



# **Information security**

**CS-3002**

## **Assignment 04**

### **ELLIPTIC CURVE**

**Submitted by:** Minam Faisal, Momenah Saif

**Roll number:** 21i-1901, 21i-1909.

**Date:** 29<sup>th</sup> April 2024.

## Table of Contents

• INTRODUCTION .....	2
• VERIFICATION OF ELLIPTIC CURVE.....	2
• VERIFICATION OF POINT .....	4
• IMPLEMENTATION OF ECC BASED DH.....	5
• IMPLEMENTATION OF PUBLIC KEY BASED ENCRYPTION.....	10
• SUMMARY.....	15
• REFERENCES.....	15

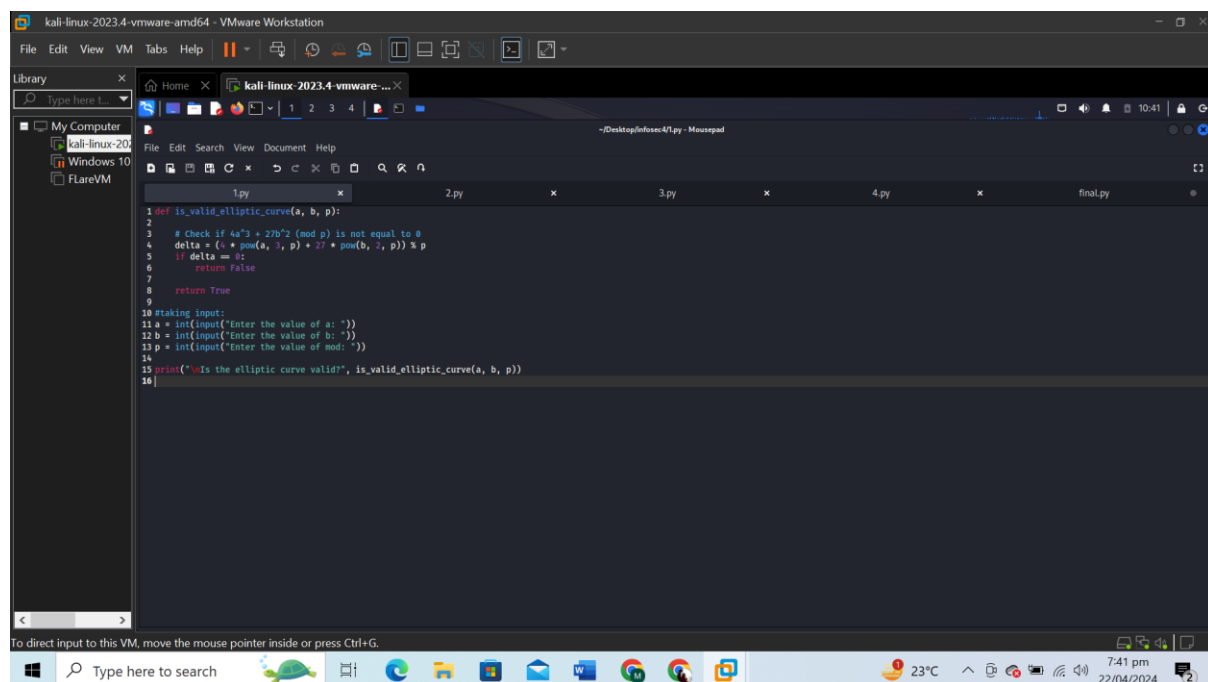
## • INTRODUCTION

This assignment focuses on implementing various concepts of Elliptic Curve Cryptography (ECC) using a language of choice, without relying on external libraries. The objectives include verifying the validity of an elliptic curve, determining whether a given point lies on the curve, implementing ECC-based Diffie-Hellman key exchange, and realizing public key-based encryption and decryption using ECC. Through this implementation, we aim to gain a deeper understanding of ECC principles and their practical application in cryptographic protocols. The implementation of these functions will provide a hands-on experience in ECC, reinforcing the theoretical knowledge acquired in class.

## • VERIFICATION OF ELLIPTIC CURVE

Function to verify the Elliptic Curve is valid or not.

### Overall code:



```
1 def is_valid_elliptic_curve(a, b, p):
2
3     # Check if 4a^3 + 27b^2 (mod p) is not equal to 0
4     delta = (4 * pow(a, 3, p) + 27 * pow(b, 2, p)) % p
5     if delta == 0:
6         return False
7     return True
8
9
10 #taking input:
11 a = int(input("Enter the value of a: "))
12 b = int(input("Enter the value of b: "))
13 p = int(input("Enter the value of mod: "))
14
15 print("\nIs the elliptic curve valid:", is_valid_elliptic_curve(a, b, p))
16
```

### Explanation of code:

```
1 def is_valid_elliptic_curve(a, b, p):
```

- This line defines a function named *is\_valid\_elliptic\_curve* that takes three parameters: a, b, and p. These parameters represent the coefficients of the elliptic curve equation:  $y^2 = x^3 + ax + b \text{ modulo } p$ .

```
2
3 # Check if  $4a^3 + 27b^2 \pmod{p}$  is not equal to 0
4 delta = (4 * pow(a, 3, p) + 27 * pow(b, 2, p)) % p
5 if delta == 0:
6     return False
7
8 return True
9
```

- The above snippet calculates the discriminant of the elliptic curve, which is:  
 $\Delta = 4a^3 + 27b^2$  modulo  $p$ .
- It uses the `pow()` function to efficiently compute the powers modulo  $p$ .
- ***If the discriminant is 0***, it means that the curve is singular, and thus not valid according to standard elliptic curve cryptography. In such a case, the function returns `False`.
- ***If the discriminant is not equal to 0***, the function returns `True`, indicating that the elliptic curve is valid.

```
9
10 #taking input:
11 a = int(input("Enter the value of a: "))
12 b = int(input("Enter the value of b: "))
13 p = int(input("Enter the value of mod: "))
14
15 print("\nIs the elliptic curve valid?", is_valid_elliptic_curve(a, b, p))
16
```

- These lines prompt the user to input values for the parameters  $a$ ,  $b$ , and the prime modulo  $p$  of the elliptic curve.
- Finally, this line calls the *is\_valid\_elliptic\_curve* function with the user-provided parameters and prints the result, indicating whether the elliptic curve defined by the parameters is valid or not.

## Output:

The screenshot shows a Kali Linux virtual machine window titled "kali-linux-2023.4 vmware-and64 - VMware Workstation". The terminal window displays the following commands and output:

```

kali@kali: ~/Desktop/infosec4
--kali@kali: ~/Desktop/infosec4
└─$ python 1.py
Enter the value of a: 4
Enter the value of b: 8
Enter the value of mod: 4

Is the elliptic curve valid? false

--kali@kali: ~/Desktop/infosec4
└─$ python 1.py
Enter the value of a: 4
Enter the value of b: 5
Enter the value of mod: 2

Is the elliptic curve valid? true

--kali@kali: ~/Desktop/infosec4
└─$

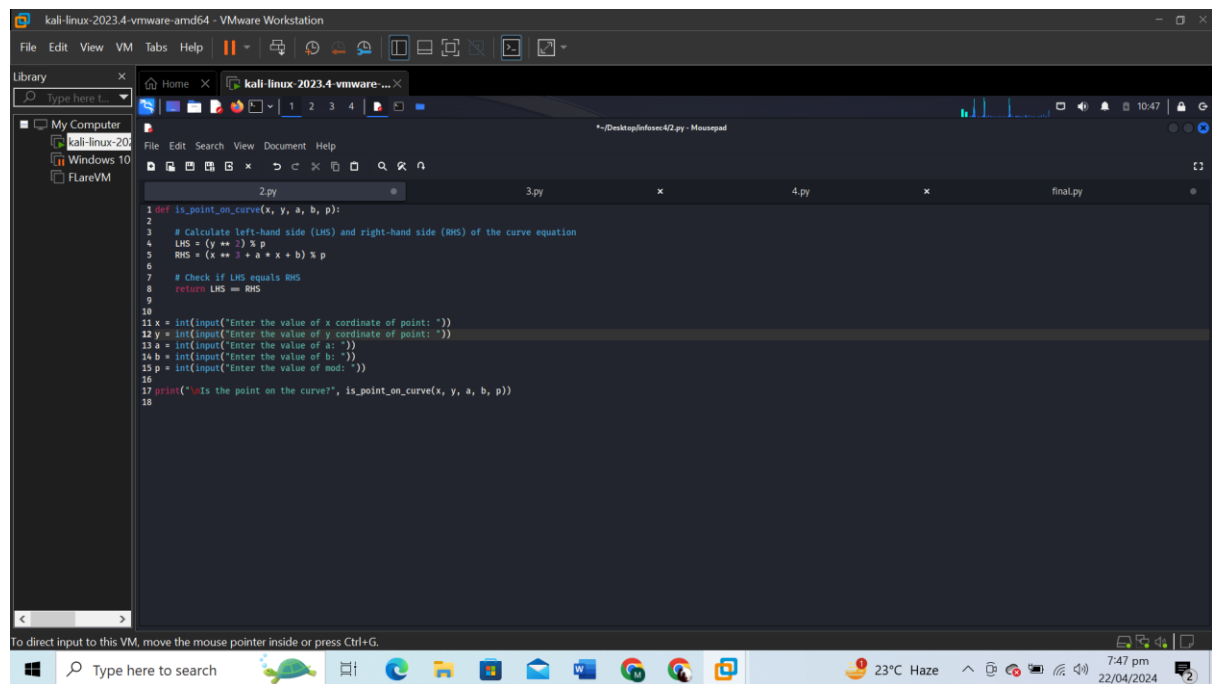
```

The left sidebar shows the virtual machine's library with "kali-linux-2023.4", "Windows 10", and "FlareVM" listed. The bottom status bar indicates the system time is 7:30 pm on 24°C.

## • VERIFICATION OF POINT

Function to verify the point in on the curve or not.

### Overall code:



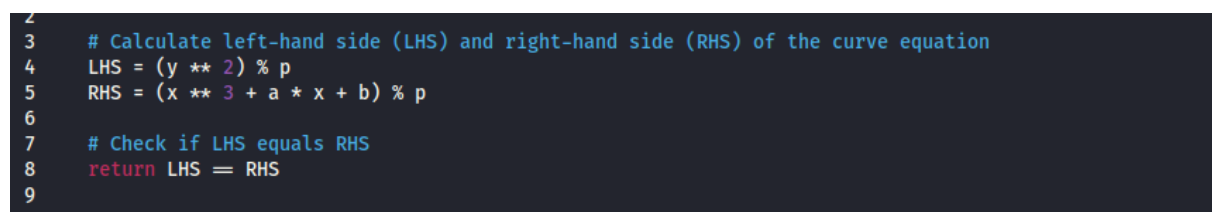
```
1 def is_point_on_curve(x, y, a, b, p):
2     # Calculate left-hand side (LHS) and right-hand side (RHS) of the curve equation
3     LHS = (y ** 2) % p
4     RHS = (x ** 3 + a * x + b) % p
5     # Check if LHS equals RHS
6     return LHS == RHS
7
8 x = int(input("Enter the value of x coordinate of point: "))
9 y = int(input("Enter the value of y coordinate of point: "))
10 a = int(input("Enter the value of a: "))
11 b = int(input("Enter the value of b: "))
12 p = int(input("Enter the value of mod: "))
13
14 print("Is the point on the curve?", is_point_on_curve(x, y, a, b, p))
```

### Explanation of code:



```
1 def is_point_on_curve(x, y, a, b, p):
```

- This line defines a function named `is_point_on_curve` that takes five parameters: `x`, `y`, `a`, `b`, and `p`. These parameters represent the coordinates of a point  $(x, y)$  and the coefficients `a`, `b`, and modulus `p` of the elliptic curve equation  $y^2 = a^3 + ax + b \text{ modulo } p$ .



```
2
3 # Calculate left-hand side (LHS) and right-hand side (RHS) of the curve equation
4 LHS = (y ** 2) % p
5 RHS = (x ** 3 + a * x + b) % p
6
7 # Check if LHS equals RHS
8 return LHS == RHS
9
```

- These lines calculate the **left-hand side (LHS)** and **right-hand side (RHS)** of the elliptic curve equation for the given point  $(x, y)$ .
- For the LHS, it calculates:  $y^2 \text{ modulo } p$ .
- For the RHS, it calculates  $x^3 + ax + b \text{ modulo } p$ .
- Then compares the calculated LHS and RHS. If they are equal, it means that the point  $(x, y)$  lies on the elliptic curve, satisfying the curve equation. In such a case, the function returns **True**. Otherwise, it returns **False**.

```

10
11 x = int(input("Enter the value of x coordinate of point: "))
12 y = int(input("Enter the value of y coordinate of point: "))
13 a = int(input("Enter the value of a: "))
14 b = int(input("Enter the value of b: "))
15 p = int(input("Enter the value of mod: "))
16
17 print("\nIs the point on the curve?", is_point_on_curve(x, y, a, b, p))
18 |

```

- The above lines prompt the user to input values for the coordinates  $x$  and  $y$  of the point, as well as the coefficients  $a$ ,  $b$ , and the prime modulus  $p$  of the elliptic curve.
- Finally, calls *the is\_point\_on\_curve* function with the user-provided parameters and prints the result, indicating whether the given point lies on the elliptic curve defined by the parameters.

## Output:

```

kali@kali: ~/Desktop/infosec4
$ python 2.py
Enter the value of x coordinate of point: 6
Enter the value of y coordinate of point: 5
Enter the value of a: 9
Enter the value of b: 8
Enter the value of mod: 4
Is the point on the curve? False

kali@kali: ~/Desktop/infosec4
$ python 2.py
Enter the value of x coordinate of point: 6
Enter the value of y coordinate of point: 4
Enter the value of a: 4
Enter the value of b: 3
Enter the value of mod: 1
Is the point on the curve? True

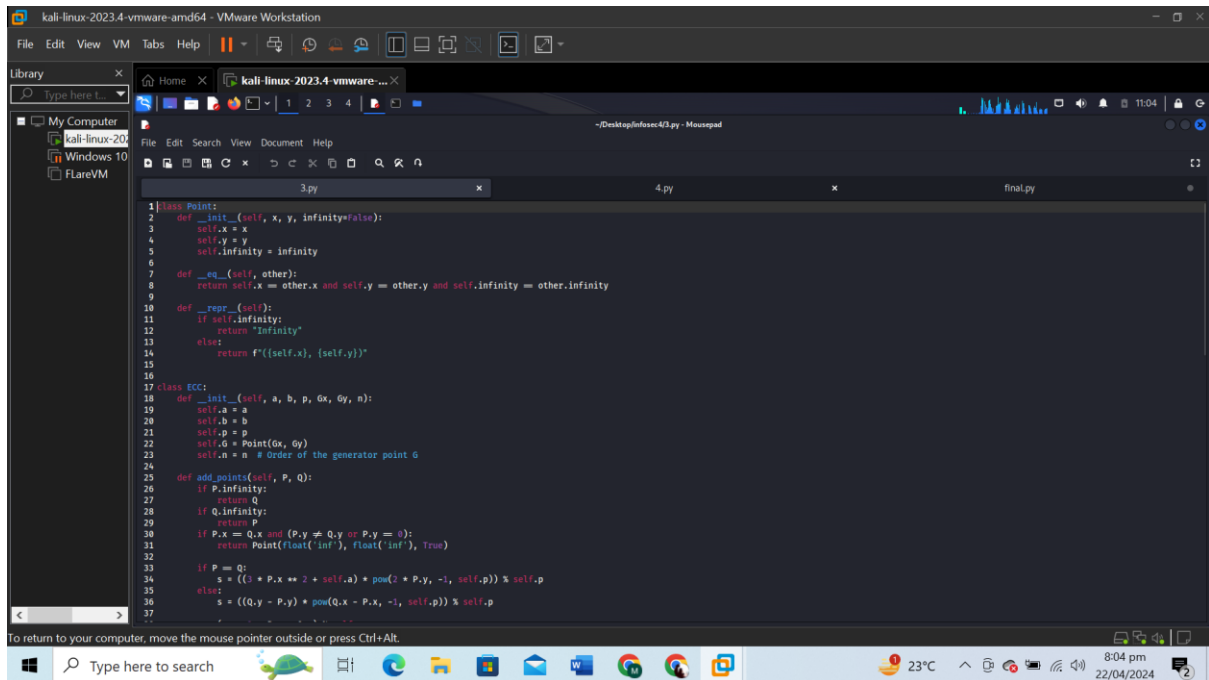
kali@kali: ~/Desktop/infosec4
$

```

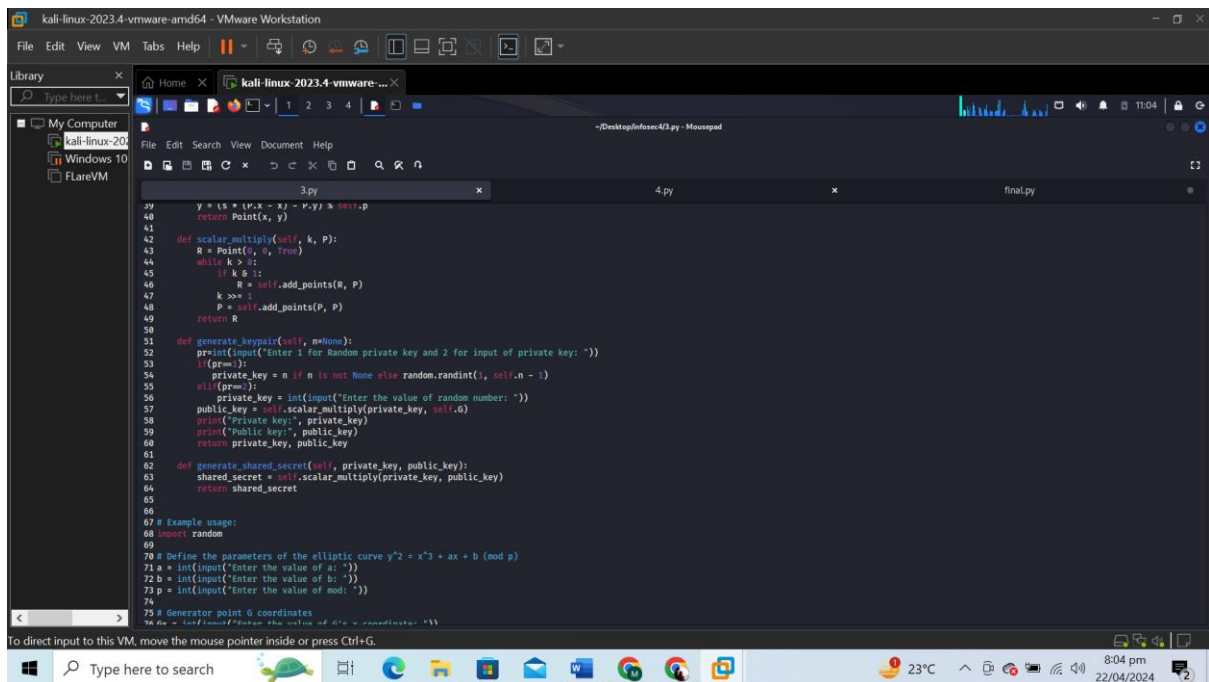
## • IMPLEMENTATION OF ECC BASED DH.

Implement ECC based DH.

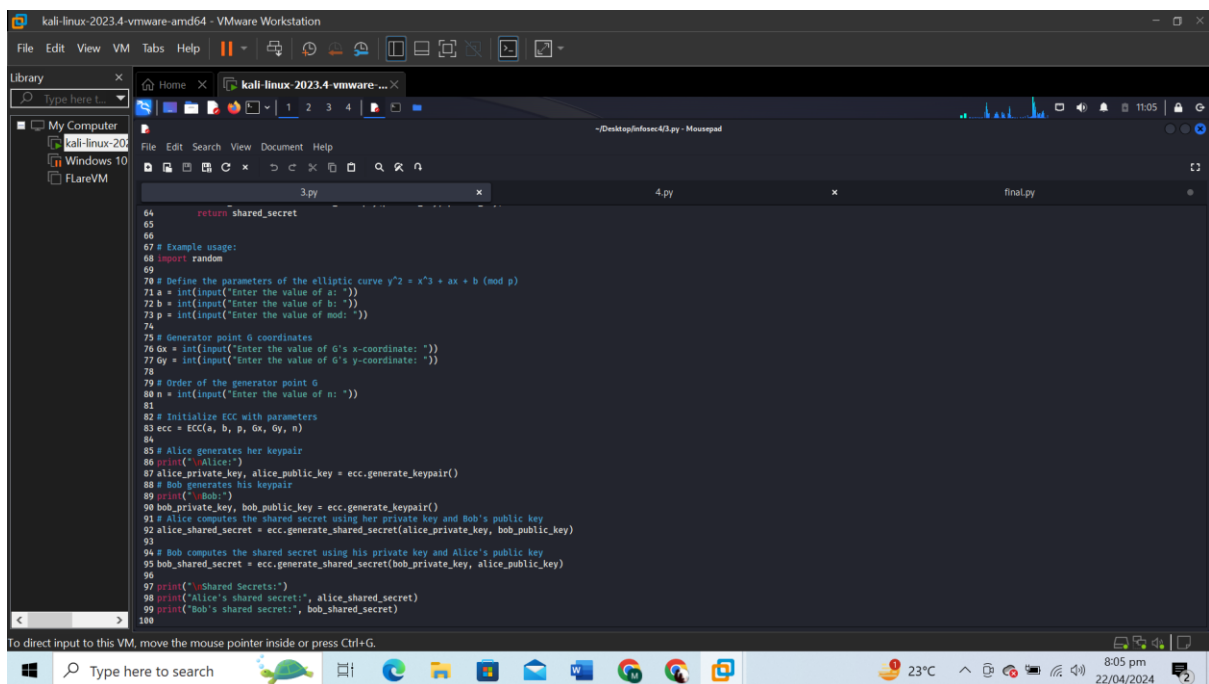
## Overall code:



```
1 class Point:
2     def __init__(self, x, y, infinity=False):
3         self.x = x
4         self.y = y
5         self.infinity = infinity
6
7     def __eq__(self, other):
8         return self.x == other.x and self.y == other.y and self.infinity == other.infinity
9
10    def __repr__(self):
11        if self.infinity:
12            return "infinity"
13        else:
14            return f'({self.x}, {self.y})'
15
16
17 class ECC:
18    def __init__(self, a, b, p, Gx, Gy, n):
19        self.a = a
20        self.b = b
21        self.p = p
22        self.G = Point(Gx, Gy)
23        self.n = n # Order of the generator point G
24
25    def add_points(self, P, Q):
26        if P.infinity:
27            return Q
28        if Q.infinity:
29            return P
30        if P.x == Q.x and (P.y != Q.y or P.y == 0):
31            return Point(float('inf'), float('inf'), True)
32
33        if P == Q:
34            s = ((3 * P.x ** 2 + self.a) * pow(2 * P.y, -1, self.p)) % self.p
35        else:
36            s = ((Q.y - P.y) * pow(Q.x - P.x, -1, self.p)) % self.p
37
38    def multiply(self, k, P):
39        R = Point(0, 0, True)
40        while k > 0:
41            if k & 1:
42                R = self.add_points(R, P)
43            k = k // 2
44            P = self.add_points(P, P)
45        return R
46
47    def generate_keypair(self, n=None):
48        print(input("Enter 1 for Random private key and 2 for input of private key: "))
49        if (prv := 1):
50            private_key = n if n is not None else random.randint(1, self.n - 1)
51        else:
52            private_key = int(input("Enter the value of random number: "))
53        public_key = self.scalar_multiply(private_key, self.G)
54        print("Private key:", private_key)
55        print("Public key:", public_key)
56        return private_key, public_key
57
58    def generate_shared_secret(self, private_key, public_key):
59        shared_secret = self.scalar_multiply(private_key, public_key)
60        return shared_secret
61
62 # Example usage:
63 # 1. Random
64 # 2. Define the parameters of the elliptic curve y^2 = x^3 + ax + b (mod p)
65 a = int(input("Enter the value of a: "))
66 b = int(input("Enter the value of b: "))
67 p = int(input("Enter the value of mod: "))
68
69 # 3. Generator point G coordinates
70 Gx = int(input("Enter the value of Gx: "))
71 Gy = int(input("Enter the value of Gy: "))
72 n = int(input("Enter the value of n: "))
73
74 # 4. Create ECC object
75 ecc = ECC(a, b, p, Gx, Gy, n)
76
77 # 5. Generate keypair
78 private_key, public_key = ecc.generate_keypair()
79
80 # 6. Generate shared secret
81 shared_secret = ecc.generate_shared_secret(private_key, public_key)
82
83 # 7. Print results
84 print("Private key:", private_key)
85 print("Public key:", public_key)
86 print("Shared secret:", shared_secret)
```



```
1 # 4.py
2
3 # 1. Define the parameters of the elliptic curve y^2 = x^3 + ax + b (mod p)
4 a = int(input("Enter the value of a: "))
5 b = int(input("Enter the value of b: "))
6 p = int(input("Enter the value of mod: "))
7
8 # 2. Generator point G coordinates
9 Gx = int(input("Enter the value of Gx: "))
10 Gy = int(input("Enter the value of Gy: "))
11 n = int(input("Enter the value of n: "))
12
13 # 3. Create ECC object
14 ecc = ECC(a, b, p, Gx, Gy, n)
15
16 # 4. Generate keypair
17 private_key, public_key = ecc.generate_keypair()
18
19 # 5. Generate shared secret
20 shared_secret = ecc.generate_shared_secret(private_key, public_key)
21
22 # 6. Print results
23 print("Private key:", private_key)
24 print("Public key:", public_key)
25 print("Shared secret:", shared_secret)
```



### Explanation of code:

```

1 class Point:
2     def __init__(self, x, y, infinity=False):
3         self.x = x
4         self.y = y
5         self.infinity = infinity
6

```

- This class represents a point on an elliptic curve. It has attributes `x` and `y` representing the coordinates of the point, and `infinity` which indicates whether the point is at infinity (for the point at infinity on the curve).

```

5
7 class ECC:
8     def __init__(self, a, b, p, Gx, Gy, n):
9         self.a = a
10        self.b = b
11        self.p = p
12        self.G = Point(Gx, Gy)
13        self.n = n # Order of the generator point G
14

```

- This class represents an Elliptic Curve Cryptography (ECC) system. It is initialized with the parameters of the elliptic curve equation (`a`, `b`, `p`), the coordinates of the generator point (`Gx`, `Gy`), and the order of the generator point (`n`).



```

def add_points(self, P, Q):
    if P.infinity:
        return Q
    if Q.infinity:
        return P
    if P.x == Q.x and (P.y != Q.y or P.y == 0):
        return Point(float('inf'), float('inf'), True)

    if P == Q:
        s = ((3 * P.x ** 2 + self.a) * pow(2 * P.y, -1, self.p)) % self.p
    else:
        s = ((Q.y - P.y) * pow(Q.x - P.x, -1, self.p)) % self.p

    x = (s ** 2 - P.x - Q.x) % self.p
    y = (s * (P.x - x) - P.y) % self.p
    return Point(x, y)

```

- This method performs point addition operation on the elliptic curve. It takes two points **P** and **Q** as input and returns their **sum**.

```

def scalar_multiply(self, k, P):
    R = Point(0, 0, True)
    while k > 0:
        if k & 1:
            R = self.add_points(R, P)
        k >>= 1
        P = self.add_points(P, P)
    return R

```

- This method performs scalar multiplication operation on the elliptic curve. It takes an integer **k** and a point **P** as input, and returns the result of multiplying point **P** by scalar **k**.

```

def generate_keypair(self, n=None):
    pr=int(input("Enter 1 for Random private key and 2 for input of private key: "))
    if(pr==1):
        private_key = n if n is not None else random.randint(1, self.n - 1)
    elif(pr==2):
        private_key = int(input("Enter the value of random number: "))
    public_key = self.scalar_multiply(private_key, self.G)
    print("Private key:", private_key)
    print("Public key:", public_key)
    return private_key, public_key

```

- This method generates a **key pair** (private key, public key) for ECC. It prompts the user to choose between generating a random private key or entering one manually. It then calculates the corresponding public key and returns both.

```

1
2 def generate_shared_secret(self, private_key, public_key):
3     shared_secret = self.scalar_multiply(private_key, public_key)
4     return shared_secret
5
6
7 # Example usage:

```

- This method generates a shared secret given a private key and a public key. It performs **scalar multiplication** of the **public key** by the **private key** and returns the result.

```

# Define the parameters of the elliptic curve  $y^2 = x^3 + ax + b \pmod{p}$ 
a = int(input("Enter the value of a: "))
b = int(input("Enter the value of b: "))
p = int(input("Enter the value of mod: "))

# Generator point G coordinates
Gx = int(input("Enter the value of G's x-coordinate: "))
Gy = int(input("Enter the value of G's y-coordinate: "))

# Order of the generator point G
n = int(input("Enter the value of n: "))

```

- These lines prompt the user to input values for the parameters of the elliptic curve and the generator point.

```

81
82 # Initialize ECC with parameters
83 ecc = ECC(a, b, p, Gx, Gy, n)
84

```

- This line initializes an instance of the ECC class with the provided parameters.

```

85 # Alice generates her keypair
86 print("\nAlice:")
87 alice_private_key, alice_public_key = ecc.generate_keypair()
88 # Bob generates his keypair

```

- `bob_private_key, bob_public_key = ecc.generate_keypair ()`

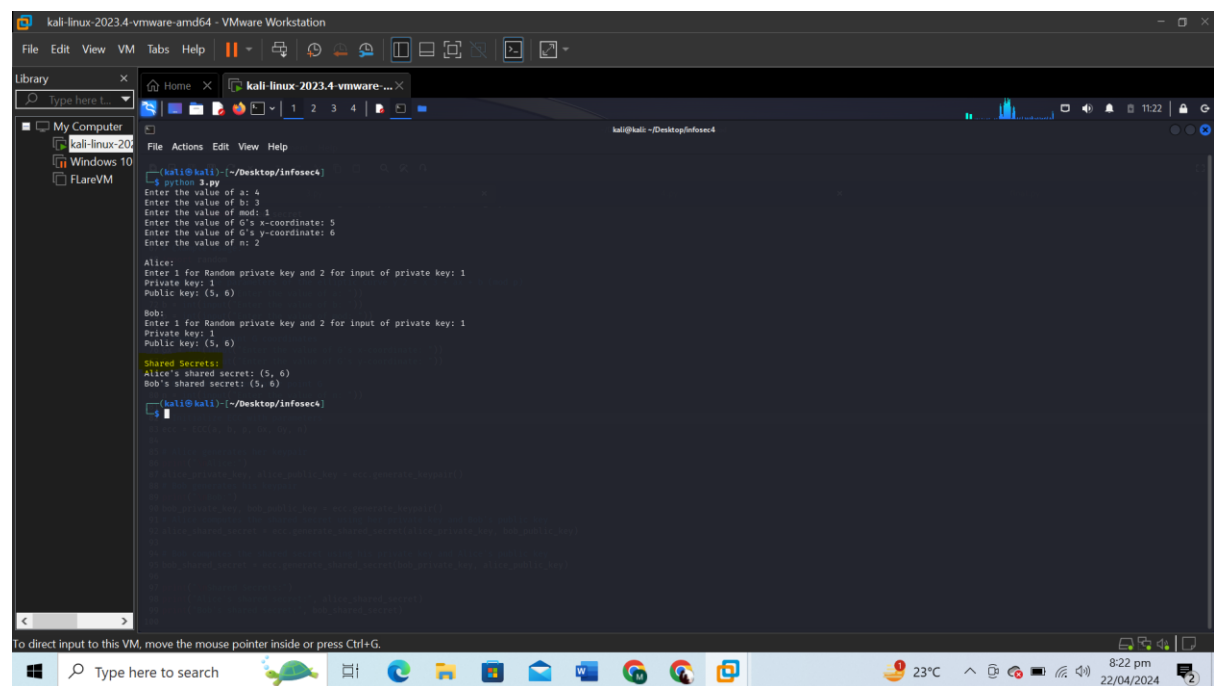
```

96
97 print("\nShared Secrets:")
98 print("Alice's shared secret:", alice_shared_secret)
99 print("Bob's shared secret:", bob_shared_secret)
100

```

- This prints the shared secrets between Alice and Bob.

## Output:



```

kali@kali:~/Desktop/infosec4
$ python 3.py
Enter the value of a: 4
Enter the value of b: 3
Enter the value of mod: 1
Enter the value of G's x-coordinate: 5
Enter the value of G's y-coordinate: 6
Enter the value of n: 2

Alice:
Enter 1 for Random private key and 2 for input of private key: 1
Private key: 1
Public key: (5, 6)

Bob:
Enter 1 for Random private key and 2 for input of private key: 1
Private key: 1
Public key: (5, 6)

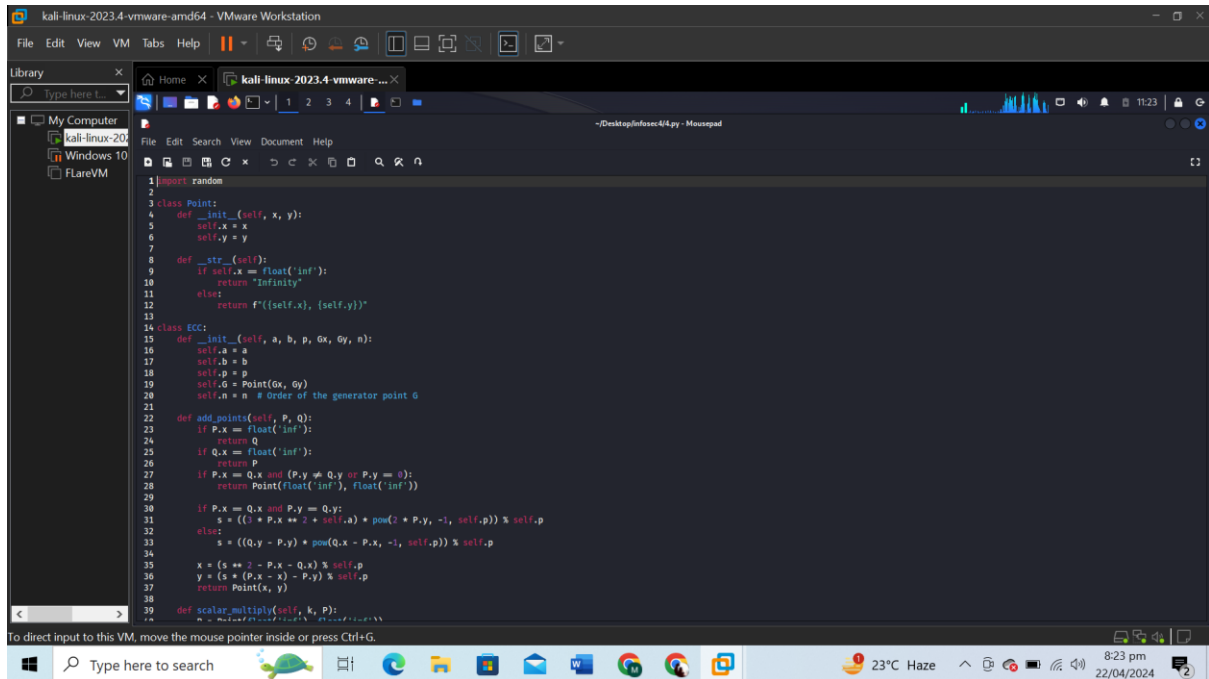
Shared Secrets:
Alice's shared secret: (5, 6)
Bob's shared secret: (5, 6)
kali@kali:~/Desktop/infosec4

```

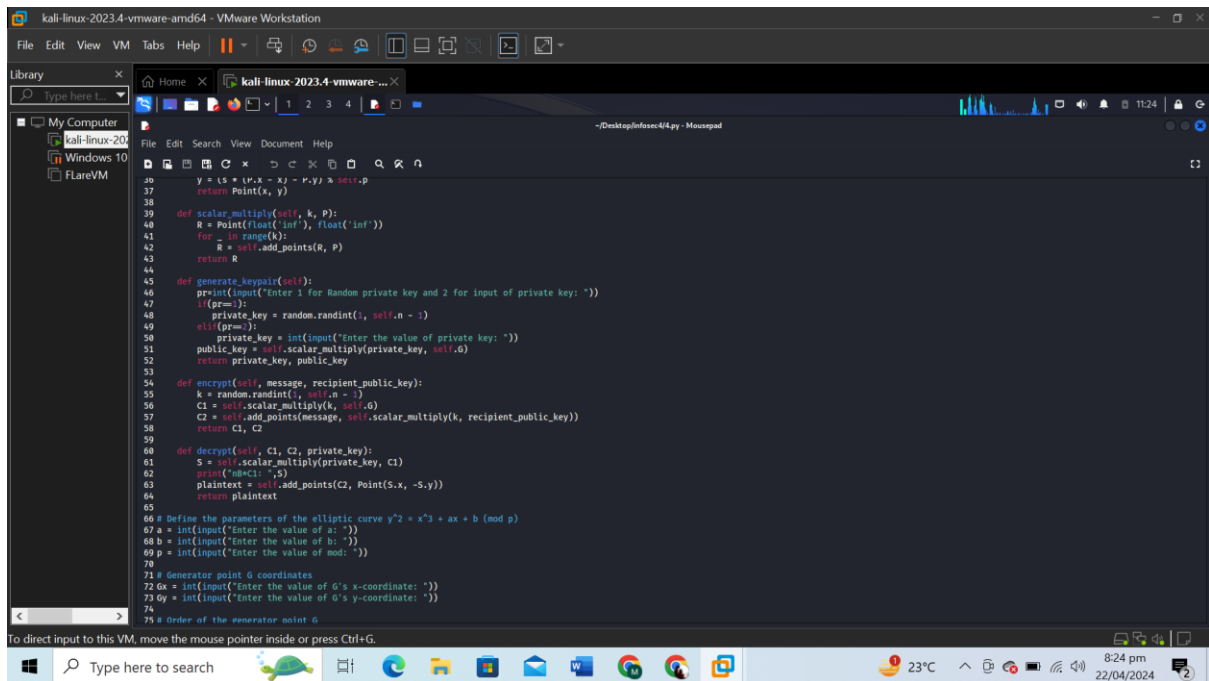
## • IMPLEMENTATION OF PUBLIC KEY BASED ENCRYPTION

Implement the Public key-based Encryption - Decryption using ECC for a given message given in (x, y) form.

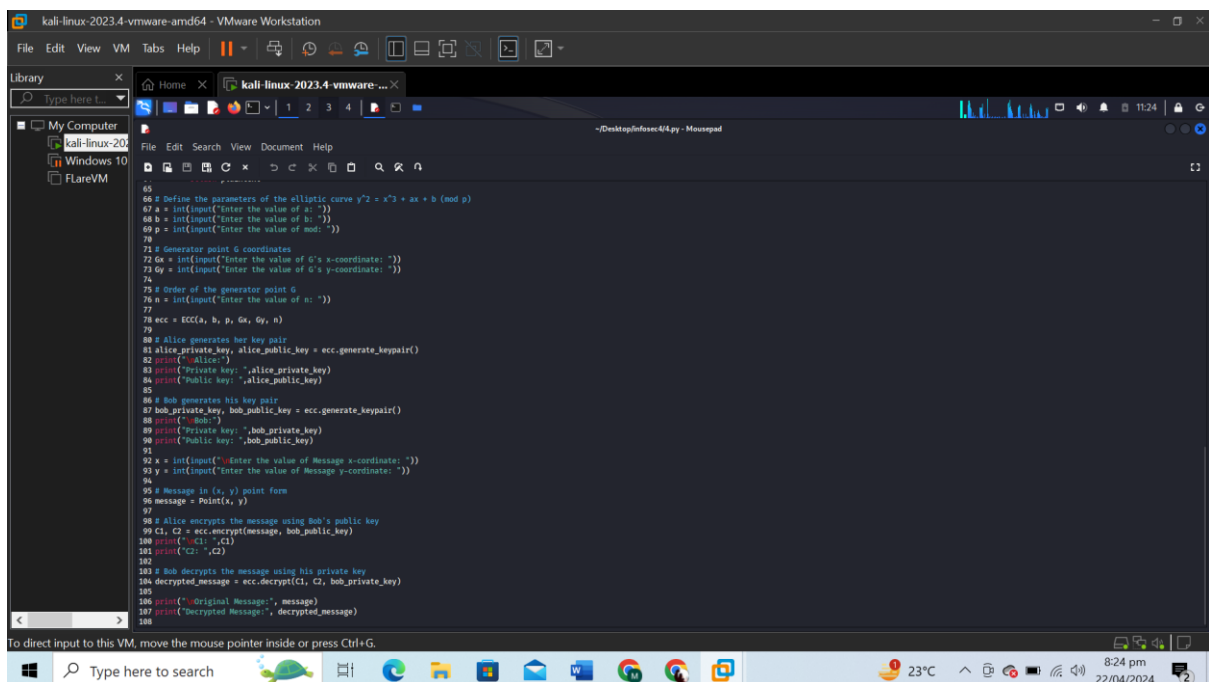
## Overall code:



```
1 import random
2
3 class Point:
4     def __init__(self, x, y):
5         self.x = x
6         self.y = y
7
8     def __str__(self):
9         if self.x == float('inf'):
10             return "Infinity"
11         else:
12             return f"({self.x}, {self.y})"
13
14 class ECC:
15     def __init__(self, a, b, p, Gx, Gy, n):
16         self.a = a
17         self.b = b
18         self.p = p
19         self.G = Point(Gx, Gy)
20         self.n = n # Order of the generator point G
21
22     def add_points(self, P, Q):
23         if P.x == float('inf'):
24             return Q
25         if Q.x == float('inf'):
26             return P
27         if P.x == Q.x and (P.y != Q.y or P.y == 0):
28             return Point(float('inf'), float('inf'))
29
30         if P.x == Q.x and P.y == Q.y:
31             s = ((3 * P.x ** 2 + self.a) * pow(2 * P.y, -1, self.p)) % self.p
32         else:
33             s = ((Q.y - P.y) * pow(Q.x - P.x, -1, self.p)) % self.p
34
35         x = (s ** 2 - P.x - Q.x) % self.p
36         y = (s * (P.x - x) - P.y) % self.p
37         return Point(x, y)
38
39     def scalar_multiply(self, k, P):
40         n = random.randint(1, self.n - 1)
41         P = self.add_points(P, self.G)
42         return P
```



```
40 y = (s * (P.x - x) - P.y) % self.p
41 return Point(x, y)
42
43 def scalar_multiply(self, k, P):
44     R = Point(float('inf'), float('inf'))
45     for i in range(k):
46         R = self.add_points(R, P)
47     return R
48
49 def generate_keypair(self):
50     print(input("Enter 1 for Random private key and 2 for input of private key: "))
51     if (pr==1):
52         private_key = random.randint(1, self.n - 1)
53     elif (pr==2):
54         private_key = int(input("Enter the value of private key: "))
55     public_key = self.scalar_multiply(private_key, self.G)
56     return private_key, public_key
57
58 def encrypt(self, message, recipient_public_key):
59     k = random.randint(1, self.n - 1)
60     C1 = self.scalar_multiply(k, self.G)
61     C2 = self.add_points(message, self.scalar_multiply(k, recipient_public_key))
62     return C1, C2
63
64 def decrypt(self, C1, C2, private_key):
65     s = self.scalar_multiply(private_key, C1)
66     print("m&C1: ", s)
67     plaintext = self.add_points(C2, Point(s.x, -s.y))
68     return plaintext
69
70 # Define the parameters of the elliptic curve y^2 = x^3 + ax + b (mod p)
71 a = int(input("Enter the value of a: "))
72 b = int(input("Enter the value of b: "))
73 p = int(input("Enter the value of mod: "))
74
75 # Generator point G coordinates
76 Gx = int(input("Enter the value of G's x-coordinate: "))
77 Gy = int(input("Enter the value of G's y-coordinate: "))
78
79 # Order of the generator point G
```



## Explanation of code:

```

1 class Point:
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def __str__(self):
7         if self.x == float('inf'):
8             return "Infinity"
9         else:
10            return f"({self.x}, {self.y})"

```

- This `class` defines a point on the elliptic curve. It has attributes `x` and `y` representing the coordinates of the point. The `str` method is overridden to provide a string representation of the point.

```
class ECC:
    def __init__(self, a, b, p, Gx, Gy, n):
        self.a = a
        self.b = b
        self.p = p
        self.G = Point(Gx, Gy)
        self.n = n # Order of the generator point G
```

- This class represents an Elliptic Curve Cryptography (ECC) system. It is initialized with the parameters of the elliptic curve equation ( $a, b, p$ ), the coordinates of the generator point ( $Gx, Gy$ ), and the order of the generator point ( $n$ ).

```
def add_points(self, P, Q):
    if P.x == float('inf'):
        return Q
    if Q.x == float('inf'):
        return P
    if P.x == Q.x and (P.y != Q.y or P.y == 0):
        return Point(float('inf'), float('inf'))

    if P.x == Q.x and P.y == Q.y:
        s = ((3 * P.x ** 2 + self.a) * pow(2 * P.y, -1, self.p)) % self.p
    else:
        s = ((Q.y - P.y) * pow(Q.x - P.x, -1, self.p)) % self.p

    x = (s ** 2 - P.x - Q.x) % self.p
    y = (s * (P.x - x) - P.y) % self.p
    return Point(x, y)
```

- This method performs point addition operation on the elliptic curve. It takes two points  $P$  and  $Q$  as input and returns their sum.

```
def scalar_multiply(self, k, P):
    R = Point(float('inf'), float('inf'))
    for _ in range(k):
        R = self.add_points(R, P)
    return R
```

- This method performs scalar multiplication operation on the elliptic curve. It takes an integer  $k$  and a point  $P$  as input, and returns the result of multiplying point  $P$  by scalar  $k$ .

```
def generate_keypair(self):
    pr=int(input("Enter 1 for Random private key and 2 for input of private key: "))
    if(pr==1):
        private_key = random.randint(1, self.n - 1)
    elif(pr==2):
        private_key = int(input("Enter the value of private key: "))
    public_key = self.scalar_multiply(private_key, self.G)
    return private_key, public_key
```

- This method generates a **key pair** (private key, public key) for ECC. It prompts the user to choose between generating a random private key or entering one manually. It then calculates the corresponding public key and returns both.

```
def encrypt(self, message, recipient_public_key):
    pr=int(input("Enter 1 for Random k value and 2 for input of k value: "))
    if(pr==1):
        k = random.randint(1, self.n - 1)
    elif(pr==2):
        k = int(input("Enter the value of k: "))

    C1 = self.scalar_multiply(k, self.G)
    C2 = self.add_points(message, self.scalar_multiply(k, recipient_public_key))
    return C1, C2
```

- This method **encrypts** a message using ECC. It generates a random number or user input k, calculates the ciphertext points **C1** and **C2**, and returns them.

```
def decrypt(self, C1, C2, private_key):
    S = self.scalar_multiply(private_key, C1)
    print("\nB*C1: ",S)
    plaintext = self.add_points(C2, Point(S.x, -S.y))
    return plaintext
```

- This method **decrypts** a message encrypted using ECC. It computes the shared secret S, then computes the plaintext message by subtracting the shared secret from the ciphertext.

```
# Alice encrypts the message using Bob's public key
C1, C2 = ecc.encrypt(message, bob_public_key)
print("\nC1: ",C1)
print("C2: ",C2)
```

- This line **encrypts** the message using **Bob's public key**.

```
3 # Bob decrypts the message using his private key
4 decrypted_message = ecc.decrypt(C1, C2, bob_private_key)
5
```

- This line **decrypts** the message using **Bob's private key**.

```
105
106 print("\nOriginal Message:", message)
107 print("Decrypted Message:", decrypted_message)
108
```

- These lines print the original message and the decrypted message.

### Output:

```
File Actions Edit View Help
(kali@kali)-[~/Desktop/Infosec_A4]
$ python3 4.py
Enter the value of a: 0
Enter the value of b: -4
Enter the value of mod: 257
Enter the value of G's x-coordinate: 2
Enter the value of G's y-coordinate: 2
Enter the value of n: 150

Bob:
Enter 1 for Random private key and 2 for input of private key: 2
Enter the value of private key: 101
Private key: 101
Public key: (197, 167)

Alice:
Enter the value of Message x-coordinate: 112
Enter the value of Message y-coordinate: 26
Enter 1 for Random k value and 2 for input of k value: 2
Enter the value of k: 41

C1: (136, 128)
C2: (246, 174)
nB*C1: (68, 84)

Original Message: (112, 26)
Decrypted Message: (112, 26)
```

### • SUMMARY

We explore Elliptic Curve Cryptography (ECC) in this assignment, putting key features into practice without the need for third-party libraries. We aim to achieve the following: elliptic curve validity verification, curve membership confirmation, ECC-based Diffie-Hellman key exchange, and public key encryption and decryption realization. We hope to strengthen our understanding of ECC principles and improve our practical application of them in practical cryptography scenarios through this investigation.

### • REFERENCES

<https://chat.openai.com/>

THE END...