24.08.27 Settings of MuSig, Mithril and Lookup Argument

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

August 27 2024

1 Aggregator Node

We first introduce the setting of aggregator node in MuSig2 [NRS21]. And it is necessary to clarify the syntax of the two-round multi-signature scheme. The following contents is from original paper.

It consists of algorithms

$$(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{KeyAgg}, (\mathsf{Sign}, \mathsf{SignAgg}, \mathsf{Sign}', \mathsf{SignAgg}', \mathsf{Sign}''), \mathsf{Ver})$$

as follows. System-wide parameters *par* are generated by the setup algorithm Setup taking as input the security parameter. For notational simplicity, we assume that *par* is given as implicit input to all other algorithms. The randomized key generation algorithm takes no input and returns a secret/public key pair $(sk, pk) \leftarrow \mathsf{KeyGen}()$. The deterministic key aggregation algorithm KeyAgg takes a multiset of public keys $L = \{pk_1, ..., pk_n\}$ and returns an aggregate public key: $\tilde{pk} = \mathsf{KeyAgg}(\{pk_1, ..., pk_n\})$. The interactive signature algorithm $(\mathsf{Sign}, \mathsf{SignAgg}, \mathsf{Sign}', \mathsf{SignAgg}', \mathsf{Sign}')$is run by each signer i and proceeds in a sequence of two communication rounds.

For instance, from the point of view of signer 1, a full signing run proceeds as follows:

$$(out_1, state_1) \leftarrow \mathsf{Sign}()$$

$$out := \mathsf{SignAgg}(out_1, ..., out_n)$$

$$(out'_1, state'_1) \leftarrow \mathsf{Sign}'(state_1, out, sk_1, m, \{pk_2, ..., pk_n\})$$

$$out' := \mathsf{SignAgg}'(out'_1, ..., out'_n)$$

$$\sigma \leftarrow \mathsf{Sign}''(state'_1, out')$$

The purpose of the aggregation algorithms $\mathsf{SignAgg}$ and $\mathsf{SignAgg}'$ is to enable savings in the broadcast communication in both signing rounds: An aggregator

node [SS01; KG20], which will be untrusted in our security model and can for instance be one of the signers, can collect the outputs of all signers in both rounds, aggregate the outputs using $\mathsf{SignAgg}$ and $\mathsf{SignAgg}'$, respectively, and broadcast only the aggregate output back to all signers. This optimization is entirely optional. If it is not desired, each signer can simply broadcast its outputs directly to all signers, which then all run $\mathsf{SignAgg}$ and $\mathsf{SignAgg}'$ by themselves.

2 Mapping in Mithril

In Mithril, the relation we need to prove includes:

- For $i = 1..k : \mathsf{MSP.Eval}(msg, \mathsf{index}_i, \sigma_i) = ev_i$
- For $i = 1..k : ev_i \leq \phi(\mathsf{stake}_i)$

There are two constructions of mapping: one is based on Bulletproofs and the other based on releasing the witness.

The Bulletproofs construction uses Elligator Squared, which avoids oracle calls on user-specific data i.e. we explicitly avoid hashing σ to sidestep soundness issues in circuit-based proofs.

For the concatenation proof system(the one just releases the witness), use a random oracle $H : 0, 1* \to \mathbb{Z}_p$ to implement the mapping as: $M^R_{msg,index}(\sigma) := H(\text{"}map\text{"}||msg||\mathsf{index}||\sigma)$.

In Mithril official code, they use Blake2b hash function to map the information:

```
/// Dense mapping function indexed by the index to be evaluated.
/// We hash the signature to produce a 64 bytes integer.
/// The return value of this function refers to
/// `ev = H("map" || msg || index || σ) <- MSP.Eval(msg,index,σ)` given in paper.
pub fn eval(&self, msg: &[u8], index: Index) -> [u8; 64] {
    let hasher = Blake2b512::new()
        .chain_update(b"map")
        .chain_update(msg)
        .chain_update(index.to_le_bytes())
        .chain_update(self.to_bytes())
        .finalize();

    let mut output = [0u8; 64];
    output.copy_from_slice(hasher.as_slice());

    output
}
```

And this value will be checked in the ϕ funtion:

```rust
#[cfg(not(feature = "num-integer-backend"))]
/// The crate `rug` has sufficient optimizations to not require a taylor approximation with early
/// stop. The difference between the current implementation and the one using the optimization
/// above is around 10% faster. We perform the computations with 117 significant bits of
/// precision, since this is enough to represent the fraction of a single lovelace. We have that
/// 1e6 lovelace equals 1 ada, and there is 45 billion ada in circulation. Meaning there are
/// 4.5e16 lovelace, so 1e-17 is sufficient to represent fractions of the stake distribution. In
/// order to keep the error in the 1e-17 range, we need to carry out the computations with 34
/// decimal digits (in order to represent the 4.5e16 ada without any rounding errors, we need
/// double that precision).
pub(crate) fn ev_lt_phi(phi_f: f64, ev: [u8; 64], stake: Stake, total_stake: Stake) -> bool {
    use rug::{integer::Order, ops::Pow, Float};

    // If phi_f = 1, then we automatically break with true
    if (phi_f - 1.0).abs() < f64::EPSILON {
        return true;
    }
    let ev = rug::Integer::from_digits(&ev, Order::LsfLe);
    let ev_max: Float = Float::with_val(117, 2).pow(512);
    let q = ev / ev_max;

    let w = Float::with_val(117, stake) / Float::with_val(117, total_stake);
    let phi = Float::with_val(117, 1.0) - Float::with_val(117, 1.0 - phi_f).pow(w);

    q < phi
}
```

So if we choose to write a circuit of Blake2b hash function, it will be heavy. Or we replace the Blake2b with Poseidon hash function, just as we did in merkle tree.

So now let's estimate the number of constraints. Assume there are k valid signers, each one will produce a valid BLS signature, and each signer offers a won lottery index. The cost will be:

- k Rescue hash functions for e_v. Since we use merkle tree with rate $= 3$, it will be $2k$ Rescue permuitation.
- k ϕ function evaluations.
- k Comparisons between e_v and $\phi(stake)$.

The actual constraints number needs further evaluation.

If we choose the approach in original paper, the cost will be:

- k Mapping evaluations for e_v(use Elligator Squared).
- k ϕ function evaluations.
- k Comparisons between e_v and $\phi(stake)$.

And the paper also give the constraints number:

- $3k \log q + 60k$ for representation function evaluations.

Because Mapping representations involve 60 constraints plus 3 range checks. One is for comparison between e_v and $\phi(stake)$ and other two for range checks for index.

And there is also a optimization for $\phi(stake)$, you can see that there is no constraints for ϕ evaluation. The following contents is from Mithril paper:

The main outlier is the evaluation of ϕ. Fortunately, we don't actually need to evaluate ϕ in the proof: we can replace stake in the tree with $\phi(stake)$ and proceed with the comparison directly. This gives us a circuit size of $O(klogq)$, and verifier complexity of $O(\frac{klog^4q}{log(klogq)})$ as verification is dominated by a multi-exponentiation based on the circuit size.

3 SNARK-based Mithril Implementation

We met a problem when implementing the following relations:

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 $e_\sigma \leftarrow \mathrm{H}(\sigma)$.

We firstly compute the weighting seed e_σ and the coefficient e_i according to the indices:

```
54      // B
55      let signatures_x_assigned: Vec<{unknown}> = signatures.iter().map(|pt: &G2Affine| {
56          ctx.load_witness(F::from_bytes_le(&pt.x.c0.to_bytes()))
57      }).collect::<Vec<_>>();
58      let gate_chip = GateChip::<F>::default();
59      let weighting_seed_comp = self.poseidon_chip.hash_fix_len_array(ctx, &gate_chip, &signatures_x_assigned[..]);
60      let weighting_seed_assigned = ctx.load_witness(weighting_seed);
61      // B_1 : verify weighting seed
62      let verify_B_1 = gate_chip.is_equal(ctx, weighting_seed_assigned, weighting_seed_comp);
63      // e_i = H(i,weighting_seed) for i in 0..n where n is the number of public keys
64      let e_is: Vec<AssignedValue<F>>  = pubkeys.iter().enumerate().map(|(i: usize, _)| {
65          let i_assigned = ctx.load_witness(F::from(i as u64));
66          self.poseidon_chip.hash_fix_len_array(ctx, &gate_chip, &[i_assigned, weighting_seed_assigned])
67      }).collect::<Vec<_>>();
```

And use the e_i coefficient to combine all the public keys and signatures:

```
halo2-lib > halo2-ecc > src > bn254 > msp.rs > {} impl MspChip<'chip, F> > msp_verify
25  impl<'chip, F: BigPrimeField> MspChip<'chip, F> {
38      pub fn msp_verify(
68
69          let g1_chip = EccChip::new(self.bls_signature_chip.fp_chip);
70          let fp2_chip = Fp2Chip::new(self.bls_signature_chip.fp_chip);
71          let g2_chip = EccChip::new(&fp2_chip);
72          // B_2 : verify ivk, isig
73          // ivk = \sum_{i=0}^{n-1} e_i * mvk_i
74          let ivk_assigned = self.bls_signature_chip.pairing_chip.load_private_g1(ctx, ivk);
75          let mvks = pubkeys.iter().map(|pt| self.bls_signature_chip.pairing_chip.load_private_g1(ctx, *pt)).collect::<Vec<_>>();
76          let products = mvks.iter().zip(e_is.iter()).map(|(mvk, e_i)| {
77              g1_chip.scalar_mult(ctx, mvk.clone(), e_is.clone(),64,12)
78          }).collect::<Vec<_>>();
79          let ivk_comp = g1_chip.sum(ctx, products);
80          let verify_B_2 = g1_chip.is_equal(ctx, ivk_assigned, ivk_comp);
81          // isig = \sum_{i=0}^{n-1} e_i * sig_i
82          let isig_assigned = self.bls_signature_chip.pairing_chip.load_private_g2(ctx, isig);
83          let sigs = signatures.iter().map(|pt| self.bls_signature_chip.pairing_chip.load_private_g2(ctx, *pt)).collect::<Vec<_>>();
84          let products = sigs.iter().zip(e_is.iter()).map(|(sig, e_i)| {
85              g2_chip.scalar_mult(ctx, sig.clone(), e_is.clone(),64,12)
86          }).collect::<Vec<_>>();
87          let isig_comp = g2_chip.sum(ctx, products);
88
89          let verify_B_3 = g2_chip.is_equal(ctx, isig_assigned, isig_comp);
```

There is an error when we try to make a scalar multiplication between σ and coefficient e_i. Maybe it's because e_i is a $\mathbb{G}_1$ scalar value, and it cannot be multiplied by a $\mathbb{G}_2$ element directly. It is a problem remains to be solved.

4 Shuffle Argument

The public keys(**pk**) we use are points on an elliptic curve. We can compress a point on the elliptic curve.

For any finite point $P = (x_P, y_P)$ on the elliptic curve, this point can be succinctly represented by storing only the x-coordinate x_P and a specific bit derived from x_P and y_P, known as the compressed representation of the point.

Specifically, let $P = (x_P, y_P)$ be a point on the elliptic curve $E : y^2 = x^3 + ax + b$ defined over $\mathbb{F}_p$. If y_{rb} is the rightmost bit of y_P, then the point P can be represented by x_P and y_{rb}.

According to our previous approach, for shuffling **pk** , we used the following strategy. A compressed representation of a **pk** in the set is denoted as (pk_x, pk_{yb}). We treat the vector PK' as a shuffle of the vector PK, introducing two challenge values γ and ζ. We compute:

1. $\prod_{i=1}^{n}(pk_x + \gamma \cdot pk_{yb} + \zeta)$.
2. $\prod_{i=1}^{n}(pk'_x + \gamma \cdot pk'_{yb} + \zeta)$.

If the following equality holds:

$$\prod_{i=1}^{n}(pk_x + \gamma \cdot pk_{yb} + \zeta) = \prod_{i=1}^{n}(pk'_x + \gamma \cdot pk'_{yb} + \zeta)$$

then PK' is a permutation of PK.

4.1 Cost Discussion for Different Scheme Combinations

Consider a set of N elements in the PK collection, from which we select m elements for signature aggregation:

When we use Merkle path for verification, the cost is $O(m \log N)$.

When we use shuffle arguments for verification, the cost is $O(N)$. The cost for proving the Merkle root of the PK set is $2N$, so the total cost is $O(N)$.

In this context, if m is linear in N, then the cost of the second scheme is smaller. However, if m is sub-linear or smaller, then the cost of the first scheme is lower.