25.02.18 Improving MSP.B

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

February 18 2025

1 MSP.B in Mithril

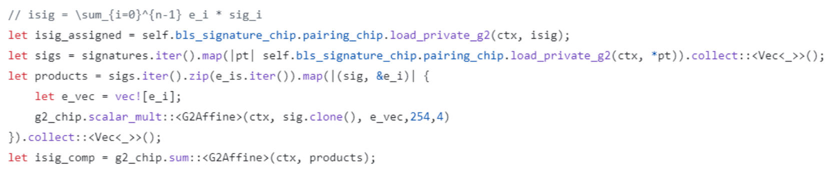
First of all, we found a new way(actually a new function) to implement the MSP.B relations in Mithril. The MSP.B is as following:

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

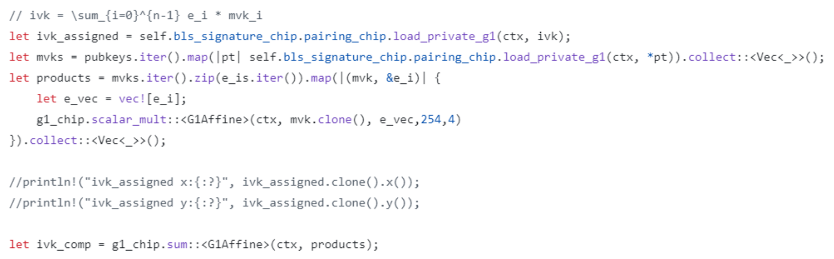
And we also verify the MSP.B multisignature μ using aggregated verification key ivk in the circuit.

2 Original Implementation and Benchmark Results

For the original implementation of MSP.B, we use a ”scalar_mult” function to aggregate the individual signatures and pubilc keys. The aggregation relations is as following:



```
// isig = \sum_{i=0}^{n-1} e_i * sig_i
let isig_assigned = self.bls_signature_chip.pairing_chip.load_private_g2(ctx, isig);
let sigs = signatures.iter().map(|pt| self.bls_signature_chip.pairing_chip.load_private_g2(ctx, *pt)).collect::<Vec<_>>();
let products = sigs.iter().zip(e_is.iter()).map(|(sig, &e_i)| {
    let e_vec = vec![e_i];
    g2_chip.scalar_mult::<G2Affine>(ctx, sig.clone(), e_vec,254,4)
}).collect::<Vec<_>>();
let isig_comp = g2_chip.sum::<G2Affine>(ctx, products);
```

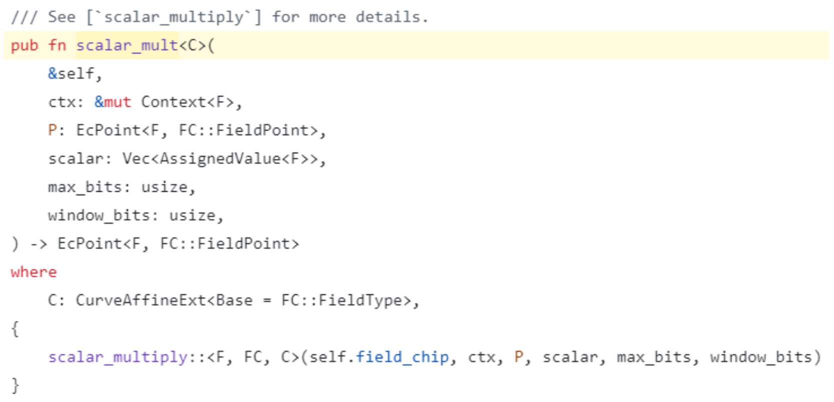


```
// ivk = \sum_{i=0}^{n-1} e_i * mvk_i
let ivk_assigned = self.bls_signature_chip.pairing_chip.load_private_g1(ctx, ivk);
let mvks = pubkeys.iter().map(|pt| self.bls_signature_chip.pairing_chip.load_private_g1(ctx, *pt)).collect::<Vec<_>>();
let products = mvks.iter().zip(e_is.iter()).map(|(mvk, &e_i)| {
    let e_vec = vec![e_i];
    g1_chip.scalar_mult::<G1Affine>(ctx, mvk.clone(), e_vec,254,4)
}).collect::<Vec<_>>();

//println!("ivk_assigned x:{:?}", ivk_assigned.clone().x());
//println!("ivk_assigned y:{:?}", ivk_assigned.clone().y());

let ivk_comp = g1_chip.sum::<G1Affine>(ctx, products);
```

We can swap the group if needed. And the "scalar_mult" function is like:



```
/// See [`scalar_multiply`] for more details.
pub fn scalar_mult<C>(
    &self,
    ctx: &mut Context<F>,
    P: EcPoint<F, FC::FieldPoint>,
    scalar: Vec<AssignedValue<F>>,
    max_bits: usize,
    window_bits: usize,
) -> EcPoint<F, FC::FieldPoint>
where
    C: CurveAffineExt<Base = FC::FieldType>,
{
    scalar_multiply::<F, FC, C>(self.field_chip, ctx, P, scalar, max_bits, window_bits)
}
```

However, this function is not so efficient, because we need to do a MSM(multi-scalar-multiplication) to every group element. And finally compute the sum of all group elements. But in the relations, one group element is only multiplied by one scalar.

So "e_vec" vector in the code only include one scalar variable, and we need to do n times MSM in the circuit to Both $\mathbb{G}_1$ and $\mathbb{G}_2$ element. This approach caused significant proving cost.

Below is our former benchmark results:

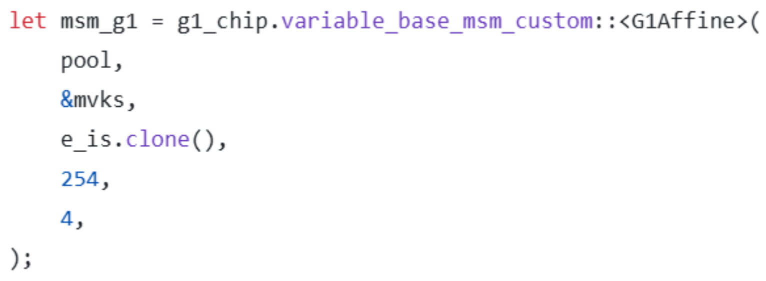
Note: The configuration with Degree , Advice , Lookup Bits , Limb Bits , Num Limbs is 19, 6, 18, 90, and 3, respectively.

num_aggregation	proving_time	verification_time
4	70.5080s	14.4204ms
8	100.3936s	17.5523ms
16	145.7191s	18.6467ms
32	251.0654s	21.7318ms
64	471.5117s	30.2121ms

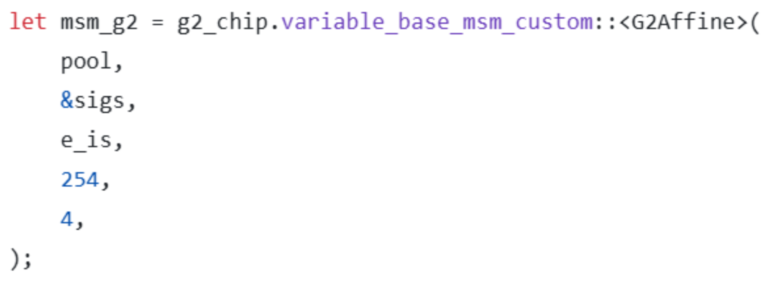
When attempting to sign with 128 public keys, the server's 126 GB of RAM and 8 GB swap partition become full, leading to the program being killed.

3 Improved MSP.B Implementation

We conducted a comprehensive inspection of the “halo2-lib” library in response to the previous issues. And we choose a new (implemented) function ”variable_base_msm_custom” to complete our aggregation phase. The core code is as following:



```
let msm_g1 = g1_chip.variable_base_msm_custom::<G1Affine>(
    pool,
    &mvks,
    e_is.clone(),
    254,
    4,
);
```



```
let msm_g2 = g2_chip.variable_base_msm_custom::<G2Affine>(
    pool,
    &sigs,
    e_is,
    254,
    4,
);
```

Where ”e_is” is a Vec¡Vec¡¿¿ type consisting of all aggregation parameters. At first glance, this implementation seems to be the same as the previous one. However, this function is a standard MSM implementation, we don’t need to sum the scalar multiplication results manually. See the function details below:

```
if P.len() <= 25 {
    multi_scalar_multiply::<F, FC, C>(
        self.field_chip,
        builder.main(),
        P,
        scalars,
        max_bits,
        window_bits,
    )
} else {
    /*let mut radix = (f64::from((max_bits * scalars[0].len()) as u32)
        / f64::from(P.len() as u32))
    .sqrt()
    .floor() as usize;
    if radix == 0 {
        radix = 1;
    }*/
    // guessing that is always better to use parallelism for >25 points
    pippenger::multi_exp_par::<F, FC, C>(
        self.field_chip,
        builder,
        P,
        scalars,
        max_bits,
        window_bits, // clump_factor := window_bits
```

When the length of MSM is bigger than 25, it use a pippenger algorithm to accelerate the MSM.

4 New Benchmark

The improved MSP.B benchmark result is as following:

num_aggregation	proving_time	verification_time
8	44.01s	14.84ms
16	58.69s	9.57ms
32	80.52s	10.23ms
64	125.82s	15.56ms
128	217.63s	18.24ms
256	422.99s	27.62ms

We achieved a 4× speed-up in this relation. And the parameters are totally same as previous implementation.