25.01.14 ATMS with Product of All Keys

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

January 14 2025

1 ATMS with Product

First we introduce the original version of a multisignature-based t-ATMS construction:

Given d individual signatures $\sigma_1, \ldots, \sigma_d$, created using secret keys belonging to (not necessarily unique) public keys $vk_1, \ldots, vk_d$ can be combined into a multisignature $\sigma = \prod_{i=1}^{d} \sigma_i$ that can then be verified using an aggregated public key $avk = \prod_{i=1}^{d} vk_i$.

Assuming S is a set and $\langle S \rangle$ is a a Merkle-tree commitment to the set. There are some functions:

- AKey: given a sequence of public keys $\mathcal{VK} = \{vk_i\}_{i=1}^{n}$, returns $avk = (\prod_{i=1}^{d} vk_i, \langle \mathcal{VK} \rangle)$.
- $\mathsf{ACheck}(\mathcal{VK}, avk)$: simply recomputes it to verify avk.
- ASig: takes the message m, d pairs of signatures with their respective public keys $\{\sigma_i, vk_i\}_{i=1}^{d}$ and $n-d$ additional public keys $\{\widehat{vk_i}\}_{i=1}^{n-d}$ and produces an aggregate signature:

$$\sigma = \left(\prod_{i=1}^{d} \sigma_i, \{\widehat{vk_i}\}_{i=1}^{n-d}, \{\pi_{\widehat{vk_i}}\}_{i=1}^{n-d} \right)$$

  where $\{\pi_{\widehat{vk_i}}\}_{i=1}^{n-d}$ denotes the (unique) inclusion proof of $\{\widehat{vk_i}\}_{i=1}^{n-d}$ in the Merkle commitment.

- AVer: takes a message m, an aggregate key avk, and an aggregate signature σ parsed as above and does the following:
  1. verifies that each of the public keys $\widehat{vk_i}$ indeed belongs to a different leaf in the commitment $\langle \mathcal{VK} \rangle$ in avk using membership proofs $\pi_{\widehat{vk_i}}$.
  2. computes avk' by dividing the first part of avk by $\prod_{i=1}^{n-d} \widehat{vk_i}$.
  3. returns true if and only if $d \geq t$ and the first part of σ verifies as a Π_{MGS}-signature under avk'

2 Relations

In a SNARK version of ATMS with product(original ATMS), we have a avk as the root of a Merkle tree containing $\mathcal{VK}$. And let $S' = \{s_i\}$ be the signatures generated by a sequence $\mathcal{VK}'$ containing keys in $\mathcal{VK}$. The signing algorithm reconstructs the Merkle tree from $\mathcal{VK}$ and determines the membership proof π_i for each $vk_i \in \mathcal{VK}'$.

The the statements of SNARK is $x = (avk, m)$, and the witness of SNARK is $w = \{\pi_i, (s_i, vk_i)\}_{i \in S'}$. The SNARK prove the following relations:

- For all i, the signature verifying algorithm $\mathsf{Ver}(vk_i, m, s_i)$ returns true.
- For all i, proof π_i is a valid Merkle path proof pointing to a unique leaf.
- $|S'| \geq t$.

This is very similar to the code implemented by Inigo, but the original code replace the Merkle tree by a straight hash.

3 SNARK-based ATMS Implementation

According to the relations mentioned in the paper, the code implementation is as following:

```
let mut flattened_pks: Vec<AssignedCell<Scalar, …>> = Vec::new();
for pk: &AssignedEccPoint in pks {
    flattened_pks.push(pk.x.clone());
}

let hashed_pks: AssignedCell<Scalar, Scalar> = self &AtmsVerifier…
    .schnorr_gate SchnorrVerifierGate
    .rescue_hash_gate RescueCrhfGate<Scalar, RescueParametersBls>
    .hash(ctx, input: &flattened_pks)?;
```

Compute a straight hash of all public keys firstly, and compare it to the public commitment.

Then, just as described in the paper, we verify the individual signature one by one. And if there exits a signature corresponding to a public keys, which can also be verified, we add the counter(to compute the threshold).

```
let mut counter: AssignedCell<Scalar, Scalar> = self &AtmsVerifier…
    .schnorr_gate SchnorrVerifierGate
    .ecc_gate EccChip
    .main_gate MainGate<Scalar>
    .assign_constant(ctx, constant: Base::ZERO)?;

for (sig: &Option<(AssignedEccPoint, …)>, pk: &Assigned…) in signatures.iter().zip(pks.iter()) {
    if let Some(signature: &(AssignedEccPoint, ScalarVar)) = sig {
        self.schnorr_gate.verify(ctx, signature, pk, msg)?;
        counter = self.schnorr_gate.ecc_gate.main_gate.add_constant(
            ctx,
            a: &counter,
            constant: Base::one(),
        )?;
    }
}
```

Finally we constraint the counter to be equal with threshold.

```
self.schnorr_gate SchnorrVerifierGate
    .ecc_gate EccChip
    .main_gate MainGate<Scalar>
    .assert_equal(ctx, a: &counter, b: threshold)?;
```

4 New Approach

If we use the settings in the ATMS paper, which means that there do exit a Merkle tree that commits $\mathcal{VK}$. And we need a product of all public keys $ivk = \prod_{i=1}^{n} vk_i$ at first. Now, let the avk be aggregated public key $avk = \prod_{i=1}^{d} vk_i$, and $\sigma = \prod_{i=1}^{n} \sigma_i$ be the aggregated signature.

Then the The the statements of SNARK is $x = (avk, ivk, m, \sigma)$, and the witness of SNARK is $w = \{\{\pi_i, vk_i\}_{i \notin S'}, \{\sigma_i\}_{i \in S'}\}$. The SNARK prove the following relations:

- $avk = ivk - \prod_{i=1}^{n-d} vk_i$.
- For all i, proof π_i is a valid Merkle path proof pointing to a unique leaf.
- $\sigma = \prod_{i=1}^{d} \sigma_i$.
- The signature verifying algorithm $\mathsf{Ver}(avk, m, \sigma)$ returns true.

This is a normal multisignature-like proving approach.

5 Advantage

The SNARK-based ATMS implementation use Schnorr signature as its core signature scheme, but when we turn to **BLS** signature, the proving cost will increase rapidly.

Assuming the threshold is d, the original approach in paper need to prove(in BLS setting):

- d BLS signature verification.

Which means it need to prove d times of pairing check, it is very costly.
In our approach, we need to prove:

- Product of unused keys, signatures and the result of avk.
- $n - d$ times of Merkle path verification.
- 1 BLS signature verification.

The main cost is Merkle path verification, it may can be optimized by our shuffle argument.

6 Test Result

Below are our test results. We continued using the same parameter settings as before, meaning we signed with half of the total number of keys.

num	**proof_time**	**proof_size**	**verify_time**
16 of 32	29.24s	13152	15.07ms
32 of 64	30.75s	14528	12.93ms
64 of 128	36.50s	17984	16.33ms
128 of 256	51.53s	25600	16.60ms
256 of 512	82.75s	41856	23.04ms
512 of 1024	147.84s	76672	35.33ms
1024 of 2048	289.60s	151488	56.85ms

Currently, the main bottleneck lies in the hash verification process used for Merkle tree validation. However, we need to adjust our parameters. Under the actual ATMS configuration, about two-thirds of the keys would be used for signing. Because the Merkle paths to be verified correspond to keys that are not signed, the real-world verification time will be lower than the current test results. So we need further testing.

The figure below shows that, under the current parameter settings, both the proving time and the verification time increase linearly with the number of keys.