24.06.25 Bug Discussion

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

June 25 2024

1 BLS12-381 and Jubjub

1.1 BLS12-381

The parameter of curve is:$\mathbf{z} = -$**0xd201000000010000** (hexadecimal): low hamming weight, few bits set to 1.

Field modulus: $q = \frac{1}{3}(z-1)^2(z^4 - z^2 + 1) + z$, 381-bit:

q = 0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf6730d2a0f6b0f62 41eabfffeb153ffffb9feffffffffaaab

Subgroup size: $r = (z^4 - z^2 + 1)$, 255-bit:

r = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

And the form of curve BLS12-381 is:

$$E(\mathbb{F}_q) := y^2 = x^3 + 4$$

1.2 Jubjub

Jubjub is an elliptic curve of the twisted Edward's form. It is defined over finite field $\mathbb{F}_q$ where

q = 0x73eda753299d7d483339d80809a1d80553bda402fffe5bfeffffffff00000001

with a subgroup of order r and cofactor 8.

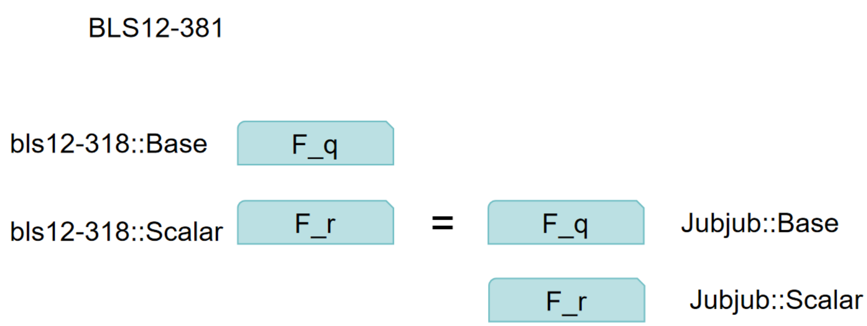
r = 0x0e7db4ea6533afa906673b0101343b00a6682093ccc81082d0970e5ed6f72cb7

Let $d = -(10240/10241)$, the Jubjub curve is defined as follows:

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

$\mathbb{F}_q$ is chosen to be the scalar field of BLS12-381 curve construction.

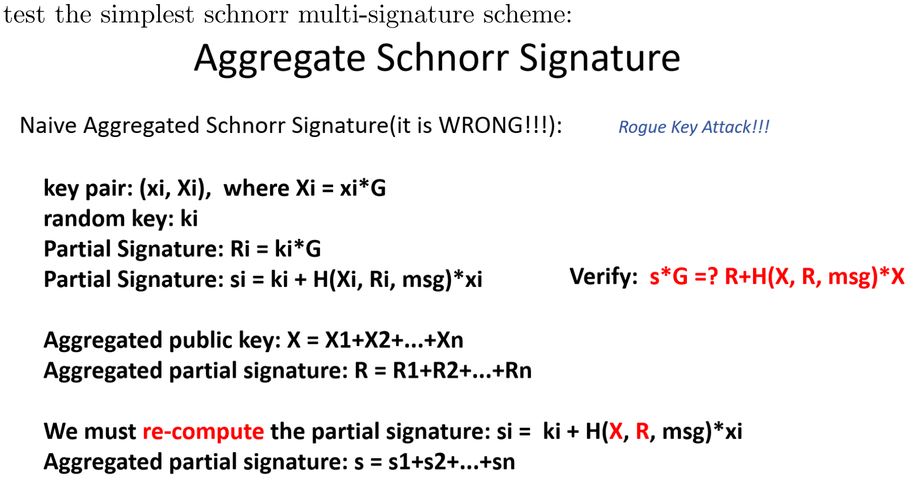
1.3 Relations in Code



2 Bug Analysis

2.1 Description

We test the simplest schnorr multi-signature scheme:



And find that when the random seed changes, the test results also change. We use random Chacha20 as a rng.

- num of parties: 16, random seed = [0u8;32]. Test **pass**.
- num of parties: 17, random seed = [0u8;32]. Test **fail**.
- num of parties: 17, random seed = [111u8;32]. Test **pass**.
- num of parties: 16, random seed = [111u8;32]. Test **fail**.

The failure happens in below equation(it is implemented in other form in code):

$$s * G = R + H(R, X, msg) * X$$

If we add the constraint for public inputs, the situation becomes more complex. We add three constraints:

1. aggregated pk
2. aggregated signature **R** in (R,s)
3. aggregated signature **s** in (R,s)

Note that the aggregated signature scalar **s** will be transformed in base field of Jubjub(which is scalar filed of BLS12-381).

And the two coordinates of aggregated signature **R** is also in base field of Jubjub.

And the test results of these three public inputs are:

1. aggregated pk: always **pass** the test.
2. aggregated signature **R**: always **pass** the test.
3. aggregated signature **s**: Always **fails**.

It is very strange that since the aggregated signature **s** always fails the test, how can the test pass(without public input constraint).

See screen shot of code:

```
real x is 0x6e174c70c777c302bed76290f897de18ca9ee41bf93b2cb43ed3f2e6e75e8d6
7
real y is 0x452563af193eb48e3909c7e6a84385ce802f863982cbb1994f62d805ead9608
3
real scalar is 0x01ff4c58629a68c30181240dbbe3cd6ff43e65666e87f330b09ad6054e
200601
base scalar is 0x01ff4c58629a68c30181240dbbe3cd6ff43e65666e87f330b09ad6054e
200601
circuit x is AssignedCell { value: Value { inner: None }, cell: Cell { regi
on_index: RegionIndex(0), row_offset: 12, column: Column { index: 0, column
_type: Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar::Sca
lar> }
circuit y is AssignedCell { value: Value { inner: None }, cell: Cell { regi
on_index: RegionIndex(0), row_offset: 12, column: Column { index: 1, column
_type: Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar::Sca
lar> }
circuit scalar is AssignedCell { value: Value { inner: None }, cell: Cell {
 region_index: RegionIndex(0), row_offset: 13, column: Column { index: 2, c
olumn_type: Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar
::Scalar> }
circuit x is AssignedCell { value: Value { inner: Some(0x6e174c70c777c302be
d76290f897de18ca9ee41bf93b2cb43ed3f2e6e75e8d67) }, cell: Cell { region_inde
x: RegionIndex(0), row_offset: 12, column: Column { index: 0, column_type:
Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar::Scalar> }
circuit y is AssignedCell { value: Value { inner: Some(0x452563af193eb48e39
09c7e6a84385ce802f863982cbb1994f62d805ead96083) }, cell: Cell { region_inde
x: RegionIndex(0), row_offset: 12, column: Column { index: 1, column_type:
Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar::Scalar> }
circuit scalar is AssignedCell { value: Value { inner: Some(0x0b34554724c07
3005314df516a5208e6d89665ada19abad0944fc51e209b5b25) }, cell: Cell { region
_index: RegionIndex(0), row_offset: 13, column: Column { index: 2, column_t
ype: Advice } }, _marker: PhantomData<halo2curves::bls12_381::scalar::Scala
r> }
```

2.2 A Possible Reason

I think the a possible reason is that there is no struct for a **Scalar** value in Jubjub curve.

Since the **Scalar** is much smaller than **Base**, it is handy to use a **Base** type value as **Scalar**. See picture below:

```rust
/// Structure representing a `Scalar` used in variable-base multiplication.
#[derive(Clone, Debug)]
2 implementations
pub struct ScalarVar(pub(crate) AssignedValue<Base>);
```

In this way, there will be bugs in the calculation of signature **s**:

$$s = s_1 + s_2 + \ldots + s_n$$

This is because the addition of partial signature s_i will be under the modulus of **Jubjub::Base**, but what we need is the modulus of **Jubjub::Scalar**.

2.3 Solution

We just do a modular arithmetic on every addition. Here we multiplexing the code of checking equality:

$$s * G = R + H(X, R, msg) * X$$

The challenge, or $H(X, R, msg)$ will be reduces to **Scalar** filed to multiply **X**. We do the same thing to the addition results of every two s_i. See code below:

```
pub fn change_field(
    &self,
    ctx: &mut RegionCtx<'_, Base>,
    scalar: &AssignedValue<Base>,
)-> Result<AssignedValue<Base>, Error>{
    // consider a better way for this

    // transform the scalar value into scalar field
    let jub_jub_scalar_bytes: [u8; 32] = [
        183, 44, 247, 214, 94, 14, 151, 208, 130, 16, 200, 204, 147, 32, 104, 166, 0, 59, 52,
        1, 1, 59, 103, 6, 169, 175, 51, 101, 234, 180, 125, 14,
    ];

    let jubjub_mod: Scalar =
        Base::from_bytes(&jub_jub_scalar_bytes).expect(msg: "Failed to deserialise modulus");


    let mult_remainder: Vec<Value<Scalar>> = scalar &AssignedCell<Scalar, S…
        .value() Value<&Scalar>
        .map(|&val: Scalar| {
            let (mult: BigUint, remainder: BigUint) = fe_to_big(fe: val).div_rem(&fe_to_big(fe: jubj
            [big_to_fe::<Base>(mult), big_to_fe(remainder)]
        }) Value<[Scalar; 2]>
        .transpose_vec(length: 2);

    let real_scalar: AssignedCell<Scalar, Scalar> = self.schnorr_gate.ecc_gate.main_gate.assign_val

    self.schnorr_gate.ecc_gate.main_gate.assert_zero_sum(
        ctx,
        terms: &[
            Term::Assigned(&scalar, -Base::ONE),
            Term::Unassigned(mult_remainder[0], jubjub_mod),
            Term::Assigned(&real_scalar, Base::ONE),
        ],
        constant: Base::ZERO,
    )?;
```

And this will cost a constraint:

$$scalar(basefiled) = mult\_remainder * jubjub\_mod + real_scalar(scalarfield)$$

3 Benchmark

The Bug above will influence the performance of SNARK-based Aggregated Schnorr Signature scheme. Note that it is the **original** implementation of Aggregated Schnorr signature, without any optimization.

Setting	Proving Time
k = 13, n = 5	0.8408s
k = 14, n = 8	1.3719s
k = 15, n = 14	2.4313s
k = 16, n = 20	4.4653s
k = 17, n = 32	8.3078s
k = 19, n = 72	30.616s

Table 1: Proving time of Aggregated Schnorr Signature

This proof of concept implementation mainly proves the following core components of signature scheme:

1. $R = R_1 + R_2 + ... + R_n$
2. $a_i = H(L||X_i)$ for all X_i, where $L = (X_1, X_2, ..., X_n)$
3. $X = a_1 * X_1 + a_2 * X_2 + ... + a_n * X_n$
4. $s = s_1 + s_2 + ... + s_n$
5. $s * G =?R + H(X, R, msg) * X$

4 Merkle Tree Root Proof

We use the Rescue hash function to build a 2-arity merkle tree, to prove that the correctness of the root. Here is the benchmark.

n is the number of leaves.

Setting	Proving Time
k = 13, n = 16	0.6570s
k = 14, n = 64	1.1433s
k = 16, n = 256	3.8133s
k = 18, n = 1024	13.763s
k = 19, n = 2048	26.243s

Table 2: Proving time of Merkle Tree Root

5 An Optimized Aggregated Schnorr Signature

Consider the real workflow of a aggregated signature scheme, it is reasonable to add the membership-proof feature into the workflow.

The optimized Aggregated Schnorr signature scheme has two set of public keys. One of which is the public keys used to "sign" the message, the other one is the public keys participate in the merkle tree root proving(as a member of leaves).

1. $R = R_1 + R_2 + ... + R_n$
2. $(X_1, X_2, ..., X_n) \in PK$, where $PK = (X_1, X_2, ..., X_n, NX_1, NX_2, ..., NX_m)$
3. $a_i = H(L||X_i)$ for all X_i, where $L = ROOT(X_1, X_2, ..., X_n, NX_1, NX_2, ..., NX_m)$
4. $X = a_1 * X_1 + a_2 * X_2 + ... + a_n * X_n$
5. $s = s_1 + s_2 + ... + s_n$
6. $s * G =?R + H(X, R, msg) * X$

Thus we need a efficient membership-proof technique to prove the **step 2**. We plan to use the Merkle tree path to prove the relation firstly.