|  |  |
| --- | --- |
| 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:

1. A prime number p defining the finite field Fp.
2. Coefficients a and b defining the elliptic curve equation y^2 = x^3 + ax + b \mod p.
3. A base point G = (xg, yg) on the curve, and its order n.

2. Key Generation by Alice:

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

ii) Alice computes her public key: Pa = da \* G (scalar multiplication).

iii) Alice sends Pa to Bob.

3. Key Generation by Bob:

i) Bob chooses a random private key db , where d**b** \in [1, n1].

ii) Bob computes his public key: Pb = db \* G.

iii) Bob sends P\_B to Alice.

4. Shared Secret Calculation:

i) Alice computes the shared secret: Sb = db \* Pb.

ii) Bob computes the shared secret: Sb = db \* Pa.

iii) Both S\_A and S\_B are the same point on the curve due to the commutative property of scalar multiplication: S = da \* db \* 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()

main.py

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)

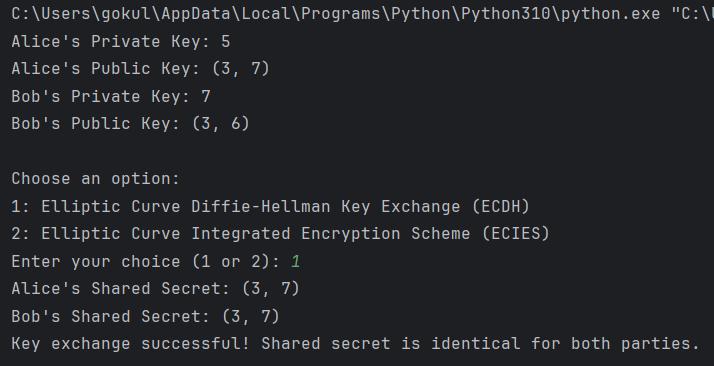
else:

print("Invalid choice. Please enter 1 or 2.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

|  |
| --- |
| **SCREEN SHOTS:** |

****

****

|  |
| --- |
| **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 |  |  |