24.05.21 SNARK-based ATMS

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

May 21 2024

1 SNARK-based ATMS

1.1 Comparison between Mithril and ATMS

Use Code from Inigo.

We provide a quick comparison to see what are the differences and Similarities between Mithril and implemented ATMS(SNARK version).

	Mithril	ATMS(SNARK version)
core signature scheme	BLS	Schnorr
threshold	stake sum	valid signature num
VK commitment	Merkle tree	Hash
hash function	blake2b	Rescue
eligibility check	Yes	No
curve	bls12-381	bls12-381

Table 1: Mithril and ATMS Comparison

1.2 Workflow of SNARK-based ATMS

Assume that there exists n committee members, and the required threshold is t.

Step 1. Each individual signer proceeds the [$keygen$] The function generates (sk_i, pk_i) as the keypair for the $signer_i$.

Step 2. Signers share their public keys with the registration authority:

- The role of the registration authority is simply to commit to all public keys of the committee in a Merkle Tree (MT).
- The Registration Authority can be a Plutus script, a trusted party, or be distributed amongst the committee members.
- The reason why it needs to be 'trusted' is because it can exclude certain participants, or include several keys it owns.

Step 3. Once all registration requests have been submitted with their corresponding public keys, $pks = [pk_1, ..., pk_n]$, the aggregated public key is created $avk = H(pk_1, \ldots, pk_n)$.

Step 4. Individual parties generate their single signature with $[sign]$ and send the signature to aggregator (does not need to be trusted). Individual signatures should be verifiable with $[verify]$

Step 5. Aggregator receives the single signatures. It collects at least threshold-many valid signatures as the aggregate signature.

Step 6. Once the aggregator receives at least t valid signatures $sig_1, ..., sig_t$ it proceeds to generate the SNARK. In particular, it proves that:

- There exists t' valid and distinct signatures, $sig_1, ..., sig_t$ for public keys $pk_1, ..., pk_t$ and message msg.
- The hash of all t public keys, together with some other set of keys, results in the corresponding avk.

1.3 Benchmark of SNARK-based ATMS

We bench the SNARK-based ATMS by Criterion crate.

k represents the degree of polynomial(which means the number of rows is about 2^k). n represents the number of parties. And th represents the threshold of signatures.

Setting	Proving Time
k = 14, n = 6, th = 3	1.2579s
k = 15, n = 9, th = 6	2.3281s
k = 15, n = 9, th = 8	2.3310s
k = 16, n = 15, th = 15	4.3034s
k = 16, n = 21, th = 14	4.3301s
k = 17, n = 21, th = 17	8.0298s
k = 17, n = 42, th = 28	8.1542s

Table 2: Proving time of ATMS

Setting	Proving Time
k = 19, n = 102, th = 72	28.761s
k = 22, n = 2001, th = 1602	$\approx$ 200s

Table 3: Proving Time of Real Situations

Note that the benchmark of last setting(k = 22, n = 2001, th = 1602) is computed by hand, since my computer does not support the long running time of function.

2 Rescue and Poseidon

Since Mithril is implemented with Poseidon and SNARK-based ATMS is on Rescue, it is necessary to compare Poseidon and Rescue.

The Poseidon functions implemented in halo2 and halo2-lib are both **Poseidon128**, which means it has a **128-bit** security level.

We provide a quick comparison between Poseidon128 and Rescue in implementation:

	Poseidon128	Rescue
security level	128bit	128bit
S-box	x^5	$x^5 \& x^{1/5}$
Support curve	BN/BLS/Ed	BN/BLS/Ed
Width	3,9,12	4
Capacity	1	1
R_f	8	12
R_p	57	-

Table 4: Comparison of Poseidon128 and Rescue

2.1 Performance

The performance we benched are in quite different settings, is for reference only.

2.2 Poseidon Benchmark

The field of halo2 official Poseidon is **pallas/vesta**.

And the **rate = width - 1**. All the results only do **one-time** permutation, because the input length is strictly equals to the rate.

It should be noted that the halo2 official implementation used a Blake2b transcript and run a real workflow(including generating PK/VK, proving and verification). So the time it cost may be longer than Rescue.

	Proving	Verification
width = 3	58.371 ms	3.3824 ms
width = 9	106.29 ms	3.7644 ms
width = 12	139.08 ms	3.9400 ms

Table 5: Halo2 Official Poseidon Benchmark

The halo2-lib crate uses a optimized Poseidon implementation described in Supplementary Material Section B of https://eprint.iacr.org/2019/458.pdf, aka Poseidon paper(full version). This involves some further computation of optimized constants and sparse MDS matrices beyond what the Scroll PoseidonSpec generates.

The rate is fixed to **2**, and the capacity is fixed to **1**. Which means that the arity(in the context of tree hashing) of this function is **2**.

And the field of halo2-lib Poseidon is **BN254**.

	Generate VK	Generate PK	Proving	Verification
maxlen = 0	59.107ms	12.434ms	162.234ms	2.584ms
maxlen = 2*2	81.841ms	21.711ms	293.331ms	3.925ms
maxlen = 2*5	127.393ms	25.423ms	370.451ms	4.608ms
maxlen = 2*2+1	94.544ms	24.769ms	312.785ms	4.069ms
maxlen = 2*5+1	125.951ms	41.236ms	349.318ms	4.319ms

Table 6: Halo2-lib Poseidon Benchmark

The halo2-lib crate uses a **base_test**().**k**().**bench_builder**() method to bench the function. It runs keygen, real prover, and verifier by providing a closure that uses a 'builder' and 'RangeChip'.

We can do very rough calculation to see the cost of one permutation under the setting **rate = 2**.

The proving time of a permutation is about:

- $(370.451ms - 293.331ms)/(5 - 2) = 25.7066ms$ or
- $(312.785ms - 293.331ms)/1 = 19.454ms$

2.3 Rescue Benchmark

The benchmark of Rescue is running under follwing settings:

- **Curve**: BLS12-381
- **RATE**: 3

Note: this benchmark is using **MockProver** :: **run** method, with parameter **k = 10**. Thus the real proving time of Rescue hash function will be **much lower**.

iteration	width	operation	total time	time per permutation
4	4	12-to-3	36.269ms	2.2668ms

Table 7: Rescue Hash Benchmark

This benchmark uses **12** BLS12-381 scalar filed element as in put , and get a output element sequence of length **3**. Of which the first element is the hash result.

A more detailed description is, the benched function "absorbs" the input elements **12** times, and "squeezes" an output of length **3**.

Since the rate of Rescue is 3, take **3** field element as a input group, and straightly do a addition operation on **state**. That means the benched function do **4** permutations in total.

3 Discussion

1. The SNARK-based ATMS uses Jubjub for in-circuit elliptic curve operations since it provides efficient EC operations within the proof. Jubjub is s an elliptic curve of the twisted Edward's form:

   $$E_d : -u^2 + v^2 = 1 + du^2v^2.$$

   Define the Jubjub curve over the field $\mathbb{F}_q$ where q is represented in hexadecimal as follows:

   $$q = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001$$

   seSet the Jubjub curve as the embedded curve of BLS12-381. Meaning that, Jubjub curve is defined over a prime which is also the prime that defines the scalar field of BLS12-381.

2. Since the we only want to get a commitment of public keys, there is "no need to of a merkle tree inside a SNARK, with a hash it is sufficient". So basically, $avk = H(pk_1, pk_2, ..., pk_n)$.

3. A little problem: The proof must include only threshold-many valid signatures even if the prover has more valid signatures.