Master of Applied Computer Science

Selected topics in Embedded 2

Winter Term 2021/22

Use of the CryptoCore for generating ECC key material in GF (p) using Jacobi coordinates

Rosine Kingni Kingni
Adil Ouzzane
Ahmad Hdaib
Abdullah Asif

15 February 2022

Contents

1 Introduction

Elliptic Curve Cryptography (ECC) has gained much recognition over the last decades and has established itself among the well known public-key cryptography schemes, not least due its smaller key size and relatively lower computational effort compared to RSA. Cryptography is the practice and study of the techniques used to communicate and/or store information or data privately and securely, without being intercepted by third parties. Cryptography uses complex mathematics which converts data into unreadable form, so that only intended user can access this information. In this projects we defined asymmetric cryptographic systems, which per- form operations on elliptic curves over finite fields, then we calculate the rule of an ECC Point Addition as well as Point Doubling in affine coordinate representation requires division-operations which is a quite time-consuming calculation. For a time-efficient calculation the transformation into the projective Jacobi coordinates has proven itself since no division operation is required here in the named operations.

2 Objectives

In this project we are planing to do the following:

- Understanding the Elliptic Curve and its usage in cryptography.
- Montgomery Transformation.
- Affine-to-Jacobi Transformation.
- Point Doubling Point Addition.
- Jacobi-to-Affine Transformation Montgomery Back-transformation.
- validation of the calculation inside of the CryptoCore the By SageMath.
- install the code into the driver.

3 Tools Used in the Project

For the giving Project we was allowed to used many application that

- Sage Math: It is a open-source Mathematics software. with features covering many aspects of Mathematics including Algebra Elliptic Curve, numerical analysis, Calcul and Statistics
- Cryptools: Cryptool is an open-source and freeware program that can be used in various aspects of cryptographic and cryptanalytic concepts. There are no other programs like it available over the internet where you can analyze the encryption and decryption of various algorithms.
- Jcryptool: It is an open-source e-learning platform, developed to not only let everybody experiment with cryptography, but to develop and extend the JCrypTool.
- JupyterNotebook: This platform help us to verify if ours code for the Generating the Point Doubling is correct.
- Putty: this Software help us to go to the board and use Sage Math, Application and the driver.

4 Point Doubling

4.1 Description of Point Doubling

Point doubling is one of the group of Operation of ECC over Prime Field(Fp) . Let Fp is a prime Field so that

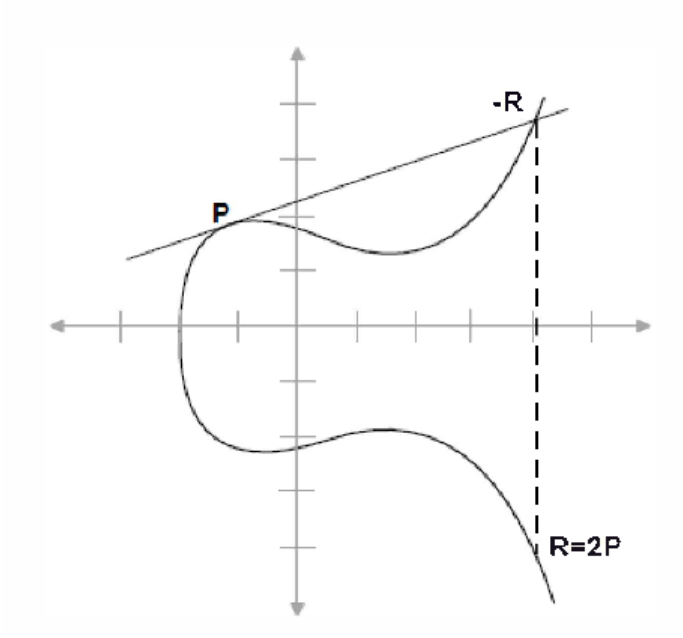

Figure 1: Point Doubling in E(Fp)

E: $y^2 = x^3 + ax + b mod p$.

E is an example of ECC over Finite Field Fp with a and b This Curve must satisfy the condition $4a^3 + 27 bequalto 0$.

If this condition is respected, that means that the ECC is Non-Singular and a Non-Singular Elliptic Curve we target on every Point. Fp: 0,1,2.....(p-1)

Point Doubling or ”adding a point to itself ” follows a very simple equation , and of course it requires the slope and the domain parameters. But basically it goes like This: considered 2 Points P (X1,Y1) and -R (X2, Y2) on the curve. For the Point Doubling P+P=2P=R(X3, Y3) in Affine Coordinate to find the coordinates of X3 and Y3, we need to calculate the Slope

$S = 3X1^3 + a/2Y1 mod p$

$X3 = S^2 - X1 - X2 mod p$

Y3 = S(X1-X3)-Y1 modp

...

4.2 Montgomery'sTransformation of Point Doubling

```
def double_jacobi_MD(P_MD,a,p,prec) :\n",
    "    \n",
    "    R = (2**prec) %p\n",
    "    r2 = (R*R) % p\n",
    "    rinv = pow(R,-1,p);\n",
    "    if P == None :\n",
    "        return None\n",
    "    X1=P_MD.x\n",
    "    Y1=P_MD.y\n",
    "    Z1=P_MD.z\n",
    "    XX = (X1*X1*rinv) % p\n",
    "    YY = (Y1*Y1*rinv) % p\n",
    "    YYYY =(YY*YY*rinv) % p\n",
    "    ZZ = (Z1*Z1*rinv) % p\n",
    "    S1 = (X1+YY) % p\n",
    "    S2 = (S1*S1*rinv) % p\n",
    "    S3 = (S2-XX) % p\n",
    "    S4 = (S3-YYYY) % p\n",
    "    S = (2*R*S4*rinv) % p\n",
    "    ZZ2 =( ZZ*ZZ*rinv) % p\n",
    "    M1 = (3*R*XX*rinv) % p\n",
    "    M2 = (a*R*ZZ2*rinv) % p\n",
    "    M = (M1+M2) % p\n",
    "    T1 = (M*M*rinv) % p\n",
    "    T2 = (2*R*S*rinv) % p\n",
    "    X3 = (T1-T2) % p\n",
    "    Y31 =( 8*R*YYYY*rinv) % p\n",
    "    Y32 =( S-X3) % p\n",
    "    Y33 =( M*Y32*rinv) % p\n",
    "    Y3 = (Y33-Y31) % p\n",
    "    Z31 =( Y1+Z1) % p\n",
    "    Z32 =( Z31*Z31*rinv) % p\n",
    "    Z33 =( Z32-YY) % p\n",
    "    Z3 = (Z33-ZZ) % p\n",
    "    return Point_3d(X3,Y3,Z3)"
  ]
```

...

4.3 Implementation of point Doubling in Crypto Core

For the implementation of Point Doubling, we need to Design the Driver by using the source address and destination address

- the Source Address are B, P(Prime or Modulo),Tc and Ts
- The destination address are A(A1.....A7), E(E1....E7) and X(X1....X7)

to make the calculation of the operation in the Driver,we need to use simple operation but in the slope we have division and to make it faster and easier, we need to avoid the division . The Crypto Core does not recognize the

simple Operations. That is the reason why we have to montgomerise all Operations. For the point Addition, we will use the MontMult, MontSub and MontAdd in the Driver. first of all we need to Montgomerise the number 2,3 and 8 and put them in there in Register A5,A6,A7.

```
Copy(MWMAC_RAM_E3,MWMAC_RAM_B1,COPYV2H,mwmac_cmd_prec);

 MontMult(MWMAC_RAM_B1,MWMAC_RAM_E3,mwmac_cmd_prec);

 Copy(MWMAC_RAM_B1,MWMAC_RAM_X1,COPYH2V,mwmac_cmd_prec); // XX in reg_X1
```

5 Point Addition

5.1 Description of Point Addition

It is basically addition of 2 distinct points, P and Q. To add the points P and Q, a line is drawn through the two points. This line will intersect the elliptic curve in exactly one more point, call -R. The point -R is reflected in the x-axis to the point R.

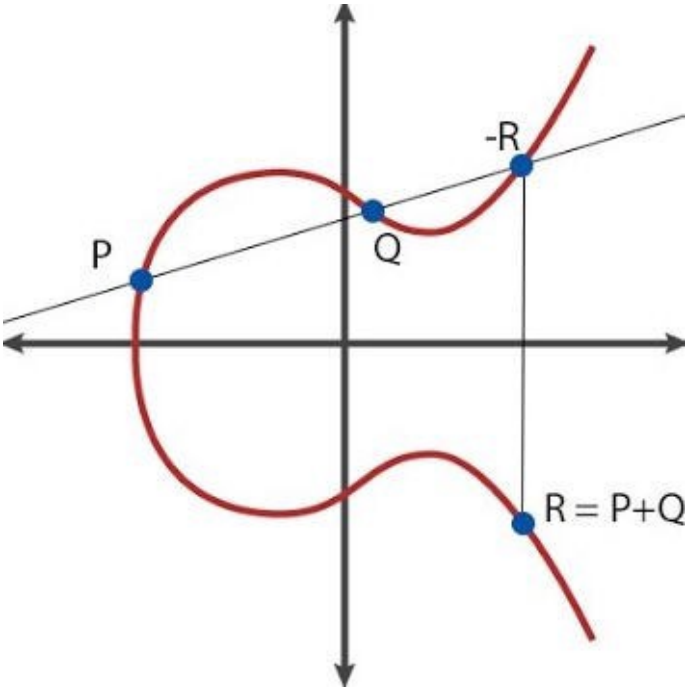

Figure 2: Point Addition in E(Fp)

P + Q = R

(Xp,Yp) + (Xq, Yq) = (Xr, Yr)

Assuming the elliptic curve, E, is given by y2 = x3 + ax + b, this can be calculated as:

m = (Yq – Yp) ÷ (Xq – Xp)

Xr = m2 – Xp – Xq

Yr = m(Xp – Xr) - Yp

5.2 Montgomery of Point Addition

It is the process of fast modular addition of coordinates. After affine to Jacobian conversion all the coordinates are montgomerized. Hence, these coordinates are used in further calculations of Point Addition operations as crypto core do not allow any division operations. For montgomerization in Point Addition, Montmult() operation is frequently used with modulus P to reduce the size of the number.

The following operation is a Montmult(), which multiplies the values in two crypto core registers A and B:

C= Montmult(A, B, P) =A * B * R-1 mod P

5.3 Implementation of Point Addition

Listing 1: Montgomery of Point Addition

```
# Point addition in Jacobian coordinate using Montgomery arithmetic

def add_jacobi_MD(P_MD,Q_MD,a,p,prec) :

    R = (2**prec) % p
    r2 = (R*R)%p
    rinv = pow(R,-1,p) #rinv = inverse_mod(r,p)

    if (P_MD==None or P_MD.z==0) and (Q_MD==None or Q_MD.z==0) :
        return None
    if P_MD==None or P_MD.z==0 : # z=0 means that the point is at infinity
        return Q_MD
    if Q_MD==None or Q_MD.z==0 :
        return P_MD
    if P_MD==Q_MD :
        return double_jacobi_MD(P_MD,a,p,prec)
    X1 = P_MD.x
    Y1 = P_MD.y
    Z1 = P_MD.z

    X2 = Q_MD.x
    Y2 = Q_MD.y
    Z2 = Q_MD.z

    Z1Z1 = (Z1*Z1*rinv) % p
    Z2Z2 = (Z2*Z2*rinv) % p
    U1 = (X1*Z2Z2*rinv) % p
    U2 = (X2*Z1Z1*rinv) % p
    S11 = (Y1*Z2*rinv) % p
    S1 = (S11*Z2Z2*rinv) % p
    S21 = (Y2*Z1*rinv) % p
    S2 = (S21*Z1Z1*rinv) % p
```

```
    H = (U2-U1) % p
    I1  = (2*R*H*rinv) % p
    I  = (I1*I1*rinv )  % p
    J  = (H*I*rinv)  % p
    r1  = (S2-S1) % p
    r  = (2*R*r1*rinv)  % p
    V = (U1*I*rinv)  % p
    X31 = (r*r*rinv )  % p
    X32 = (2*R*V*rinv) % p
    X33 = (X31-J) % p
    X3 = (X33-X32) % p
    Y31 = (V-X3) % p
    Y32 = (2*R*S1*rinv) % p
    Y33 = (r*Y31*rinv)  % p
    Y34 = (Y32*J*rinv)  % p
    Y3 = (Y33-Y34) % p
    Z31 = (Z1+Z2) % p
    Z32 = (Z31*Z31*rinv) % p
    Z33 = (Z32-Z1Z1) % p
    Z34 = (Z33-Z2Z2) % p
    Z3 = (Z34*H*rinv) % p
    return  Point_3d(X3,Y3,Z3)
```

5.4 Implementation in J-Cryptool for very large values

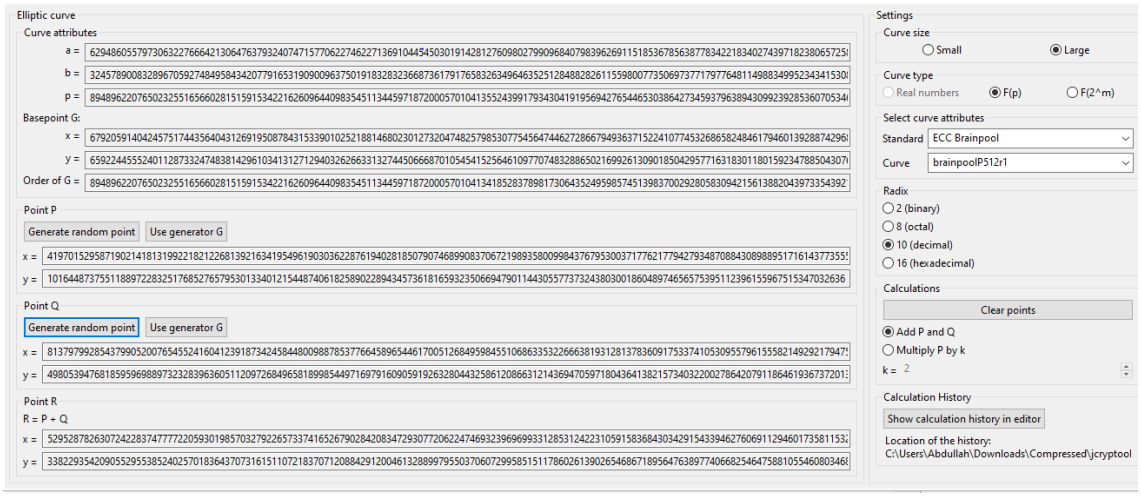

Figure 3: Point Addition in J-Cryptool

Figure 4: Point Addition in Cryptocore Application

5.5 Implementation in Cryptocore Application

6 Affine to Jacobi Transformation

6.0.1 Description of Affine to Jacobi Transformation

This involves transformation of 2D to 3D coordinates system. For simplicity, Z = 1 is assumed which simplifies the x and y coordinates formulas. Hence no division operations are involved.

In Jacobian Coordinates the triple (X', Y', Z) represents (X, Y, 1)

For Affine to Jacobian conversion all the coordinates are montgomerized using Montmult() for X and Y coordinates and MontR() for Z coordinate.

6.0.2 Implementation of Affine to Jacobi Transformation

For Affine to Jacobian conversion all the coordinates are montgomerized using Montmult() for X and Y coordinates and MontR() for Z coordinate.

Listing 2: Affine to Jacobi Transformation

```
# Affine to Jacobi_MD conversion :

def affine_to_jacobi_MD (P,p,prec) :
    r = (2**prec) % p
    r2 = (r*r)%p
    rinv = rinv = pow(r,-1,p) #rinv = inverse_mod(r,p)

    X_MD = (P.x*r)%p #MontMult(P.x,r2,p)
    Y_MD = (P.y*r2*rinv)%p #MontMult(P.y,r2,p)
    Z_MD = r%p; #MontR(p)
    return Point_3d(X_MD,Y_MD,Z_MD)
```

6.0.3 Implementation of Affine to Jacobi Transformation in Crypto core

For Affine to Jacobian conversion implementation in cryptocore MWMAC affine to jacobi function is called. x and y affine coordinates are Written to E1 and E2 Register Memory, respectively.The montgomerized Jacobian

coordinates X, Y and Z are in E3, E4, E5 registers respectively.

Listing 3: Affine to Jacobi Transformation in Crytocore

```
1
2  static void MWMAC_affine_to_jacobi( affine_to_jacobi_params_t * affine_to_jacobi_params_ptr ){
3      u32 i;
4      // u32 mwmac_cmd = 0;
5      u32 mwmac_cmd_prec = 0;
6      u32 mwmac_f_sel = 1;
7      u32 mwmac_sec_calc = 0;
8      u32 rw_prec = 0;
9
10     if( affine_to_jacobi_params_ptr ->f_sel == 0) {
11         mwmac_f_sel = 0;
12         for(i=0; i<16; i++){
13             if(BINARY_PRECISIONS[i][0]==affine_to_jacobi_params_ptr->prec){
14                 mwmac_cmd_prec = BINARY_PRECISIONS[i][1];
15                 if( affine_to_jacobi_params_ptr ->prec % 32) {
16                     rw_prec = ( affine_to_jacobi_params_ptr ->prec/32 + 1) * 32;
17                 } else {
18                     rw_prec = affine_to_jacobi_params_ptr ->prec;
19                 }
20             }
21         }
22     }
23
24     else {
25         mwmac_f_sel = 1;
26         for(i=0; i<13; i++){
27             if(PRIME_PRECISIONS[i][0]==affine_to_jacobi_params_ptr->prec){
28                 mwmac_cmd_prec = PRIME_PRECISIONS[i][1];
29                 rw_prec = affine_to_jacobi_params_ptr ->prec;
30             }
31         }
32     }
33
34     if( affine_to_jacobi_params_ptr ->sec_calc == 0) {
35         mwmac_sec_calc = 0;
36     }
37     else {
38         mwmac_sec_calc = 1;
39     }
40
41     // Write Parameter x_affine to E1 Register Memory
42     for(i=0; i<rw_prec/32; i++){
43         iowrite32( affine_to_jacobi_params_ptr ->x[rw_prec/32-1-i], (MWMAC_RAM_ptr+0x280+i));
44     }
45
46     // Write Parameter y_affine to E2 Register Memory
47     for(i=0; i<rw_prec/32; i++){
48         iowrite32( affine_to_jacobi_params_ptr ->y[rw_prec/32-1-i], (MWMAC_RAM_ptr+0x290+i));
49     }
50
51     MontR2(MWMAC_RAM_P1,MWMAC_RAM_X1,mwmac_cmd_prec);
52      Copy(MWMAC_RAM_B1,MWMAC_RAM_X1,COPYH2V,mwmac_cmd_prec);
53     MontMult(MWMAC_RAM_B1,MWMAC_RAM_E1,mwmac_cmd_prec);
```

```
54    Copy(MWMAC_RAM_B1,MWMAC_RAM_E3,COPYH2V,mwmac_cmd_prec);
55    Copy(MWMAC_RAM_X1,MWMAC_RAM_B1,COPYV2H,mwmac_cmd_prec);
56    MontMult(MWMAC_RAM_B1,MWMAC_RAM_E2,mwmac_cmd_prec);
57    Copy(MWMAC_RAM_B1,MWMAC_RAM_E4,COPYH2V,mwmac_cmd_prec);
58
59    MontR(MWMAC_RAM_P1,MWMAC_RAM_B1,mwmac_cmd_prec);
60    Copy(MWMAC_RAM_B1,MWMAC_RAM_E5,COPYH2V,mwmac_cmd_prec);
61
62    // Read Result x_md from E3 Register Memory
63    for(i=0; i<rw_prec/32; i++){
64        affine_to_jacobi_params_ptr ->x_md[rw_prec/32-1-i] = ioread32(MWMAC_RAM_ptr+0x280+2*0x10+i); //
              0x280+0x10+0x10 = 0x2A0
65    }
66
67    // Read Result y_md (copy) from E4 Register Memory
68    for(i=0; i<rw_prec/32; i++){
69        affine_to_jacobi_params_ptr ->y_md[rw_prec/32-1-i] = ioread32(MWMAC_RAM_ptr+0x280+3*0x10+i);
70
71    }
72
73    // Read Result z_md (copy) from E5 Register Memory
74    for(i=0; i<rw_prec/32; i++){
75        affine_to_jacobi_params_ptr ->z_md[rw_prec/32-1-i] = ioread32(MWMAC_RAM_ptr+0x280+4*0x10+i);
76    }
```

6.0.4 Implementation of Affine to Jacobi Transformation in Crypto core Application

```
P : 0xaadd9db8dbe9c48b3fd4e6ae33c9fc07cb308db3b3c9d20ed6639cca703308717d4d9b009bc66842aecda12ae6a380e62881ff2f2d82c68528aa6056583a48f3
A : 0x7830a3318b603b89e2327145ac234cc594cbdd8d3df91610a83441caea9863bc2ded5d5aa8253aa10a2ef1c98b9ac8b57f1117a72bf2c7b9e7c1ac4d77fc94ca
B : 0x3df91610a83441caea9863bc2ded5d5aa8253aa10a2ef1c98b9ac8b57f1117a72bf2c7b9e7c1ac4d77fc94cadc083e67984050b75ebae5dd2809bd638016f723
xp: 0x81aee4bdd82ed9645a21322e9c4c6a9385ed9f70b5d916c1b43b62eef4d0098eff3b1f78e2d0d48d50d1687b93b97d5f7c6d5047406a5e688b352209bcb9f822
yp: 0x7dde385d566332ecc0eabfa9cf7822fdf209f70024a57b1aa000c55b881f8111b2dcde494a5f485e5bca4bd88a2763aed1ca2b2fa8f0540678cd1e0f3ad80892
xq: 0x81b2354c0bf74652500882c820021b907d9a46bc111315f1b1d2d915c6dcde9e0d9019a6a04eeff3bbba6aeae51d5968ce2a74d229b8851f3d1d15c87146414b
yq: 0x803f72ee0ca52bcc020f708c3da3f99628681c6c870a1a288042cc77cd2d32ecb129c648e6d6bf367ed3d78cdd7e20972987fcb0748d6cdfe605819db16d1985
Affine to Jacobi Results :
Xp_MD: 0x5a2ba14c0994e981871cb5ca006d4573b2b6ea37f36d3cf72433d76f905c87378550539514c01fc834ab04146df55e8f683e4d64272c02a4c4ce96095161d9d3
Yp_MD: 0x8c50c9d12acb72819a5ed7da870f3f9b585d2b77cd9d3f8c7c170b888fe62fdc360ec775598ecc3ebf8455534c8594907518df6f4742f3252f90662925042a6d
Zp_MD: 0x5522624724163b74c02b1951cc3603f834cf724c4c362df1299c63358fccf78e82b264ff643997bd51325ed5195c7f19d77e00d0d27d397ad7559fa9a7c5b70d
Xq_MD: 0x7d768be42c53c236770f7b186706cd77e123b1248e3f56f490960de5342bf797361c86f7cda024c708a254976647e5b49c59a53287ebccdeed14531a990def6c
Yq_MD: 0x8d3225556ca2fc697359ff05186498df10d21d280e09b9b6f99111f2ea0c1860c3347d2bb8f7e38c19600b43750fd602496da454e1ed163f10ce97083f3afce9
Zq_MD: 0x5522624724163b74c02b1951cc3603f834cf724c4c362df1299c63358fccf78e82b264ff643997bd51325ed5195c7f19d77e00d0d27d397ad7559fa9a7c5b70d
```

Figure 5: Affine to Jacobi Implementation in Cryptocore Application

...

7 Point Multiplication

7.1 Introduction

The operation of Point Multiplication in ECC consist of adding a point to itself multiple times. This operation is used to generate a one-way function P=n*G where n is the scalar and G is the generator point. The power of the ECC Point multiplication is that given P and Q, it's very hard to find the scalar n (if the curve parameters are well chosen and the scalar n is big). This is known as the elliptic curve discrete logarithm problem.

We implementation of point multiplication is done mainly in two steps, scalar generation and double and add method, which are described in the following.

7.2 Scalar generation

The first step of point multiplication is the scalar N generation which is the private key of our the ECC algorithm. The main characteristic of the scalar N is that it must be very big to ensure a high level of security and it must be less than the Generator point order to avoid going to infinity point during the multiplication process.

In order to fulfill above conditions, we will generate the scalar N by following below steps:

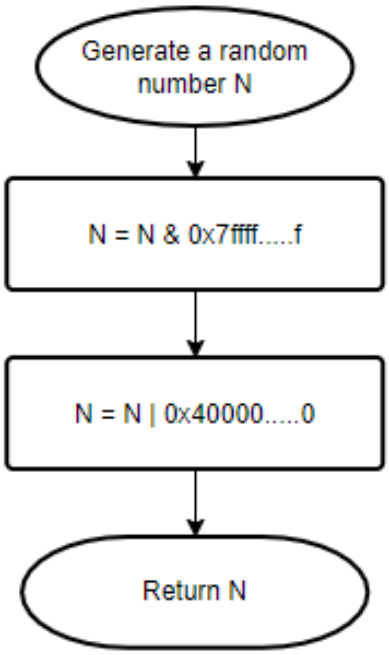

Figure 6: Scalar generation process

1 - We randomly generate the scalar N of a specific precision (512 bits in our case). We use the random generator TRNG function of the Cryptocore that will generate a random number for the given precision.

2 - We force the MSB to be 0 to make sure that the scalar is less than the order of the Generator Point. To achieve that, we will do a bitwise AND of the scalar N and number (that has the same precision as the scalar) composed with ones except the MSB is 0, e.g. (01111111 or 0x7f in case of precision = 8)

3-We force the next bit after the MSB to 1. This will make the scalar big enough to ensure a strong encryption. For that, we will do a bitwise OR operation with a number (that has the same precision as the scalar) composed of zeros except the second bit after the MSB is 1, e.g. (01000000 or 0x40 in case of precision = 8)

7.3 Double and Add

After generating the scalar N, we implement the point multiplication operation using the double and add algorithm as described in the following:

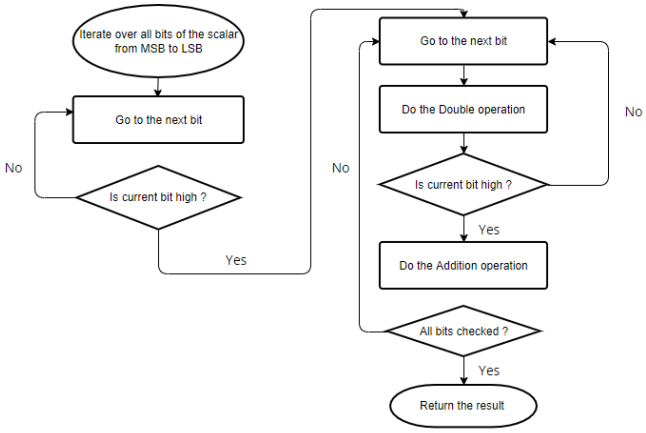

Figure 7: Scalar generation process

The process of double and add is described as follow :

1- Iterate over all bits of the scalar N, starting from the MSB to the LSB, until we find the 1st high bit. 2- Iterate over all remaining bits (ignore the 1st high bit found) and do the following operations until 3- Do the Point Double operation 4- Check If the current bit is high, do the Point Addition operation. Otherwise go to the next bit 5- Return the result once all bits have been checked.

7.4 PM Implementation in the driver

As described in the previous paragraph, the point multiplication consist of scalar generation and double and add method (which is a combination of point double and point addition operations). For the scalar generation, we used the TRNG function of the driver to generate a random number of 512 precision and then we changed the MSB to 0 and the 2nd bit to 1 as shown below :

After generating the scalar, the next step is to implement the double and add algorithm. This is done by iterating over all bits of the scalar and doing the corresponding operation as shown below:

```
// Generate random scalar n from TRNG FIRO
i = 0;
while (i < PointMult_512_test.prec/32) {
    ret_val = ioctl(dd, IOCTL_READ_TRNG_FIFO, &trng_val);
    if(ret_val == 0) {
        PointMult_512_test.n[i] = trng_val;
        i++;
    } else if (ret_val == -EAGAIN) {
        printf("TRNG FIFO empty\n");
    } else {
        printf("Error occured\n");
    }
}

// Forcing the 1st bit (MSB) to 0 and the 2nd bit to 1 :

PointMult_512_test.n[0] = PointMult_512_test.n[0] & 0x7fffffff;
PointMult_512_test.n[0] = PointMult_512_test.n[0] | 0x40000000;
```

Figure 8: Scalar generation in Cryptocore

```
for(i=0; i<PointMult_512_test.prec/32; i++){
    for(j = 1<<31; j>0; j=j>>1){
        if(msb_found) {

            // call PointDouble()
            ret_val = ioctl(dd, IOCTL_MWMAC_PODOUBLE, &PointDouble_512_test);
            if(ret_val != 0) {
                printf("Error occured\n");
            }
            // check if the current bit is high
            if(PointMult_512_test.n[i] & j) {

                // call PointAdd()
                ret_val = ioctl(dd, IOCTL_MWMAC_POADD, &PointAdd_512_test);
                if(ret_val != 0) {
                    printf("Error occured\n");
                }

            }
        }
        else if (PointMult_512_test.n[i] & j) msb_found = 1;
    }
}
```

Figure 9: Double and Add algorithm in Cryptocore

7.5 Jacobi to affine transformation

After getting the Point Multiplication results in Jacobian coordinate, we will transform it to the affine coordinate. For that purpose, we need to perfom the inversion of the Z coordinate. However, as the Cryptocore does not support such operation (inversion), we can use the Modular exponentiation where the exponent is the modulus minus 2 (ModExp(p-2)). Finally we perform the Montgomery back-transformation by multiplying the result with Rinv as shown in below code snippet:

8 Project management

For the Management of Project we have created 200 Tasks and the meeting was 3 time per week because we wanted to give the possibility of all the members to work on his task.

```python
def jacobi_MD_to_affine(P_MD,p,prec) :

    if P_MD == None or P_MD.z == 0 :
        return None

    r = (2**prec) % p
    r2 = (r*r)%p
    rinv = pow(r,-1,p) #rinv = inverse_mod(r,p)

    zinv_MD = (pow(P_MD.z,p-2,p) * r2) % p # zinv_MD = inverse_mod(P.z,p) # P.z is on Montgomery domain ModExp(Z,p-2)
    zinv2_MD = (zinv_MD*zinv_MD*rinv) %p # Z^-2
    zinv3_MD = (zinv2_MD*zinv_MD*rinv) %p

    x_affine_MD = (P_MD.x*zinv2_MD*rinv) %p # P.x is on Montgomery domain
    y_affine_MD = (P_MD.y*zinv3_MD*rinv) %p # P.y is on Montgomery domain

    # Back transformation :

    x_affine = (x_affine_MD * rinv) %p
    y_affine = (y_affine_MD * rinv) %p

    return Point(x_affine,y_affine)
```

Figure 10: Jacobi to affine conversion and Montgomery-back transformation

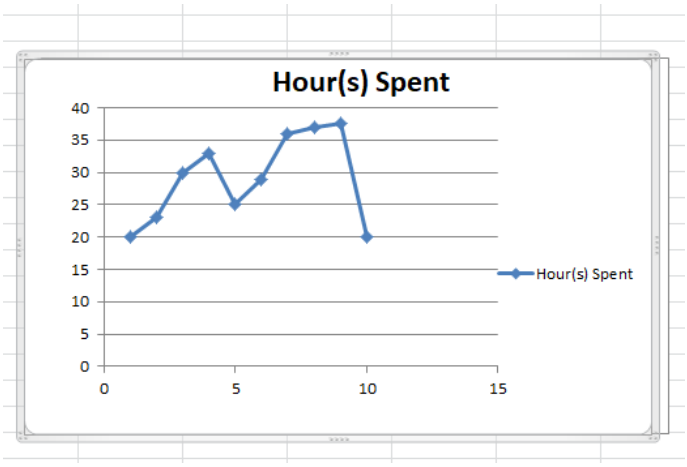

Figure 11: Spend Hours

9 Conclusion

To sum up, we get that the computation on elliptic curves is done by the operations Point Addition, Point Doubling and Point Multiplication in which the latter is based on a sequence of Point Doubling and Point Additions, transformation between coordinates has to be done in order to reduce the time of calculations, using validation tools and cryptools, in addition to other tools like putty and Jcryptool,also we included Real Time Library support (-lrt) in order to be able to illustrate the time required for ECC key generation with different precision widths. At the end we conclude that the application could be extended to use different curves with different precision.

10 References

https://crypto.stackexchange.com/questions/19598/how-can-convert-affine-to-jacobian-coordinates

https://en.wikipedia.org/wiki/Elliptic$_c urve_p oint_m ultiplication$