_i^{ai}$, **<L>** is some unique encoding of the multiset of public keys $L = \{ \boldsymbol{X}_1 = g^{x_1}, \ldots, \boldsymbol{X}_n = g^{x} \}$ Each signer randomly generates $R_i = g^{r}$ and sends $t_i = H_{com}(R_i)$ to cosigners then each of them checks $t_i = H_{com}(R_i)$ for received R_i and computes $R = \boldsymbol{\Pi}^{n}_{i=1} R_i$
- Verification. $g^{s} = R\boldsymbol{\Pi}^{n}_{i=1}\boldsymbol{X}_i^{ai\,c}$ for a signature (R,s) on message m for public keys L

The verification is similar to standard Schnorr signatures with respect to the “aggregated” public key $\boldsymbol{X} = \boldsymbol{\Pi}_{i=1}^{n} \boldsymbol{X}_i^{ai}$. However, MuSig uses less block space but it requires more interactivity between the participants.

8. Applications to Bitcoin

After all, let's dig into some current and future use cases of multi-signatures in Bitcoin. One of the favorite use cases of multi-signatures is Lightning Network channels that allow transactions between parties not on the blockchain network. Let's say that every morning Bob buys a cup of coffee on his way to work. He can set up a payment channel with the coffee shop by Lightning Network. To do that, both the coffee shop and Bob deposit a certain amount of Bitcoin in what is called a multi-signature address. This multi-signature address is basically like a safe that can only be opened when both parties agree.

Another use case is multisig wallet which is a lock that can only be unlocked with enough keys out of a set of predefined keys. In a common multi-signature wallet, instead of one private key there are three keys associated with the wallet. So a rule can be set that at least two of three keys are required in order to move funds from the wallet.

9. Implementation

To introduce the basic backbone of our implementation, firstly we need to explain our design choices and show important connections and differences between theoretical and practical approaches.

a) Design Choices

Among the significant changes was the implementation of **Schnorr Subgroup** with **prime order *q***. The motivation for that decision was taken from Schnorr's paper [3] Efficient signature generation by smart cards. The non formal definition can be described as: **Schnorr groups** are subgroups of prime order ***q*** of the multiplicative group of the finite field $\mathbb{Z}_p$ such that ***p*** is large enough (at least 2048 bits) to resist index calculus on the field and ***q*** is large enough (at least 256 bits) to resist Pollard's ***ρ*** or other generic discrete log attacks on the subgroup.

To create that kind of subgroup it is necessary to generate ***p, q, k*** such that ***p = qk + 1*** and ***p,q*** are primes. The next step is to find a generator by choosing a random **k** in the range ***(1,p)*** until finding one such that $\mathbf{h^k \neq 1 \bmod p}$. Generator of a subgroup of $\mathbb{Z}_p$ of order ***q*** *is* $\boldsymbol{g = h^k \bmod p}$.

Object	Description	Theory	Our Implementation
p	Prime number	At least 2048 bit size	Maximum 256 bit size
q	Prime number	At least 256 bit size	Maximum 128 bit size
k	Random number	-	Maximum 128 bit size

As a result of the Schnorr group being used, the **mod p** in some cases is changed into a **mod q** according to Maxwell Multi-Signature:

- Calculation of signer signature $\mathbf{s_i = r_i + ca_ix_i \bmod q}$
- Computation of aggregated multisignature $\mathbf{s = \sum_{i=1}^{n} s_i \bmod q}$

In our solution we used **Python** 3.10 as a programming language. Since version **3.0** of this language, it is possible to define integers with more than **32 bits**. The implementation process was made significantly easier by this feature. Due to physical resource limitations, we decided to limit ***p*** and ***q*** to 256 bits and 128 bits respectively. In addition for simplicity we used only one hash function “SHA-256”, so that in our case functions H_{com}, H_{agg}, H_{sig} are the same.

b) Code Structure

The project employs object-oriented programming paradigms. Code is organized in three main classes:

- CyclicGroup - responsible for generating ***p, q, k*** and generator in setup phase. In later stages it gives access to that shared information.

```
class CyclicGroup:
    def __init__(self):
        self.p = 0
        self.q = 0
        self.elements = []
        self.generator = 0
        self.hash_name = "sha256"
        self.k = 0

    def generate_prime_order_subgroup(self):
        ...

    def select_generator(self, generate_elements=False):
        ...
```

- Signer - abstract class that gives basic attributes and methods (that are working in key generation phase) to all concrete implementations of signatures. Generates private and public keys.

```
class Signer(ABC):
    def __init__(self, cyclic_group):
        self.cyclic_group = cyclic_group
        self.private_key = 0
```

```
        self.public_key = 0

    def generate_keys(self):
        self.private_key = random.randrange(0, self.cyclic_group.q)
        self.public_key = pow(cyclic_group.generator, self.private_key, cyclic_group.p)
        return self.public_key

    def compute_challenge(self, X, R, message):
        ...

    @abstractmethod
    def create_aggregated_multisig(self, R, si_list):

    @abstractmethod
    def verify_message(self, message, signers, R, s):
```

- MaxwellSigner - subclass of Signer, implements all 3 rounds of Maxwell scheme and verification phase. Create a signature and verify its correctness.

```
class MaxwellSigner(Signer):
    # Round 1
    def calculate_aggregated_public_key(self, data, L):
        ...

    # Round 2
    def send_hashed_random_value(self, data):
        ...

    # Round 2
    def send_random_value(self, data):
        ...

    # Round 2
    def verify_committed_values(self, ti_list, Ri_list):
        ...

    # Round 3
    def sign_message(self, data, Ri_list, message):
        aggregated_R = 1
        for Ri in Ri_list:
            aggregated_R *= Ri
        aggregated_R = aggregated_R % self.cyclic_group.p

        c = self.compute_challenge(data.X_aggregated, aggregated_R, message)
        si = (data.ri + c * data.ai * self.private_key) % self.cyclic_group.q
```

```
        return aggregated_R, si

    # Round 3
    def create_aggregated_multisig(self, R, si_list):
        s = sum(si_list) % self.cyclic_group.q
        return R, s

    # Verification
    def verify_message(self, message, L, R, s):
        X_aggregated = 1
        for key in L:
            ai = hash_data(self.cyclic_group.hash_name, str(L), key)
            X_aggregated *= pow(key, ai, self.cyclic_group.p)

        X_aggregated = X_aggregated % self.cyclic_group.p
        c = self.compute_challenge(X_aggregated, R, message)
        left_side = pow(cyclic_group.generator, s, cyclic_group.p)
        right_side = (R * pow(X_aggregated, c, cyclic_group.p)) % cyclic_group.p
        return left_side == right_side
```

c) Execution Flow

Program execution is divided into 4 main phases:

1. **Setup** - Generating p, q and other public parameters in CyclicGroup class instance.
2. **Key Generation** - Initializing users (signers). Each user creates private and public keys then adds the public one to list **L**.
3. **Signature** - Every signer in round 1 calculates $a_i = H_{agg}(\langle L \rangle, X_i)$ and aggregated public key $X = \Pi_{i=1}^{n} X_i^{a_i}$. In round 2 users send calculated hashed value $H'(\mathbf{R_i})$. After receiving all hashed values signers send value $\mathbf{R_i}$. With full list of $\mathbf{R_i}$ every user checks integrity of received $\mathbf{R_i}$ with $H'(\mathbf{R_i})$. In round 3 each user creates signature (R, s_i) which afterwards is aggregated in one multisignature $\mathbf{s = \sum_{i=1}^{n} s_i \bmod q}$.
4. **Verification** - one user which does not take part in signing the message, verifies message signature.

10. Results

Taking into account factors such as the number of signers and phase, we would like to present the runtime of our solution here.

Number of signers	Setup time [ms]	Key Generation time [ms]	Signature time [ms]	Verification time [ms]	Overall time [ms]
10	55,99	1,55	16,67	2,44	76,65
30	41,96	3,49	139,65	4,97	190,07
100	51,43	11,27	1950,04	18,18	2030,91
300	70,96	30,71	26757,31	93,43	26952,41

11. Conclusion

The purpose of this report is to provide an introduction to the Schnorr signature algorithm and describe some of its amazing applications to bitcoin and the benefits and improvements that would result from its implementation. Improved efficiency (smaller signatures, batch validation, cross-input aggregation) and privacy (multi-signatures and threshold signatures would be indistinguishable from a single signature) are the benefits that Schnorr would bring to Bitcoin.

12. Acknowledgment

We would like to thank our supervisor, Dr. Rajeev Anand SAHU, for his valuable discussions and feedback.

References

1. Maxwell, G., Poelstra, A., Seurin, Y., & Wuille, P. (2019). Simple Schnorr multi-signatures with applications to Bitcoin. Designs, Codes and Cryptography, 87(9), 2139–2164. https://doi.org/10.1007/s10623-019-00608-x

2. Bellare, Mihir & Neven, Gregory. (2006). Multi-signatures in the plain public-Key model and a general forking lemma. Proceedings of the ACM Conference on Computer and Communications Security. 390-399. 10.1145/1180405.1180453.
   Multi-Signatures in the Plain Public-Key Model

3. Schnorr, C. P. (1991). Efficient signature generation by smart cards. Journal of Cryptology, 4(3), 161–174. https://doi.org/10.1007/bf00196725

4. DUFKA, Antonín. *Schnorr Signatures with Application to Bitcoin* [online]. Brno, 2020 [cit. 2022-02-02]. Available from: https://theses.cz/id/53xlvr/ Master's thesis. Masaryk University, Faculty of Informatics. Thesis supervisor doc. RNDr. Petr Švenda, Ph.D.

5. The MuSig Schnorr Signature Scheme. (2019, January 20). Tarilabs. https://tlu.tarilabs.com/cryptography/The_MuSig_Schnorr_Signature_Scheme

6. Đức Phong, Lê & Yang, Guomin & Ghorbani, Ali. (2019). A New Multisignature Scheme with Public Key Aggregation for Blockchain. 1-7. 10.1109/PST47121.2019.8949046. https://www.researchgate.net/publication/338451351_A_New_Multisignature_Scheme_with_Public_Key_Aggregation_for_Blockchain