Assignment-07

ECC-Elliptical Curve Cryptography
LABTask-7

CSE 459: Cryptography and Network Security

Submitted By:

Jayanth CT

AP22110011147

CSE-Y

Lab Date: 10/03/2025

Submission Date: 11/03/2025

Department Computer Science and Engineering

School of Engineering and Sciences

SRM University–AP

Amaravati, Andhra Pradesh – 522 240, India

GitHub Link :

https://github.com/jayanthct/CryptographyAndNetworks/tree/main/LabTask7

Q1) ECC

</>

Code Java:

```
package LabTask7;
   import java.math.BigInteger;
import java.security.SecureRandom;

public class ECC {

      private static final BigInteger P = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
", 16); // Prime field
      private static final BigInteger A = BigInteger.ZERO; // Curve coefficient 'a'
      private static final BigInteger B = new BigInteger("7"); // Curve coefficient 'b'
      private static final BigInteger Gx = new
BigInteger("79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
", 16); // Base point x
      private static final BigInteger Gy = new
BigInteger("483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8
", 16); // Base point y
      private static final BigInteger N = new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD03641
41", 16); // Order of G

      // Point class to represent ECC points
      static class Point {
         BigInteger x, y;

         Point(BigInteger x, BigInteger y) {
            this.x = x;
            this.y = y;
         }
      }

      // Point addition on the elliptic curve
      private static Point pointAdd(Point p1, Point p2) {
         if (p1.x.equals(BigInteger.ZERO) && p1.y.equals(BigInteger.ZERO)) return p2;
         if (p2.x.equals(BigInteger.ZERO) && p2.y.equals(BigInteger.ZERO)) return p1;

         BigInteger lambda = p2.y.subtract(p1.y).multiply(p2.x.subtract(p1.x).modInverse(P)).mod(P);
         BigInteger xr = lambda.pow(2).subtract(p1.x).subtract(p2.x).mod(P);
         BigInteger yr = lambda.multiply(p1.x.subtract(xr)).subtract(p1.y).mod(P);

         return new Point(xr, yr);
      }

      // Point doubling on the elliptic curve
```

```
    private static Point pointDouble(Point p) {
        BigInteger lambda = p.x.pow(2).multiply(BigInteger.valueOf(3)).add(A)
                .multiply(p.y.multiply(BigInteger.valueOf(2)).modInverse(P)).mod(P);
        BigInteger xr = lambda.pow(2).subtract(p.x.multiply(BigInteger.valueOf(2))).mod(P);
        BigInteger yr = lambda.multiply(p.x.subtract(xr)).subtract(p.y).mod(P);

        return new Point(xr, yr);
    }

    // Scalar multiplication using double-and-add
    private static Point scalarMultiply(BigInteger k, Point p) {
        Point result = new Point(BigInteger.ZERO, BigInteger.ZERO); // Neutral element
        Point addend = p;

        while (k.compareTo(BigInteger.ZERO) > 0) {
            if (k.and(BigInteger.ONE).equals(BigInteger.ONE)) {
                result = pointAdd(result, addend);
            }
            addend = pointDouble(addend);
            k = k.shiftRight(1);
        }

        return result;
    }

    public static void main(String[] args) {
        SecureRandom random = new SecureRandom();

        // Generate private keys for Alice and Bob
        BigInteger dA = new BigInteger(256, random).mod(N); // Alice's private key
        BigInteger dB = new BigInteger(256, random).mod(N); // Bob's private key

        // Calculate public keys
        Point G = new Point(Gx, Gy);
        Point QA = scalarMultiply(dA, G); // Alice's public key
        Point QB = scalarMultiply(dB, G); // Bob's public key

        // Calculate shared secrets
        Point sharedSecretA = scalarMultiply(dA, QB);
        Point sharedSecretB = scalarMultiply(dB, QA);

        // Display results
        System.out.println("Alice's Private Key: " + dA.toString(16));
        System.out.println("Bob's Private Key: " + dB.toString(16));
        System.out.println("Alice's Public Key: (" + QA.x.toString(16) + ", " + QA.y.toString(16) +
")");
        System.out.println("Bob's Public Key: (" + QB.x.toString(16) + ", " + QB.y.toString(16) +
")");
        System.out.println("Shared Secret (Alice): (" + sharedSecretA.x.toString(16) + ", " +
sharedSecretA.y.toString(16) + ")");
        System.out.println("Shared Secret (Bob): (" + sharedSecretB.x.toString(16) + ", " +
```

```
sharedSecretB.y.toString(16) + ")");
        }
    }
```

```
Alice's Private Key: f7b6aaa8fb0bc7bd9f3e4598fa5bf00512ecf7e7d1214109eaea6e286cc42f94
Bob's Private Key: f0206a937c8375502ba1a3f1e58e8aecc0bc4f899d84795d508dd1f46540bf6
Alice's Public Key: (9f893da912ea068b1d3083d66aee1aa5181f8e065f5bc82ca798d85b8e362b1c, bed1a9851de2cb7ecaea4c83f07730c7eb2e95759729280d8f5dd81d84ab93b6)
Bob's Public Key: (f4f944cc53b11a080022f5359ec2fb42a7e5d1b3b5060c69e18bd515c6a6c68f, 99c8577b7d563ac18b383fe5aafaadf0877d027e71135fe109a82bf91ce5746c)
Shared Secret (Alice): (c810f295eefa9c75f12f5942cd0b338da328f3580783810309b124fa74c40c0c, 42d6035da1ef3ad108abb2652c0b6261d67237ebe59fbc4c13e1293d32693475)
Shared Secret (Bob): (c810f295eefa9c75f12f5942cd0b338da328f3580783810309b124fa74c40c0c, 42d6035da1ef3ad108abb2652c0b6261d67237ebe59fbc4c13e1293d32693475)
```

Q2) ECC and AES

</>

Java Code:

```
package LabTask7;

import java.math.BigInteger;
import java.security.SecureRandom;
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class MathematicalECC_AES {
    public static void main(String[] args) {
        String message = "Hell0 SRM AP";

        System.out.println("Curve 1: secp256k1");
        ECCCurve curve1 = new ECCCurve(
                new BigInteger("0"),                // a
                new BigInteger("7"),                // b
                new
BigInteger("FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
", 16) // p
        );
        performECC_AES(curve1, message);

        System.out.println("\nCurve 2: secp192r1");
        ECCCurve curve2 = new ECCCurve(
                new BigInteger("-3"),               // a
                new BigInteger("2455155546008943817740293915197451784769108058161191238065"),
// b
                new BigInteger("6277101735386680763835789423207666416083908700390324961279")
// p
        );
        performECC_AES(curve2, message);
    }

    private static void performECC_AES(ECCCurve curve, String message) {
        try {
            // Generate a random private key
            BigInteger privateKey = new BigInteger(curve.p.bitLength(), new SecureRandom());
            ECCPoint publicKey = curve.multiplyPoint(curve.basePoint(), privateKey);

            // Create a shared secret from the public key and private key
            ECCPoint sharedSecret = curve.multiplyPoint(publicKey, privateKey);
            byte[] sharedKey = sharedSecret.x.toByteArray();

            // AES Encryption
            SecretKeySpec secretKey = new SecretKeySpec(sharedKey, 0, 16, "AES");
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            cipher.init(Cipher.ENCRYPT_MODE, secretKey);
            byte[] encryptedMessage = cipher.doFinal(message.getBytes());
```

```
            System.out.println("Encrypted: " + Base64.getEncoder().encodeToString(encryptedMessage));

            // AES Decryption
            cipher.init(Cipher.DECRYPT_MODE, secretKey);
            byte[] decryptedMessage = cipher.doFinal(encryptedMessage);
            System.out.println("Decrypted: " + new String(decryptedMessage));

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class ECCCurve {
    public BigInteger a, b, p;

    public ECCCurve(BigInteger a, BigInteger b, BigInteger p) {
        this.a = a;
        this.b = b;
        this.p = p;
    }

    public ECCPoint basePoint() {
        // Returning a fixed base point for simplicity
        return new ECCPoint(BigInteger.valueOf(4), BigInteger.valueOf(20));
    }

    public ECCPoint addPoints(ECCPoint P, ECCPoint Q) {
        if (P.isInfinity()) return Q;
        if (Q.isInfinity()) return P;

        BigInteger lambda;
        if (P.x.equals(Q.x)) {
            if (!P.y.equals(Q.y) || P.y.equals(BigInteger.ZERO)) return ECCPoint.infinity();
            lambda = (P.x.pow(2).multiply(BigInteger.valueOf(3)).add(a))
                    .multiply(P.y.multiply(BigInteger.valueOf(2)).modInverse(p)).mod(p);
        } else {
            lambda = (Q.y.subtract(P.y)).multiply(Q.x.subtract(P.x).modInverse(p)).mod(p);
        }

        BigInteger x3 = lambda.pow(2).subtract(P.x).subtract(Q.x).mod(p);
        BigInteger y3 = lambda.multiply(P.x.subtract(x3)).subtract(P.y).mod(p);

        return new ECCPoint(x3, y3);
    }

    public ECCPoint multiplyPoint(ECCPoint P, BigInteger n) {
        ECCPoint R = ECCPoint.infinity();
        ECCPoint Q = P;

        while (n.compareTo(BigInteger.ZERO) > 0) {
            if (n.and(BigInteger.ONE).equals(BigInteger.ONE)) {
                R = addPoints(R, Q);
            }
            Q = addPoints(Q, Q);
```

```
            n = n.shiftRight(1);
        }
        return R;
    }
}

class ECCPoint {
    public BigInteger x, y;

    public ECCPoint(BigInteger x, BigInteger y) {
        this.x = x;
        this.y = y;
    }

    public static ECCPoint infinity() {
        return new ECCPoint(null, null);
    }

    public boolean isInfinity() {
        return x == null || y == null;
    }
}
```

```
Curve 1: secp256k1
Encrypted: YAG8pJTjR1BWl5M3TateNg==
Decrypted: Hell0 SRM AP

Curve 2: secp192r1
Encrypted: Pgz0BtaKyhMDPf9dekBQ2w==
Decrypted: Hell0 SRM AP
```