secp256r1 review – test plan

Arash Afshar Samuel Ranellucci

Feb 14, 2024

After reviewing the code we need to perform the following tests to ensure that the code matches our understanding of the paper and verify the validity of the potential bugs.

Link to the code: FCL.sol

We need the following helper functions first:

- A way to generate valid (x, y) coordinates (you can use other Go or Rust libraries or implement it directly as follows)

```
Unset
The (x, y) coordinates are valid if they are on the elliptic curve and pass the
ecAff_isOnCurve(x, y) check. Since it is for testing we don't care about
uniform radomness and we generate the points the easy way.

To compute them randomly do the following:

// note: a, b, and p are the constant in the code
choose random x
let x = x % p
let yy = addmod(mulmod(mulmod(x, x, p), x, p), mulmod(x, a, p), p); // x^3+ax
// computing the square root of yy if one exists (therefore this should be done
in a loop. In other words, if the following fails, increment x and try again)
if yy has square root (yy^{(p-1)/2} == 1) then,
let y = yy^{ (p+1)/4 } % p
output x, y
```

- A way to convert affine coordinates (x, y) to projective coordinates (x', y', zz, zzz)

```
Unset
The goal is to convert (x, y) to (x', y', zz, zzz) such that x=x'/zz and
y=y'/zzz where zz is the square and zzz is the cube of some value z > 0
Let p be the constant value in the code


Given (x, y)
Choose random z
let zz = mulmod(z, z, p)
```

```
let zzz = mulmod(zz, z, p)
let x' = mulmod(x, zz, p)
let y' = mulmod(y, zzz, p)

output (x', y', zz, zzz)
```

Testing the inverse functions:

- I have written the test for FCL_nModInv we need to write the same thing for FCL_pModInv (using p instead of n)

```
function test_basecasesFC_nModInv() public {
    // array of base cases
    uint[] memory basecases = new uint[](8);
    uint[] memory results = new uint[](8);
    for (uint i = 0; i < 4; i++) {
        basecases[i] = i;
        results[i] = 1;
    }
    for (uint i = 0; i < 4; i++) {
        basecases[7 - i] = N - i;
        results[4+i] = 1;
    }
    // 0 and N are not invertible
    results[0] = 0;
    results[8] = 0;

    for (uint i = 0; i < 8; i++) {
        uint inv = FCL_ecdsa.FCL_nModInv(basecases[i]);
        uint mul = mulmod(basecases[i], inv, N);
        assertEq(mul, results[i]);
    }
}
```

1. Testing the addition and doubling functions:

```
Unset
choose valid (x, y) on the curve
convert it to projective: (x', y', zz, zzz)
verify that:
```

```
ecZZ_Dbl(x', y', zz, zzz) = ecZZ_AddN(x', y', zz, zzz, x, y)

UPDATE: since ecZZ_AddN does not work when its two inputs are the same, we
decided to test ecZZ_Dbl and ecZZ_AddN separately against a Go library
```

2. Testing ecZZ_mulmuladd_S_asm involves testing that its inlined parts work as expected:
    a. Testing lines 199-247 with ecAff_add

```
Unset
choose valid (x, y), (t1, t2) on the curve
convert (x, y) to projective: (x', y', zz, zzz)
verify that:
lines199to247(x', y', zz, zzz, t1, t2) = ecAff_add(x', y', zz, zzz, t1, t2)
```

    b. Testing lines with 165-176 ecZZ_Dbl

```
Unset
Choose valid (x, y) on the curve
convert it to projective (x', y', zz, zzz)
(x1, y1) = lines165to176(x, y)
(x2, y2) = ecZZ_Dbl(x', y', zz, zzz)
Verify that:
x1 = x2
y1 = -y2 // sub(p, y2)
```

    c. Testing lines 217-231 with ecZZ_Dbl

```
Unset
Choose valid (x, y) on the curve
convert it to projective (x', y', zz, zzz)

(x1, y1) = lines217to231(x, y)
(x2, y2) = ecZZ_Dbl(x', y', zz, zzz)
Verify that:
x1 = x2
y1 = y2
```

d. Testing lines 237-245 with ecZZ_AddN

```
Unset
lines237to245(x, y, zz, zzz, t1, t2) = ecZZ_AddN(x, y, zz, zzz, t1, t2)
```

e. Testing lines 248-268

```
Unset
choose x, z > 0 randomly
x = x % p
uint256 zInv = FCL_pModInv(z); // 1/z
uint256 zzInv = mulmod(zInv, zInv, p); // 1/zz
x1 = mulmod(x, zzInv, p); // x/zz

x2 = lines248to268(x, z, p)

verify that x1 = x2
```

f. Testing lines 235-241 (NOTE: we originally suspected that there was a bug here, but after attempting to write a test for it, we noticed that it is working correctly. But as we were working on this, we found another bug. See 2.g below). To see why we originally thought this might be the case, see the next section.

```
Unset
let u = 16
let v = 1
let Q = (p-16)G -> you can use the Go library to compute this
let Q0 be the x coordinate of Q
let Q1 be the y coordinate of Q

Then, verify:
ecZZ_mulmuladd_S_asm(Q0, Q1, u, v) != 0
   (once the bug is fixed, then the condition should be == 0)
```

g. ecZZ_mulmuladd_S_asm loops infinitely for:
   - -Gx, -Gy, u == v

Notes about test 2.f

The given code snippet in fcl_ecda_verisfy.sol is suspected to be invalid.
Note: After running the test, it seems that this issue is prevented by the continue statement on line 184.

```
Unset
if iszero(zz) {
      X := T1
      Y := T2
      zz := 1
      zzz := 1
      continue
}
```

If iszero(zz) is indeed a verification that y1 is zero. Then, if y1 is zero, it means that the previous point that was accumulated is indeed the zero point. Since $t_4 = u_i + 2 \cdot v_i$, we have that when $t_4 \neq 0$ that x, y, zz, zzz becomes $u_i \cdot G + v_i \cdot Q$. This is indeed correct.

This leaves only the case that when $t_4 = 0$

However, when $t_4 = 0$, things go wrong.

In this case, $t_2 = M(X_3 - S)$ where $M = 3 \cdot (X1 - ZZ1) \cdot (X1 + ZZ1)$. However this is non-zero. This means that (x, y, z, zz) become non-zero even though it should be zero.

To confirm this hypothesis, we should set Q = $(p - 16) \cdot G$ and let $u = 16$ and $v = 1$ and see if $u \cdot G + v \cdot Q$ equals the point at infinity.