Implementation of Elliptic Curve Diffie–Hellman Key Exchange in c

Sourav Ghosh

Guide :Dr. Sabyasachi Karati

Indian Statistical Institute

December 2022

Abstract

In this report we will focus on the implementation ECDH (Elliptic Curve Diffie–Hellman Key Exchange)in C language. The ECDH is anonymous key agreement scheme, which allows two parties, each having an elliptic-curve public–private key pair, to establish a shared secret over an insecure channel. ECDH is very similar to the classical DHKE (Diffie–Hellman Key Exchange) algorithm, but it uses ECC point multiplication instead of modular exponentiations. ECDH is based on the following property of EC points:

$$(a * G) * b = (b * G) * a$$

Contents

1 Introduction **4**

2 Change of Bases of digits **5**
2.1 Correctness Checking with an Example 6

3 Addition, Subtraction & Multiplication Of two 2^{30} base Numbers **7**
3.1 Addition 7
3.2 Subtraction 7
3.3 Multiplication 8
3.3.1 Karatsuba algorithm 8
3.4 Code 9
3.4.1 Sage Output 10
3.4.2 C Code Output 10

4 Barrett Reduction **11**
4.0.1 Sage Output 12
4.0.2 C Code Output 12

5 Square & Multiply **13**

6 ECC ADDITION **14**

7 Ecc Scalar Multiplication **17**

8 Elliptic Curve Diffie–Hellman Key Exchange **19**

1 Introduction

The Elliptic Curve Cryptography (ECC) is modern family of public-key cryptosystems, which is based on the algebraic structures of the elliptic curves over finite fields and on the difficulty of the Elliptic Curve Discrete Logarithm Problem (ECDLP). ECC implements all major capabilities of the asymmetric cryptosystems: encryption, signatures and key exchange. The ECC cryptography is considered a natural modern successor of the RSA cryptosystem, because ECC uses smaller keys and signatures than RSA for the same level of security and provides very fast key generation, fast key agreement and fast signatures.

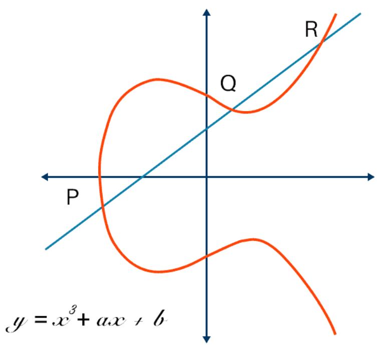


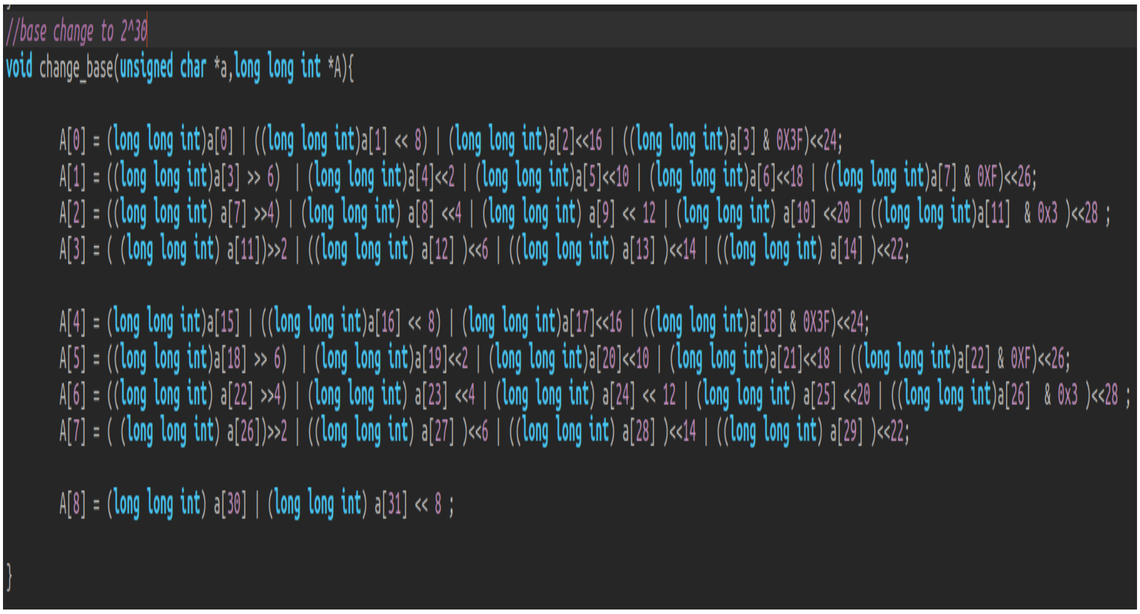
Figure 1: ECC CURVE

If we have two secret numbers a and b (two private keys, belonging to Alice and Bob) and an ECC elliptic curve with generator point G, we can exchange over an insecure channel the values (a * G) and (b * G) (the public keys of Alice and Bob) and then we can derive a shared secret:

$$\text{secret} = (\text{a} * \text{G}) * \text{b} = (\text{b} * \text{G}) * \text{a}.$$

For implementation purpose at first I build the elementary function individually then I merge them to build the whole algorithm. To do less number of elementary operations, I have taken numbers in 256 base then I converted them to 2^{30} base using sagemath.

2 Change of Bases of digits



```c
//base change to 2^30
void change_base(unsigned char *a,long long int *A){

    A[0] = (long long int)a[0] | ((long long int)a[1] << 8) | (long long int)a[2]<<16 | ((long long int)a[3] & 0X3F)<<24;
    A[1] = ((long long int)a[3] >> 6)  | (long long int)a[4]<<2 | (long long int)a[5]<<10 | (long long int)a[6]<<18 | ((long long int)a[7] & 0XF)<<26;
    A[2] = ((long long int) a[7] >>4) | (long long int) a[8] <<4 | (long long int) a[9] << 12 | (long long int) a[10] <<20 | ((long long int)a[11]  & 0x3 )<<28 ;
    A[3] = ( (long long int) a[11])>>2 | ((long long int) a[12] )<<6 | ((long long int) a[13] )<<14 | ((long long int) a[14] )<<22;

    A[4] = (long long int)a[15] | ((long long int)a[16] << 8) | (long long int)a[17]<<16 | ((long long int)a[18] & 0X3F)<<24;
    A[5] = ((long long int)a[18] >> 6)  | (long long int)a[19]<<2 | (long long int)a[20]<<10 | (long long int)a[21]<<18 | ((long long int)a[22] & 0XF)<<26;
    A[6] = ((long long int) a[22] >>4) | (long long int) a[23] <<4 | (long long int) a[24] << 12 | (long long int) a[25] <<20 | ((long long int)a[26]  & 0x3 )<<28 ;
    A[7] = ( (long long int) a[26])>>2 | ((long long int) a[27] )<<6 | ((long long int) a[28] )<<14 | ((long long int) a[29] )<<22;

    A[8] = (long long int) a[30] | (long long int) a[31] << 8 ;

}
```

Figure 2: 2^8 to 2^{30} base change

Intuition is if I consider any binary representation of a number then if I keep taking 2 consecutive bits from lsb side that will give me conversion to base 2^2. Similarly if I take 3 consecutive bits from lsb onwards that will give me conversion to base 2^3. So here I have taken 30 consecutive bits from lsb to get the conversion to 2^{30} base.

Mathematically we can easily see the conversion of any number to 2^h base:

$x = a_0 + 2 * a_1 + 2^2 * a_2 + ... + a_n * 2^{n-1} = (a_0 + 2 * a_1 + 2^2 * a_2 + ... + a_h * 2^{h-1}) + 2^h(a_h + 2 * a_{h+1} + 2^2 * a_{h+2} + ... + a_{2*h-1}) + (2^h)^2(....) + ...$

2.1 Correctness Checking with an Example

I have taken a number in 2^8 base using sagemath. Then I verified my algorithm to check it is converting the number in 2^{30} base correctly or not.

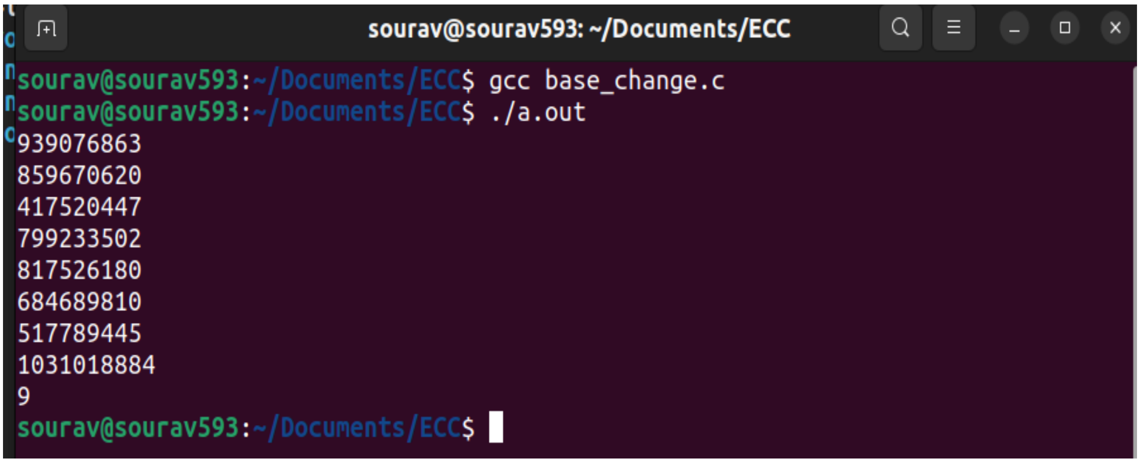


Figure 3: output of C code of Base conversion

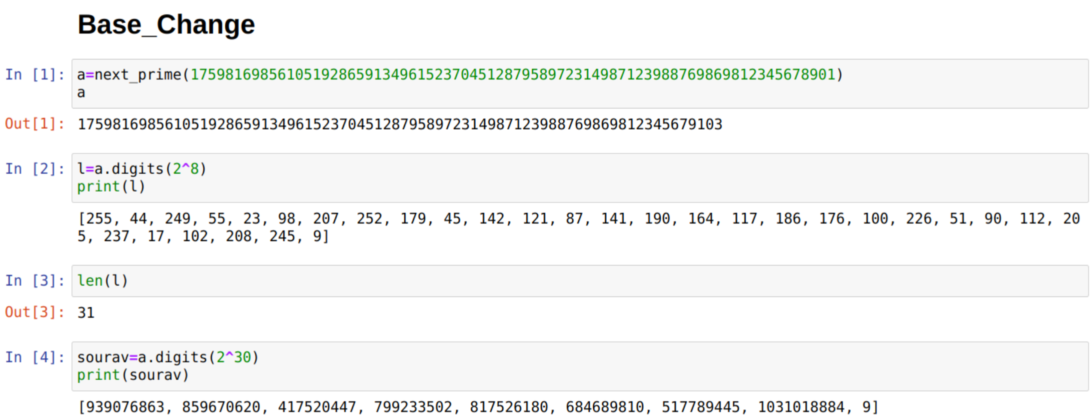


Figure 4: Example taken and checked the answer in sage

3 Addition, Subtraction & Multiplication Of two 2^{30} base Numbers

3.1 Addition

Here I have taken long long int array as inputs. Since long long int has 64 bit size so I have enough space to compute my operations. I did the addition like we generally did in the case for decimal numbers. For example if We add 19 with 9 then We will add the unit places (9+9=18) then we keep 8 in unit place and keep 1 in carry to add in the next step.

```
void addition( long long int *r, long long int *s,
long long int *t,int n){
        int i;
        long long int carry=0;
        for(i=0;i<n;i++){
                t[i]=r[i]+s[i]+carry;
                carry=(t[i]>>30);
                t[i]=t[i] & 0x3fffffff;
        }
        t[9]=carry;
}
```

Here in the code also I add the numbers and to get the carry I have taken the bits after 30^{th} bits. Then I add the carry to the next position.

3.2 Subtraction

Here like addition I used school book technique to do subtraction. Like in decimal subtraction 21-9, we don't do 1-9, we have to borrow 10 from the next digit so that we can subtract 9 from 10+1. So the unit place becomes 11-9=2 & the answer becomes 12. So in decimal we borrow 10, here we have to borrow 2^{30} since we have taken the number in 2^{30} base.

```
void mod_subtract(long long int *r, long long int *s,
long long int *t){
    for(int i=0;i<9;i++){
            t[i]=r[i]-s[i];
            if((t[i]>>63) & 1){
                    t[i]+= 1<<30;
                    r[i+1]-=1;
            }}}
```

But this can subtract a from b correctly if a is greater than b. So I defined a function to compare a & b.

Suppose I want to calculate x-y if x is greater than y then its fine But if y is greater than x then I have to think something. Finally I have to do subtraction in mod p for for some given large prime p. So what I did is for the case for y greater than x is calculate (y-x) then p-(y-x). That will equal to the number x-y in mod p.

3.3 Multiplication

I implemented schoolbook technique as well here. Just multiply to same position block and then if any carry exists I add them in the next block.

```
void multi(long long int *r, long long int *s, long long int *t,
int n){
        for(int i=0;i<n;i++){
                for(int j=0;j<n;j++){
                        t[i+j]+=r[i]*s[j];
                        t[i+j+1]+= (t[i+j]>>30);
                        t[i+j]=t[i+j] & 0x3fffffff;
                }
        }
}
```

3.3.1 Karatsuba algorithm

It is used for multiplication. It is faster than school book method. Since addition is cheaper than multiplication here we do more elementary addition than multiplication.

Take

$$a = x_1 * B^m + x_0$$
$$b = y_1 * B^m + y_0$$
$$a * b = x_1 * y_1 * B^{2m} + (z) * B^m + x_0 * y_0$$
$$z = x_0 * y_1 + y_0 * x_1 = (x_0 + y_0) * (y_0 + y_1) - x_1 * y_1 - x_0 * y_0$$

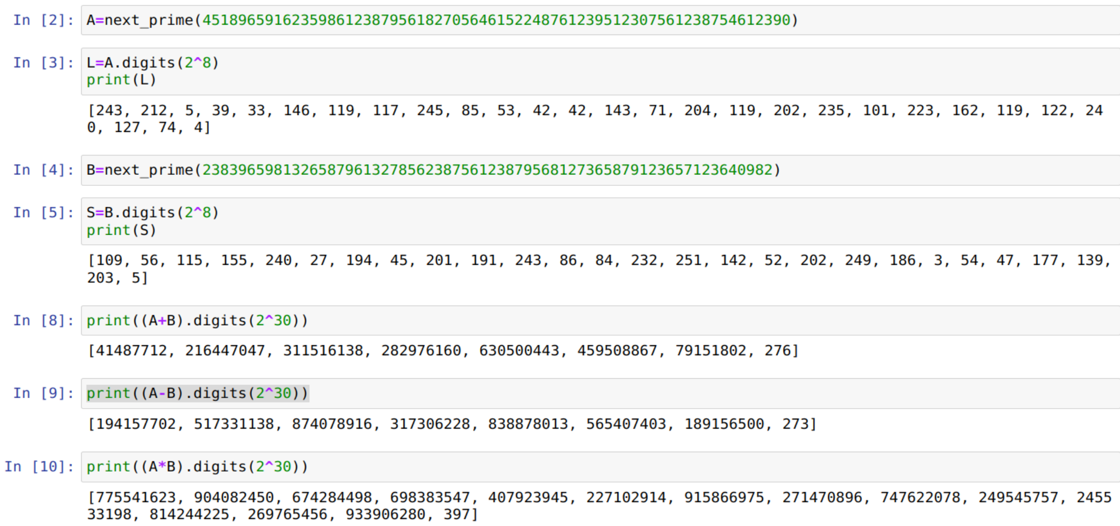
If we write z in this form then we only required 3 multiplications instead of 4. For implementation purpose it is easy to implement karatsuba using recursion. But in cryptology implementation we try to avoid recursion. So if someone want to implement karatsuba then they have to break the blocks according to their inputs.

3.4 Code

Combine the three function I verified the result of my functions.

```
    int main(){
unsigned char B[32]={243, 212, 5, 39, 33, 146, 119, 117, 245, 85, 53,
unsigned char A[32]={109, 56, 115, 155, 240, 27, 194, 45, 201, 191, 24
long long int R[9],S[9],T[10]={0},R_dash[9]={0},S_dash[9]={0};
long long int w[10]={0},MULTI[18]={0};
int i;
    //base change to 2^30
        change_base(A,R);
        change_base(B,S);
   for(int i=0;i<9;i++)
        R_dash[i]=R[i];
   for(int i=0;i<9;i++)
        S_dash[i]=S[i];
        addition(R_dash,S_dash,T);
        printf("\naddition\n");
     //after addition result
        for(i=0;i<9;i++){
                printf("%lld \t", T[i]);
        }
        printf("\nsubstraction:::%d\n",Is_a_BiggerThan_b(R,S));
     //after substraction result
     if(Is_a_BiggerThan_b(R,S)==1)
                mod_subtract(R_dash,S_dash,w);
     if(Is_a_BiggerThan_b(R,S)==-1)
                mod_subtract(S_dash,R_dash,w);
                //further we can add the prime factor while needed
     //after substraction result
        for(i=0;i<9;i++){
                printf("%lld \t", w[i]);
        }
        multi(R,S,MULTI);
        printf("\nmultiplication\n");
     //after multiplication result
        for(i=0;i<18;i++){
                printf("%lld \t", MULTI[i]);
        }
}
```

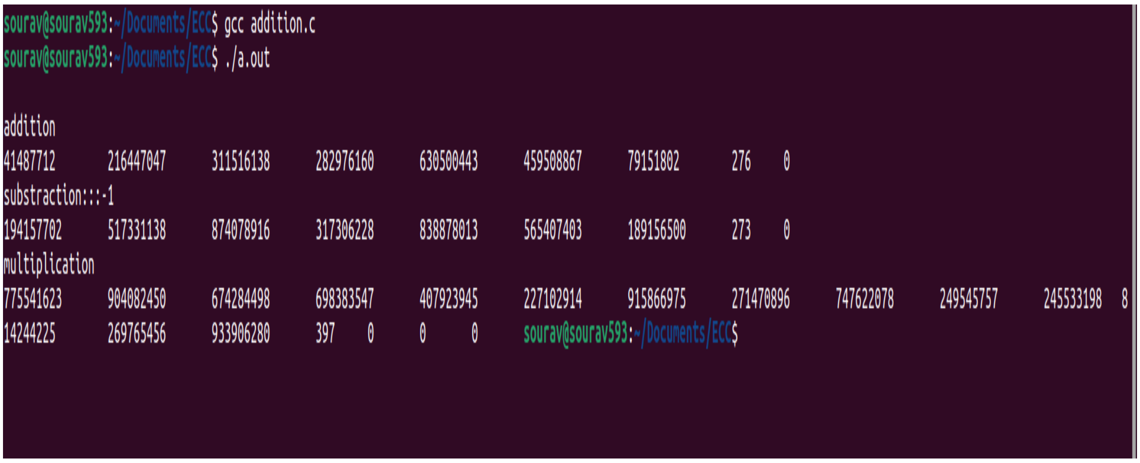
3.4.1 Sage Output



```
In [2]: A=next_prime(451896591623598612387956182705646152248761239512307561238754612390)

In [3]: L=A.digits(2^8)
        print(L)

        [243, 212, 5, 39, 33, 146, 119, 117, 245, 85, 53, 42, 42, 143, 71, 204, 119, 202, 235, 101, 223, 162, 119, 122, 24
        0, 127, 74, 4]

In [4]: B=next_prime(238396598132658796132785623875612387956812736587912365712364098 2)

In [5]: S=B.digits(2^8)
        print(S)

        [109, 56, 115, 155, 240, 27, 194, 45, 201, 191, 243, 86, 84, 232, 251, 142, 52, 202, 249, 186, 3, 54, 47, 177, 139,
        203, 5]

In [8]: print((A+B).digits(2^30))

        [41487712, 216447047, 311516138, 282976160, 630500443, 459508867, 79151802, 276]

In [9]: print((A-B).digits(2^30))

        [194157702, 517331138, 874078916, 317306228, 838878013, 565407403, 189156500, 273]

In [10]: print((A*B).digits(2^30))

        [775541623, 904082450, 674284498, 698383547, 407923945, 227102914, 915866975, 271470896, 747622078, 249545757, 2455
        33198, 814244225, 269765456, 933906280, 397]
```

Figure 5: output of Sage of Operation

3.4.2 C Code Output



```
sourav@sourav593:~/Documents/ECC$ gcc addition.c
sourav@sourav593:~/Documents/ECC$ ./a.out

addition
41487712    216447047    311516138    282976160    630500443    459508867    79151802    276    0
substraction:::-1
194157702   517331138    874078916    317306228    838878013    565407403    189156500   273    0
multiplication
775541623   904082450    674284498    698383547    407923945    227102914    915866975   271470896   747622078   249545757   245533198   8
14244225    269765456    933906280    397    0    0    0    sourav@sourav593:~/Documents/ECC$
```

Figure 6: output of C code of elementary function

Here I compared the result and they are matching. I tried for 2 or 3 more numbers also they are working fine. So good to go for the next steps.

4 Barrett Reduction

Barrett Reduction helped me to calculate values in mod p.

```
void barret(long long int *r,long long int *R){
    int i;
    //p=0xffffffff00000001000000000000000000000000
0ffffffffffffffffffffffff ;; k=256//30 ;; B=2^30
    long long int p[10]={1073741823, 1073741823,
1073741823, 63, 0, 0, 4096, 1073725440, 65535}; //calculated in sage
    long long int T[10]={805306368, 0, 0, 1073741820, 1073741807,
1073741759, 1073741567, 1073741823, 4095, 16384};
// T=B^2k/p calculated in sage
    long long int Q[10]={0},Q1[20]={0},Q2[10]={0},QP[20]={0};
    //Q=[x/B^k-1]=[r/2^240]
    for(i=0;i<10;i++)
        Q[i]=r[8+i];
    multi(Q,T,Q1,10);
    //Q2 <- Q1/B^k+1=Q/2^300
    for(i=0;i<10;i++)
        Q2[i]=Q1[10+i];
    multi(Q2,p,QP,10);
    long long int r_dash[10]={0},QP_dash[10]={0};
    for(i=0;i<10;i++){
        r_dash[i]=r[i]; //r mod2^300
        QP_dash[i]=QP[i]; //QP mod2^300
    }
    if(Is_a_BiggerThan_b(r_dash,QP_dash,10)==1)
        mod_subtract(r_dash,QP_dash,R,10);
    if(Is_a_BiggerThan_b(r_dash,QP_dash,10)==-1){
        long long int thikthik[10]={0};
        mod_subtract(QP_dash,r_dash,thikthik,10);
        mod_subtract(p,thikthik,R,10);
        }
    while(1==Is_a_BiggerThan_b(R,p,10)){
        long long int te[10]={0};
        mod_subtract(R,p,te,10);
        for(i=0;i<10;i++)
            R[i]=te[i];
        }
}
```

4.0.1 Sage Output

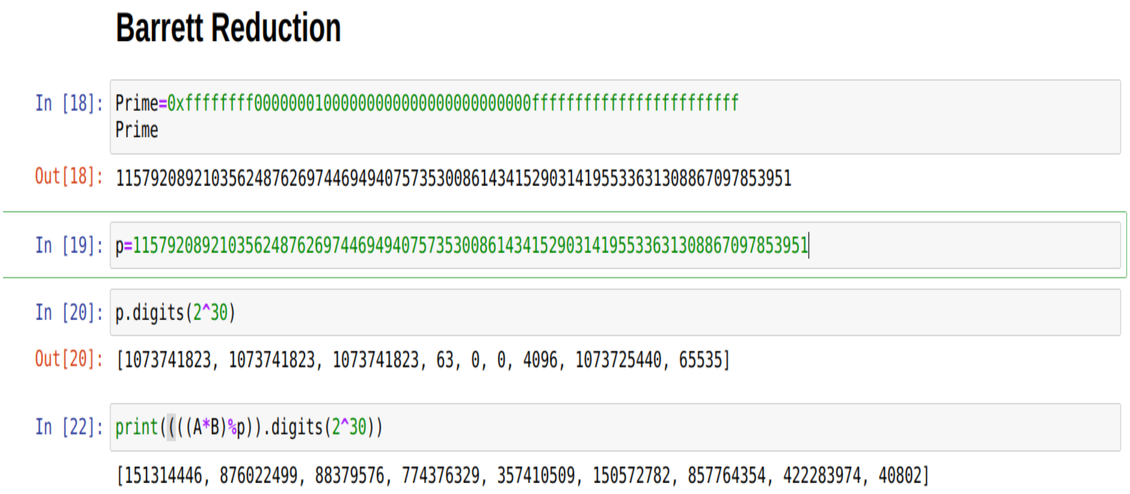


Figure 7: output of Sage for Barrett

4.0.2 C Code Output

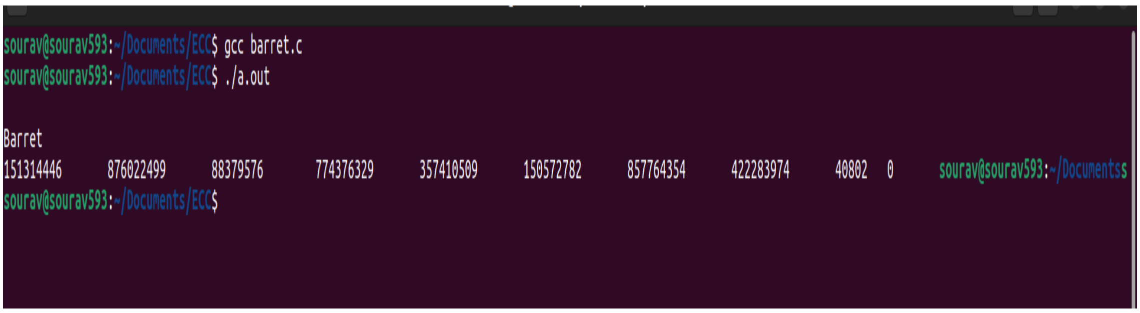


Figure 8: output of C code for Barrett

Here I have taken the prime from 256 bit prime field Weierstrass curve.I proceed according to barrett algorithm: Here B=2^{30},k=$\lceil 256/30 \rceil$=9,T=$\lceil B^{2k}/p \rceil$ $=2^{540}/p$, Input=x(512 bit),Q=$\lceil x/B^{k-1} \rceil = x/2^{240}$;
Then Q1=Q*T;Q2=Q1/2^{300};R=x-Q2*p(mod B^{k+1})=x-Q2*p(mod 2^{300});
Next is run a while loop to decrement R by p until $R < p$.

5 Square & Multiply

Briefly square and multiply method for calculating x^a (mod p) is start with z=1, then according to their bit representation of a from the side of MSB square in every iteration and multiply with x if bit=1. One can also check the correctness of the algorithm by checking for any number a, a^{p-1}=1 or not.

```
void square_multiply(long long int *x, long long int *a,
long long int *result){
        long long int zid[9]={1},temp[18]={0},temp1[18]={0};
        for(int i=8;i>=0;i--){
                for(int j=29;j>=0;j--){
                        for(int k=0;k<18;k++)
                                temp[k]=0;
                        multi(zid,zid,temp,9);
                        for(int k=0;k<9;k++)
                                zid[k]=0;
                        barret(temp,zid);
                        if((a[i]>>j) & 1){
                                for(int k=0;k<18;k++)
                                        temp1[k]=0;
                                multi(zid,x,temp1,9);
                                for(int k=0;k<9;k++)
                                        zid[k]=0;
                                barret(temp1,zid);
                        }}}
        for(int i=0;i<9;i++)
                result[i]=zid[i];
}
```

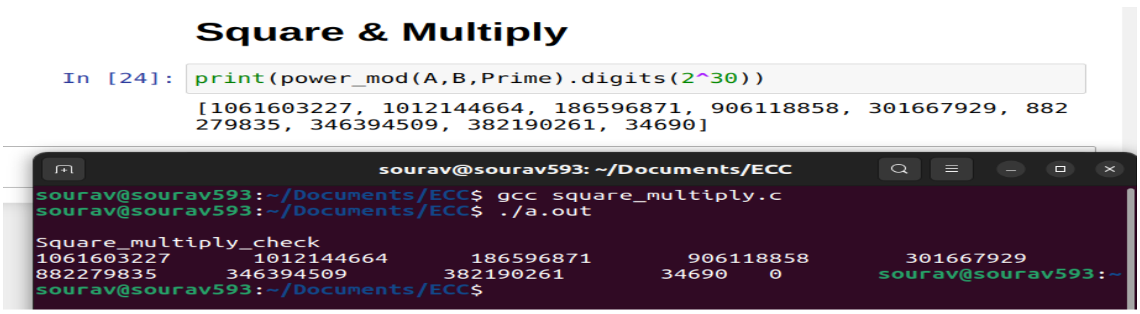


Figure 9: output of Square and Multiply

6 ECC ADDITION

Eliptic Curve Addition Algorithm

$$E := Y^2 = X^3 + AX + B$$

be an eliptic curve and let P1 & P2 be two point on E.

- Input $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$
- Output $P_1 + P_2 = P_3 = (x_3, y_3)$
- if $P_1 = 0$ then $P_1 + P_2 = P_2$
- if $P_2 = 0$ then $P_1 + P_2 = P_1$
- If $x_1 = x_2$, $y_1 = -y_2$ then $P_1 + P_2 = 0$
- Define $\lambda = (y_2 - y_1)/(x_2 - x_1)$ if $P1 \neq P2$
- if P1=P2, $\lambda = (3x^2 + p - 3)/2y_1$

$x_3 = \lambda^2 - x_1$ - x_2
$y_3 = \lambda(x_1$ - $x_3)$-y_1
Here I just implemented the above algorithm line by line and have taken help from sagemath to calculate powers and verify the result. I am showing the important part of calculating λ.

```
if((Is_a_BiggerThan_b(a_x,b_x,9)==0) &&
(Is_a_BiggerThan_b(a_y,b_y,9)==0)){
            multi(a_x,a_x,e1,9);
            barret(e1,x_sq);
            multi(x_sq,three,three_x,9);
            barret(three_x,three_x_barret);
            addition(three_x_barret,p_3,num,9);

            multi(two,a_y,den,9);
            barret(den,den_bar);
            square_multiply(den_bar,p_2,sol);
            multi(sol,num,lamda,9);
    }
    else{
            if(Is_a_BiggerThan_b(b_y,a_y,9)==1)
                    mod_subtract(b_y,a_y,y,9);
            if(Is_a_BiggerThan_b(b_y,a_y,9)==-1){
                    long long int thik[9]={0};
```

```
            mod_subtract(a_y,b_y,thik,9);
            mod_subtract(p,thik,y,9);
}
if(Is_a_BiggerThan_b(b_x,a_x,9)==1)
            mod_subtract(b_x,a_x,x,9);
if(Is_a_BiggerThan_b(b_x,a_x,9)==-1){
            long long int thik1[9]={0};
            mod_subtract(a_x,b_x,thik1,9);
            mod_subtract(p,thik1,x,9);

}
square_multiply(x,p_2,x_inv);
multi(x_inv,y,lamda,9);
```

ECC ADDITION

```
In [29]: ###### NIST P-256
p256 = 2^256-2^224+2^192+2^96-1
a256 = p256 - 3
b256 = 41058363725152142129326129780047268409114441015993725554835256314039467401291
## Base point
gx = 48439561293906451759052585252797914202762949526041747995844080717082404635286
gy = 36134250956749795798585127919587881956611106672985015071877198253568414405109
## Curve order
qq = 115792089210356248762697446949407573529996955224135760342422259061068512044369
FF = GF(p256)
EC = EllipticCurve([FF(a256), FF(b256)])
EC.set_order(qq)
# Base point
G = EC(FF(gx), FF(gy))
## Alice's private key
a = 545456567897987
## Alice's public key
A = a*G
print (A)

(85843274658334699305043628116802730568687465077193738908167644400996701448582 : 112718582919972684608820334688657393631572712981800368689363891854254539376747 : 1)

In [30]: print(85843274658334699305043628116802730568687465077193738908167644400996701448582.digits(2^30))

[822618502, 839769931, 1026660504, 406381500, 813785078, 904312073, 299183961, 613803269, 48585]

In [31]: print(112718582919972684608820334688657393631572712981800368689363891854254539376747.digits(2^30))

[413763691, 1072626329, 790597169, 524569649, 465465364, 885174302, 230441977, 490776745, 63796]
```

Figure 10: ECC

```
In [26]: b=54545656789798986
         b*G

Out[26]: (24488783781291031289044693414742267011758631797059251508710471768703341781691 : 40226669629065347600844197469478495199919812062201795325281959960505552766687 : 1)

In [33]: print(24488783781291031289044693414742267011758631797059251508710471768703341781691.digits(2^30))

         [850223803, 164210561, 32244799, 704277377, 1017435543, 478851327, 879500052, 172265376, 13860]

In [34]: print(40226669629065347600844197469478495199919812062201795325281959960505552766687.digits(2^30))

         [616859359, 1031785013, 520361210, 738402539, 162460059, 22895108, 692511935, 524158533, 22767]

In [27]: c=a+b

In [28]: c*G

Out[28]: (424276640822084966573446071852479557359529511786210742483291313821172333436 : 26905321244620271162745029002783338378226335762420493407339252786313472686587 : 1)

In [35]: print(424276640822084966573446071852479557359529511786210742483291313821172333436.digits(2^30))

         [881818492, 1023820762, 820256646, 659935734, 739752735, 682811376, 696138780, 141807735, 240]

In [36]: print(26905321244620271162745029002783338378226335762420493407339252786313472686587.digits(2^30))

         [426424827, 478459199, 182449024, 625667172, 261989006, 461626777, 232341110, 936484334, 15227]
```

Figure 11: Ecc points

I have taken this points from sagemath to do the calculations further and in c i got the correct result. I have already implemented addition function

```
sourav@sourav593: ~/Documents/ECC
sourav@sourav593:~/Documents/ECC$ gcc ecc_addition.c
sourav@sourav593:~/Documents/ECC$ ./a.out

x3
881818492	1023820762	820256646	659935734	739752735	682811376	696138780	141807735	240
y3
426424827	478459199	182449024	625667172	261989006	461626777	232341110	936484334	15227	sourav@sourav593:~/Documents/ECC$
```

Figure 12: ecc addition c output

for general case. So I can use this thing to implement doubling. That is for 2*x=x+x.

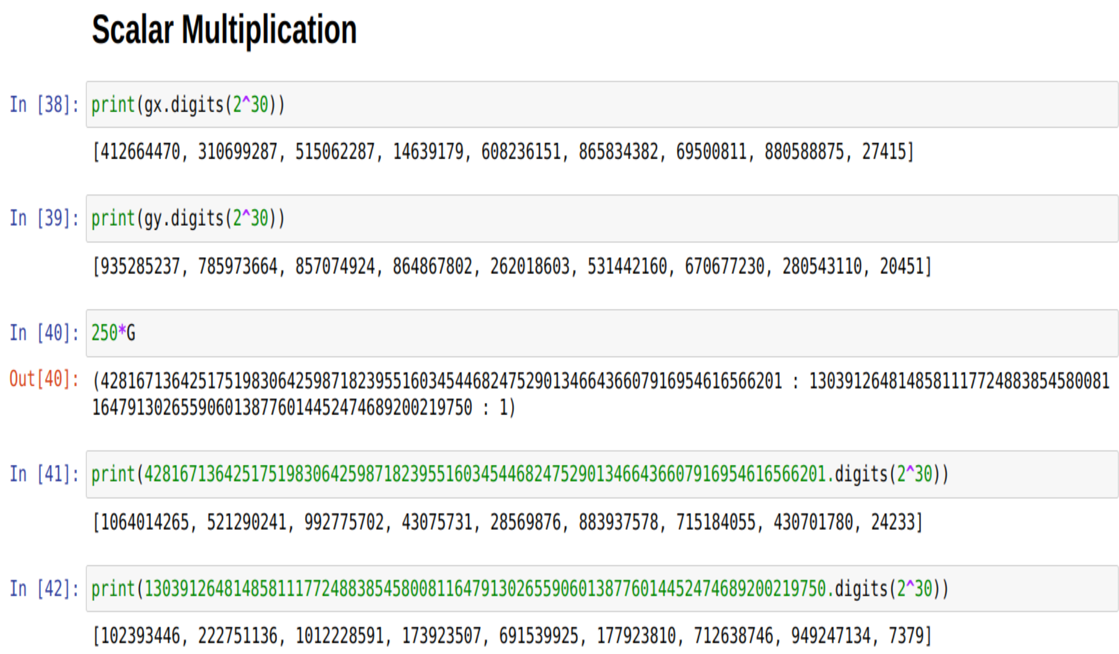
7 Ecc Scalar Multiplication

Just like square and multiply I implemented this ecc scalar multiplication also. To calculate a*G the difference from square and multiply is here we have to start with 0 and then for every bit we have to double the number and if the bit is 1 then we have to add G. We have to be carefull regarding the starting step I have set a flag to start the process only after i got a bit=1.

```
void ecc_scalar_multiply(long long int *x, long long int *y,
long long int *c, long long int *a, long long int *b){
    long long int z1[9]={0},z2[9]={0},temp[9]={0},
temp1[9]={0},sou1[9]={0},sou2[9]={0};
    int flag=0,i,j,B,k;
    for(i=0;i<9;i++){
        z1[i]=x[i];
        z2[i]=y[i];
    }
    for(i=8;i>=0;i--){
        for(j=29;j>=0;j--){
            B=(c[i]>>j) & 1;
            printf("\nbit %d\n",B);
            if(flag==1){
                ecc_addition(z1,z1,z2,z2,temp,temp1);
                for(k=0;k<9;k++){
                    z1[k]=temp[k];
                    z2[k]=temp1[k];

                    temp[k]=0;
                    temp1[k]=0;
                }
                if(B==1){
                    ecc_addition(z1,x,z2,y,sou1,so
                    for(k=0;k<9;k++){
                        z1[k]=sou1[k];
                        z2[k]=sou2[k];

                        sou1[k]=0;
                        sou2[k]=0;
                        }}}
            if(B==1)
                flag=1;
```

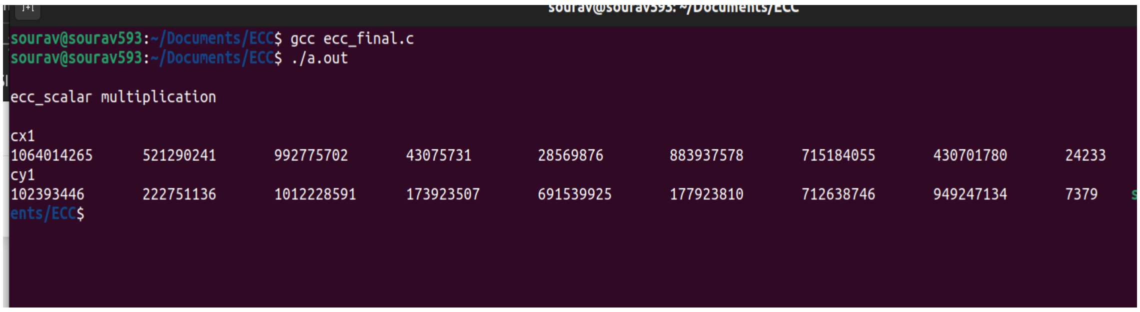
```
			}
		}
		for(i=0;i<9;i++){
			a[i]=z1[i];
			b[i]=z2[i];
		}
}
```

Like the previous cases I can verify this using sagemath.



```
Scalar Multiplication

In [38]: print(gx.digits(2^30))
[412664470, 310699287, 515062287, 14639179, 608236151, 865834382, 69500811, 880588875, 27415]

In [39]: print(gy.digits(2^30))
[935285237, 785973664, 857074924, 864867802, 262018603, 531442160, 670677230, 280543110, 20451]

In [40]: 250*G
Out[40]: (42816713642517519830642598718239551603454468247529013466436607916954616566201 : 1303912648148581117724883854580081
1647913026559060138776014452474689200219750 : 1)

In [41]: print(42816713642517519830642598718239551603454468247529013466436607916954616566201.digits(2^30))
[1064014265, 521290241, 992775702, 43075731, 28569876, 883937578, 715184055, 430701780, 24233]

In [42]: print(13039126481485811177248838545800811647913026559060138776014452474689200219750.digits(2^30))
[102393446, 222751136, 1012228591, 173923507, 691539925, 177923810, 712638746, 949247134, 7379]
```

Figure 13: Ecc scalar multiply sage output



```
sourav@sourav593:~/Documents/ECC$ gcc ecc_final.c
sourav@sourav593:~/Documents/ECC$ ./a.out

ecc_scalar multiplication

cx1
1064014265	521290241	992775702	43075731	28569876	883937578	715184055	430701780	24233
cy1
102393446	222751136	1012228591	173923507	691539925	177923810	712638746	949247134	7379
ents/ECC$
```

Figure 14: Ecc scalar multiply C output

8 Elliptic Curve Diffie–Hellman Key Exchange

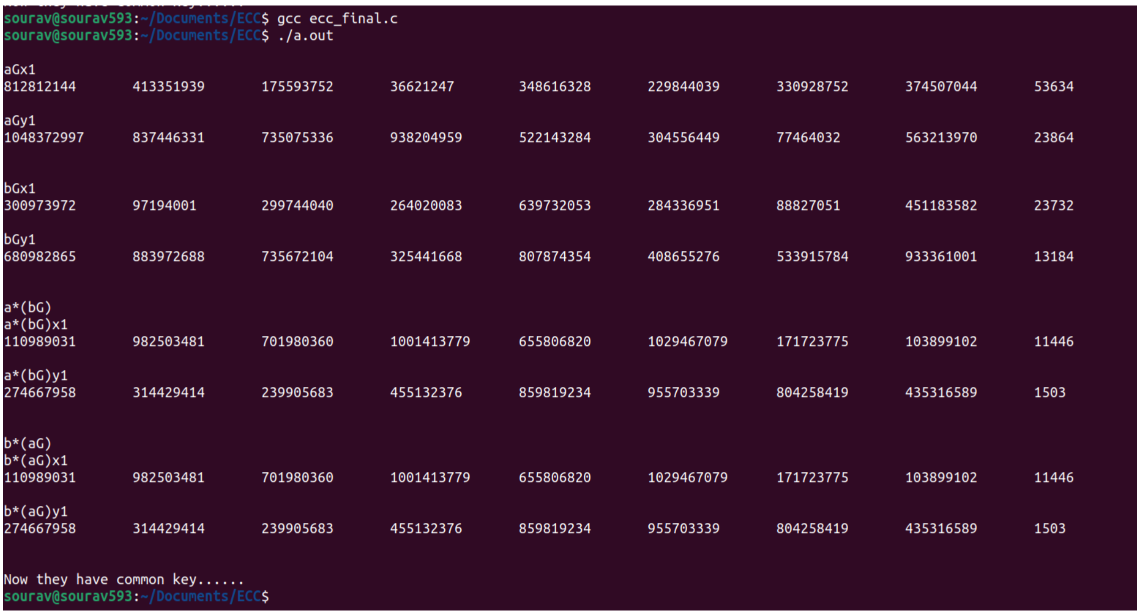
Now I have all the function to do key exchange.

(a * G) * b = (b * G) * a

In the program I first calculated a*G then b*(a*G). Similarly for the other party. Then they have the common key.

```
int main(){
    unsigned char r[32]={243, 212, 5, 39, 33, 146, 119, 117, 245,
71, 204, 119, 202, 235, 101, 223, 162, 119, 122, 240, 127, 74, 4},s[32
84, 232, 251, 142, 52, 202, 249, 186, 3, 54, 47, 177, 139, 203, 5};
    long long int a[9]={0},b[9]={0};
    //base change to 2^30
    change_base(r,a);
    change_base(s,b);
    int i;
    long long int gx[9]={412664470, 310699287, 515062287, 14639179
    long long int aG_x[9]={0},aG_y[9]={0};
    //calculating aG
    ecc_scalar_multiply(gx,gy,a,aG_x,aG_y);
    printf("\naGx1\n");
    print_array(aG_x,9);
    printf("\naGy1\n");
    print_array(aG_y,9);
    printf("\n");
    //calculating bG
    long long int bG_x[9]={0},bG_y[9]={0};
    ecc_scalar_multiply(gx,gy,b,bG_x,bG_y);
    printf("\nbGx1\n");
    print_array(bG_x,9);
    printf("\nbGy1\n");
    print_array(bG_y,9);
    printf("\n");
    //calculate a*(bG)
    printf("\na*(bG)");
    long long int aBG_x[9]={0},aBG_y[9]={0};
    ecc_scalar_multiply(bG_x,bG_y,a,aBG_x,aBG_y);
    printf("\na*(bG)x1\n");
    print_array(aBG_x,9);
    printf("\na*(bG)y1\n");
    print_array(aBG_y,9);
```

```
        printf("\n");
        //calculate b*(aG)
        printf("\nb*(aG)");
        long long int bAG_x[9]={0},bAG_y[9]={0};
        ecc_scalar_multiply(aG_x,aG_y,b,bAG_x,bAG_y);
        printf("\nb*(aG)x1\n");
        print_array(bAG_x,9);
        printf("\nb*(aG)y1\n");
        print_array(bAG_y,9);
        printf("\n");
        if(Is_a_BiggerThan_b(bAG_x,aBG_x,9)==0 && Is_a_BiggerThan_b(bA
                printf("\nNow they have common key......\n");
        else
                printf("\nError in key exchange.........\n");


}
```



```
sourav@sourav593:~/Documents/ECC$ gcc ecc_final.c
sourav@sourav593:~/Documents/ECC$ ./a.out

aGx1
812812144	413351939	175593752	36621247	348616328	229844039	330928752	374507044	53634

aGy1
1048372997	837446331	735075336	938204959	522143284	304556449	77464032	563213970	23864


bGx1
300973972	97194001	299744040	264020083	639732053	284336951	88827051	451183582	23732

bGy1
680982865	883972688	735672104	325441668	807874354	408655276	533915784	933361001	13184


a*(bG)
a*(bG)x1
110989031	982503481	701980360	1001413779	655806820	1029467079	171723775	103899102	11446

a*(bG)y1
274667958	314429414	239905683	455132376	859819234	955703339	804258419	435316589	1503


b*(aG)
b*(aG)x1
110989031	982503481	701980360	1001413779	655806820	1029467079	171723775	103899102	11446

b*(aG)y1
274667958	314429414	239905683	455132376	859819234	955703339	804258419	435316589	1503

Now they have common key......
sourav@sourav593:~/Documents/ECC$
```

Figure 15: ecc addition c output