Diffie-Hellman Kex Exchange in Multiplicative Group and Elliptic Curve Group

Author: Prabal Das

Date: December 2, 2024

Abstract

This documentation is based on what I have studied and implemented in our class at ISI, Kolkata. In our class, I have implemented the arithmetic and multiplicative operations in a finite Field and then the Diffie-Hellman Key Exchange in both multiplicative group and Elliptic Curve group. In this documentation, I have discussed all the preliminary concepts of Diffie-Hellman Key Exchange in both the groups along with various security implications and how all of these are implemented in basic C language. I have also produced the pseudocode corresponding to all the code I have done in the course. I have tried my best to make this documentation as simple as possible without losing its essence. For the coding part, you may refer to my GitHub link here.

Contents

0.1 Introduction

Diffie–Hellman key exchange (DHKEX) is a mathematical method of securely generating a symmetric cryptographic key over a insecure public channel and was one of the first public-key protocols named after **Whitfield Diffie** and **Martin Hellman**. Diffie-Hellman is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman 1, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key. In 2015, both of them were honored by **Turing Award** for their revolutionary work in cryptography.
In 2006, Hellman suggested the algorithm be called **Diffie–Hellman–Merkle key exchange** in recognition of **Ralph Merkle**'s contribution to the invention of public-key cryptography (Hellman, 2006), writing:

> *The system...has since become known as Diffie–Hellman key exchange. While that system was first described in a paper by Diffie and me, it is a public key distribution system, a concept developed by Merkle, and hence should be called 'Diffie-Hellman–Merkle key exchange' if names are to be associated with it. I hope this small pulpit might help in that endeavor to recognize Merkle's equal contribution to the invention of public key cryptography.*

0.2 General overview

Diffie–Hellman key exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. An analogy illustrates the concept of public key exchange by using colors instead of very large numbers:

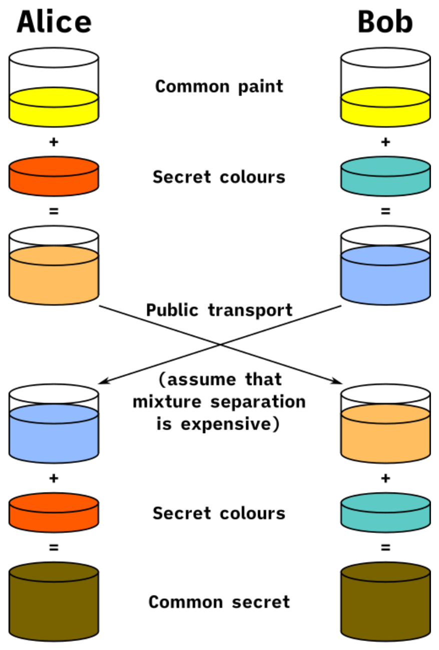


Figure 1: Illustration of the concept behind Diffie–Hellman key exchange

The process begins by having the two parties, Alice and Bob, publicly agree on an arbitrary starting color that **does not** need to be kept secret. In this example 1, the color is yellow. Each person also selects a secret color that they keep to themselves – in this case, red and cyan. The crucial part of the process is that Alice and Bob each mix their own secret color together with their mutually shared color, resulting in orange-tan and light-blue mixtures respectively, and then publicly exchange the two mixed colors. Finally, each of them mixes the color they received from the partner with their own private color. The result is a final color mixture (yellow-brown in this case) that is identical to their partner's final color mixture.
If a third party listened to the exchanges, they would only know the common color (yellow) and the first mixed colors (orange-tan and light-blue), but it would be very hard for them to find out the final secret color (yellow-brown). Bringing the analogy back to a real-life exchange using large numbers rather than colors, this determination is computationally expensive. It is impossible to compute in a practical amount of time even for modern supercomputers.

0.2.1 Uses of DH-KEX

Although Diffie–Hellman key exchange itself is a non-authenticated key-agreement protocol, it provides the basis for a variety of authenticated protocols, and is used to secure a variety of Internet services, including SSL/TLS and SSH.

- **Public Key Encryption :** Public key encryption schemes based on the Diffie–Hellman key exchange have been proposed and the first such scheme is the ElGamal encryption2.
- **Password-authenticated key agreement :** When Alice and Bob share a password, they may use a password-authenticated key agreement (PK) form of Diffie–Hellman to prevent man-in-the-middle attacks.
- **Public key :** It is also possible to use Diffie–Hellman as part of a public key infrastructure, allowing Bob to encrypt a message so that only Alice will be able to decrypt it, with no prior communication between them other than Bob having trusted knowledge of Alice's public key. By this they can establish a common shared secure symmetric key for further communications.

0.2.2 Security Considerations

Although DH KEX has many application in cryptographic security, like all cryptographic systems, it is not completely immune to attacks and vulnerabilities. Some potential vulnerabilities of the Diffie-Hellman key exchange include:

- **Man-in-the-middle attacks :** If an attacker is able to intercept and modify the messages exchanged between Alice and Bob during the key exchange, they may be able to impersonate Alice or Bob and establish a secure channel with the other party.
- **Small subgroup attacks :** If the prime number(which we will discuss shortly) used in the key exchange has a small subgroup, an attacker may be able to use this to their advantage to recover the shared secret key.

- **Exponent attacks :** If the secret exponents used in the key exchange are not chosen randomly, an attacker may be able to use this to their advantage to recover the shared secret key.

To avoid these vulnerabilities, some authors recommend use of elliptic curve cryptography, for which no similar attack is known. Failing that, they recommend that the order of the Diffie–Hellman group should be at least 2048 bits.

From the seminal paper of Diffie-Hellman, a new domain, **Public Key Cryptosystem** developed and this is still serving a foundation to many cryptographic primitives. Now we will move the different mathematical versions of Diffie-Hellman Key Exchange and discuss them in context of our implementation.

0.3 DH-KEX in Multiplicative Group

The simplest and the original implementation, later formalized as Finite Field Diffie–Hellman in RFC 7919, and in this section we will discuss in context of finite order multiplicative cyclic group.

0.3.1 Overview

The protocol uses the multiplicative group of integers modulo P, where P is prime, and G is a primitive root modulo P. These two values are chosen in this way to ensure that the resulting shared secret can take on any value from 1 to P–1. Here is an example of the protocol, with non-secret values in blue, and secret values in red.

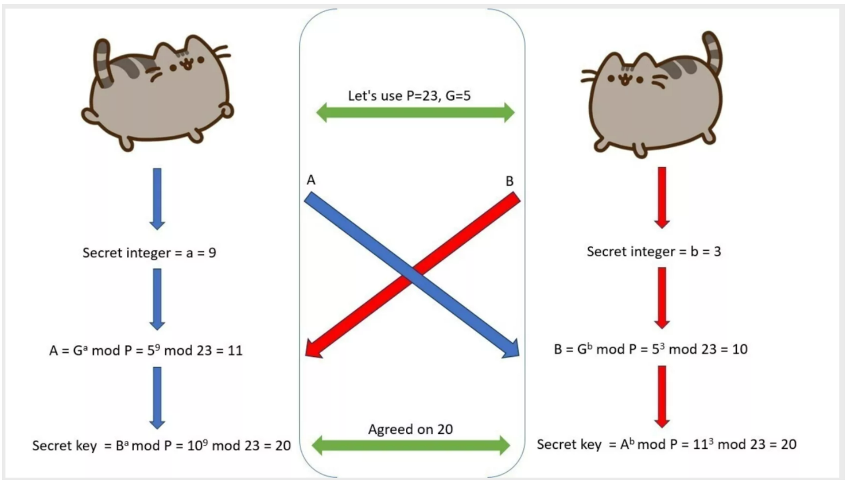

Let's use P=23, G=5
A
B
Secret integer = a = 9
Secret integer = b = 3
A = G^a mod P = 5^9 mod 23 = 11
B = G^b mod P = 5^3 mod 23 = 10
Agreed on 20
Secret key = B^a mod P = 10^9 mod 23 = 20
Secret key = A^b mod P = 11^3 mod 23 = 20

Figure 2: Diffie–Hellman key exchange in Multiplicative group

1. Alice and Bob publicly agree to use a modulus $P = 23$ and base $G = 5$ (which is a primitive root modulo 23).
2. Alice chooses a secret integer $a = 9$, then sends Bob $A = G^a \bmod P$. Here in our example, $A = 5^9 \bmod 23$ i.e. $A = 11$.
3. Bob chooses a secret integer $b = 3$, then sends Alice $B = G^b \bmod P$. Here in our example, $B = 5^3 \bmod 23$ i.e. $B = 10$.

4. Alice computes $S = B^a \bmod P$. In our example, $S = 10^9 \bmod 23$ i.e. $S = 20$.
5. Bob computes $S = A^b \bmod P$. In our example, $S = 11^3 \bmod 23$ i.e. $S = 20$.
6. Alice and Bob now share a secret, S.

Both Alice and Bob have arrived at the same values because under mod P,

$$A^b \bmod P = G^{ab} \bmod P = G^{ba} \bmod P = B^a \bmod P$$

Only a and b are kept secret. All the other values – P, G, $G^a \bmod P$, and $G^b \bmod P$ – are sent in the insecure channel. The strength of the scheme comes from the fact that $G^{ab} \bmod P = G^{ba} \bmod P$ take extremely long times to compute by any known algorithm just from the knowledge of P, G, $G^a \bmod P$, and $G^b \bmod P$. Once Alice and Bob compute the shared secret they can use it as an encryption key, known only to them, for sending messages across the same open communications channel.

0.3.2 Our Implementation

Of course, much larger values of a and b and P would be needed to make this example secure, since there are only 23 possible results of $n \bmod 23$. However, if P is a prime of at least 600 digits, then even the fastest modern computers using the fastest known algorithm cannot find a given only G, P and $G^a \bmod P$.
For the DH KEX implementation, we need the operations like modulus, exponentiation, arithmetics all in finite group and next we are going to discuss how we implemented these operations in C language.

Preliminary:

In our implementation, we have taken the prime P to be 256 bit long and the prime in decimal is 105470615072424007464777057006017113535036866827082468263120632948849084329973.
Since, all computation will be in multiplicative group $\mathbb{Z}_P$, we need to store all 256-bit integers in our 64-bit operating system. For that we will represent all integers in base, 2^{29} and store it in an `long long` type array (which is 64-bit long) to avoid buffer overflow.
In base $\theta = 2^{29}$, each 256-bit long integer, say a, can be written as

$$a = a_0 + a_1\theta + a_2\theta^2 + \cdots + a_7\theta^7 + a_8\theta^8,\ 0 \leq a_i < \theta,\ \forall i$$

Note that, a_8 will be of 24-bit long and other a_i will be 29-bit long.
We have also pre-computed the value `TString` $= \theta^{2*9}/P =$ 130174601815239752763103701526270888022515529381378518624138659694433114167181336(using SageMath).

Algorithms:

For DHKEX, we mainly need to implement to operations, one is **exponentiation**(i.e. computing G^a) and another is **modular reduction** in P(mod P). We will define some atomic functions, which will help us to do the exponentiation and for reduction part we will discuss and implement **Barrett's Reduction**.
What we will discuss next will be based on our implementation and what we have done. The itemized points are the functions from our implementation

- `MultSchoolBook():` This is function which takes two 256-bit long integers and do the school book multiplication and returns a `long long` type array.
  Since, the resultant of the previous function become 512-bit long and we need to store it in base 2^{29}, we need another function which will convert the resultant in our base.

- `ConversionTo29bit():` This function takes an array and for each cell extract the last 29 bits and add remaining with the next cell and do iteratively so that the resultant array lies in our base. The pseudocode of our implementation is given below:

Algorithm 1 `ConversionTo29bit()`

```
1: Input: Array $C$ of length $l$.
2: Output: The converted array $C$ with 29-bit values.
3: $temp \leftarrow C[0]$
4: for $i = 0$ to $l - 1$ do
5:     $C[i] \leftarrow temp \& 0x1FFFFFFF$          ▷ Extract the last 29 bits
6:     $temp \leftarrow temp \gg 29$                 ▷ Shift temp right by 29 bits
7:     $temp \leftarrow temp + C[i + 1]$             ▷ Add the next value in the array to temp
8: end for
9: $C[l] \leftarrow temp \& 0x1FFFFFFF$              ▷ Store last 29 bits
10: $temp \leftarrow temp \gg 29$                    ▷ Shift temp right by 29 bits
11: $C[l + 1] \leftarrow temp$                       ▷ Store remaining bits
```

- `subtract_any():` This function does the subtraction, $(a - b)$ of two 256-bit long integers a and b. Notice, if $a < b$, then it need to add with the modulus P.

- `divide_by_base_exp():` This function takes a number and *exp* as input and divides number by θ^{exp}. The output is basically right shift by $\theta \times exp$ bits.

Barrett Reduction: In modular arithmetic, Barrett reduction is a reduction algorithm introduced in 1986 by P.D. Barrett3. A naive way of computing $x(mod\ m)$ is by division algorithm $x = q \cdot m + r, 0 \leq r < m$ and objective to find r i.e $x \equiv r(mod\ m)$. Barrett reduction is an algorithm designed to optimize this operation by approximating the value q such that $q - 2 \leq Q \leq q$, where Q is a candidate for q and is obtained by Barrett reduction algorithm. This algorithm also assure that w.h.p (with prob.0.95) $Q = q$. The algorithm is given below:
Input: positive integers $x = (x_{2k-1} \cdots x_1 x_0)_b$, $m = (m_{k-1} \cdots m_1 m_0)_b$ and $\mu = \lfloor b^{2k}/m \rfloor$.
Output: $r \equiv x \bmod m$.

1. $q_1 \leftarrow \lfloor x/b^{k-1} \rfloor$, $q_2 \leftarrow q_1 \cdot \mu$, $q_3 \leftarrow \lfloor q_2/b^{k+1} \rfloor$.
2. $r_1 \leftarrow x \bmod b^{k+1}$, $r_2 \leftarrow q_3 \cdot m \bmod b^{k+1}$, $r \leftarrow r_1 - r_2$.
3. If $r < 0$ then $r \leftarrow r + b^{k+1}$.
4. While $r \geq m$ do: $r \leftarrow r - m$.
5. Return r.

Now, we will see our implementation of this barrett reduction.

Algorithm 2 `barrett_reduction()`

```
Require: Arrays $x$ and $m$ representing integers of lengths $x\_len$ and $m\_len$ respectively,
    $\mu$, and an empty array $result$. Also, $b = 2^{29}$ and $L = 9$.
Ensure: $result$ holds $x \bmod m$.
 1: Initialize $q1$, $q2$, $q3$, and $r1$ arrays to 0.
 2: Step 1: Compute $q1 = \lfloor x/b^{L-1} \rfloor$ using divide_by_base_exp.
 3: Step 2: Compute $q2 = q1 \cdot \mu$ using MultSchoolBook.
 4: Convert $q2$ into 29-bit size cells using ConversionTo29bit.
 5: Step 3: Compute $q3 = \lfloor q2/b^{L+1} \rfloor$ using divide_by_base_exp.
 6: Step 4: Compute $r1 = q3 \cdot m$ using MultSchoolBook.
 7: Convert $r1$ into 29-bit size cells using ConversionTo29bit.
 8: Step 5: Compute $r = x - r1$ using subtract_any.
 9: Step 6: While $r \geq m$, reduce $r$ by $m$ using subtract_any.
10: Copy the final $r$ into $result$.
```

So, we have solve one of our two problem, the problem of reduction. Now, we will move to the exponentiation part. Our task is to compute $x^{exp} (mod\ p)$, where x, exp both are integer and $p = 256$. For that we will see three algorithms, all of them runs in order of size of exp and those are *LeftToRight exp, RightToLeft exp and Montgomary exp.*

Left to Right Exponentiation: The algorithm for left to right exponentiation is given below:
Input: Positive integers g, $m = (m_{l-1}m_{l-2} \cdots m_1 m_0)$.
Output: g^m.

1. Initialize $h \leftarrow 1$.
2. **For** i from $l-1$ down to 0, **do**:
   (a) $h \leftarrow h^2$
   (b) **If** $m_i = 1$, **then** $h \leftarrow h \cdot g$
3. **Return** h.

Now, we will see the pseudocode which we have implemented.

- `IntToBin()`: This function takes an array and an integer and outputs the integer's binary representation in the array.

Right to Left Exponentiation: The algorithm for right to left exponentiation is given below:
Input: Positive integers g, $m = (m_{l-1}m_{l-2} \cdots m_1 m_0)$.
Output: g^m.

1. Initialize $h \leftarrow 1$.
2. **For** i from 0 down to $l-1$, **do**:
   (a) **If** $m_i = 1$, **then** $h \leftarrow h \cdot g$
   (b) $g \leftarrow g^2$
3. **Return** h.

Since, both of the above algorithm almost same, we are only giving the pseudocode for the first one.

Algorithm 3 `L2R_Exponentiation()`

```
Require: Arrays prod, prime, TString, num, and exp.
Ensure: prod contains the result of modular exponentiation.
 1: Initialize prod_temp to 0 (temporary memory).
 2: Initialize bin to 0 (array for binary representation, size 29).
 3: Set $j \leftarrow 8$, $count \leftarrow 0$, $k \leftarrow 28$.
 4: for $i \leftarrow 0$ to 260 do                    ▷ Loop iterates 261 times ($29 \times 9$).
 5:     prod_temp ← prod * prod using MultSchoolBook().
 6:     Convert prod_temp into 29-bit cells using ConversionTo29bit().
 7:     Reset prod to zero.
 8:     Perform modular reduction: prod ← barrett_reduction().
 9:     Reset prod_temp to zero.
10:     if $count = 0$ then
11:         Set $count \leftarrow 29$.
12:         Convert exp[j] to binary: IntToBin(bin, exp[j]).
13:         Set $k \leftarrow 28$.
14:         Decrement $j \leftarrow j - 1$.
15:     end if
16:     if bin[k] $= 1$ then                          ▷ Check if the $k$-th bit of exp[j] is 1.
17:         prod_temp ← prod * num using MultSchoolBook().
18:         Convert prod_temp into 29-bit cells: ConversionTo29bit().
19:         Perform modular reduction: prod ← barrett_reduction().
20:         Reset prod_temp to zero: InitializeToZero(prod_temp).
21:     end if
22:     Decrement $count \leftarrow count - 1$, $k \leftarrow k - 1$.
23: end for
```

Montgomery Exponentiation: In modular arithmetic computation, Montgomery modular multiplication, more commonly referred to as **Montgomery multiplication**, is a method for performing fast modular multiplication. It was introduced in 1985 by the American mathematician Peter L. Montgomery4. The algorithm for Montgomery multiplication is given below:
Input: Positive integers g, $m = (m_{l-1}m_{l-2}\cdots m_1m_0)$.
Output: $S = g^m$.

1. Initialize $S \leftarrow g, R \leftarrow g^2$.
2. **For** i from $l-2$ down to 0, **do**:
    (a) **If** $m_i = 0$, **then** $R \leftarrow R \cdot S$
    (b) $S \leftarrow S^2$
    (c) **Else** $S \leftarrow R \cdot S$
    (d) $R \leftarrow R^2$
3. **Return** S.

Our implementation for Montgomery exponentiation is given below.

Algorithm 4 `Montgomery_Exponentiation()`

```
Require: prime, TString, num, and exp.
Ensure: Compute $num^{exp}$
1: Initialize R to 0.
2: Compute $R \leftarrow$ num · num using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
3: for $i \leftarrow 1$ to 255 do          ▷ Iterate over bits of exp, from MSB to LSB.
4:     if exp[i] = '0' then
5:         Compute R ← num · R using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
6:         Compute num ← num · num using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
7:     else
8:         Compute num ← num · R using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
9:         Compute R ← R · R using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
10:    end if
11: end for
```

This was the Montgomery algorithm which can efficiently compute the exponentiation. Since, we have implemented some algorithms which can efficiently compute modulus and exponentiation, so we have donevthe DHKEX in a cyclic multiplicative group $\mathbb{Z}_p$ and now we will move to the DHKEX based on elliptic curve group which is ECDHKEX.

0.4 DH-KEX in Elliptic Curve Group

Elliptic Curve Diffie-Hellman (ECDH) is based on the mathematical structure of elliptic curves and offers several advantages over traditional Diffie-Hellman key exchange using cyclic multiplicative group. Here's why ECDH is widely used and important:

- ECDH achieves the same level of security as traditional Diffie-Hellman with much **smaller key sizes**.
- While all public-key cryptography schemes based on integer factorization (e.g., RSA) or discrete logarithms (e.g., traditional Diffie-Hellman) are vulnerable to quantum computers, elliptic curves provide a higher **quantum resistance** per bit of key length.
- ECDH is foundational for more advanced cryptographic systems, including ECDSA, PFS etc.

So, let's delve into the concept of Elliptic curve and Elliptic curve group.

0.4.1 Elliptic Curve

Let, $\mathbb{F}$ be a field and $a, b, c, d, e, f, g, h, i, j \in \mathbb{F}$. Then the general equation of a qubic in two variable is

$$f(x, y) = ax^3 + by^3 + cx^2y + dxy^2 + ex^2 + fy^2 + gxy + hx + iy + j \tag{1}$$

An elliptic curve is the solution set of the nonsingular cubic polynomial equation in two unknowns over the field i.e.

$$\mathbb{E} = \{(x, y) \in \mathbb{F} \times \mathbb{F} : f(x, y) = 0\} \tag{2}$$

If the characteristic of the field, char($\mathbb{F}$) $\neq 2, 3$, then the function $f(x, y)$ can be written as

$$y^2 = x^3 + ax + b; a, b \in \mathbb{F} \tag{3}$$

The definition of elliptic curve requires that the curve be non-singular. Geometrically, this means that the graph has no cusps, self-intersections, or isolated points. Algebraically, this holds if and only if the discriminant, Δ is not equal to zero i.e.

$$\Delta = -16(4a^3 + 27b^2) \neq 0$$

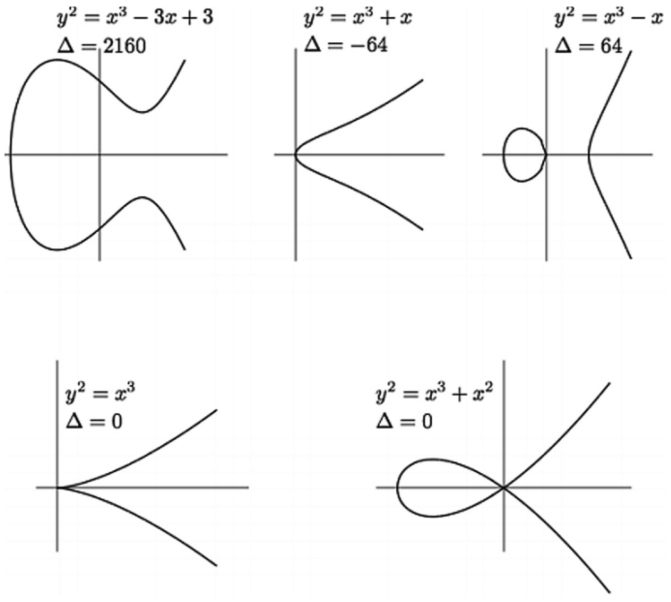


Figure 3: Some non-singular and singular curves

In the above figure we can see in geometrically how singular and non-singular curves look like. In above curves, there is no cusps or self-intersection, but in the below curves there is cusp and self-intersection. So, above three curves are three different Elliptic curves based on different a, b.
Equation 3 is also known as **Short Weierstrass form**, also the **Edwards curves** and **Montgomery curves** are useful in cryptography.

0.4.2 Elliptic Curve Group

Since, in equation 3 $\lim_{x\to\infty} y = \infty$, so the **point at infinity**, written as $\mathcal{O} = (\infty, \infty)$, is also considered as one of the solutions of the equation. For a finite field $\mathbb{F}_p$, the elliptic curve $\mathbb{E}$ along with $\mathcal{O}$ forms a group with group operation $\oplus$. We can use geometry to make the points of an elliptic curve into a group and we will visualize how the operations are happening. For illustration purpose, we have taken the elliptic curve to be

$$\mathbb{E} : y^2 = x^3 - 5x + 8 \tag{4}$$

and $P, Q \in \mathbb{E}$ distinct.

- **Addition of two points:** To find $P \oplus Q$, we first draw a line L joining P, Q. Then by **Bezout Theorem**, L will cut the curve $\mathbb{E}$ at a point, say R. Then, $P \oplus Q$ will be the reflection of R on $\mathbb{E}$. Geometrically, it will be like:

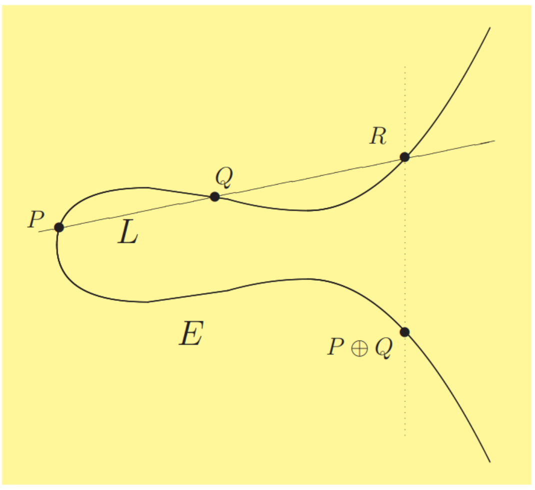


Figure 4: Addition of P and Q i.e. $P \oplus Q$

- **Point Doubling:** For closure property of a group, we also need to show $P \oplus Q \in \mathbb{E}$. To find $P \oplus P$, we first draw a **tangent** L at point P. Then L will cut the curve $\mathbb{E}$ at a point, say R. Then, $P \oplus P$ will be the reflection of R on $\mathbb{E}$. Geometrically, it will be like:

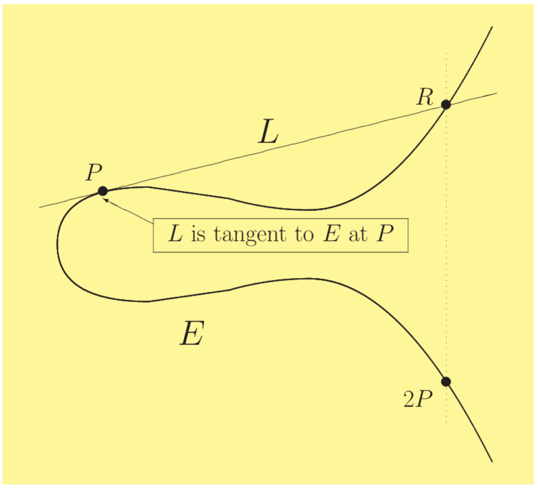


Figure 5: Point Doubling i.e. $P \oplus P$

- **Point Inverse:** For inverse property of a group element, we also need to show, $\exists$ $Q \in \mathbb{E}$ s.t. $P \oplus Q = \mathcal{O}$, where $\mathcal{O}$ is the identity element of $\mathbb{E}$. To find Q, we will draw a **vertical line** L at point P. Then L will cut the curve $\mathbb{E}$ at a point, say R. The line L has no third intersection point, as it is vertical line. So, the intersection point R is the desired point Q, which is denoted as $(-P)$. Geometrically, it will be like:

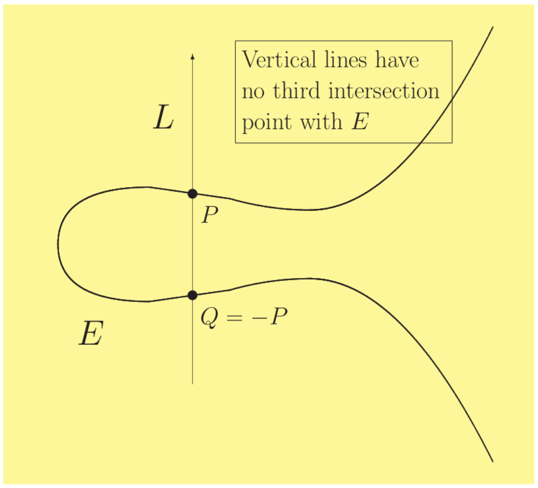


Figure 6: Point Inverse i.e. $(-P)$

Now we will see the group operations properties on elliptic curve group elements.

Properties of $\oplus$ operation on group elements

Let, $P, Q, R \in \mathbb{E}$. The addition law on $\mathbb{E}$ has the following properties:

- $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P$
- $P + (-P) = \mathcal{O}$
- $P + (Q + R) = (P + Q) + R$
- $P + Q = Q + P$

In other words, the addition law, $\oplus$ makes the points of $\mathbb{E}$ into a commutative group. Since, we have bagged a lot of concept about Elliptic curve and Elliptic curve group, we will see how Diffie-Hellman Key Exchange is happening in this group.

0.4.3 Overview

For DHKEX in multiplicative group, the hard problem was based on computing the exponent x, given g^x, where $g \in \mathbb{Z}^*{}_p$, known as **Discrete Log problem**, in elliptic curve the hard problem is also to find n, given nP, where $P \in \mathbb{E}$ is a generator of the elliptic curve group. Let's see the algorithm how this is happening:

- A and B agree on the elliptic curve group $\mathbb{E}$ of order n and a primitive element $g \in \mathbb{E}$ of order n, all of these are public elements.
- A selects integer $x \in [0, n-1]$, calculate $P = xg$ and sends to B.
- B selects integer $y \in [0, n-1]$, calculate $Q = yg$ and sends to A.
- A computes xQ, B computes yP and the shared secret is $xQ = yP = (xy)(mod\ n)g$.

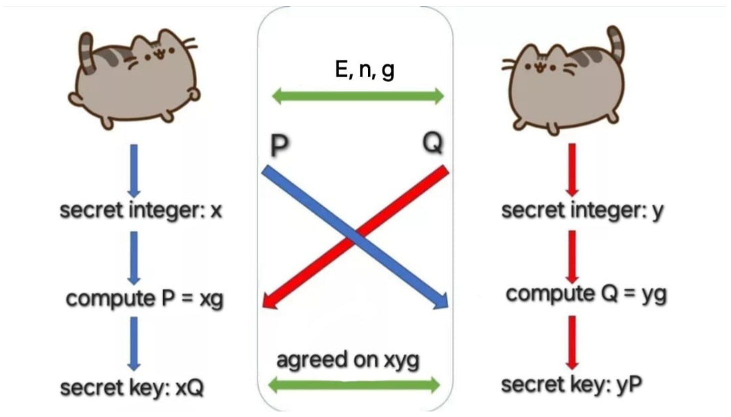


Figure 7: Diffie–Hellman key exchange in Elliptic curve group

So, in the protocol, we have seen two parties A and B, given a group element $g \in \mathbb{E}$, need to compute xg, yg, xP and yQ where $mP = P \oplus P \oplus P \cdots \oplus P(m \text{ times})$. If we can compute those values, then we are done with the key exchange. We have seen the geometric interpretation of the point addition and point doubling, now we will see how to compute those values mathematically.

Point Addition Formula

Let, $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ be two points on elliptic curve $\mathbb{E}$ and $P_3 = (x_3, y_3) := P_1 \oplus P_2$, where $x_1, y_1, x_2, y_2, x_3, y_3 \in \mathbb{F}_p$. Then point addition is defined as:

- For $P_3 = (x_3, y_3)$, $-P_3 = (x_3, -y_3)$.
- For point of infinity $\mathcal{O}$, $P_1 \oplus \mathcal{O} = \mathcal{O} \oplus P_1 = P_1$ and $P_1 \oplus (-P_1) = (-P_1) \oplus P_1 = \mathcal{O}$.
- Otherwise, first compute the slope m of the straight line that connects P_1 and P_2 using $m = \frac{y_2 - y_1}{x_2 - x_1}$, if $x_2 \neq x_1$ otherwise $m = \frac{3x_1^2 + a}{2y_1}$,if $x_1 = x_2$ and $y_1 = y_2$. Then compute $x_3 = m^2 - x_1 - x_2$, $y_3 = m(x_1 - x_3) - y_1$.

0.4.4 Our Implementation

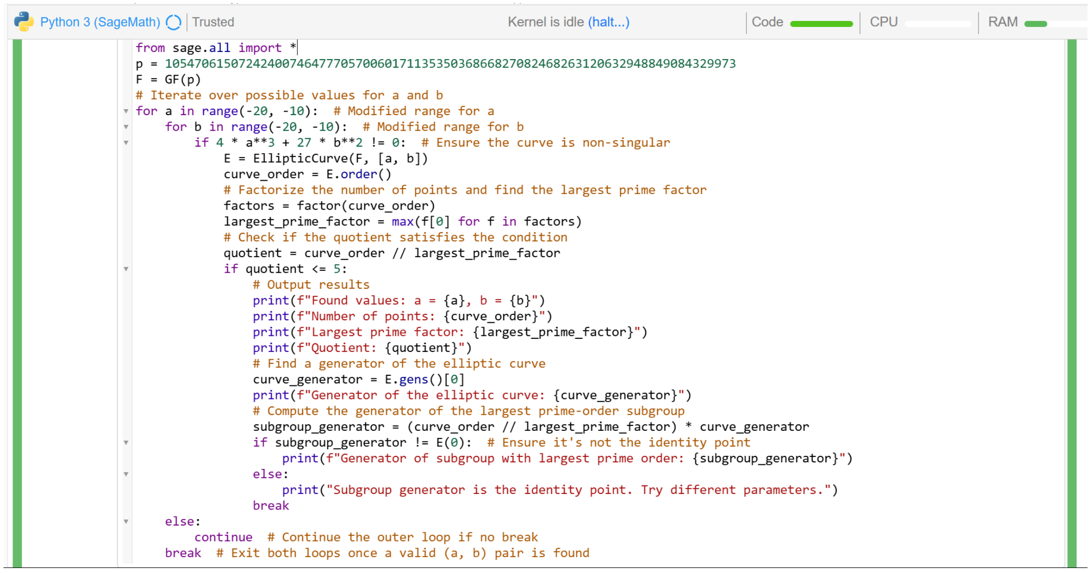
To happen the key exchange, we need to implement first the point addition then scalar multiplication. For that first we need to choose a, b, p and g.
In our implementation, we have taken the prime p to be the same as define before i.e. $p =$ 105470615072424007464777057006017113535036866827082468263120632948849084329973, a 256-bit string.

How to choose a, b and g

Different choice of a, b will give different curve $y^2 = x^3 + ax + b$, but the curves might be singular i.e $\Delta = 0$ leading the curve to be non-elliptic. So, we need to choose a, b s.t the curve become elliptic. The following process ensures that the curve's order has a large prime factor and that the cofactor is small (usually 2 or 3). This is important for security, as it helps protect against certain attacks, like those based on small subgroups. The generator of the largest prime order subgroup will be the candidate for public generator g. The process is follows:

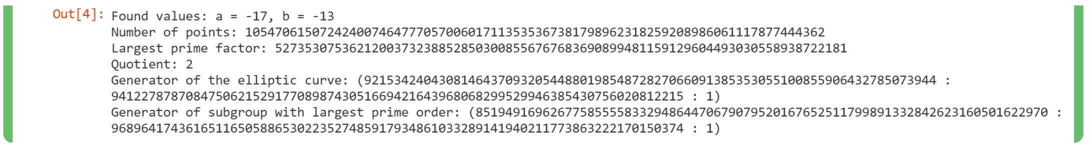
- We will begin by selecting random values for the curve parameters a and b. These will define an curve over a finite field. Then check whether $\Delta \neq 0$ or not.
- Once we have chosen a valid elliptic curve, we need to determine the order of the group, which can be done by **point counting algorithms** like **Müller's algorithm** or **Schoof's algorithm**.
- Then we need to find a generator of the largest prime order subgroup of the elliptic curve group, which can be be done by first factoring the order, then pick a non-zero element of order n by **Brute-Force method**.
- Calculate the cofactor i.e the order of the group divided by the order of the subgroup. If it is 2 or 3, we will stop otherwise re-sample a, b and repeat.

This is how we find the public parameters a, b and g, which will generate **safe elliptic curve** instance. Now we will see how we have chosen those public parameters in our implementation. Following is the code for finding the public parameters in **SageMath**.



```python
from sage.all import *
p = 105470615072424007464777057006017113535036866827082468263120632948849084329973
F = GF(p)
# Iterate over possible values for a and b
for a in range(-20, -10):  # Modified range for a
    for b in range(-20, -10):  # Modified range for b
        if 4 * a**3 + 27 * b**2 != 0:  # Ensure the curve is non-singular
            E = EllipticCurve(F, [a, b])
            curve_order = E.order()
            # Factorize the number of points and find the largest prime factor
            factors = factor(curve_order)
            largest_prime_factor = max(f[0] for f in factors)
            # Check if the quotient satisfies the condition
            quotient = curve_order // largest_prime_factor
            if quotient <= 5:
                # Output results
                print(f"Found values: a = {a}, b = {b}")
                print(f"Number of points: {curve_order}")
                print(f"Largest prime factor: {largest_prime_factor}")
                print(f"Quotient: {quotient}")
                # Find a generator of the elliptic curve
                curve_generator = E.gens()[0]
                print(f"Generator of the elliptic curve: {curve_generator}")
                # Compute the generator of the largest prime-order subgroup
                subgroup_generator = (curve_order // largest_prime_factor) * curve_generator
                if subgroup_generator != E(0):  # Ensure it's not the identity point
                    print(f"Generator of subgroup with largest prime order: {subgroup_generator}")
                else:
                    print("Subgroup generator is the identity point. Try different parameters.")
                break
    else:
        continue  # Continue the outer loop if no break
    break  # Exit both loops once a valid (a, b) pair is found
```

Figure 8: Python code for finding a, b and g



```
Out[4]: Found values: a = -17, b = -13
        Number of points: 105470615072424007464777057006017113535367381798962318259208986061117877444362
        Largest prime factor: 52735307536212003732388528503008556767683690899481159129604493030558938722181
        Quotient: 2
        Generator of the elliptic curve: (92153424043081464370932054488019854872827066091385353055100855906432785073944 :
        94122787870847506215291770898743051669421643968068299529946385430756020812215 : 1)
        Generator of subgroup with largest prime order: (85194916962677585555833294864470679079520167652511799891332842623160501622970 :
        96896417436165116505886530223527485917934861033289141940211773863222170150374 : 1)
```

Figure 9: Output of the above code

From on wards in our implementation, we will use the public parameters $a = -17 (\text{mod p})$, $b = -13 (\text{mod p})$, $g = (85194916962677585555833294864470679079520167652511799891332842623160501622970, 96896417436165116505886530223527485917934861033289141940211773863222170150374)$(the last coordinate in the output is z-axis, projective coordinate).

Algorithms:

After selecting a Elliptic Curve and the public parameters, we need to do the scalar multiple of that generator. For that first we will implement two point addition, then extend it to scalar multiple.

- `is_zero():` This function takes an array and check whether the integer corresponding to the array is zero or not by iterating all the cells as zero or not. Returns 1 if all is zero, otherwise 0.

Algorithm 5 `is_zero()`

```
1: Input: An array of integers array of size MAX_SIZE
2: Output: 1 if all elements of the array are zero, 0 otherwise
3: for $i = 0$ to MAX_SIZE - 1 do
4:     if array[$i$] $\neq 0$ then
5:         Return 0                                   ▷ Not zero
6:     end if
7: end for
8: Return 1                                           ▷ All elements are zero
```

- `Inverse_Using_Fermat():` For computing the slope m, we need to find the multiplicative inverse of field element, which will do by **Fermat's Algorithm**. By this theorem, given an field element x, the inverse will be x^{p-2}, as

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

Since, we need to do the exponentiation x^{p-2} and all in modular p, we will use `L2R_Exponentiation()` function defined previously0.3.2.The corresponding pseudocode is as follows:

Algorithm 6 `Inverse_Using_Fermat()`

```
1: Input: An array x of size MAX_SIZE, prime, TString.
2: Output: An array x_inv of size MAX_SIZE, representing the modular inverse of x.
3: Initialize: exp_for_inv[MAX_SIZE] ← (prime $- 2$),the exponents for inversion.
4: Initialize: prod[MAX_SIZE] with prod[0] = 1 and other elements as zero.
5: prod ← L2R_Exponentiation(prod, prime, TString, x, exp_for_inv)
6: Step 2: Copy the result: x_inv ← prod
```

- `Addition_256():` This function takes two integer and add it in modulo p.

Algorithm 7 `Addition_256():`

```
1: Input: Two arrays $A$, $B$ of size MAX_SIZE, prime, TString.
2: Output: The resulting array $C$, where $C = (A + B) \bmod prime$.
3: $result \leftarrow [0, 0, \dots]$
4: for $i = 0$ to MAX_SIZE - 1 do
5:     $result[i] \leftarrow A[i] + B[i]$
6: end for
7: $C \leftarrow result$ using-
8: MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
```

- `elliptic_curve_add()`: Now we are ready to implement the point addition which we have already defined previously.

Algorithm 8 `elliptic_curve_add()`

```
Input: Two elliptic curve points $P = (x_1, y_1), Q = (x_2, y_2)$, prime, TString.
Output: The resulting point $(x_3, y_3) = (x_1, y_1) \oplus (x_2, y_2)$.
initialize: $a \leftarrow 0, m \leftarrow 0$, $temp1, temp2, temp3 \leftarrow 0$.
Define constants: $array\_three \leftarrow [3, 0, \dots]$, $array\_two \leftarrow [2, 0, \dots]$.
if $(x_1, y_1)$ is the point at infinity then
    $x_3 \leftarrow x_2$, $y_3 \leftarrow y_2$ return
end if
Case 2: If $Q$ is the point at infinity, return $P$:
if $(x_2, y_2)$ is the point at infinity then
    $x_3 \leftarrow x_1$, $y_3 \leftarrow y_1$ return
end if
Case 3: If $P = -Q$, return the point at infinity:
Compute: $temp1 \leftarrow y_1 + y_2$
if $x_1 = x_2$ and $temp1 = 0$ then
    $x_3 \leftarrow 0$, $y_3 \leftarrow 0$ return
end if
Case 4: If $P = Q$ (Point Doubling):
if $x_1 = x_2$ and $y_1 = y_2$ then
    $temp1 \leftarrow 3 \cdot x_1^2 + a$ using MultSchoolBook(),ConversionTo29bit(), barrett_reduction(), Addition_256():.
    $temp2 \leftarrow 2 \cdot y_1$ using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().
    $m \leftarrow temp1/temp2 \pmod{prime}$ using MultSchoolBook(),ConversionTo29bit(), barrett_reduction().    ▷ Slope of the tangent line
else
    $temp1 \leftarrow y_2 - y_1$ using subtract_any()
    $temp2 \leftarrow x_2 - x_1$ using subtract_any()
    $m \leftarrow temp1/temp2 \pmod{prime}$    ▷ Slope of the secant line
end if
Step 5: Compute $x_3$:
$temp1 \leftarrow m^2 - x_1 - x_2$ using MultSchoolBook(),ConversionTo29bit(), barrett_reduction(),subtract_any()
$x_3 \leftarrow temp1 \pmod{prime}$
Step 6: Compute $y_3$:
$temp1 \leftarrow x_1 - x_3$
$temp2 \leftarrow m \cdot temp1 - y_1$ using MultSchoolBook(),ConversionTo29bit(), barrett_reduction(),subtract_any()
$y_3 \leftarrow temp2 \pmod{prime}$
Return: The resulting point $(x_3, y_3)$.
```

- `L2R_Elliptic_Multiple()`: Based on the previous point algorithm, we will implement the scalar multiple which will be our targeted function.

Algorithm 9 `L2R_Elliptic_Multiple()`

```
1: Input: Initial point $(x_3, y_3)$ and scalar $exp$.
2: Output: Resulting point $(x_4, y_4)$ after scalar multiplication.
3: Initialize:
4: $bin[29] \leftarrow [0]$                ▷ Array for storing binary representation of integers
5: $j \leftarrow 8$, $count \leftarrow 0$, $k \leftarrow 28$
6: $x5 \leftarrow [0, 0, \dots]$, $y5 \leftarrow [0, 0, \dots]$                ▷ Temporary variables
7: for $i = 0$ to $260$ do                ▷ Iterate $261 = 29 \times 9$ times
8:     $(x_5, y_5) \leftarrow$ elliptic_curve_add($(x_4, y_4), (x_4, y_4)$)
9:     Update $(x_4, y_4) \leftarrow (x_5, y_5)$
10:    Reset $x_5 \leftarrow 0$, $y_5 \leftarrow 0$
11:    if $count = 0$ then
12:        $count \leftarrow 29$
13:        $bin \leftarrow$ IntToBin($bin, exp[j]$)                ▷ Convert integer $exp[j]$ to binary
14:        $k \leftarrow 28$, $j \leftarrow j - 1$
15:    end if
16:    if $bin[k] = 1$ then
17:        $(x_5, y_5) \leftarrow$ elliptic_curve_add($(x_4, y_4), (x_3, y_3)$)
18:        Update $(x_4, y_4) \leftarrow (x_5, y_5)$
19:        Reset $x_5 \leftarrow 0$, $y_5 \leftarrow 0$
20:    end if
21:    $count \leftarrow count - 1$, $k \leftarrow k - 1$
22: end for
23: Return: The point $(x_4, y_4)$.
```

0.5 Conclusion

So, after a long journey, we have arrived in a situation where we can now do the Diffie-Hellman Key Exchange in both Multiplicative Group as well as Elliptic Curve Group. Both parties will compute those things locally and share in a public channel to establish a common shared secret key. From next time onwards, when they want to communicate, they will use that key for encryption as well as decryption.

Agreed upon

0.6 Reference

1. New Directions in Cryptography, DH'76 Link

2. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms Link

3. Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor, CRYPTO'86 Link

4. Modular Multiplication Without Trial Division, MC'85 Link

5. Images are taken from **G o o g l e**

6. Some Materials are taken from **WikipediA**

7. Class Materials.