Experiment 6

ELLIPTIC CURVE CRYPTOGRAPHY

6.1 Aim

To implement elliptic curve cryptography algorithm.

6.2 Theory

Elliptic-curve cryptography (ECC) is an approach to public-key cryptography based on the algebraic structure of elliptic curves over finite fields. ECC allows smaller keys compared to non-EC cryptography to provide equivalent security.It works based on the equation $y^2=x^3 + ax + b$.

6.3 Algorithm

1. START
2. Encode the plaintext message 'm'to be send as an x-y point 'pm'.It is the point pm that will be encrypted as a ciphertext and subsequently decrypted.
3. With the key exchange system ,an encryption/decryption system requires a point G and an elliptic group Eq(a,b) as parameters.
4. Each user A selects a private key nA and generates a public key PA = nA*G.
5. To encryt and send a message pm to B ,A chooses a random positive integer k and produce the ciphertext cm consisting of the points. cm = kG,pm + kpB,A has used B's public key pB
6. To decrypt the ciphertext,B multiplies the first point in the pair by B's secret key and subtracts the result from the second point.pm+kpB-nB(kG)=pm+k(nB G)-nB(kG)=pm.
7. STOP

6.4 Program

```
from random import randint
from hashlib import sha256

class ECPoint():

    def __init__(self, x, y, inf=0):
        self.x = x
```

```
        self.y = y
        self.inf = inf

    def __eq__(self, other):
        if (self.inf == 1) and (other.inf == 1):
            return True

        return (self.x == other.x) and (self.y == other.y)

    def __repr__(self):
        if self.inf == 1:
            return 'O'
        return '({}, {})'.format(self.x, self.y & 1)

    def __hash__(self):
        return hash(str(self))


class EllipticCurve():

    def __init__(self, p, g, a, b):
        self.p = p
        self.g = g
        self.a = a
        self.b = b

    def identity(self):
        return ECPoint(0, 0, 1)

    def is_valid(self, p):
        return p.y**2  % self.p == (p.x**3 + self.a*p.x + self.b) % self.p

    def random_point(self):
        m = randint(1, self.p)
        p = self.mul(self.g, m)
        while p == self.identity():
            m = randint(1, self.p)
            p = self.mul(self.g, m)

        return p

    def egcd(self, a, b):
        if a == 0:
            return (b, 0, 1)
        else:
            g, y, x = self.egcd(b % a, a)
            return (g, x - (b // a) * y, y)

    def modinv(self, a, m):
        g, x, y = self.egcd(a, m)
        if g != 1:
            raise Exception('Modular inverse does not exist')
        else:
```

```
            return x % m

    def add(self, p1, p2):
        if p1.inf == 1 and p2.inf == 1:
            return self.identity()
        if p1.inf == 1:
            return p2
        if p2.inf == 1:
            return p1

        if p1.x != p2.x:
          lam = ((p2.y - p1.y) * self.modinv((p2.x - p1.x) % self.p, self.p))
          % self.p
        else:
            if (p1 == self.neg(p2)):
                return self.identity()
            if (p1.y == 0):
                return self.identity()
            lam = ((3*p1.x**2 + self.a) * self.modinv(2 * p1.y, self.p))
            % self.p


        x3 = (lam**2 - p1.x - p2.x) % self.p
        y3 = ((p1.x - x3) * lam - p1.y) % self.p
        return ECPoint(x3, y3)

    def neg(self, p):
        return ECPoint(p.x, self.p - p.y)

    def sub(self, p1, p2):
        return self.add(p1, self.neg(p2))

    def mul(self, p, m):
        result = self.identity()
        addend = p

        while m:
            if m & 1:
                result = self.add(result, addend)

            addend = self.add(addend, addend)
            m >>= 1

        return result


def keygen():
    p = int('FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
    FFFFEFFFFFC2F', 16)
    a = 0
    b = 7
    g = ECPoint(int('79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DC
    E28D959F2815B16F81798', 16),
```

```
        int('483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47
        D08FFB10D4B8', 16))
    G = EllipticCurve(p, g, a, b)

    # generate private key
    x = randint(1, p)
    h = G.mul(g, x)


    return (x, G, g, p, h)


def encrypt(m, G, g, p, h):

    y = randint(1, p)
    c1 = G.mul(g, y)
    s = G.mul(h, y)
    hs = sha256(repr(s).encode('utf-8')).digest()
    c2 = bytearray([i ^ j for i,j in zip(m, hs)])
    return (c1, bytes(c2))


def decrypt(c, x, G):

    c1, c2 = c
    s = G.mul(c1, x)
    # m = c2 xor H(c1*x)
    hs = sha256(repr(s).encode('utf-8')).digest()
    m = bytearray([i ^ j for i,j in zip(c2, hs)])
    return bytes(m)


x, G, g, p, h = keygen()
m = input('Enter message: ').encode('utf-8')

c = encrypt(m, G, g, p, h)
print('Encrypted: {}'.format(c))
mp = decrypt(c, x, G)
print('Decrypted:\t{}'.format(mp.decode()))
assert m == mp
```

6.5 Output

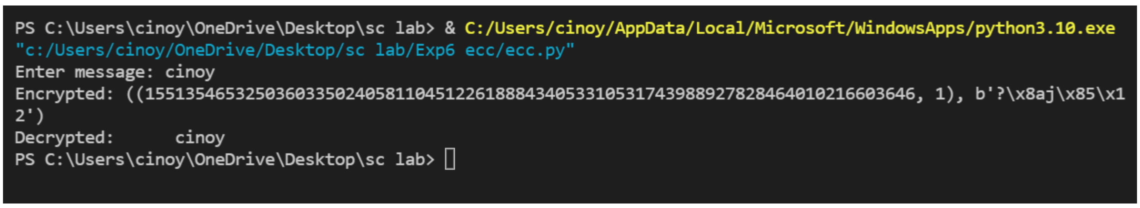

Figure 1: Encryption and Decryption using ECC

6.6 Result

The elliptic curve cryptography algorithm was implemented successfully.