24.07.30 Schnorr MuSig and BLS Benchmark

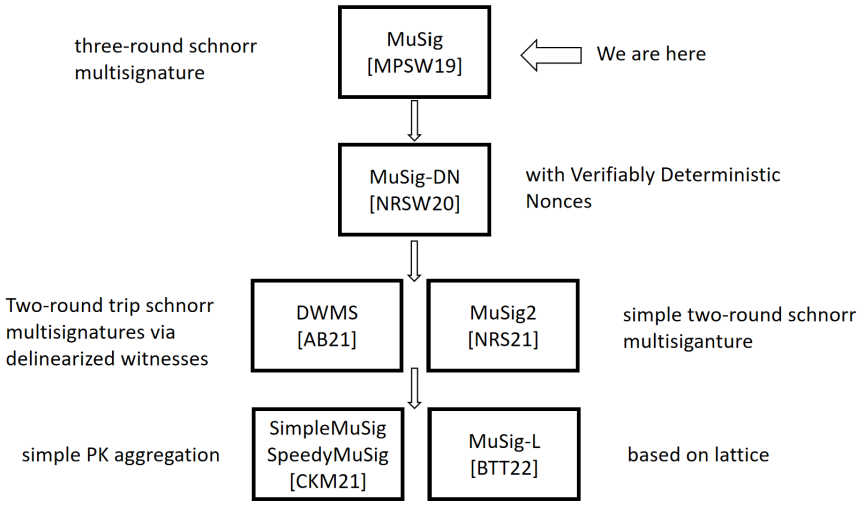
Xun Zhang Wuyun Siqin Bingsheng Zhang
Zhejiang University, CHN
22221024@zju.edu.cn 3210101763@zju.edu.cn bingsheng@zju.edu.cn

July 30 2024

1 Schnorr Multi-signature Scheme

1.1 Roadmap

We give a roadmap to the existing Schnorr Multi-signature scheme(it's called 'MuSig' specifically).



The Schnorr multisignature we implemented is the paper [MPSW19], the scheme is called *MuSig*. It is a three-round pratical schnorr multisignature with public key aggregation.

1.2 MuSig [MPSW19]

We try to give a detailed description to the MuSig scheme. Note that all the multisignature schemes involve all public keys to participate in signing.

Round 1.

A group of n signers want to cosign a message m. Let X_1 and x_1 be the public and private key of a specific signer, let $X_1, ..., X_n$ be the public keys of other cosigners and let $\langle L \rangle$ be the multi-set of all public keys involved in the signing process.

For $i \in \{1, ..., n\}$, the signer computes the following:

$$a_i = \mathrm{H}_{agg}(\langle L \rangle, X_i)$$

as well as the "aggregated" public key:

$$\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$$

Round 2.

The signer generates a random private nonce r_1, computes $R_1 = g^{r_1}$ (the public nonce) and commitment $t_1 = \mathrm{H}_{com}(R_1)$ and sends t_1 to all other cosigners.

When receiving the commitments $t_2, ..., t_n$ from the other cosigners, the signer sends R_1 to all other cosigners. This ensures that the public nonce is not exposed until all commitments have been received.

Upon receiving $R_2, ..., R_n$ from other cosigners, the signer verifies that $t_i = \mathrm{H}_{com}(R_i)$ for all $i \in \{2, ..., n\}$.

The protocol is aborted if this is not the case.

Round 3.

If all commitment and random challenge pairs can be verified with H_{agg}, the following is computed:

$$R = \prod_{i=1}^{n} R_i$$

$$c = \mathrm{H}_{sig}(\tilde{X}, R, m)$$

$$s_1 = r_1 + ca_1x_1$$

Signature s_1 is sent to all other cosigners. When receiving $s_2, ..., s_n$ from other cosigners, the signer can compute $s = \sum_{i=1}^{n} s_i \bmod p$. The signature is $\sigma = (R, s)$.

In order to verify the aggregated signature $\sigma = (R, s)$, given a lexicographically encoded multi-set of public keys $\langle L \rangle$and message m, the verifier computes:

$$a_i = \mathrm{H}_{agg}(\langle L \rangle, X_i) \quad \text{for} \quad i \in \{1, ..., n\}$$

$$\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$$

$$c = \mathrm{H}_{sig}(\tilde{X}, R, m)$$

then accepts the signature if:

$$g^s = R \prod_{i=1}^{n} X_i^{a_i c} = R\tilde{X}^c$$

1.3 MuSig2 [NRS21]

For the convenience of explanation, we just use a two-party setting.

Round 1.

This round is completely same as the original *MuSig* scheme. But it can be more efficient by use a hash function:

$$L = \mathrm{H}(X_1||X_2)$$

$$a_i = \mathrm{H}_{agg}(L, X_i)$$

$$X = a_1X_1 + a_2X_2$$

Round 2. The signer Alice generates **TWO** random private nonce r_1' and r_1'', computes $R_1' = g^{r_1'}$ and $R_1'' = g^{r_1''}$(the public nonce), and sends them to Bob.

After receiving Bob's nonce R_2' and R_2'', Alice do the follwing calculation:

$$R' = R_1' + R_2'$$

$$R'' = R_1'' + R_2''$$

the coefficient used for nonce combination will be:

$$b = \mathrm{H}_{agg2}(X||R'||R''||m)$$

and the final nonce as following:

$$R_1 = R_1' + bR_1''$$

$$R_2 = R_2' + bR_2''$$

$$R = R_1 + R_2$$

the challenge will be:

$$c = \mathrm{H}_{sig}(\tilde{X}, R, m)$$

Now it's time for Alice to create her response $s1$ (Bob will create $s2$ in a similar fashion):

$$r_1 = r_1' + br_1''$$

$$s_1 = r_1 + ca_1x_1$$

and the final multisignature is $(R, s) = (R, s_1 + s_2)$

One can verify the signature just as a normal schnorr signature(after computing a aggregated public key X).

1.4 SpeedyMuSig [CKM21]

This work is from **Elizabeth Crites** et al. It construct a variant of the two-round multisignature scheme **MuSig2** that includes proofs of possession. And the public key aggregation is very simple(just multiply all the pks).

Key Generation.

Each signer will compute their own hash $\overline{c} = \mathrm{H}_{reg}(X, X, \overline{R})$. Where $\overline{R}$ is the commitment of random nonce $\overline{R} = g^{\overline{r}}$. Signer compute the signature $\overline{z} = \overline{r} + \overline{c}x$. Their proof of possession is the signature $\pi = (\overline{R}, \overline{z})$.

Key Verification.

On input a public key (X, π), the verifier computes $\overline{c} = \mathrm{H}_{reg}(X, X, \overline{R})$ and accepts if $\overline{R}X^{\overline{c}} = g^{\overline{z}}$.

Round 1.

Same as **MuSig2**, each signer generate their own R_i' and R_i''.

Round 2.

Compute the simple aggregated public key:

$$\tilde{X} = \prod_{i=1}^{n} X_i$$

The other opration is totally same as **MuSig2**.

And the final signature is :

$$s_i = r_i + cx_i$$

This is a basic Schnorr signature. And the final multisignature is $(R, s) = (R, \sum s_i)$.

One can verify the signature just as **MuSig2**, but with a **simple** publick key aggregation.

2 BLS Benchmark

2.1 Version 1: Built-in BLS Signatures in halo2-lib

2.1.1 Data Generation

The data generation process for this version includes: Randomly generating secret keys (*sk*). Deriving the public keys (*pk*) and signatures (*sig*) from the secret key. Generating a random message hash (*msg_hash*).

2.1.2 Code Description

The BLS signature verification code is implemented in the `BlsSignatureChip` struct. The overview is as following.

Input Parameters G_1:{g_1, pubkeys} , G_2 {signatures, msghash}.

Verification Process The function ensures that $e(g_1, \text{signature}) = e(\text{pubkey}, H(m))$ by verifying that $e(g_1, \text{signature}) \cdot e(\text{pubkey}, -H(m)) = 1$, where e is the optimal Ate pairing.

2.1.3 Benchmark Results

The following table summarizes the benchmark results for different parameter settings,

Note: For all benchmark results in Table **??**, the *Num Fixed* is 1, *Num Limbs* is 3, and *Num Aggregation* is 2.

Degree	Advice	Lookup	Lookup Bits	Limb Bits	Proof Time	Proof Size	Verify Time
14	211	27	13	91	19.9237s	95808	103.104ms
15	105	14	14	90	17.3673s	48000	78.1614ms
16	50	6	15	90	16.3698s	22752	70.8313ms
17	25	3	16	88	16.6448s	11520	72.8319ms
18	13	2	17	88	18.9069s	6080	82.6032ms
19	6	1	18	90	19.2544s	3072	85.9115ms
20	3	1	19	88	26.3072s	1920	96.7935ms
21	2	1	20	88	40.4636s	1344	131.228ms
22	1	1	21	88	67.5031s	960	203.219ms

Table 1: Benchmark results for various degrees and parameters (Version 1)

Note: The configuration with Degree , Advice , Lookup , Fixed , Lookup Bits , Limb Bits , Num Limbs is 17, 25, 3, 1, 16, 88, and 3, respectively.

Num Aggregation	Proof Time	Proof Size	Verify Time
2	17.4515s	11520	72.7713ms
200	22.3638s	15008	89.0890ms
2000	67.0237s	45952	286.869ms
4000	116.439s	79680	327.100ms
6000	165.035s	113760	551.478ms
8000	210.119s	147840	664.927ms
10000	256.993s	181920	841.507ms

Table 2: Benchmark results for varying aggregation sizes (Version 1)

2.2 Version 2: BLS with Merkle Tree and Hash of Message Verification

2.2.1 Data Generation

The data generation process involves: Generating a Merkle tree based on the specified depth d with random sks and pks. Recording the *pk* and *sk* values.

Note: Data generation time increases linearly with depth; thus, data is pre-generated and read from JSON files.

2.2.2 Code Description

The combined BLS signature and Merkle tree verification code is implemented in the `CombineBlsMtChip` struct. The overview is as following.

Input Parameters :

g1, **signatures**, **pubkeys** as the above.

message: An element of type F representing the message.

root: The root of the Merkle tree.

merkle_infos: Representing the Merkle tree information (leaf, path, and index).

Verification Process :

BLS Signature Verification: As the above, where the hash of message is generated by the original message by calculate its poseidon hash.

Merkle Tree Verification: The function verifies the Merkle tree by comparing the computed hash from the leaf to the root of the Merkle tree for each path in the *merkle_infos*.

Combining Results: The results from the BLS signature verification and the Merkle tree verification are combined using a logical AND operation. Additionally, the function ensures that the x coordinate of each public key matches the corresponding Merkle tree leaf.

2.2.3 Benchmark Results

The following table summarizes the benchmark results for different parameter settings:

Note: The configuration with *Degree, Advice, Lookup, Fixed, Lookup Bits, Limb Bits, Num Limbs* is fixed at 17, 25, 3, 1, 16, 88, and 3 respectively.

Num Aggregation	Num MT leaf	Proof Time	Proof Size	Verify Time
16	32	20.5646s	13152	80.7299ms
32	64	21.1587s	14528	74.7994ms
64	128	26.8527s	17984	106.079ms
128	256	38.0101s	25600	142.173ms
256	512	62.4086s	41632	223.739ms
512	1024	110.867s	76448	485.612ms
1024	2048	217.888s	150912	750.038ms
2048	4096	456.445s	310336	1.599s

Table 3: Benchmark results for various aggregation sizes (Degree = 17)

Note: The configuration with *Degree, Advice, Lookup, Fixed, Lookup Bits, Limb Bits, Num Limbs* is fixed at 19, 6, 1, 1, 18, 90, and 3 respectively.

Num Aggregation	Num MT leaf	Proof Time	Proof Size	Verify Time
16	32	21.2527s	3424	93.9743ms
32	64	23.7070s	4000	99.2152ms
64	128	26.5444s	4576	123.859ms
128	256	39.8745s	7008	173.010ms
256	512	59.3749s	10688	248.074ms
512	1024	102.120s	19104	490.098ms
1024	2048	199.642s	37664	878.392ms
2048	4096	415.307s	77312	1.491s

Table 4: Benchmark results for various aggregation sizes (Degree = 19)

Note: The configuration with *Degree, Advice, Lookup, Fixed, Lookup Bits, Limb Bits, Num Limbs* is fixed at 21, 2, 1, 1, 20, 88, and 3 respectively.

Num Aggregation	Num MT leaf	Proof Time	Proof Size	Verify Time
16	32	45.8947s	1696	196.018ms
32	64	46.6231s	1696	185.981ms
64	128	50.6344s	1920	202.661ms
128	256	57.9435s	2272	229.011ms
256	512	79.8219s	3424	356.260ms
512	1024	118.136s	5376	480.207ms
1024	2048	208.207s	9984	798.579ms
2048	4096	416.241s	19904	1.538s

Table 5: Benchmark results for various aggregation sizes (Degree = 21)

2.2.4 Benchmark Environment

OS	Ubuntu 20.04.6 LTS (x86_64)
Kernel	5.4.0-169-generic
CPU	Intel Xeon Silver 4214 (48 cores) @ 3.200GHz
GPU	4 x NVIDIA GeForce RTX 2080 Ti
Memory	128546 MiB