Elliptic Curve Diffie-Hellman(ECDH) KE Protocol

Implementation in C

Cryptographic and Security Implementations

T. K. Garai

Indian Statistical Institute

Elliptic Curve Diffie-Hellman(ECDH) KE Protocol

Implementation in C

by

Student Name	Roll Number
Tamas Kanti Garai	CrS2116

Course:	M.Tech
Depertment:	Cryptology and Security
Instructor:	**Dr. Sabyasachi Karati**
Project Duration:	August-September, 2022

Contents

List of Figures

1 Overview

Elliptic Curve Cryptography (ECC) is the most recent member of the three families of established public-key algorithms of practical relevance, the other two are Integer-Factorization Scheme like RSA and Discrete Logarithm Scheme like Diffie-Hellman key exchange or Elgamal encryption.[1]

ECC provides the same level of security as RSA or discrete logarithm systems with considerably shorter operands (approximately $160 - 256$ bit vs. $1024 - 3072$ bit). ECC is based on the generalized descrete logarithm problem and thus DL-protocol such as Diffie-Hellman key exchange can also be realized using elliptic curves. In many cases, ECC has performance advantages (fewer computations) and bandwidth advantages (shorter signatures and keys) over RSA and DL schemes. However RSA operations which involves shorter public keys are still much faster than ECC operations.

The mathematics of elliptic curves are considerably more involved than those of RSA and DL schemes. But those deatails are not our concern in this project. We mainly focus on the implementation part of Elliptic Curve. We will implement the NIST standard Elliptic Curve $p - 256$ at the first part of the project. Then we implement Diffie-Hellman Key Exchange Protocol on our implemented $p - 256$.

To implement $p - 256$ we have to start by implementing big integer arithmetic, since that would the most basic operation. Then we will have to implement the modular arithmetic. We will use Barrett reduction for that. To find inverse of an element in $\mathbb{F}_p$ we will implement square and multiply algorithm.

At last we will implement the addition of two points and doubling of a point on elliptic curve. Then using these two implementation we will implement scalar multiplication on a point of elliptic curve using analogous version of square and multiply algorithm, say adding and doubling algorithm.

The whole project repository can be found in the link: https://github.com/tkgarai/ECDH.git

2

Big Integer Arithmetic

From the documentation of $p - 256$ we know that the prime p considered in this elliptic curve is 256-bit long. So we have to implement addition subtraction multiplication of 256-bit numbers in this project.

2.1. Representation

We will implement the project in c programming and it does not support any data type of more than 64 bit. Thats why we need to think about the representation of 256-bit number. We need to take the base of the number in a way that will help us reducing the number of operations needed for basic addition subtraction and multiplication. Mainly we will focus on reducing number of multiplication since that is the most expensive operation. Also multiplication of two n-bit number gives 2n-bit number. So we cannot take the base more than 2^{32} since multiplying 2 more than 32-bit numbers gives rise to a more than 64-bit number, and we cannot store a number of more than 64 bit in c. That is why we will take the base to be 2^{30} so that we can store the number easily and no overflow ever affect our result.

Let n is the integer and B is the base of our integer representation, then we will have $log_B n = l$ bits in our B base representation of n , so it can be written in the form:

$$n = a_{l-1}B^{l-1} + a_{l-2}B^{l-2} + ... + a_1 B + a_0$$

for unique a_i where $0 \le a_i < B$ and $0 \le i < l$.

2.2. Addition

Suppose a and b are two 256-bit integers. By the above representation we have:

$$a = a_{l-1}B^{l-1} + a_{l-2}B^{l-2} + ... + a_1 B + a_0$$

and

$$b = b_{l-1}B^{l-1} + b_{l-2}B^{l-2} + ... + b_1 B + b_0$$

where $0 \le a_i, b_i < B$, $0 \le i < l$ and $l = 10$, $B = 2^{30}$. To obtain $c = a + b$ we can perform coefficient wise polynomial addition. But in that case sometimes $c_i = a_i + b_i > B$. To handle this case we need to consider the carry to the $(i+1)^{th}$-bit. So after adding a and b we can get a polynomial of degree at most l.

Implementation

```
void add(long long int *a, long long int *b, long long int *c)
{
    for (int i = 0; i < 9; i++)
    {
        c[i] += (a[i] + b[i]);//adding corresponding limbs
        c[i + 1] = (c[i] >> 30) & 1;//checking for carry
        c[i] = c[i] & 0x3fffffff;//putting last 30 bits in the result
    }
}
```

2.3. Subtraction

My implementation of subtraction can only subtract smaller number from a larger number. We did the same thing as addition here,i.e., coefficient wise subtraction. But here we have to check each coefficient after subtraction if they become negative.

To obtain $c = a - b$ (where of course $a \geq b$ in my implementation), we can perform coefficient wise polynomial subtraction. But in that case sometimes $c_i = a_i - b_i < 0$. To handle this case we need to borrow 1 from $(i+1)^{th}$-bit of a.

Implementation

```
void subt(long long int *a, long long int *b, long long int *c, int limb)//number of limbs to
     represent the bigger input is taken on "limb"
{
    for (int i = 0; i < limbs; i++)
    {
        c[i] = a[i] - b[i];//subtracting corresponding limbs
        if ((c[i] >> 63) & 1)//checking if the corresponding limb negative
        {
            c[i] += (1 << 30);//if corresponding limb is negative adding 2^30 with it
            a[i + 1] -= 1;//and taking 1 borrow from next limb
        }
    }
}
```

2.4. Multiplication

Now we have to implement multiplication of two 10 limb number in 2^{30} base. For that we have two different procedure. One is schoolbook multiplication which is quite straightforward and the other is Karatsuba algorithm, which is asymptotically faster than the general schoolbook one.

$X \leftarrow$ n limb number in 2^{30} base;
$X \leftarrow$ n limb number in 2^{30} base;
if $n = 1$ **then**
  $P \leftarrow X * Y$
end
else
  split X and Y in half;
  $X =: B^{n/2}X_1 + X_2$;
  $Y =: B^{n/2}Y_1 + Y_2$;
  $U \leftarrow karatsuba(X_1, Y_1)$;
  $V \leftarrow karatsuba(X_2, Y_2)$;
  $W \leftarrow karatsuba((X_1 + X_2), (Y_1 + Y_2))$;
  $Z \leftarrow (W - U - V)$;
  $P \leftarrow B^n U + B^{n/2} Z + W$;
end
return P;

Algorithm 1: Karatsuba Multiplication Algorithm

To implement Karatsuba in 10 limb, we have to break the number in two parts, one of 2 limbs and the other of 8 limbs. Then apply Karatsuba algorithm on 2 limbs directly and to calculate the 8 limb part we use 4 limb Karatsuba and then again 2 limb Karatsuba to calculate the 4 limb multiplication.

Implementation

```
void mult(long long int *a, long long int *b, long long int *c)
{
    int i;
    long long int a0[2] = {0}, b0[2] = {0}, c0[16] = {0};
    long long int a1_dash[8] = {0}, b1_dash[8] = {0}, c1_dash[16] = {0};
```

```
    long long int a2[8] = {0}, b2[8] = {0}, c2[16] = {0};
    long long int c1_dash_dash[16] = {0}, c1[16] = {0};
    for (i = 0; i < 2; i++)
    {
        a0[i] = a[i];
        b0[i] = b[i];
    }
    for (i = 2; i < 10; i++)
    {
        a2[i - 2] = a[i];
        b2[i - 2] = b[i];
    }
    for (i = 0; i < 2; i++)
    {
        a1_dash[i] = a0[i] + a2[i];
        b1_dash[i] = b0[i] + b2[i];
    }
    for (i = 2; i < 8; i++)
    {
        a1_dash[i] = a2[i];
        b1_dash[i] = b2[i];
    }
    karatsuba_2limb(a0, b0, c0);
    karatsuba_8limb(a1_dash, b1_dash, c1_dash);
    karatsuba_8limb(a2, b2, c2);
    subt(c1_dash, c0, c1_dash_dash, 16);
    subt(c1_dash_dash, c2, c1, 16);
    c[0] = c0[0];
    c[1] = c0[1];
    c[2] = c0[2] + c1[0];
    c[3] = c0[3] + c1[1];
    c[4] = c1[2] + c2[0];
    c[5] = c1[3] + c2[1];
    c[6] = c1[4] + c2[2];
    c[7] = c1[5] + c2[3];
    c[8] = c1[6] + c2[4];
    c[9] = c1[7] + c2[5];
    c[10] = c1[8] + c2[6];
    c[11] = c1[9] + c2[7];
    c[12] = c1[10] + c2[8];
    c[13] = c1[11] + c2[9];
    c[14] = c1[12] + c2[10];
    c[15] = c1[13] + c2[11];
    c[16] = c1[14] + c2[12];
    c[17] = c1[15] + c2[13];
    c[18] = c2[14];
    c[19] = c2[15];
    for (i = 0; i < 20; i++)
    {
        c[i + 1] += (c[i] >> 30);
        c[i] = c[i] & 0x3fffffff;
    }
}
```

3

Modular Arithmetic

To implement p-256 we need the addition subtraction and multiplication in $\mathbb{F}_p$ where the p is specified as 115792089210356248762697446949407573530086143415290314195533631308867097853951 in the NIST documentation. For that we will use the very famous algorithm Barrett reduction. Then to find the inverse of an element we will implement square and multiply algorithm.

3.1. Barrett Reduction

Introduced by Barrett, this method is based on the idea of fixed point arithmetic. The principle is to estimate the quotient x/m where x has 2n digits and m has n digits with operations that can either be pre-computed or are less expensive than a multi-precision division. The remainder r of x modulo m is equal to $r = x - m\left\lfloor \frac{x}{m} \right\rfloor$. Using the fact that divisions by a power of b , we have:

$$r = x - m\left\lfloor \frac{\frac{x \cdot b^{2n}}{b^{n-1} \cdot m}}{b^{n+1}} \right\rfloor = x - m\left\lfloor \frac{\frac{x \cdot \mu}{b^{n-1}}}{b^{n+1}} \right\rfloor$$

,where $\mu = \left\lfloor \frac{b^{2n}}{m} \right\rfloor$ a pre-calculated value that depends on the modulus. Let $\hat{q}$ be the estimation of the quotient of x/m. Barrett improves further the reduction using only partial multi-precision multiplication when needed. The estimate $\hat{r}$ of the remainder of x modulo m is

$$\hat{r} = (x - m\hat{q}) \mod (b^{n+1})$$

This estimation implies that at most two subtractions of m are required to obtain the correct remainder r. In general, we will take the value of b as 2, but for a general b the procedure is given below.For $x = x_{2n-1}x_{2n-2}....x_0$ and $m = m_{n-1}m_{n-2}m_0$ and first we calculate $\mu = \left\lfloor \frac{b^{2n}}{m} \right\rfloor$

Implementation

```
void barrett(long long int *x, long long int *reduced_x)
{
    for (int i = 0; i < 10; i++)
        reduced_x[i] = 0;
    int i;
    long long int p[10] = {1073741823, 1073741823, 1073741823, 63, 0, 0, 4096, 1073725440,
        65535, 0};
    long long int T[10] = {805306368, 0, 0, 1073741820, 1073741807, 1073741759, 1073741567,
        1073741823, 4095, 16384};//precomputed T value
    long long int q0[10] = {0}, q1[20] = {0}, q2[10] = {0}, qp[20] = {0}, r[11] = {0}, r1[11]
         = {0}, r2[11] = {0};
    long long int arr[11] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1};
    for (i = 0; i < 10; i++)
        q0[i] = x[i + 8];
    mult(q0, T, q1);
    for (i = 0; i < 10; i++)
```

Data: $x = (x_{2n-1}, x_{2n-2}, ..., x_0)_b, m = (m_{n-1}, m_{n-2}, ..., m_0)_b, \mu = \left\lfloor \frac{b^{2n}}{m} \right\rfloor$
Result: $x \mod n$
$q_1 \leftarrow \lfloor x/b^{n-1} \rfloor;$
$q_2 \leftarrow \mu q_1;$
$q_3 \leftarrow \lfloor q_2/b^{n+1} \rfloor;$
$r_1 \leftarrow x \mod b^{n+1};$
$r_2 \leftarrow m q_3 \mod b^{n+1};$
$r \leftarrow (r_1 - r_2) \mod (b^{n+1});$
while $r \geq m$ **do**
  $r \leftarrow r - m;$
end
return r;

Algorithm 2: Barrett Reduction Algorithm

```
        q2[i] = q1[i + 10];
    mult(q2, p, qp);
    if (compare(x, qp) == 1)
        subt(x, qp, r1, 10);
    else
    {
        subt(qp, x, r, 10);
        subt(arr, r, r1, 11);
    }
    while (compare(r1, p) == 1)
    {
        subt(r1, p, r2, 10);
        for (int j = 0; j < 10; j++)
        {
            r1[j] = r2[j];
            r2[j] = 0;
        }
    }
    for (int j = 0; j < 10; j++)
        reduced_x[j] = r1[j];
}
```

3.2. Square and Multiply Algorithm

Data: $x, c = (c_{l-1}, c_{l-2}, ..., c_0), n$
Result: $x^c \mod n$
$z \leftarrow 1;$
for $i \leftarrow (l-1)$ *to* 0 **do**
  $z \leftarrow z^2 \mod n;$
  if $c_i = 1$ **then**
    $z \leftarrow z * x \mod n$
  end
end
return z;

Algorithm 3: Square and Multiply Algorithm

3.2.1. Inverse of an element

Fermat's Little Theorem: If p is a prime and a is any integer not divisible by p, then $a^{p-1} - 1$ is divisible by p, i.e, $a^{p-1} \equiv 1 \mod p$.

So,By Fermat's little theorem for any field of prime order p and for any element x in the field ,the inverse of x is nothing but x^{p-2} because:

$$x * x^{(p-2)} \equiv x^{1+p-2} = x^{p-1} \equiv 1 \mod p$$

So we need to implement square and multiply algorithm to find $a^{p-2} \mod p$ and that will give us the inverse of a.

Implementation

```
void inverse(long long int *a, long long int *a_inv)
{
    int i, j, k;
    long long int p_minus_2[9] = {1073741821, 1073741823, 1073741823, 63, 0, 0, 4096,
        1073725440, 65535};//(p-2)
    long long int x1[10] = {0}, x[10] = {1, 0};
    for (i = 8; i >= 0; i--)
    {
        for (j = 0; j < 30; j++)
        {
            mult_fp(x, x, x1);//squaring for each bit position
            for (k = 0; k < 10; k++)
            {
                x[k] = x1[k];
                x1[k] = 0;
            }
            if ((p_minus_2[i] >> (29 - j)) & 1)//finding the bit from msb to lsb, if the bit
                is 1 then multiply
            {
                mult_fp(x, a, x1);
                for (k = 0; k < 10; k++)
                {
                    x[k] = x1[k];
                    x1[k] = 0;
                }
            }
        }
    }
    for (k = 0; k < 9; k++)
        a_inv[k] = x[k];
}
```

4

Elliptic Curve Diffie-Hellman

4.1. Elliptic Curve

Definition(Elliptic Curve): The elliptic curve over $\mathbb{F}_p$, $p > 3$, is the set of all pairs $(x, y) \in \mathbb{F}_p$ which fulfill

$$y^2 \equiv x^3 + ax + b \mod p$$

together with an imaginary point of infinity $\mathcal{O}$, where $a, b \in \mathbb{F}_p$ and the condition $4a^3 + 27b^2 \neq 0 \mod p$.

Elliptic Curve Addition Algorithm[2]: Let $E : Y^2 = X^3 + AX + B$ be an elliptic curve and let P_1 and P_2 be points on E.

(a) If $P_1 = \mathcal{O}$, then $P_1 + P_2 = P_2$.
(b) Otherwise, if $P_2 = \mathcal{O}$, then $P_1 + P_2 = P_1$.
(c) Otherwise, write $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$.
(d) If $x_1 = x_2$ and y_1 =-y_2 , then $P_1 + P_2 = \mathcal{O}$.
(e) Otherwise, define λ by

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & P_1 \neq P_2 \\ \frac{3x_1^2 - A}{2y_1} & P_1 = P_2 \end{cases}$$

and let $x_3 = \lambda^2$-x_1-x_2 and $y_3 = \lambda(x_1$-$x_3)$-y_1. Then $P_1 + P_2 = (x_3, y_3)$.

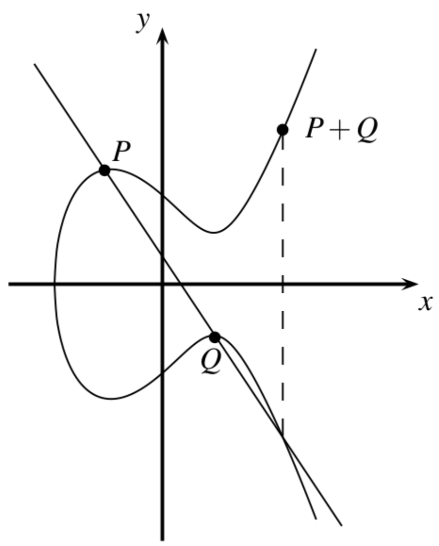

Figure 4.1: Addition of two points on an elliptic curve over real number field

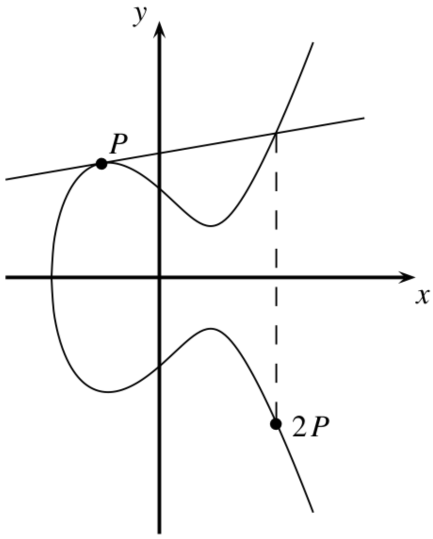

Figure 4.2: Point doubling on an elliptic curve over real number field

Implementation

```c
void pt_add(long long int *x1, long long int *y1, long long int *x2, long long int *y2, long
    long int *p_plus_q_x, long long int *p_plus_q_y)
{//lambda = (y_2 - y_1)/(x_2 - x_1), x_3 = lambda^2 - x_1 - x_2, y_3 = lambda * (x_1 - x_3 )
    - y1
    long long int p[10] = {1073741823, 1073741823, 1073741823, 63, 0, 0, 4096, 1073725440,
        65535, 0};
    long long int subt_y2y1[11] = {0}, subt_y1y2[11] = {0};
    long long int subt_x2x1[11] = {0}, subt_x1x2[11] = {0}, subt_x2x1_inv[11] = {0};
    long long int lambda[10] = {0}, lambda_sq[10] = {0};
    long long int add_x1x2[11] = {0};
    long long int subt_add_x1x2_lambda_sq[10] = {0};
    long long int subt_x1_p_plus_q_x[10] = {0}, subt_p_plus_q_x_x1[10] = {0};
    long long int mult_fp_lambda_subt_x1_p_plus_q_x[10] = {0},
        subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x[11] = {0};
    int l;
    if (compare(y2, y1) == 1)
        subt(y2, y1, subt_y2y1, 9);
    else
    {
        subt(y1, y2, subt_y1y2, 9);
        subt(p, subt_y1y2, subt_y2y1, 10);//(y_2 - y_1)
    }
    if (compare(x2, x1) == 1)
        subt(x2, x1, subt_x2x1, 9);//(x_2 - x_1)
    else
    {
        subt(x1, x2, subt_x1x2, 9);
        subt(p, subt_x1x2, subt_x2x1, 10);
    }
    inverse(subt_x2x1, subt_x2x1_inv);//1/(x_2 - x_1)
    mult_fp(subt_y2y1, subt_x2x1_inv, lambda);//(y_2 - y_1)/(x_2 - x_1)
    mult_fp(lambda, lambda, lambda_sq);//lambda^2
    add_fp(x1, x2, add_x1x2);//(x_1 + x_2)
    if (compare(lambda_sq, add_x1x2) == 1)
        subt(lambda_sq, add_x1x2, p_plus_q_x, 9);//lambda^2 - (x_1 + x_2)
    else
    {
        subt(add_x1x2, lambda_sq, subt_add_x1x2_lambda_sq, 9);
        subt(p, subt_add_x1x2_lambda_sq, p_plus_q_x, 10);
    }
    if (compare(x1, p_plus_q_x) == 1)
        subt(x1, p_plus_q_x, subt_x1_p_plus_q_x, 9);//(x_1 - x_3)
    else
```

```
    {
        subt(p_plus_q_x, x1, subt_p_plus_q_x_x1, 9);
        subt(p, subt_p_plus_q_x_x1, subt_x1_p_plus_q_x, 10);
    }
    mult_fp(lambda, subt_x1_p_plus_q_x, mult_fp_lambda_subt_x1_p_plus_q_x);//lambda * (x_1 -
        x_3)
    if (compare(mult_fp_lambda_subt_x1_p_plus_q_x, y1) == 1)
        subt(mult_fp_lambda_subt_x1_p_plus_q_x, y1, p_plus_q_y, 9);//lambda * (x_1 - x_3) -
            y_1
    else
    {
        subt(y1, mult_fp_lambda_subt_x1_p_plus_q_x, subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x
            , 9);
        subt(p, subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x, p_plus_q_y, 10);
    }
}

void pt_double(long long int *x1, long long int *y1, long long int *p_plus_q_x, long long int
     *p_plus_q_y)
{//lambda = (3 * x_1^2 + A) / (2 * y_1)
    long long int p[10] = {1073741823, 1073741823, 1073741823, 63, 0, 0, 4096, 1073725440,
        65535, 0};
    long long int a[10] = {1073741820, 1073741823, 1073741823, 63, 0, 0, 4096, 1073725440,
        65535, 0};
    long long int add_x1_x1[10] = {0};
    long long int x1_sq[10] = {0};
    long long int three[10] = {3}, mult_fp_three_x1_sq[10] = {0};
    long long int add_mult_fp_three_x1_sq_a[10] = {0};
    long long int two_y1[10] = {0}, two_y1_inv[10] = {0};
    long long int lambda[10] = {0}, lambda_sq[10] = {0};
    long long int add_x1x2[11] = {0};
    long long int subt_add_x1x2_lambda_sq[10] = {0};
    long long int subt_x1_p_plus_q_x[10] = {0}, subt_p_plus_q_x_x1[10] = {0};
    long long int mult_fp_lambda_subt_x1_p_plus_q_x[10] = {0},
        subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x[11] = {0};
    int l;
    add_fp(x1, x1, add_x1_x1);
    mult_fp(x1, x1, x1_sq);
    mult_fp(three, x1_sq, mult_fp_three_x1_sq);
    add_fp(mult_fp_three_x1_sq, a, add_mult_fp_three_x1_sq_a);
    add_fp(y1, y1, two_y1);
    inverse(two_y1, two_y1_inv);
    mult_fp(add_mult_fp_three_x1_sq_a, two_y1_inv, lambda);
    mult_fp(lambda, lambda, lambda_sq);//lambda^2
    add_fp(x1, x1, add_x1x2);//(x_1 + x_1)
    if (compare(lambda_sq, add_x1x2) == 1)
        subt(lambda_sq, add_x1x2, p_plus_q_x, 9);//lambda^2 - (x_1 + x_2)
    else
    {
        subt(add_x1x2, lambda_sq, subt_add_x1x2_lambda_sq, 9);
        subt(p, subt_add_x1x2_lambda_sq, p_plus_q_x, 10);
    }
    if (compare(x1, p_plus_q_x) == 1)
        subt(x1, p_plus_q_x, subt_x1_p_plus_q_x, 9);//(x_1 - x_3)
    else
    {
        subt(p_plus_q_x, x1, subt_p_plus_q_x_x1, 9);
        subt(p, subt_p_plus_q_x_x1, subt_x1_p_plus_q_x, 10);
    }
    mult_fp(lambda, subt_x1_p_plus_q_x, mult_fp_lambda_subt_x1_p_plus_q_x);//lambda * (x_1 -
        x_3)
    if (compare(mult_fp_lambda_subt_x1_p_plus_q_x, y1) == 1)
        subt(mult_fp_lambda_subt_x1_p_plus_q_x, y1, p_plus_q_y, 9);//lambda * (x_1 - x_3) -
            y_1
    else
    {
        subt(y1, mult_fp_lambda_subt_x1_p_plus_q_x, subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x
            , 9);
        subt(p, subt_y1_mult_fp_lambda_subt_x1_p_plus_q_x, p_plus_q_y, 10);
    }
}
```

Scalar Multiplication: Multiplying a point on the Elliptic curve P by a scalar c is actually $c * P = P + P + ... + P$(c times), where the addition is the addition defined on the elliptic curve.

To implement this we can use the addition and doubling function we already implemented in c and do adding and doubling similar to square and multiply algorithm by keeping an eye on the binary representation of the scalar c.

Implementation

```c
void scalar_mult(long long int *G_x, long long int *G_y, long long int *A_x, long long int *
    A_y, long long int *b)
{
    int i, j, k, l, a[2];
    long long int x1[20] = {0}, B_x[10] = {0}, B_y[10] = {0};
    for (i = 0; i < 10; i++)
    {
        A_x[i] = G_x[i];
        A_y[i] = G_y[i];
    }
    first_1(b, a);
    int tk = a[1] + 1;//since first we have to ignore the first one as we are initializing (
        A_x, A_y) with (G_x, G_y)
    for (i = a[0]; i >= 0; i--)
    {
        for (j = tk; j < 30; j++)
        {
            pt_double(A_x, A_y, B_x, B_y);
            for (k = 0; k < 10; k++)
            {
                A_x[k] = B_x[k];
                A_y[k] = B_y[k];
                B_x[k] = 0;
                B_y[k] = 0;
            }
            if ((b[i] >> (29 - j)) & 1)//if the j-th bit is one then only we have to add
            {
                pt_add(A_x, A_y, G_x, G_y, B_x, B_y);
                for (k = 0; k < 10; k++)
                {
                    A_x[k] = B_x[k];
                    A_y[k] = B_y[k];
                    B_x[k] = 0;
                    B_y[k] = 0;
                }
            }
        }
        tk = 0;//since only for the last limb we have to consider first one
    }
}
```

4.2. Elliptic Curve Diffie-Hellman

ECDH Domain Parameters[1]:

1. Choose a prime p and the elliptic curve

$$E : y^2 \equiv x^3 + ax + b \mod p$$

2. Choose a primitive element $P = (x_p, y_p)$.

The prime p, the curve given by its coefficients a, b and the primitive element P are the domain parameters.

In our implementation we have taken all these domain parameters as specified in the elliptic curve p-256.

Implementation

```c
void main()
{
```

Alice	**Bob**
choose $k_{prA} = a \in \{2, 3, \ldots, \#E-1\}$	choose $k_{prB} = b \in \{2, 3, \ldots, \#E-1\}$
compute $k_{pubA} = aP = A = (x_A, y_A)$	compute $k_{pubB} = bP = B = (x_B, y_B)$

$$\xrightarrow{\quad A \quad}$$

$$\xleftarrow{\quad B \quad}$$

compute $aB = T_{AB}$ | compute $bA = T_{AB}$

Joint secret between Alice and Bob: $T_{AB} = (x_{AB}, y_{AB})$.

Figure 4.3: Elliptic Curve Diffie-Hellman Key Exchange Protocol

```c
    clock_t start = clock();
    long long int gx[10] = {412664470, 310699287, 515062287, 14639179, 608236151, 865834382, 69500811, 880588875, 27415};//x-co-ordinate of the generator
    long long int gy[10] = {935285237, 785973664, 857074924, 864867802, 262018603, 531442160, 670677230, 280543110, 20451};//y-co-ordinate of the generator
    long long int n[10] = {0x3c632551, 0xee72b0b, 0x3179e84f, 0x39beab69, 0x3fffffbc, 0x3fffffff, 0xfff, 0x3fffc000, 0xffff};//this is the 2^30 base representation of the order of the ellitic curve
    long long int a_gx[10] = {0}, a_gy[10] = {0};
    long long int b_gx[10] = {0}, b_gy[10] = {0};
    long long int ab_gx[10] = {0}, ab_gy[10] = {0};
    long long int ba_gx[10] = {0}, ba_gy[10] = {0};
    long long int a[10] = {0}, b[10] = {0};
    int i, l;
    srand(time(NULL));
    for(i = 0; i < 9; i++)//assigning random values to the scalar, at the same time making sure the value of the scalar must be less than the order of the elliptic curve
    {
        a[i] = rand() & n[i];
        b[i] = rand() & n[i];
    }
    printf("a: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", a[l]);
    printf("\n");
    printf("b: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", b[l]);
    printf("\n\n");
    scalar_mult(gx, gy, a_gx, a_gy, a);//aG
    printf("a_gx: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", a_gx[l]);
    printf("\n");
    printf("a_gy: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", a_gy[l]);
    printf("\n\n");
    scalar_mult(a_gx, a_gy, ba_gx, ba_gy, b);
    printf("ba_gx: ");//baG
    for (l = 0; l < 9; l++)
        printf("%10lld\t", ba_gx[l]);
    printf("\n");
    printf("ba_gy: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", ba_gy[l]);
    printf("\n\n");
    scalar_mult(gx, gy, b_gx, b_gy, b);//bG
    printf("b_gx: ");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", b_gx[l]);
    printf("\n");
    printf("b_gy: ");
    for (l = 0; l < 9; l++)
```

```c
        printf("%10lld\t", b_gy[l]);
    printf("\n\n");
    scalar_mult(b_gx, b_gy, ab_gx, ab_gy, a);//abG
    printf("ab_gx:␣");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", ab_gx[l]);
    printf("\n");
    printf("ab_gy:␣");
    for (l = 0; l < 9; l++)
        printf("%10lld\t", ab_gy[l]);
    printf("\n\n");
    clock_t end = clock();
    double elapsed = ((float)end - (float)start) / CLOCKS_PER_SEC;
    printf("Time␣measured:␣%.3f␣seconds.\n", elapsed);
}
```

4.3. Elliptic Curve p-256[3]

556 The elliptic curve P-256 is a Weierstrass curve Wa,b defined over the prime field GF(p) that has
557 order hn, where h=1 and where n is a prime number. This curve has domain parameters D=(p,
558 h, n, Type, a, b, G, Seed, c), where the Type is "Weierstrass curve" and the other parameters
559 are defined as follows:
560
561 p: 2256 2224 + 2192 + 296 561 1
562 = 1157920892103562487626974469494075735300
563 0861434152903141955336313088670978539510
564 (=0xffffffff 00000001 00000000 00000000 00000000 ffffffff ffffffff
565 ffffffff)
566 h: 1
567 n: 1157920892103562487626974469494075735299
568 9969555224135760342422259061068512044369
569 (=0xffffffff 00000000 ffffffff ffffffff bce6faad a7179e84 f3b9cac2
570 fc632551)
571 tr: 89188191154553853111372247798585809583
572 (=(p+1) h n = 0x43190553 58e8617b 0c46353d 039cdaaf)
573 a: 3
574 = 1157920892103562487626974469494075735300
575 0861434152903141955336313088670978539480
576 (=0xffffffff 00000001 00000000 00000000 00000000 ffffffff ffffffff
577 fffffffc)
578 b: 4105836372515214212932612978004726840910
579 1144410159937255548352563140394674012910
580 (=0x5ac635d8 aa3a93e7 b3ebbd55 769886bc 651d06b0 cc53b0f6 3bce3c3e
581 27d2604b)
582 Gx: 4843956129390645175905258525279791420276
583 2949526041747995844080717082404635286
584 (=0x6b17d1f2 e12c4247 f8bce6e5 63a440f2 77037d81 2deb33a0 f4a13945
585 d898c296)
586 Gy: 3613425095674979579858512791958788195661
587 1106672985015071877198253568414405109
588 (=0x4fe342e2 fe1a7f9b 8ee7eb4a 7c0f9e16 2bce3357 6b315ece cbb64068
589 37bf51f5)
590 Seed: 0xc49d3608 86e70493 6a6678e1 139d26b7 819f7e90
591 c: 5743601147020015596417353403826606187
592 4404262441590381759559473094645957903490
593 (=0x7efba166 2985be94 03cb055c 75d4f7e0 ce8d84a9 c5114abc af317768
594 0104fa0d)

References

[1] Jan Pelzl (auth.) Christof Paar. *Understanding cryptography: a textbook for students and practitioners*. 1st ed. Springer, 2010. ISBN: 3642041000; 9783642041006; 3642446493; 9783642446498; 3642041019; 9783642041013.

[2] J.H. Silverman Jeffrey Hoffstein Jill Pipher. *An Introduction to Mathematical Cryptography*. 1st ed. Undergraduate Texts in Mathematics. Springer, 2008. ISBN: 0387779930; 9780387779935.

[3] NIST. *NIST*. 2022. URL: `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186-draft.pdf` (visited on 12/08/2022).