MANIPAL INSTITUTE OF TECHNOLOGY
BENGALURU
(A constituent unit of MAHE, Manipal)

B.TECH. SIXTH SEMESTER

Information Technology
(B. Tech CSE (CYBER SECURITY))

APPLIED CRYPTOGRAPHY LAB

IT_3261

LABORATORY MANUAL

CONTENTS

Course Objectives

- To defend the security attacks on information systems with secure algorithms.
- To learn advanced concepts of cryptography
- To study concepts of security

Course Outcomes

At the end of this course, students will have the

- Analyze cryptographic concepts such as encryption, decryption, key management, cryptographic algorithms, and protocols.
- Analyze the security of cryptographic systems, including understanding potential vulnerabilities and threats.
- Apply cryptography in various real-world scenarios, including data protection, secure communication, and authentication.
- Implementing cryptographic algorithms and protocols using programming languages or cryptographic libraries.

Evaluation plan

- Internal Assessment Marks : 60 marks

  Continuous evaluation: 36 Marks
  □ The Continuous evaluation assessment will depend on punctuality, designing right algorithm, converting algorithm into an efficient program, maintaining the observation note and answering the questions in viva voce.

  □ Project Evaluation :24 Marks

- End semester assessment of 2 hour duration: 40 marks

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session
2. Be in time and follow the institution dress code
3. Must Sign in the log register provided
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum

In- Lab Session Instructions

- Follow the instructions on the allotted exercises
- Show the program and results to the instructors on completion of experiments
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Programs should perform input validation (Data type, range error, etc.) and give appropriate error messages and suggest corrective actions.
    - Comments should be used to give the statement of the problem and every function should indicate the purpose of the function, inputs and outputs.
    - Statements within the program should be properly indented.
    - Use meaningful names for variables and functions.
    - Make use of constants and type definitions wherever needed.
    - Programs should include necessary time analysis part (Operation count /Step count method)
- Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
    - Solved exercise

    - Lab exercises - to be completed during lab hours
    - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition lab with the permission of the faculty concerned.
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.
- A sample note preparation is given as a model for observation.
- You may write scripts/programs to automate the experimental analysis of algorithms and compare with the theoretical result.
- You may use spreadsheets to plot the graph.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

LAB NO: 1 **Date:**

Finite Field Operations in Cryptography

Objectives:

In this lab, student will be able to:

- Recall the concepts learnt in Cryptography
- Implement basic techniques

Description: A cryptosystem is an implementation of cryptographic techniques and their accompanying infrastructure to provide information security services. A cryptosystem is also referred to as a cipher system. The Finite Field Operations program is designed to demonstrate fundamental operations within a finite field, a concept widely used in cryptography and error-correcting codes. A finite field, often denoted as GF(p), is a mathematical structure where arithmetic operations are performed modulo a prime number. In this program, the user is prompted to input a prime modulus and two elements of the finite field. The implemented operations include addition and multiplication, both executed within the confines of the specified finite field.

I. SOLVED EXERCISE:

1) Write a Java program to perform addition and multiplication in a finite field with a given prime modulus. Test your program with different inputs.

Description : The "Finite Field Operations in Cryptography" program is designed to provide a practical exploration of fundamental operations within finite fields, a key concept in cryptographic applications. This lab exercise allows users to input a prime modulus and two finite field elements, demonstrating addition and multiplication operations performed within the specified finite field structure.

```
import java.util.Scanner;

public class FiniteFieldOperations {
    static int primeModulus;

    // Function to perform addition in a finite field
    static int add(int a, int b) {
        return (a + b) % primeModulus;
    }

    // Function to perform multiplication in a finite field
    static int multiply(int a, int b) {
        return (a * b) % primeModulus;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Get the prime modulus as input
        System.out.print("Enter the prime modulus for the finite field: ");
        primeModulus = scanner.nextInt();

        // Get two elements of the finite field as input
        System.out.print("Enter the first element of the finite field: ");
        int element1 = scanner.nextInt();

        System.out.print("Enter the second element of the finite field: ");
        int element2 = scanner.nextInt();

        // Perform addition
        int sum = add(element1, element2);
        System.out.println("Sum in the finite field: " + sum);

        // Perform multiplication
        int product = multiply(element1, element2);
        System.out.println("Product in the finite field: " + product);

        scanner.close();
    }
```

II LAB EXERCISES

1) Write a Java program to find the greatest common divisor (GCD) of two numbers using Euclid's algorithm. Test your program with various pairs of numbers.

2) Implement a Java program that performs arithmetic operations (addition, subtraction, multiplication, and division) within a finite field GF(p), where "p" is a prime number provided by the user. The program should prompt the user to input two elements of the finite field and then perform the specified operations. Ensure that division by zero is handled appropriately, and the results are displayed modulo the prime modulus.

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

[OBSERVATION SPACE – LAB1]

LAB NO: 2 **Date:**

Number Theory

Objectives:

In this lab, student will be able to:

- Familiarize with fundamentals of problem solving with the help of algorithms.
- Realize that for a problem there can be multiple solutions with different techniques.

Description: The Finite Fields and Number Theory lab is an educational and hands-on exercise designed to introduce participants to the fundamental concepts and practical implementation of finite fields and number theory in the realm of cryptography. The lab aims to familiarize learners with various classic mathematical concepts and their cryptographic significance.

I. SOLVED EXERCISE:

1) Write a Java program to determine whether a given number is a probable prime using Fermat's Little Theorem. Allow the user to input the number and the desired number of iterations for the test.

Description: The "Fermat's Little Theorem Probable Primality Test" Java program is designed to assess the likelihood of a given number being a prime using Fermat's Little Theorem. Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then $a_{p-1}\equiv 1(\mathrm{mod}\,p)$. This theorem forms the basis for a probabilistic primality test.

Program

```
import java.util.Scanner;
import java.util.Random;
import java.math.BigInteger;

public class FermatPrimalityTest {

    // Function to perform modular exponentiation (a^b mod m)
    static BigInteger power(BigInteger a, BigInteger b, BigInteger m) {
        BigInteger result = BigInteger.ONE;
        a = a.mod(m);

        while (!b.equals(BigInteger.ZERO)) {
            // If b is odd, multiply result with a
            if (b.mod(BigInteger.valueOf(2)).equals(BigInteger.ONE)) {
                result = result.multiply(a).mod(m);
            }

            // b must be even now
            b = b.divide(BigInteger.valueOf(2));
            a = a.multiply(a).mod(m);
        }

        return result;
    }
    // Function to perform Fermat's Little Theorem test
    static boolean fermatTest(BigInteger n, int iterations) {
        if (n.equals(BigInteger.valueOf(2)) || n.equals(BigInteger.valueOf(3))) {
            return true;
        }

        if (n.equals(BigInteger.ONE) || n.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO)) {
            return false;
        }

        Random random = new Random();

        for (int i = 0; i < iterations; i++) {
            // Choose a random integer between 2 and n-2
            BigInteger a = new BigInteger(n.bitLength() - 2, random).add(BigInteger.valueOf(2));

            // Check if a^(n-1) is congruent to 1 mod n
            if (!power(a, n.subtract(BigInteger.ONE), n).equals(BigInteger.ONE)) {
                return false; // n is composite
            }
        }

        return true; // n is probably prime
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Get the number to be tested from the user
        System.out.print("Enter the number to be tested: ");
        BigInteger number = scanner.nextBigInteger();

        // Get the number of iterations for the test
        System.out.print("Enter the number of iterations for the test: ");
        int iterations = scanner.nextInt();

        // Perform Fermat's Little Theorem test
        boolean isProbablePrime = fermatTest(number, iterations);

        // Display the result
        if (isProbablePrime) {
            System.out.println(number + " is probably a prime number.");
        } else {
            System.out.println(number + " is composite.");
        }

        scanner.close();
    }
}
```

II. LAB EXERCISES

1). Implement a Java program to solve a system of linear congruences using the Chinese Remainder Theorem.

2). Write a Java program to find the modular inverse of a number using the Extended Euclidean Algorithm.

3). Write a Java program to calculate Euler's Totient Function (phi) for a given positive integer.

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

[OBSERVATION SPACE – LAB2]

LAB NO: 3 **Date:**

Asymmetric Encryption

Objectives:

In this lab, student will be able to:

- Understanding Asymmetric Encryption Concepts
- Key Generation and Management

Description: The objectives of an Asymmetric Encryption lab typically involve introducing students to the fundamental concepts of classical or traditional encryption techniques. These objectives aim to provide students with a foundational understanding of how historical encryption methods work and their vulnerabilities.

I. SOLVED EXERCISE:

1) Develop a Java program to generate RSA public and private key pairs.

Description: In this program, the program generates RSA public and private key pairs. The user is not prompted for input in this specific program, as it focuses on the key pair generation process.

Program

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.NoSuchAlgorithmException;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;

public class GenerateRSAKeyPair {

    public static void main(String[] args) {
        try {
            // Generate RSA key pair
            KeyPair keyPair = generateRSAKeyPair();

            // Get public and private keys
            PublicKey publicKey = keyPair.getPublic();
            PrivateKey privateKey = keyPair.getPrivate();

            // Convert keys to base64-encoded strings
            String publicKeyBase64 = encodeToBase64(publicKey.getEncoded());
            String privateKeyBase64 = encodeToBase64(privateKey.getEncoded());
```

II. LAB EXERCISES

1). Write a Java program to create and verify digital signatures using the RSA.
2). Implement a system that can detect and recover from key compromises without exposing sensitive information.
3). Develop a program that incorporates a zero-knowledge proof to demonstrate possession of a private key without revealing the key itself.

[OBSERVATION SPACE – LAB3]

[OBSERVATION SPACE – LAB3]

[OBSERVATION SPACE – LAB3]

[OBSERVATION SPACE – LAB3]

[OBSERVATION SPACE – LAB3]

[OBSERVATION SPACE – LAB3]

LAB NO: 4 **Date:**

Elliptic Curve Cryptography

Objectives:

In this lab, student will be able to:

- Understand the algebraic properties of elliptic curves
- Analyze the key generation, encryption, and decryption performance of ECC

I. SOLVED EXERCISE:

1). Develop a program to generate Elliptic Curve Cryptography (ECC) key pairs. Support different elliptic curves and key lengths.

Description: Elliptic Curve Cryptography (ECC) is a modern and efficient public-key cryptography algorithm widely used for securing communications, digital signatures, and key exchange. Unlike traditional public-key algorithms such as RSA or DSA, ECC is based on the mathematics of elliptic curves over finite fields.

Program

```
#include <stdio.h>
#include <stdlib.h>
#include <openssl/ec.h>
#include <openssl/obj_mac.h>

void generateECCKeyPair(const char *curveName) {
    EC_KEY *eccKey = EC_KEY_new_by_curve_name(OBJ_sn2nid(curveName));
    if (!eccKey) {
        fprintf(stderr, "Error creating ECC key\n");
        return;
    }

    if (EC_KEY_generate_key(eccKey) != 1) {
        fprintf(stderr, "Error generating ECC key\n");
        EC_KEY_free(eccKey);
        return;
    }

// Get public and private keys
    const EC_POINT *publicKey = EC_KEY_get0_public_key(eccKey);
    const BIGNUM *privateKey = EC_KEY_get0_private_key(eccKey);

    // Print public key
    char *publicKeyHex = EC_POINT_point2hex(EC_KEY_get0_group(eccKey), publicKey,
POINT_CONVERSION_UNCOMPRESSED, NULL);
    printf("Public Key:\n%s\n", publicKeyHex);
    OPENSSL_free(publicKeyHex);
    // Print private key
    char *privateKeyHex = BN_bn2hex(privateKey);
    printf("\nPrivate Key:\n%s\n", privateKeyHex);
    OPENSSL_free(privateKeyHex);

    // Clean up
    EC_KEY_free(eccKey);
}

int main() {
    const char *curveName = "secp256r1";
    generateECCKeyPair(curveName);

    return 0;
}
```

II. LAB EXERCISES

1). Write a program to perform digital signatures using the Elliptic Curve Digital Signature Algorithm.
2). Implement the ECDSA algorithm for signing and verifying messages using an elliptic curve. Your implementation should be able to handle key generation, signature generation, and signature verification.
3). Write a program to compress and decompress elliptic curve points. The compression function should take a point on the curve and output a compressed form, while the decompression function should reverse the process.

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

[OBSERVATION SPACE – LAB4]

LAB NO: 5 **Date:**

Elliptic Curve Cryptography II

I. SOLVED EXERCISE:

1). Implement a basic homomorphic encryption scheme, such as Paillier encryption. Allow users to perform addition or multiplication on encrypted values and decrypt the result.

Description: Paillier encryption is a probabilistic asymmetric algorithm used for public-key cryptography. Named after its creator Pascal Paillier, this encryption scheme is primarily designed for homomorphic encryption, which allows computations on encrypted data without decrypting it. Paillier encryption is particularly useful in privacy-preserving applications and secure multiparty computations.

Program

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

public class PaillierEncryption {

    private static final int BIT_LENGTH = 512; // Adjust the bit length as needed
    private static final int CERTAINTY = 64; // Adjust the certainty as needed

    private BigInteger p, q, n, nSquared, g, lambda, mu;

    public PaillierEncryption() {
        // Key generation
        generateKeyPair();
    }
    private void generateKeyPair() {
        // Step 1: Choose two large random prime numbers p and q
        p = generateRandomPrime();
        q = generateRandomPrime();

        // Step 2: Compute n = p * q
        n = p.multiply(q);
        nSquared = n.multiply(n);

        // Step 3: Compute Carmichael's totient function lambda = lcm(p-1, q-1)
        lambda = lcm(p.subtract(BigInteger.ONE), q.subtract(BigInteger.ONE));

        // Step 4: Choose a random integer g where 1 < g < n^2 and g is in the group Zn^2*
        g = generateRandomZnStar();

        // Step 5: Compute mu, the modular multiplicative inverse of lambda (lambda^-1 mod
n)
        mu = lambda.modInverse(n);
    }
```

```
  private BigInteger generateRandomPrime() {
    return BigInteger.probablePrime(BIT_LENGTH, new SecureRandom());
  }

  private BigInteger generateRandomZnStar() {
    BigInteger rand;
    do {
      rand = new BigInteger(BIT_LENGTH, new SecureRandom());
    } while (rand.compareTo(nSquared) >= 0 || rand.gcd(n).intValue() != 1);
    return rand;
  }
  private BigInteger lcm(BigInteger a, BigInteger b) {
    return a.multiply(b).divide(a.gcd(b));
  }

  public BigInteger encrypt(BigInteger plaintext) {
    BigInteger r = generateRandomZnStar();
    BigInteger ciphertext = g.modPow(plaintext, nSquared).multiply(r.modPow(n,
nSquared)).mod(nSquared);
    return ciphertext;
  }

  public BigInteger add(BigInteger ciphertext1, BigInteger ciphertext2) {
    return ciphertext1.multiply(ciphertext2).mod(nSquared);
  }

  public BigInteger multiply(BigInteger ciphertext, int scalar) {
    return ciphertext.modPow(BigInteger.valueOf(scalar), nSquared);
  }

  public BigInteger decrypt(BigInteger ciphertext) {
    BigInteger L = ciphertext.modPow(lambda,
nSquared).subtract(BigInteger.ONE).divide(n);
    return L.multiply(mu).mod(n);
  }

  public static void main(String[] args) {
    PaillierEncryption paillier = new PaillierEncryption();

    // Encryption
    BigInteger plaintext1 = new BigInteger("123");

    BigInteger plaintext2 = new BigInteger("456");

    BigInteger ciphertext1 = paillier.encrypt(plaintext1);
    BigInteger ciphertext2 = paillier.encrypt(plaintext2);

    // Homomorphic addition
    BigInteger sumCiphertext = paillier.add(ciphertext1, ciphertext2);
    BigInteger decryptedSum = paillier.decrypt(sumCiphertext);
System.out.println("Plaintext 1: " + plaintext1);
        System.out.println("Plaintext 2: " + plaintext2);
        System.out.println("Encrypted 1: " + ciphertext1);
        System.out.println("Encrypted 2: " + ciphertext2);
        System.out.println("Homomorphic Addition Result: " + decryptedSum);

        // Homomorphic multiplication (scalar)
        int scalar = 3;
        BigInteger productCiphertext = paillier.multiply(ciphertext1, scalar);
        BigInteger decryptedProduct = paillier.decrypt(productCiphertext);

        System.out.println("\nHomomorphic Multiplication (Scalar) Result: " +
decryptedProduct);
    }
}
```

II. LAB EXERCISES

1). Design a program to that performs key exchange using Elliptic Curve Diffie-Hellman.
2). Explore and implement protocols based on elliptic curve isogenies, such as SIDH (Supersingular Isogeny Diffie-Hellman). Develop a program that demonstrates key exchange using isogenies and handles public key generation, private key generation, and shared secret computation.
3). Develop functions for signing and verifying homomorphically encrypted messages while preserving the confidentiality of the underlying data.

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

[OBSERVATION SPACE – LAB5]

LAB NO: 6 **Date:**

Hash Functions and MAC

I. SOLVED EXERCISE:

1). Write a program to detect collisions in a set of hash values.

Description : A collision in the context of hash functions refers to the situation where two different inputs produce the same hash value. Ideally, in a good hash function, each unique input should result in a unique hash output. However, due to the finite size of hash values and an infinite number of possible inputs, collisions are inevitable. The probability of collisions depends on the quality of the hash function and the size of the hash space.

Program

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.HashSet;

public class HashCollisionDetection {

   public static String calculateMD5(String input) throws NoSuchAlgorithmException {
      MessageDigest md = MessageDigest.getInstance("MD5");
      byte[] hashedBytes = md.digest(input.getBytes());

      // Convert bytes to hexadecimal representation
      StringBuilder sb = new StringBuilder();
      for (byte b : hashedBytes) {
         sb.append(String.format("%02x", b));
      }

      return sb.toString();
   }
   public static void main(String[] args) throws NoSuchAlgorithmException {
      int numIterations = 100000;
      HashSet<String> hashSet = new HashSet<>();

      for (int i = 0; i < numIterations; i++) {
         String input = Integer.toString(i);
         String hashValue = calculateMD5(input);

         if (hashSet.contains(hashValue)) {
            System.out.println("Collision found for inputs: " + input + " and " +
hashSet.stream()
                  .filter(existingHash ->
existingHash.equals(hashValue)).findFirst().orElse(null));
            break;
         } else {
            hashSet.add(hashValue);
         }
      }
   }
```

II. LAB EXERCISE:

1) Implement a MAC algorithm that uses a cryptographic hash function (e.g., SHA-256) to generate the code. Ensure that the MAC provides both message integrity and authenticity.

2) Write a program to generate two different inputs that result in the same MD5 hash.

3) Create a program that verifies the integrity of a message using a MAC. Demonstrate how an attacker attempting to modify the message would be detected.

4) Write a program that performs authenticated encryption using a mode like GCM (Galois/Counter Mode). Encrypt a message and generate an authentication tag, then decrypt and verify the authenticity.

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

[OBSERVATION SPACE – LAB6]

LAB NO: 7 **Date:**

Public-key Cryptography

Objectives:

In this lab, student will be able to:

- Understanding of the fundamental concepts behind public-key cryptography
- Practice the processes of encrypting and decrypting messages

Description: Number theory is a branch of mathematics that deals with the properties and relationships of integers and their fundamental properties. It has numerous applications in various fields, including cryptography, computer science, and algorithms. When it comes to number theory algorithms, the main objectives are to efficiently solve problems and perform computations related to integers and their properties.

I. SOLVED EXERCISE:

1). Write a java program to demonstrate a basic timing attack on RSA decryption.

Description: A basic timing attack on RSA decryption is a type of side-channel attack that exploits the variation in execution times of cryptographic algorithms based on the input data. In the context of RSA decryption, the attack focuses on observing and analyzing the time it takes for the decryption operation to complete for different ciphertexts. By exploiting these timing differences, an attacker may gain information about the private key and potentially compromise the security of the RSA implementation.

Program

```
import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.Signature;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;
import java.util.Arrays;

public class RSATimingAttackDemo {

    public static void main(String[] args) throws Exception {
        // Generate RSA key pair
        KeyPair keyPair = generateKeyPair();

        // Get the private key
        PrivateKey privateKey = keyPair.getPrivate();

        // Simulate a message that has been encrypted with the public key
        byte[] encryptedMessage = simulateEncryptedMessage(privateKey);

        // Measure the time it takes to decrypt the message
        long startTime = System.nanoTime();
        byte[] decryptedMessage = decryptMessage(encryptedMessage, privateKey);
        long endTime = System.nanoTime();

        // Demonstrate the timing attack by comparing the time taken for different decryption
attempts
        for (int i = 0; i < 5; i++) {
            long attackStartTime = System.nanoTime();
            decryptMessageWithRandomKey(encryptedMessage);
            long attackEndTime = System.nanoTime();

            long decryptionTime = endTime - startTime;
            long attackTime = attackEndTime - attackStartTime;

            System.out.println("Attempt " + (i + 1) + ": Decryption Time = " + decryptionTime + "
ns, Attack Time = " + attackTime + " ns");
        }

        // Compare the decrypted message to verify correctness
        System.out.println("Decrypted Message: " + Arrays.toString(decryptedMessage));
    }

    private static KeyPair generateKeyPair() throws Exception {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048); // Adjust the key size as needed
        return keyPairGenerator.generateKeyPair();
    }
    private static byte[] simulateEncryptedMessage(PrivateKey privateKey) throws Exception {
        // Simulate a message that has been encrypted with the public key
        byte[] message = "Hello, RSA timing attack!".getBytes();
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initSign(privateKey);
        signature.update(message);
        return signature.sign();
    }

    private static byte[] decryptMessage(byte[] encryptedMessage, PrivateKey privateKey) throws
Exception {
        // Simulate the decryption process
        Cipher rsaCipher = Cipher.getInstance("RSA");
        rsaCipher.init(Cipher.DECRYPT_MODE, privateKey);
        return rsaCipher.doFinal(encryptedMessage);
    }

    private static byte[] decryptMessageWithRandomKey(byte[] encryptedMessage) throws
Exception {
        // Simulate the decryption process with a random key
        KeyPair randomKeyPair = generateKeyPair();
        PublicKey randomPublicKey = randomKeyPair.getPublic();

        Cipher rsaCipher = Cipher.getInstance("RSA");
        rsaCipher.init(Cipher.DECRYPT_MODE, randomPublicKey);
        return rsaCipher.doFinal(encryptedMessage);
    }
}
```

II. LAB EXERCISES:

1) Implement the Diffie-Hellman key exchange algorithm in a programming language of your choice.
2) Implement the Chinese Remainder Theorem for efficient RSA decryption.
3) Implement a hybrid encryption scheme using ElGamal for key exchange and symmetric-key encryption for data transmission.
4) Implement a program to demonstrate the feasibility of lattice-based attacks on ElGamal.
5) Implement a padding oracle attack on ElGamal encryption, exploiting vulnerabilities in the padding scheme.

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

[OBSERVATION SPACE – LAB 7]

LAB NO: 8 **Date:**

Digital Signatures

Objectives:

In this lab, student will be able to:

- Understanding of the Key Generation for Digital Signatures
- Practice the processes of encrypting and decrypting messages

I. SOLVED EXERCISE:

1). Design and implement a java program that combines digital signatures with biometric authentication for enhanced security.

Description: The program employs digital signatures to ensure the integrity and authenticity of digital messages or transactions. A user's private key is used to sign messages, and the corresponding public key is used to verify the signatures. The program incorporates biometric authentication, leveraging unique physical or behavioral characteristics of individuals, such as fingerprints, facial features, or voice patterns. Biometric data is used to verify the identity of the user before allowing access or authorization.

Program

```
import org.bouncycastle.jce.provider.BouncyCastleProvider;

import java.security.*;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.X509EncodedKeySpec;

public class BiometricSignatureAuthentication {

    public static void main(String[] args) {
        Security.addProvider(new BouncyCastleProvider());

        try {
            // Generate or load a user's public and private key pair for digital signatures
            KeyPair keyPair = generateKeyPair();
            PublicKey publicKey = keyPair.getPublic();
            PrivateKey privateKey = keyPair.getPrivate();

            // Placeholder for biometric authentication
            boolean isBiometricAuthenticated = authenticateBiometric();

            if (isBiometricAuthenticated) {
                // Generate a digital signature using the private key
                byte[] digitalSignature = signData("Hello, World!", privateKey);

                // Verify the digital signature using the public key
                boolean isSignatureValid = verifySignature("Hello, World!", digitalSignature,
publicKey);

                if (isSignatureValid) {
                    System.out.println("Biometric authentication and digital signature verification
successful.");
                } else {
                    System.out.println("Digital signature verification failed.");
                }
            } else {
                System.out.println("Biometric authentication failed.");
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private static KeyPair generateKeyPair() throws NoSuchAlgorithmException {
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048);
        return keyPairGenerator.generateKeyPair();
    }
    private static boolean authenticateBiometric() {
        // Placeholder for biometric authentication logic
        // This could involve interaction with biometric hardware or a biometric database
        // Return true if authentication is successful, false otherwise
        return true;
    }
    private static byte[] signData(String data, PrivateKey privateKey) throws Exception {
        Signature signature = Signature.getInstance("SHA256withRSA");
        signature.initSign(privateKey);
        signature.update(data.getBytes());
        return signature.sign();
    }

    private static boolean verifySignature(String data, byte[] signature, PublicKey publicKey)
throws Exception {
        Signature verifier = Signature.getInstance("SHA256withRSA");
        verifier.initVerify(publicKey);
        verifier.update(data.getBytes());
        return verifier.verify(signature);
    }
}
```

II. LAB EXERCISES:

1) Design and implement a secure digital signature system using a widely-accepted cryptographic library.
2) Create a program that verifies the integrity of a message using a MAC.
3) Implement a side-channel attack on a digital signature system, such as a timing attack or power analysis.
4) **Scenario:** You are building an email system, and you want to implement a mechanism to authenticate the sender of an email.
   Task: Develop a program that signs outgoing emails with a digital signature and another program that verifies the signature of incoming emails.

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

[OBSERVATION SPACE – LAB 8]

LAB NO: 9 **Date:**

Encryption Techniques

Objectives:

In this lab, student will be able to:

- Understanding Encryption Schemes
- Implementing Basic symmetric encryption Operations
- Evaluate Performance and Efficiency

I. SOLVED EXERCISE:

1) Implement a program that securely encrypts and decrypts files using a symmetric encryption algorithm.

Description: A symmetric encryption algorithm is a cryptographic technique that uses the same key for both the encryption and decryption of data. In symmetric encryption, the sender and the recipient must both possess the secret key and keep it confidential to ensure the security of the communication.

Program

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.security.Key;
import java.security.MessageDigest;
import java.util.Arrays;

public class FileEncryptor {

    private static final String ALGORITHM = "AES";
    private static final String TRANSFORMATION = "AES/ECB/PKCS5Padding";

    public static void main(String[] args) {
        String key = "MySecretKey123"; // Replace with a secure key generation mechanism

        // Input and output file paths
        String inputFile = "path/to/input/file.txt";
        String encryptedFile = "path/to/output/encryptedFile.enc";
        String decryptedFile = "path/to/output/decryptedFile.txt";
        try {
            // Encrypt the file
            encryptFile(key, inputFile, encryptedFile);

            // Decrypt the file
            decryptFile(key, encryptedFile, decryptedFile);

            System.out.println("File encryption and decryption completed successfully.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static void encryptFile(String key, String inputFile, String outputFile) throws
Exception {
        byte[] rawKey = getRawKey(key);
        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        SecretKeySpec secretKeySpec = new SecretKeySpec(rawKey, ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec);

        byte[] fileBytes = Files.readAllBytes(Paths.get(inputFile));
        byte[] encryptedBytes = cipher.doFinal(fileBytes);

        Files.write(Paths.get(outputFile), encryptedBytes, StandardOpenOption.CREATE);
    }
private static void decryptFile(String key, String inputFile, String outputFile) throws Exception
{
        byte[] rawKey = getRawKey(key);
        Cipher cipher = Cipher.getInstance(TRANSFORMATION);
        SecretKeySpec secretKeySpec = new SecretKeySpec(rawKey, ALGORITHM);
        cipher.init(Cipher.DECRYPT_MODE, secretKeySpec);
        byte[] encryptedBytes = Files.readAllBytes(Paths.get(inputFile));
        byte[] decryptedBytes = cipher.doFinal(encryptedBytes);

        Files.write(Paths.get(outputFile), decryptedBytes, StandardOpenOption.CREATE);
    }

    private static byte[] getRawKey(String key) throws Exception {
        MessageDigest md = MessageDigest.getInstance("SHA-256");
        byte[] keyBytes = md.digest(key.getBytes("UTF-8"));
        return Arrays.copyOf(keyBytes, 16); // AES key size is 128 bits (16 bytes)
    }
}
```

II. LAB EXERCISES:

1) Write a program that verifies the digital signature of a given message using a public key.

2) **Scenario:** You are working on a file-sharing application, and you want to ensure that shared files are not tampered with during transit.ng transit.
   Task: Implement a program that signs files before they are uploaded. The recipient's program should verify the digital signature before allowing access to the downloaded file.

3) **Scenario:** You are managing a database containing sensitive information (e.g., personal details, financial records), and you want to protect it from unauthorized access.
   Task: Implement database encryption to secure the stored data. This can include encrypting specific fields, entire tables, or the entire database.

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

[OBSERVATION SPACE – LAB 9]

LAB NO: 10 **Date:**

Authentication Techniques

Objectives:

In this lab, student will be able to:

- Understand Authentication Concepts
- Understand the Internet Firewalls for Trusted Systems

Description: This course aims to provide students with a comprehensive understanding of security practices and system security, covering essential topics such as authentication applications, Kerberos, X.509 authentication services, internet firewalls, intrusion detection systems, and countermeasures against viruses and threats.

I. SOLVED EXERCISE:

1) Write a simple java program to develop virus scanner that scans files for known virus signatures. Use a signature-based approach for detection.

Description: A "signature-based approach for detection" refers to a method used in cybersecurity and intrusion detection systems to identify known patterns or signatures associated with malicious activities. This approach relies on pre-existing knowledge of specific patterns or characteristics of known threats, and it is particularly effective in recognizing well-defined, previously identified attacks.

Program

```
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class VirusScanner {

    // Known virus signatures (dummy signatures for demonstration)
    private static List<String> virusSignatures = new ArrayList<>();

    static {
        // Add some dummy virus signatures
        virusSignatures.add("malicious_pattern_1");
        virusSignatures.add("malicious_pattern_2");
        // Add more signatures as needed
    }

    public static void main(String[] args) {
        // Specify the directory to scan
        String directoryPath = "/path/to/scan";

        // Scan the directory for viruses
        scanDirectory(directoryPath);
    }
private static void scanDirectory(String directoryPath) {
        File directory = new File(directoryPath);
        if (!directory.isDirectory()) {
            System.out.println("Invalid directory path.");
            return;
        }
File[] files = directory.listFiles();
        if (files != null) {
            for (File file : files) {
                if (file.isFile()) {
                    if (isInfected(file)) {
                        System.out.println("Infected file detected: " + file.getAbsolutePath());
                        // Implement actions to take when a virus is detected (e.g., quarantine, delete)
                    } else {
                        System.out.println("File is clean: " + file.getAbsolutePath());
                    }
                } else if (file.isDirectory()) {
                    scanDirectory(file.getAbsolutePath()); // Recursive call for subdirectories
                }
            }
        }
    }

    private static boolean isInfected(File file) {
        try (FileInputStream fis = new FileInputStream(file)) {
            byte[] fileBytes = new byte[(int) file.length()];
            fis.read(fileBytes);

            // Check for virus signatures
            for (String signature : virusSignatures) {
                if (containsSignature(fileBytes, signature.getBytes())) {
                    return true;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return false;
    }

    private static boolean containsSignature(byte[] data, byte[] signature) {
        for (int i = 0; i <= data.length - signature.length; i++) {
            boolean match = true;
            for (int j = 0; j < signature.length; j++) {
                if (data[i + j] != signature[j]) {
                    match = false;
                    break;
                }
            }
            if (match) {
                return true;
            }
        }
        return false;
    }
}
```

II. LAB EXERCISES:

1). Write a program that simulates a network intrusion, and design an intrusion detection system to detect and respond to the simulated attack.

2). Develop a program that digitally signs and verifies messages using asymmetric cryptography for secure communication.

3). Write a program to develop multi-factor authentication system that combines something the user knows (password) with something the user has (one-time token).

4). Implement a program that detects DNS spoofing attacks.

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]

[OBSERVATION SPACE – LAB 10]