Ex.No.8 23/09/2024	**IMPLEMENTATION OF ELLIPTIC CURVE CRYPTOSYSTEM**

AIM:

To implement elliptic curve key exchange and elliptic curve encryption algorithms.

THEORY

Affine points:

In elliptic curve cryptography (ECC), an affine point is represented as a pair of coordinates (x, y) that lie on the elliptic curve. These points are defined over a finite field and follow the curve's equation, typically y^2 = x^3 + ax + b . Affine points are used in ECC operations.

Elliptic Curve Discrete logarithms:

The elliptic curve discrete logarithm problem (ECDLP) involves finding the integer k given two points P and Q on an elliptic curve such that Q = k*P.ECDLP is computationally hard, making ECC secure for encryption, digital signatures, and key exchange.

Point Addition:

Point addition in ECC refers to adding two distinct points P and Q on an elliptic curve to get another point R. This operation follows specific algebraic rules that depend on the curve's equation. If P=Q, it becomes "point doubling."

Scalar Multiplication:

Scalar multiplication in ECC is the process of repeatedly adding a point P to itself k times, denoted as k*P . This is the fundamental operation in elliptic curve cryptography, used in key generation, encryption, and signatures.

Inverse of an affine point:

The inverse of an affine point P = (x, y) on an elliptic curve is -P = (x, -y). This is used in elliptic curve operations, such as point subtraction, where P - Q is defined as P+(-Q).

ALGORITHM:

Elliptic Curve Key Exchange

1. Elliptic Curve Parameters Selection:

Both parties agree on the public parameters:

a) A prime number p defining the finite field F_p.

b) Coefficients a and b defining the elliptic curve equation y^2 = x^3 + ax + b \mod p.

c) A base point G = (x_g, y_g) on the curve, and its order n.

2. Key Generation by Alice:

i) Alice chooses a random private key d_a , where d_a \in [1, n1].

ii) Alice computes her public key: $P_a = d_a * G$ (scalar multiplication).

iii) Alice sends P_a to Bob.

3. Key Generation by Bob:

i) Bob chooses a random private key d_b , where d_b \in [1, n1].

ii) Bob computes his public key: $P_b = d_b * G$.

iii) Bob sends P_B to Alice.

4. Shared Secret Calculation:

i) Alice computes the shared secret: $S_b = d_b * P_b$.

ii) Bob computes the shared secret: $S_b = d_b * P_a$.

iii) Both S_A and S_B are the same point on the curve due to the commutative property of scalar multiplication: $S = d_a * d_b * G$.

5. Shared Secret Extraction:

Both parties extract the x coordinate of the shared point S as the shared secret key.

Elliptic Curve Encryption:

1. Elliptic Curve Parameters:

Both parties agree on public parameters:

A prime p defining the finite field F_p.

Coefficients a and b for the elliptic curve equation y^2 = x^3 + ax + b mod p.

A base point G = (x_G, y_G) on the curve, and its order n.

2. Key Generation (By the Receiver):

The receiver (Bob) chooses a private key d_B where d_B in [1, n1].
The receiver computes their public key P_B = d_B* G and shares P_B with the sender (Alice).

3. Message Encoding (By the Sender):
Alice represents the plaintext message M as a point on the elliptic curve Min E(F_p). If the message is too large, a hashing or encoding function is used to map it to the curve.

4. Encryption (By the Sender):
Alice chooses a random integer kin [1, n1].
Alice computes two values:
C_1 = k* G (a point on the curve).
C_2 = M + k* P_B (the encrypted message point).
The ciphertext is (C_1, C_2), and Alice sends this to Bob.

5. Decryption (By the Receiver):
Upon receiving (C_1, C_2), Bob computes the shared secret S = d_B* C_1.
Bob then recovers the original message point by computing: M = C_2 S.
Since S = d_B* C_1 = d_B* (k* G) = k* P_B, the original message M is recovered correctly.

CODING:

ECC.py

```
import random
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad,unpad

class ECC:
    def __init__(self,a,b,p,g):
        self.a=a
        self.b=b
        self.p=p
        self.g=g
    def is_point_on_curve(self,point):
        x,y=point
        return (y**2)%self.p==(x**3+self.a*x+self.b)%self.p
```

```
    def point_addition(self,P,Q):
        if P==Q:
            return self.point_doubling(P)
        x1,y1=P
        x2,y2=Q
        if x1==x2 and y1 != y2:
            return None
        try:
            slope=(y2 - y1) * pow(x2 - x1,-1,self.p) % self.p
        except ValueError:
            raise ValueError("Cannot compute the modular inverse (points are not suitable for addition).")
        xr=(slope ** 2 - x1 - x2) % self.p
        yr=(slope * (x1 - xr) - y1) % self.p
        return xr,yr
    def point_doubling(self,P):
        x,y=P
        slope=(3*x**2+self.a)*pow(2*y,-1,self.p)%self.p
        xr=(slope**2-2*x)%self.p
        yr=(slope*(x-xr)-y)%self.p
        return xr,yr
    def scalar_multiplication(self,k,P):
        result=None
        addend=P
        while k:
            if k & 1:
                if result is None:
                    result=addend
                else:
                    result=self.point_addition(result,addend)

            addend=self.point_doubling(addend)
            k >>= 1
        return result
    def generate_private_key(self):
        return random.randint(1,self.p-1)
    def generate_public_key(self,private_key):
        return self.scalar_multiplication(private_key,self.g)
```

```
    def derive_shared_secret(self,private_key,public_key):
        return self.scalar_multiplication(private_key,public_key)

    def hash_shared_secret(self,secret_point):
        secret_x=secret_point[0]
        shared_key=hashlib.sha256(str(secret_x).encode()).digest()
        return shared_key

    def encrypt_message(self,public_key,message):
        ephemeral_private_key=self.generate_private_key()
        ephemeral_public_key=self.generate_public_key(ephemeral_private_key)
        shared_secret=self.derive_shared_secret(ephemeral_private_key,public_key)
        shared_key=self.hash_shared_secret(shared_secret)
        cipher=AES.new(shared_key,AES.MODE_CBC)
        ciphertext=cipher.encrypt(pad(message.encode(),AES.block_size))
        return (ciphertext,cipher.iv,ephemeral_public_key)

    def decrypt_message(self,private_key,ciphertext,iv,ephemeral_public_key):
        shared_secret=self.derive_shared_secret(private_key,ephemeral_public_key)
        shared_key=self.hash_shared_secret(shared_secret)
        cipher=AES.new(shared_key,AES.MODE_CBC,iv)
        plaintext=unpad(cipher.decrypt(ciphertext),AES.block_size)
        return plaintext.decode()
```

<u>main.py</u>

```
from ECC import ECC
def run_ecdh(ecc,alice_private_key,alice_public_key,bob_private_key,bob_public_key):
    alice_shared_secret=ecc.derive_shared_secret(alice_private_key,bob_public_key)
    print("Alice's Shared Secret:",alice_shared_secret)

    bob_shared_secret=ecc.derive_shared_secret(bob_private_key,alice_public_key)
    print("Bob's Shared Secret:",bob_shared_secret)

    if alice_shared_secret == bob_shared_secret:
        print("Key exchange successful! Shared secret is identical for both parties.")
    else:
        print("Key exchange failed! Shared secret mismatch.")
```

```
def run_ecies(ecc,bob_public_key,bob_private_key):
    message=input("Enter the Message to be encrypted.")
    ciphertext,iv,ephemeral_public_key=ecc.encrypt_message(bob_public_key,message)
    print("Ciphertext:",ciphertext)


decrypted_message=ecc.decrypt_message(bob_private_key,ciphertext,iv,ephemeral_public_
key)
    print("Decrypted Message:",decrypted_message)


def main():
    a=2
    b=3
    p=13
    g=(3,6)
    ecc=ECC(a,b,p,g)
    alice_private_key=ecc.generate_private_key()
    alice_public_key=ecc.generate_public_key(alice_private_key)
    bob_private_key=ecc.generate_private_key()
    bob_public_key=ecc.generate_public_key(bob_private_key)
    print("Alice's Private Key:",alice_private_key)
    print("Alice's Public Key:",alice_public_key)
    print("Bob's Private Key:",bob_private_key)
    print("Bob's Public Key:",bob_public_key)
    choices={'1': run_ecdh,'2': run_ecies}
    print("\nChoose an option:")
    print("1: Elliptic Curve Diffie-Hellman Key Exchange (ECDH)")
    print("2: Elliptic Curve Integrated Encryption Scheme (ECIES)")
    choice=input("Enter your choice (1 or 2): ")
    if choice == '1':
        choices[choice](ecc,alice_private_key,alice_public_key,bob_private_key,bob_public_key)

    elif choice == '2':
        choices[choice](ecc,bob_public_key,bob_private_key)
    else:
        print("Invalid choice. Please enter 1 or 2.")
```

```
if __name__ == "__main__":
    main()
```

SCREEN SHOTS:

```
C:\Users\gokul\AppData\Local\Programs\Python\Python310\python.exe "C:\
Alice's Private Key: 5
Alice's Public Key: (3, 7)
Bob's Private Key: 7
Bob's Public Key: (3, 6)

Choose an option:
1: Elliptic Curve Diffie-Hellman Key Exchange (ECDH)
2: Elliptic Curve Integrated Encryption Scheme (ECIES)
Enter your choice (1 or 2): 1
Alice's Shared Secret: (3, 7)
Bob's Shared Secret: (3, 7)
Key exchange successful! Shared secret is identical for both parties.
```

```
Choose an option:
1: Elliptic Curve Diffie-Hellman Key Exchange (ECDH)
2: Elliptic Curve Integrated Encryption Scheme (ECIES)
Enter your choice (1 or 2): 2
Enter the Message to be encrypted.gokul
Ciphertext: b'z\n\xa9\x1ey\x08\x7f\xd0T\xb0Z\x84\x92PB\xd3'
Decrypted Message: gokul
```

RESULT:

Thus, we have implemented elliptic curve key exchange and elliptic curve encryption algorithms successfully.

Evaluation

Parameter	Max Marks	Marks Obtained
Uniqueness of the Code	50	
Completion of experiment on time	10	
Documentation	15	
Total	75	
Signature of the faculty with Date		