24.08.20 MuSig2, SpeedyMuSig and Shuffle Arguement

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

August 20 2024

1 MuSig2 Benchmark

We implemented the MuSig2 scheme, this SNARK-based multi-signature scheme prove the following statements:

1. $a_i = \mathrm{H}_{agg1}(L||X_i)$ for all X_i, where $L = (X_1, X_2, ..., X_n)$
2. $X = a_1 * X_1 + a_2 * X_2 + ... + a_n * X_n$
3. $R' = R'_1 + R'_2 + ... + R'_n$
4. $R'' = R''_1 + R''_2 + ... + R''_n$
5. $b = \mathrm{H}_{agg2}(X||R'||R''||m)$
6. $R_i = R'_i + bR''_i$ for all (R'_i, R''_i)
7. $R = R_1 + R_2 + ... + R_n$
8. $s = s_1 + s_2 + ... + s_n$
9. $s * G =?R + \mathrm{H}_{sig}(X, R, msg) * X$

Note that this is the whole workflow of MuSig2. The benchmark results are as follows:

Settings	Proof Time
k=14, n=6	1.3837s
k=15, n=15	2.5438s
k=16, n=30	4.8002s
k=17, n=60	8.6175s
k=18, n=120	17.392s

Table 1: Benchmark results for MuSig2

2 SpeedyMuSig Benchmark

We implemented the SpeedyMuSig scheme, this SNARK-based multi-signature scheme prove the following statements:

1. $X = \sum_{i=1}^{n} X_i$
2. $R' = R'_1 + R'_2 + ... + R'_n$
3. $R'' = R''_1 + R''_2 + ... + R''_n$
4. $b = \mathrm{H}_{agg2}(X||R'||R''||m)$
5. $R_i = R'_i + bR''_i$ for all (R'_i, R''_i)
6. $R = R_1 + R_2 + ... + R_n$
7. $s = s_1 + s_2 + ... + s_n$
8. $s * G =?R + \mathrm{H}_{sig}(X, R, msg) * X$

We assume that the aggregator has verified all the proof-of-possession signatures from signers. Thus this part of computation is not included in circuit.

Settings	Proof Time
k=13, n=6	0.7817s
k=14, n=15	1.3665s
k=15, n=30	2.4466s
k=16, n=60	4.5086s
k=17, n=120	8.3763s

Table 2: Benchmark results for SpeedyMusig

3 Comparison

We offer a quick comparison of three Schnorr multi-signature schemes:

	MuSig	MuSig2	SpeedyMuSig
Signature Scheme	Schnorr	Schnorr	Schnorr
Assumption	DL	AOMDL	AOMDL
Offline Round	1	1	1
Online Round	2	1	1
PK Aggregation	Heavy	Heavy	Simple

Table 3: MuSig Comparison

Due to the simple aggregation method of PKs, the SpeedyMuSig scheme is about 2x faster than other schemes in Halo2 proving.

4 Mithril Discussion

We offer a discussion about Mithril, about its public keys aggregation, security consideration and potential SNARK-version realization.

4.1 Public Keys Aggregation

The aggregation technology of Mithril is similar to MuSig and MuSig2. In the original paper:

- MSP.BKey($\mathbf{mvk}, \mathbf{e}_\sigma$):Takes a vector **mvk** of (previously checked) verification keys and weighting seed e_σ, and returns an intermediate aggregate public key $ivk = \prod mvk_i^{e_i}$, where $e_i \leftarrow \mathrm{H}(i, e_\sigma)$.
- MSP.BSig(σ):Takes as input a vector of signatures σ and returns (μ, e_σ) where $\mu \leftarrow \prod \sigma_i^{e_i}$, where $e_i \leftarrow \mathrm{H}(i, e_\sigma)$ and $e_\sigma \leftarrow \mathrm{H}(\sigma)$.

The paper also said that:

The MSP.BKey *and* MSP.BSig *aggregation functions enforce more stringent checking than that of standard multisignatures by utilizing the short random exponent batching of Bellare et al. The difference from standard multisignature aggregation, is that the randomized check will fail with overwhelming probability if any of the individual signatures is invalid, whereas the simpler aggregation allows for erroneous individual signatures if the aggregate is correct.*

4.2 Mithril and SpeedyMuSig

In SpeedyMuSig scheme, there is a same proof-of-possession process, just as same as Mithril. But SpeedyMuSig avoid the complex public keys aggregation method, replace it by a simple product of public keys.

The SpeedyMuSig paper claimed that they prove the EUF-CMA security of SpeedyMuSig in the programmable random oracle model under the one-more discrete logarithm assumption and the Schnorr knowledge of exponent assumption.

4.3 SNARK-based Mithril

we give a quick review of what we have done to SNARK-based Mithril:

1. $ivk = \prod mvk_i$
2. $\mu \leftarrow \prod_{i=1}^{d} \sigma_i$
3. $e(g1, \sigma) = e(ivk, \mathrm{H}(m))$
4. root $= \mathrm{H}(mvk, ...)$, and all $mvk_i \in (mvk, ...)$

Note that the original Mithril paper verify the BLS signature like: $e(\sigma, g2) = e(\mathrm{H}_{\mathbb{G}_1}(''M''||msg), ivk)$. Since it is a preliminary implementation, we can modify it later.

Our preliminary plan is to implement the following form of SNARK-based Mithril.

1. $ivk = \prod mvk_i^{e_i}$, where $e_i \leftarrow \mathrm{H}(i, e_\sigma)$.
2. $\mu \leftarrow \prod \sigma_i^{e_i}$, where $e_i \leftarrow \mathrm{H}(i, e_\sigma)$ and $e_\sigma \leftarrow \mathrm{H}(\sigma)$.
3. $e(g1, \mu) = e(ivk, \mathrm{H}(m))$
4. root $= \mathrm{H}((mvk, stake), ...)$, and all $(mvk_i, stake_i) \in ((mvk, stake), ...)$

This could be challenging. And we are still searching for better solution of zero-knowledge bridge for Cardano.

5 Shuffle Arguement Discussion

5.1 BG12[1]

Common Reference String: pk, ck.

Statement: $\mathcal{C}, \mathcal{C}' \in \mathbb{H}^N$ with $N = mn$.

Witness: The prover possesses a permutation $\pi \in \Sigma_N$ and randomness $\rho \in \mathbb{Z}_q^N$ such that $C' = \mathcal{E}_{\mathsf{pk}}(1; \rho)C_\pi$.

Proof Phases:

1. P: Commit to $\mathbf{a} = \{\pi(i)\}_{i=1}^N$.
2. V: Pick $x \leftarrow \mathbb{Z}_q^*$ as the challenge.
3. P: Commit to $\mathbf{b} = \{x^{\pi(i)}\}_{i=1}^N$.
4. V: Pick $y, z \leftarrow \mathbb{Z}_q^*$ as the challenge.
5. P: Compute and commit to $\mathbf{d} = y\mathbf{a} + \mathbf{b}$, $-\mathbf{z} = (-z, -z, ..., -z, \mathbf{0})$
   Compute $\rho = -\rho \cdot \mathbf{b}$ and set $\mathbf{x} = (x, x^2, \ldots, x^N)^T$.

Verification: The verifier checks the commits to a and b ,$\mathbf{c}_a, \mathbf{c}_b \in \mathbb{G}^m$ and computes $\mathbf{c}_{-z}$ and $\mathbf{c}_d$ along with $\mathbf{C}^{\mathbf{x}}$ and $\prod_{i=1}^N (yi + x^i - z)$.

Engage in a product argument for the openings of $\mathbf{d} - \mathbf{z}$, and verify the equality

$$\prod_{i=1}^N (d_i - z) = \prod_{i=1}^N (yi + x^i - z).$$

[1] Bayer, S., Groth, J.:Efficient Zero-Knowledge Argument for Correctness of a Shuffle. In:EUROCRYPT 2012.

Engage in a multi-exponentiation argument of $\mathbf{b}$ and ρ such that:

$$\mathbf{C}^{\mathbf{x}} = \mathcal{E}_{\mathsf{pk}}(1;\rho)\mathbf{C'}^{\mathbf{b}}$$

The verifier accepts if the product and multi-exponentiation arguments are both valid.

5.2 A Naive Approach

For cases where it is unnecessary to hide the ciphertext and the permutation π, a simplified method can be employed: treat the vector C' as a permutation of the vector C. Introduce a challenge value z and then compute the product of each element of C and C' after subtracting z. If the results of the two products are equal, it indicates that C' is indeed a permutation of C.

The mathematical description is as follows:

1. Compute $\prod_{i=1}^{n}(c_i - z)$.
2. Compute $\prod_{i=1}^{n}(c'_i - z)$.

If the following equality holds:

$$\prod_{i=1}^{n}(c_i - z) = \prod_{i=1}^{n}(c'_i - z)$$

then C' is a permutation of C. There's no need the possession of the permutation π.

This approach essentially constructs polynomials with respect to z from the two vectors. According to Schwartz-Zippel lemma, the prover has negligible chance over the choice of z of making a convincing argument unless indeed there is a permutation π.

6 Halo2-lib Benchmark

6.1 Bug Discussion

Figure 1: Use of TimerInfo

The issue in the code snippet is related to the end_timer! macro. The problem is that end_timer! does not actually stop the timer immediately when it is called. Instead, it stops the timer only when the elapsed function is subsequently invoked. This causes a cumulative effect on the recorded time, leading to an inaccurate total duration. Specifically, the time keeps accumulating until elapsed is called, which results in the final timing measurement including time that should not be accounted for.

```
        writeln!(
            fs_results,
            "{},{},{},{},{},{},{},{},{:?},{},{:?}",
            bench_params.degree,
            bench_params.num_advice,
            bench_params.num_lookup_advice,
            bench_params.num_fixed,
            bench_params.lookup_bits,
            bench_params.limb_bits,
            bench_params.num_limbs,
            bench_params.num_aggregation,
            stats.proof_time.time.elapsed(),
            stats.proof_size,
            stats.verify_time.time.elapsed()
        )?;
```

Figure 2: When the Timer stop

6.2 New Test Results

degree	num_aggregation	num_origin	proof_time	proof_size	verify_time
17	256	512	65.107423254s	41632	22.065098ms
17	512	1024	111.976211587s	76448	34.824188ms
17	1024	2048	223.437017328s	150912	67.46092ms
17	2048	4096	470.005966428s	310336	124.343528ms
19	256	512	61.625703969s	10688	12.830639ms
19	512	1024	104.553267256s	19104	14.696026ms
19	1024	2048	204.119754992s	37664	21.397588ms
19	2048	4096	439.596726985s	77312	41.115937ms
21	256	512	81.647671708s	3424	12.611693ms
21	512	1024	120.595740985s	5376	19.456223ms
21	1024	2048	214.113378674s	9984	19.698181ms
21	2048	4096	440.551034029s	19904	19.534606ms