
# Project Report for Endsem

*Design & Implementation of ECC*

SOHAN DAS
CRS 2119
M.tech student
csr2119@isical.ac.in

Department of Cryptology and Security,
Indian Statistical Institute, Kolkata

December 8, 2022

**Abstract**

Here we will design and implement ECC key exchange protocol in C . We will describe stepwise each and every function used to build it. Will discuss from th eroot level i.e. starting from filed arithmatic to symmetric private key generation with secure exchange. Here I am giving you short description. Let's Alice have some private key let's say a which is unknow to Bob. Similarly Bob has some privatek key let's say b which is unknown to Alice as well. Now the quetion arises that how can they communicate securely in a public chanel with each other. Here the Elliptic curve cryptography plays an important roll, actually hardness of Discrete Logarithm problem is used. Alice makes a public key a*P and then Bob computes b*(a*P) and similarly Bob makes a public key b*P and then Alice computes a*(b*P). Due to the fact that EC group is an abelian group so both of them are same and in this way Alice and Bob get a symmetric private key.

# Contents

**1 Introduction** **1**

**2 Overview of ECC key exchange protocol** **1**

**3 Integer Arithmatics** **1**

3.1 Additon . . . . . 1

3.2 Multiplication . . . . . 2

3.3 Subtraction . . . . . 4

**4 Field Arithmatics** **5**

4.1 Introduction . . . . . 5

4.2 Modulo Operation . . . . . 5

4.3 Barrett Reduction . . . . . 5

4.4 Square and Multiply Algorithm . . . . . 8

4.5 Inverse Algorithm . . . . . 9

**5 Elliptic Curve** **10**

5.1 Elliptic Curve Algorithms . . . . . 10

5.1.1 ECC_Addition() . . . . . 10

5.1.2 Doubling And Adding() . . . . . 13

**6 Elliptic Curve Key Exchange** **15**

# List of Figures

1 Elliptic Curve:Addtion Rule . . . . . 10

2 Secure Key Echange . . . . . 16

# 1 Introduction

Elliptic-curve Diffie–Hellman (ECDH) is a key agreement protocol that allows two parties, each having an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel. This shared secret may be directly used as a key, or to derive another key. The key, or the derived key, can then be used to encrypt subsequent communications using a symmetric-key cipher. It is a variant of the Diffie–Hellman protocol using elliptic-curve cryptography.

# 2 Overview of ECC key exchange protocol

In recent years the rapid deployment of applications like online banking, stock trading and corporate remote access, have seen an explosive growth in the amount of sensitive data exchanged over the internet. Moreover, these internet hosts increasingly are battery-powered, wireless, handheld devices with strict memory, CPU, latency and bandwidth constraints. Given these trends, there is a clear need for efficient, scalable security mechanisms and protocols that operate well in both wired and wireless environments. To date elliptic curve cryptography is gaining wide acceptance as an alternative to the conventional cryptosystems (DES, RSA, AES, etc.) which tend to be power hungry. Elliptic curve ciphers require less computational power, memory and communication bandwidth giving it a clear edge over the traditional crypto-algorithms. This study describes the basic design principle of Elliptic Curve Crypto (ECC), EC discrete logarithm problem, ECDH key agreement and encryption protocols.

# 3 Integer Arithmatics

## 3.1 Additon

Here we have considered $2^{28}$ base , and we are doing 256-bit addition so
$a = a_0 + a_1B + .... + a_9B^9$
$b = b_0 + b_1B + .... + b_9B^9$ Then the addition c will be
$c = c_0 + c_1B + .... + c_9B^9$ , where B = $2^{28}$

```
1    void add(long long int a[], long long int b[], long long int c[], int
         size) // a and b will be added and the sum will be stored in c
2 {
3    long long int carry = 0;
4    for(int i =0; i<size; i++)
5        c[i] = 0;
6    for (int i = 0; i < size; i++)
7    {
8        c[i] = a[i] + b[i] + carry;
9        carry = (c[i] >> 28) & 1;
```

```
        c[i] = c[i] & ((1 << 28) - 1); // (1<<28) - 1 = 2^28 - 1

        // printf(" %lld", c[i]);
    }
}
```

## 3.2 Multiplication

Here first I have implemented schoolbook multiplication and then later the Karatsuba multiplication is implemented.

**Schoolbook Multiplication**

```
    void mult(long long int a[], long long int b[], long long int m[], int
        size) // a and b will be multiplied and the product will be stored in m
{
    long long int restof28bit;
    for (int i = 0; i < 2 * size; i++)
    {
        m[i] = 0;
    }
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            m[i + j] = m[i + j] + (a[i] * b[j]);
            restof28bit = m[i + j] >> 28;
            m[i + j] = m[i + j] & ((1 << 28) - 1); // (1<<28) - 1 = 2^28 - 1
            m[i + j + 1] = m[i + j + 1] + restof28bit; // Carry forwarding
        }
    }

    // for (int k = 0; k < 2 * size; k++)
    //     printf(" %lld", m[k]);
}
```

**Karatsuba Multiplication**

```
/*  ------------------ KARATSUBA MULTIPLICATIONS ------------ */


// This karat_1M1 multiplies 1 degree polynomial with 1 degree polynomial
    using KARATSUBA multiplication
void karat_1M1(long long int a[], long long int b[], long long int k_ab[],
    int size) // size = sizeof(a) = 2 = degree+1, k_ab = AB, where A = a0+a1*x
    , B = b0+b1*x
```

```
{
    int i;
    long long int coeffi[2*size]; // it will store coefficients of x^i 's for
        i = 0,1,2
    // coeffi[3] = 0 is taken for getting advantage in building for loop in
        base management part below
    for(i=0; i<2*size; i++) // initialization
    {
        k_ab[i] = 0;
        coeffi[i] = 0;
    }
    coeffi[0] = a[0]*b[0]; //coefficient of x^0
    coeffi[2] = a[1]*b[1]; //coefficient of x^2
    coeffi[1] = (a[0] + a[1])*(b[0] + b[1]) - (coeffi[0] + coeffi[2]);
        //coefficient of x^1

    // base management part
    baseManagement(coeffi, k_ab, 2*size, 28);
}
```

There are few many karatsuba many karatsuba multiplications implemented in my code for different degrees of polynomials. Only for 1 - degree calculation in code snippet I have attached here, but actually all the others for different degree polynomials are similar. In those successive karatsuba multiplication I call the previously implemented karatsuba multiplications as well.
Here I use a **baseManagement()** function to manage the base B = $2^{28}$ of each coefficients.

```
    // This will prepare all the array elements on 2^28 base
void baseManagement(long long int input[], long long int outputbase[], int
    size, int base) // size means sizeof(outputbase) = sizeof(input)
{
    int i;
    long long int inp_coeffi[size], carry = 0;
    for (i=0; i<size; i++) // initialization
    {
        inp_coeffi[i] = input[i];
        outputbase[i] = 0;
    }
    outputbase[0] = inp_coeffi[0]&((1<<base)-1); // (1<<28) - 1 = 111.....11
        (base number of 1's)
    for(i = 1; i<size; i++)
    {
        carry = inp_coeffi[i-1]>>base; // carry from previous block
        inp_coeffi[i] = inp_coeffi[i]+carry; // update the current block
        outputbase[i] = inp_coeffi[i]&((1<<base)-1); // ((1<<size)-1) =
            1111...11 , size no. of 1's
    }
```

```
}
```

## 3.3 Subtraction

We will now implement subtraction using 2's complement method as follows :

```
    void sub(long long int a_o[], long long int b_o[], long long int s[], int
        size) // s = a_o - b_o with the assumption that a_o > b_o , o stands
        for original
{   long long int a[size];
    long long int b[size];
    int i;
    for (i = 0; i < size; i++) // initialization
    {
        s[i] = 0;
        a[i] = a_o[i];
        b[i] = b_o[i];
    }

    // for (i = 0; i < size; i++)
    // {
    //     if (a[i] < b[i])
    //     {
    //         a[i] = a[i] + (1 << 28);
    //         b[i + 1] = b[i + 1] + 1;
    //     }
    //     s[i] = a[i] - b[i];

    //     // printf(" %lld", s[i]);
    // }

    /* Using 2's complemnet method */
    long long int I[size], b_2s_com[size];
    for (i = 0; i < size; i++)
    {
        I[i] = 0;
        b[i] = (~b[i])&((1<<28)-1); // 1's complement is done
    }
    I[0] = 1; // I = 1
    add(b, I, b_2s_com, size); // 2's complement of b is done
    add(a, b_2s_com, s, size); // s = a + b_2s_com
}
```

# 4 Field Arithmatics

## 4.1 Introduction

All the operations we have to do , need to done in finite field $F_p$. That's why we need modular arithmetic operation in $F_p$. Also to perform division in $F_p$ we need inverse of an element with respect to modular operation in $F_p$.

## 4.2 Modulo Operation

For modulo operation we use Barrett's modulo reduction technique like $x \equiv a(modp)$.A naive way of computing modular operation,would be to use a fast division algorithm. Barrett reduction is an algorithm designed to optimize this operation assuming n as constant and $a < n^2$ replacing divisions by multiplications.

## 4.3 Barrett Reduction

### Algorithm

Input: $positive\, integers\, x = (x_1, x_2, ...., x_2n), m = (m_1, ....m_n), mu = \lfloor b^{2n}/m \rfloor$
Output: $x\ mod\ n$
$q_1 = \lfloor x/b^{n-1} \rfloor$
$q_2 = mu * q_1$
$\hat{q} = \lfloor q_2/b^{n+1} \rfloor$
$r_1 = x\ mod\ b^{n+1}$
$r_2 = m\hat{q}\ mod\ b^{n+1}$
$\hat{r} = r_1\ - r_2$
**if** $(r \leq 0)$ : **then**
$r = r\ +\ b^{n+1}$
**while** $(r \geq m)$ **do**:
$r = r\ - m$
**end while**
**end if**
return r;

Now I wish to describe the implementation part of Barrette Reduction
**Code Snippet for Barratte Reduction**

```
1    void Barrett(long long int x[], long long int x_Barret_p[], int size) //
        Barrett reduction with ( mod p), size means size of input x[]
2 {
3    long long int q[11]; // since size of "mu" is 28*11 bits
4    for (int i = 0; i < 11; i++)
5    {
```

```
6         q[i] = 0; // initialization
7     }
8     // q = floor(x/{B^(k-1)}) , k = number of 28-bit blocks used to store the
          value of p, so k = 10, here and B = 2^28, so to divide x by B^9 we
          will right shift x by 9*28 bits and store it in q
9     int i = 0;
10    while (i < (size - 9))
11    {
12        q[i] = x[i + 9];
13        i++;
14    }                    // q = floor(x/{B^(k-1)}) is done.
15    long long int q22[22]; // to store q*mu , both of them are of 11*28 bits
          so their product may be of 2*11*28 bit long
16    for (int i = 0; i < 22; i++)
17    {
18        q22[i] = 0; // initialization
19    }
20    mult(q, mu, q22, 11); // q22 = q*mu is done
21    long long int q11[11]; // since q11 = floor(q22/{B^(k+1)}) , q22 is of
          22*28 bits and k = 10
22    for (int i = 0; i < 11; i++)
23    {
24        q11[i] = 0; // initialization
25    }
26    int j = 0; // initialization
27    while (j < 11)
28    {
29        q11[j] = q22[j + 11];
30        j++;
31    } // q11 = floor(q22/{B^(k+1)}) is done
32    // r = x - q11p11 (mod B^(k+1)), B = 2^28, k= 10
33
34    long long int q11p11[22] = {0}; // sizeof(q11) = 11 block, sizeof(p) = 10,
          so we have to extend it to 11 blocks and we say it p11 also have to
          extend sizeof(x) = 20 to 22 blocks
35    long long int p11[11];       // extending block size of p from 10 to 11
36    for (int i = 0; i < 10; i++) // since sizeof(p) is 10 blocks means 10 long
          long int is used
37    {
38        p11[i] = p[i];
39    }
40    p11[10] = 0;
41    mult(p11, q11, q11p11, 11); // q11p11 = p11 * q11
42
43    // r = x - q11p11 (mod B^(k+1)), first we will calculate x(mod B^(k+1))
          and q11p11(mod B^(k+1)) then will subtract in mod B^(k+1)
44    long long int x11[11] = {0}; // for storing the value x (mod B^(k+1)),
          k=10 here
```

```
45      long long int qp[11] = {0}; // for storing the value q11p11 (mod B^(k+1)),
            k=10 here
46      for (int i = 0; i < 11; i++)
47      {
48          x11[i] = x[i];
49          qp[i] = q11p11[i];
50      }
51      long long int r11[11] = {0}; // r11 = x11 - qp (mod B^(k+1))
52      int flg = check(x11, qp, 11); // flg = 1 if x11>qp, flg = 2 if x11<qp,
            flag = 3 if x11=qp
53      if((flg == 1) || (flg == 3)) // if x11 >= qp
54          sub(x11, qp, r11, 11);
55      if(flg == 2) // if x11 < qp
56          {
57              long long int tmp[11] = {0};
58              sub(qp, x11, tmp, 11); // temp = qp - x11 is done
59              //to do mod B^(k+1) we will use 2's complement method as follows
60              for(int i = 0; i<11; i++)
61              {
62                  tmp[i] = (~tmp[i])&((1<<28)-1); // 1's complement in 28-bit is
                        done
63              }
64              long long int I[11] = {0};
65              I[0] = 1;
66              add(tmp, I, r11, 11); // 2's complement is done
67          }
68
69      // Checking whether r >= p or not
70      int flag = check(r11, p11, 11); // flag = 1 if r11>p11, flag = 2 if
            r11<p11, flag = 3 if r11=p11
71
72      while ((flag == 1) || (flag == 3)) // do successive subtraction when r >= p
73      {
74          // printf("\n Entered in Barrett While loop flag = %d\n", flag);
75          long long int temp[11] = {0}; // to copy r11
76          for (int i = 0; i < 11; i++)
77          {
78              temp[i] = r11[i];
79              r11[i] = 0;
80          }
81          sub(temp, p11, r11, 11); // r11 = temp - p11 = r11(old) - p11
82
83          // Checking whether r >= p or not
84          flag = check(r11, p11, 11); // flag = 1 when r > p, flag = 2 when r <
                p, flag = 0 when r =p
85      }
86      static int count = 0;
87      count++;
```

```
88      // printf("\n You are checking Barrett %d times", count);
89      for (int i = 0; i < 10; i++)
90      {
91          x_Barret_p[i] = r11[i];
92          // printf("\n x_Barrett_p[%d] = %lld", i, x_Barret_p[i]);
93      }
94  }
```

## 4.4 Square and Multiply Algorithm

We want to get $z = y^d(modp)$ .To do the exponential we will apply the square and multiply. The algorithm is the following :

Input: $y = (y_1, y_2, ..., y_n), d = (d_1, d_2, .., d_n)$
$z \leftarrow 1$
$z \leftarrow z * z$
$BarretReduction(z, p)$
**while** *Last bit position* $\neq$ *empty* **do**
**if** *Last bit of* $d == 1$ **then**
$z \leftarrow z * y$
$BarretReduction(z, p)$
**end if**
*Last bit position* $\leftarrow +1$
**end while**

Now we will describe the code snippet of **sqm_mod_p()** (square and multiply in mood p)

```
1       // Square and multiply ( used for finding inverse in Z*_p)
2   void sqm_mod_p(long long int y[], long long int d[], long long int z[], int
        size)
3   {
4       z[0] = 1;
5       long long int z2[2 * size]; // for storing the value of z^2
6       for (int i = 0; i < 2 * size; i++)
7       {
8           z2[i] = 0; // initialization
9           // printf(" z2[%d] = %lld\n", i, z2[i]);
10      }
11      for (int i = size - 1; i >= 0; i--) // since sizeof(d) = 10 block = 10 *28
            bit
12      {
13          for (int j = 27; j >= 0; j--) // since each block is of 28-bit long
14          {
15              mult(z, z, z2, size); // z2 = z*z, 10 is the size of inputs z's
16              for (int j = 0; j < 10; j++)
```

```
17                z[j] = 0;             // initialization of before storing
                      values gotten from Barrett
18            Barrett(z2, z, 2 * size); // z = z^2 (mod p) is done
19            long long int zy[2 * size]; // for storing z*y
20            for (int i = 0; i < 2 * size; i++)
21                zy[i] = 0; // initialization
22            if (((d[i] >> j) & 1) == 1)
23            {
24                mult(z, y, zy, size); // zy = z*y
25                for (int k = 0; k < 10; k++)
26                    z[k] = 0;           // initialization before storing values
                          gotten from Barrett
27                Barrett(zy, z, 2 * size); // z = zy (mod p) is done
28            }
29        }
30    }
31 }
```

## 4.5 Inverse Algorithm

Since this is a finite field operation and we are working on elliptic curve group of order P and this is a subgroup of $F_P{}^*$. So order of every group element is P-1. Hence $x^{P-1}$ =1. Therefore $x^{-1} = x^{P-2}$. So we will run $sqm\_mod\_p()withinputy = xandd = p - 2$

# 5 Elliptic Curve

The Elliptic curve can be described as a plane algebraic curve which consists of solutions $(x, y)$ for:
$y^2 = x^3 + ax + b$ , for some coefficients a and b in the respective prime filed such that $4a^3 + 27b^2 \neq 0$ holds.

## 5.1 Elliptic Curve Algorithms

In this section we will discuss about **point addition** of Elliptic curve and also discuss about about the **Doubling and Adding** method as follows:

### 5.1.1 ECC_Addition()

Let $P1 = (x_1, y_1)$ and $P2 = (x_2, y_2)$ and $P3 = P1 + P2$ and say $P3 = (x_3, y_3)$.

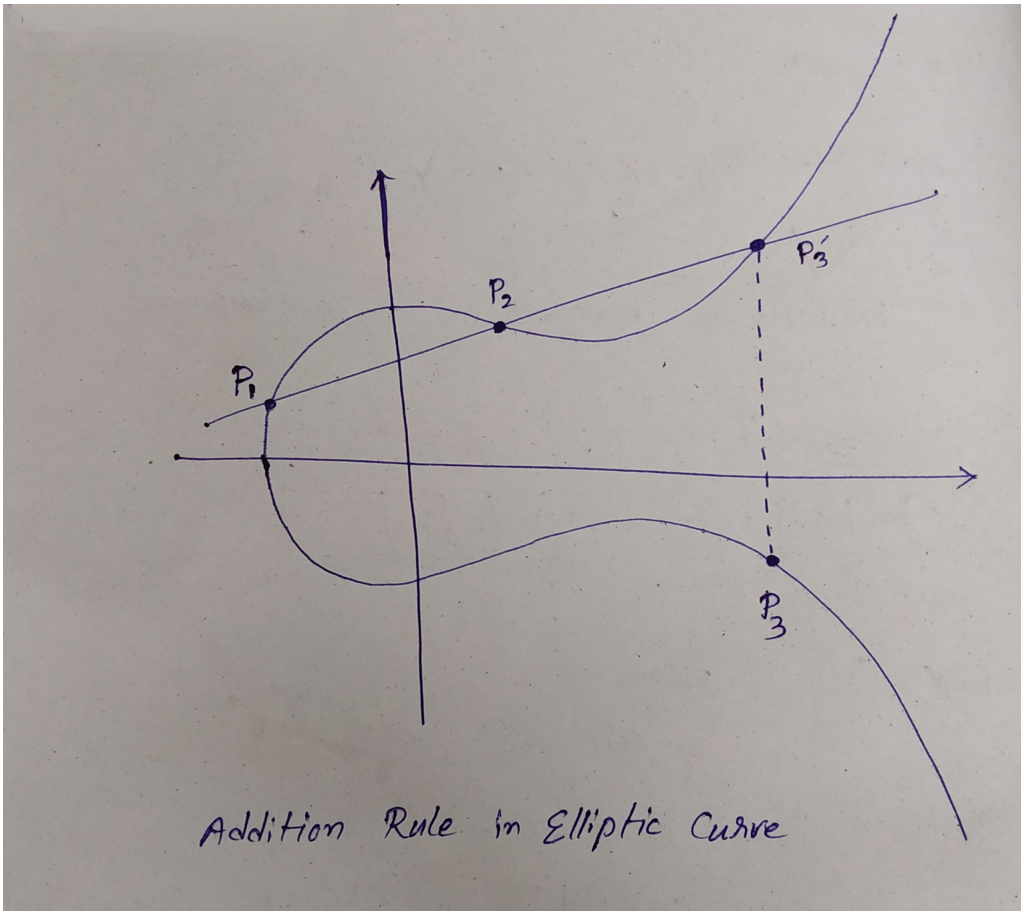


Figure 1: Elliptic Curve:Addtion Rule

```
1 void ECC_addition(long long int x_1[10], long long int y_1[10], long long int
    x_2[10], long long int y_2[10], long long int x3[10], long long int
    y3[10]) // (x1,y1)+(x2,y2) = (x3,y3) ECC Addition
```

```
{
    int i;
    long long int x1[10], y1[10], x2[10], y2[10];
    for (i = 0; i < 10; i++)
    {
        x3[i] = 0;
        y3[i] = 0;
        x1[i] = x_1[i];
        y1[i] = y_1[i];
        x2[i] = x_2[i];
        y2[i] = y_2[i];
    }
    long long int A[20] = {0}; // A = p-3, we take it's size 20 blocks since
        we will add it with 3x1^2, which is of 20 block
    for (i = 0; i < 10; i++)
        A[i] = p[i];
    A[0] = p[0] - 3; // A = (p - 3) is done
    // Now we will compare whether P1=(x1,y1) and P2=(x2,y2) are equal or not
    // lamda = (y2 - y1)/(x2 - x1) ,when P1 != P2
    // lamda = (3x1^2 + A)/(2*y1) ,when P1 = P2
    long long int lamda[10] = {0};
    // for comparing P1, P2 we will set flag_x, flag_y using check() function
        which
    // returns 1 when x>y
    //         2 when x<y
    //         3 when x=y on comparing x and y in check
    int flag_x = check(x1, x2, 10);
    int flag_y = check(y1, y2, 10);
    // printf("flag_x = %d , flag_y = %d", flag_x, flag_y);
    // if P1 == P2 calculate lamda = (3x1^2 + A)/(2*y1)
    if ((flag_x == 3) && (flag_y == 3))
    {
        long long int x1_sq[20] = {0}; // x_sq is for storing x1^2
        mult(x1, x1, x1_sq, 10);
        long long int x1_sq_2sums[20] = {0}; // will be used to store 2*x1^2
        add(x1_sq, x1_sq, x1_sq_2sums, 20); // 2*x1^2 is stored in x1_sq_2sums
        long long int x1_sq_3sums[20] = {0}; // will be used to store 3*x1^2
        add(x1_sq, x1_sq_2sums, x1_sq_3sums, 20); // 3*x1^2 is done and stored
            in x1_sq_3sums
        // Now we have add 3*x1^2 with A
        long long int sum_3x1_sq_and_A[20] = {0};
        add(x1_sq_3sums, A, sum_3x1_sq_and_A, 20); // 3*x1^2 + A is done
        // use Barrett to reduce it into (mod p)
        long long int addedVal_mod_p[10] = {0}; // will be used to store
            reduced value of sum_3x1_sq_and_A in mod p
        Barrett(sum_3x1_sq_and_A, addedVal_mod_p, 20);
        //Calculate inverse of 2*y1 in Z*_p, we will use sqm_mod_p() to find
            inverse
```

```
45          long long int y1_2sums[10] = {0};
46          add(y1,y1,y1_2sums,10);
47          long long int inverse_of_2y1[10] = {0}; // y^(p-2) will be the inverse
                of y in Z*_p
48          long long int p_2[10] = {268435453, 268435455, 268435455, 4095, 0, 0,
                16777216, 0, 268435455, 15}; //p_2 is actually p-2
49          sqm_mod_p(y1_2sums, p_2, inverse_of_2y1, 10);
50          long long int ad_in[20] = {0}; // to store addedVal_mod_p *
                inverse_of_2y1
51          mult(addedVal_mod_p, inverse_of_2y1, ad_in, 10);
52          // ad_in (mod p) will be stored in lamda
53          Barrett(ad_in, lamda, 20);
54      }
55
56      else
57      {
58          long long int y2_y1[10] = {0}; // fro y2-y1
59          sub_mod_p(y2,y1,y2_y1,10); // (y2 - y1) (mod p) is done
60          long long int x2_x1[10] = {0}; // for x2-x1
61          sub_mod_p(x2,x1,x2_x1,10); // (x2 - x1) (mod p) is done
62
63          // calculate inverse of x2-x1 in Z*_p using sqm_mod_p()
64          long long int inv_x2_x1[10] = {0}; // (x2-x1)^(p-2) will be the
                inverse of (x2-x1) in Z*_p
65          long long int p_2[10] = {268435453, 268435455, 268435455, 4095, 0, 0,
                16777216, 0, 268435455, 15}; //p_2 is actually p-2
66          sqm_mod_p(x2_x1, p_2, inv_x2_x1, 10); // calculating inverse of (x2-x1)
67          long long int pr_in[20] = {0}; // pr_in stores (y2_y1 * inv_x2_x1)
                i.e. (y2 - y1)/(x2 - x1)
68          mult(y2_y1, inv_x2_x1, pr_in, 10);
69          Barrett(pr_in, lamda, 20);
70
71          // for(i = 0; i < 10; i++)
72          //     printf("\n %lld",lamda[i]);
73      }
74      /*    VALUE OF LAMDA IS COMPUTED SUCCESSFULLY */
75
76      /* Calculation of x3 : x3 = lamda^2 - x1 - x2 */
77      long long int lamda_2[20] = {0};
78      mult(lamda, lamda, lamda_2, 10); // lamda^2 is done
79      long long int add_x1_x2[10] = {0};
80
81      add(x1, x2, add_x1_x2, 10); // x1+x2 is done
82
83      // Extend the size of add_x1_x2 to 20 blocks
84      long long int ext_add_x1_x2[20] = {0};
85      for(i =0; i<10; i++)
86          ext_add_x1_x2[i] = add_x1_x2[i]; // copied
```

```
87      long long int ext_x3[20] = {0};
88      sub(lamda_2, ext_add_x1_x2, ext_x3, 20);
89      Barrett(ext_x3, x3, 20); // x3 is calculated successfully
90
91      /* Calculation of y3 = lamda(x1 - x3) - y1 */
92      long long int x1_x3[10] = {0}; // for soring (x1 - x3)
93      sub_mod_p(x1, x3, x1_x3, 10); // (x1 - x3) is done
94      long long int pr_lam[20] = {0}; // for storing lamda*(x1 - x3)
95      mult(lamda, x1_x3, pr_lam, 10); // lamda*(x1 - x3) is done
96      long long int ext_y1[20] = {0};
97      for(i=0; i<10; i++)
98          ext_y1[i] = y1[i];  // y1 is extended to 20 block
99      long long int ext_y3[20] = {0}; // for storing y3 in ext_y3 = pr_lam -
            ext_y1
100     sub_mod_p(pr_lam, ext_y1, ext_y3, 20); // y3 = lamda(x1 - x3) - y1 is
            done, now reduce it
101     Barrett(ext_y3, y3, 20); // y3 is calculated
102
103     // for(i = 0; i < 10; i++)
104     //         printf("\n %lld   %lld",x3[i], y3[i]);
105 }
```

### 5.1.2 Doubling And Adding()

It's quit similar to square and multiply method, actually an efficient way of calculation larger sums by using **Doubling** and **Adding**. I would like to explain it with example for ease to understand.
$9a = a + a + a + .... + a(9times)$ $9a = 2 * (2 * (2 * a)) + a$ Also notice that $9 = (1001)_2$
1.Consider z = 0;
2.Point to the msb 1 so of binary representation of 9
3.$z = 2 * z + a = 2 * 0 + a = a$
4. next bit is 0 so just double the value of z and $z = 2 * a$
5. next bit is also 0 so just double the value od z and $z = 2 * (2 * a)$
6. next bit(here we reach the lsb) is 1 so first doublel it and add the value a with z and so $z = 2 * (2 * (2 * a)) + a = 9 * a$
So Done !

```
1  // Doubling and Adding for computing K*(p_x , p_y) = (k_Px , k_Py)
2  void Doubling_Adding(long long int K[10], long long int p_x[10], long long
       int p_y[10], long long int k_Px[10], long long int k_Py[10] ) // K*(p_x ,
       p_y) = (k_Px , k_Py)
3  {
4      // initialization
5      long long int k[10] = {0}, px[10] = {0}, py[10] = {0};
```

```
6     long long int z1[10], z2[10];
7     int i, r;
8     for(i=0; i<10; i++)
9     {
10        k[i] = K[i];
11        px[i] = p_x[i];
12        py[i] = p_y[i];
13        z1[i] = 0;
14        z2[i] = 0;
15
16        k_Px[i] = 0;
17        k_Py[i] = 0;
18    }
19    int j, flag = 0, first_bit = 0, bit_value;
20    long long int z3[10] = {0}, z4[10] = {0}; // will be used to store
          (temporary) sum of ECC_addition
21    for(i = 9; i>=0; i--)
22    {
23        for(j = 27; j>=0; j--)
24        {
25            bit_value = ((k[i]>>j)&1);
26            if((bit_value + flag)==0)
27                continue; // we will not perform any operation untill we get
                      the first non-zero bit
28            if((bit_value+first_bit)==1)
29            {
30                flag = 1;  // first non-zero bit is gotten, so flag is set to
                      1, also in later case if we get bit_vlue = 0 then this
                      if(...) will not be executed but flag will remain set to 1
31                first_bit = 3; // since we want only once
32            }
33            if(((bit_value + flag)>0) && ((bit_value + flag)<3)) // if
                  bit_value+flag = 1 or 2, that means first time getting non-zero
                  bit
34            {
35                flag = 6;  // since we want to execute this only at the first
                      time of getting non-zero bit
36                for(r=0; r<10; r++)
37                {
38                    z1[r] = px[r];
39                    z2[r] = py[r];
40                }
41                continue; // since we want this only once, at the first timeof
                      getting non-zero bit
42            }
43            if((bit_value + flag)>5) // this means this bit is gotten after
                  first non-zero bit
44            {
```

```
45                 ECC_addition(z1,z2,z1,z2,z3,z4); // (z3,z4) = 2*(z1,z2) is done
46                 for(int l = 0; l<10; l++)
47                 {
48                     z1[l] = z3[l];
49                     z2[l] = z4[l]; // new(z1,z2) = 2 * old(z1,z2)
50                     z3[l] = 0;
51                     z4[l] = 0;
52                 }
53                 if(bit_value)
54                 {
55                     ECC_addition(z1,z2,px, py,z3,z4); // if bit_vlaue = 1 then
                           (z3,z4) = (z1,z2) + (px,py)
56                     for(int l = 0; l<10; l++)
57                     {
58                         z1[l] = z3[l];
59                         z2[l] = z4[l]; // new(z1,z2) = old(z1,z2) + (px,py)
60                         z3[l] = 0;
61                         z4[l] = 0;
62                     }
63                 }
64             }
65
66         }
67     }
68     for(i=0;i<10;i++)
69     {
70         k_Px[i] = z1[i];
71         k_Py[i] = z2[i];
72     }
73 }
```

# 6 Elliptic Curve Key Exchange

Suppose Alice wants to establish a shared key with Bob over a public chanel which is not secure in general. Initially, the domain parameters (that is, $(p, a, b, G, n, h)$ in the prime case or $(m, f(x), a, b, G, n, h)$in the binary case) must be agreed upon. Also, each party must have a key pair suitable for elliptic curve cryptography, consisting of a private key d d (a randomly selected integer in the interval $[1, n-1]$ and a public key represented by a point Q (where $Q = d * G$ , that is, the result of adding G to itself d times). Let Alice's key pair be $(d_A, K_A)$ and Bob's key pair be $(d_B, K_B)$. Each party must know the other party's public key prior to execution of the protocol.

Alice computes point $(x_k, y_k) = d_A * K_B$. Bob computes point $(x_k, y_k) = d_B * K_A$ The

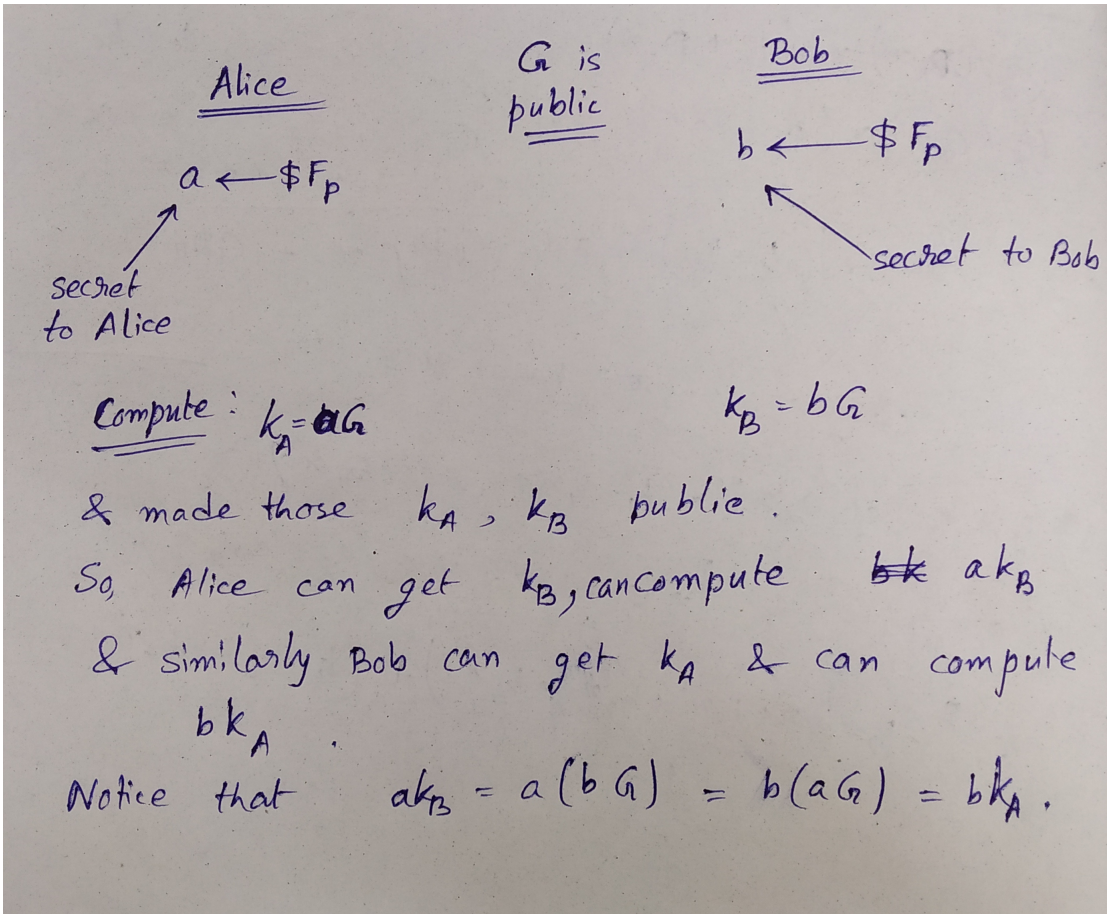


Figure 2: Secure Key Echange

shared secret is $x_k$(the x coordinate of the point). Most standardized protocols based on ECDH derive a symmetric key from $x_k x_k$ using some hash-based key derivation function.

The shared secret calculated by both parties is equal, because
$d_A * K_B = d_A d_B * G = d_B * d_A * G = d_B * K_A$

# References

Joseph H. Silverman,"(Undergraduate Texts in Mathematics) Joseph H. Silverman, John T Tate - Rational Points on Elliptic Curves-Springer (2015)", *Springer Berlin Heidelberg*

Joseph H. Silverman, "An Introduction to Mathematical Cryptography - ", *Springer Berlin Heidelberg*