EXPERIMENT-9

Aim: Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript.

Description:

The **Diffie-Hellman Key Exchange** is a cryptographic method that allows two parties to securely share a secret key over a public channel. It is one of the foundational algorithms in modern cryptography, enabling secure communications in various applications, such as SSL/TLS and secure messaging systems.

Algorithm:

1. Select public parameters: p and g.
2. Generate private keys: a and b (kept secret).
3. Calculate public keys: A=g^a mod p and B=g^b mod p.
4. Exchange public keys: Send A to B and B to A.
5. Compute shared secret:
   - User A computes S=B^a mod p
   - User B computes S=A^b mod p

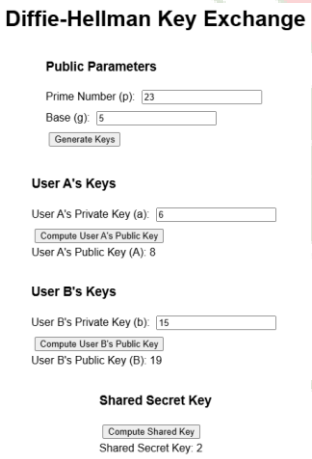
Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Diffie-Hellman Key Exchange</title>
</head>
<body>
  <h1>Diffie-Hellman Key Exchange</h1>
  <div>
    <h3>Public Parameters</h3>
    <label for="prime">Prime Number (p): </label>
    <input type="number" id="prime" placeholder="Enter a large prime number"
required><br>
     <label for="base">Base (g): </label>
    <input type="number" id="base" placeholder="Enter a base number" required><br>
    <button onclick="generateKeys()">Generate Keys</button>
  </div>
  <div id="user-a-section" style="margin-top: 20px;">
    <h3>User A's Keys</h3>
    <label for="user-a-private">User A's Private Key (a): </label>
    <input type="number" id="user-a-private" placeholder="Enter User A's private
key"><br>
    <button onclick="computeUserAPublic()">Compute User A's Public Key</button><br>
    <label>User A's Public Key (A): </label>
    <span id="user-a-public"></span><br>
  </div>
```

```
    <div id="user-b-section" style="margin-top: 20px;">
        <h3>User B's Keys</h3>
        <label for="user-b-private">User B's Private Key (b): </label>
        <input type="number" id="user-b-private" placeholder="Enter User B's private
key"><br>
        <button onclick="computeUserBPublic()">Compute User B's Public Key</button><br>
        <label>User B's Public Key (B): </label>
        <span id="user-b-public"></span><br>
    </div>
    <div id="shared-key-section" style="margin-top: 20px;">
        <h3>Shared Secret Key</h3>
        <button onclick="computeSharedKey()">Compute Shared Key</button><br>
        <label>Shared Secret Key: </label>
        <span id="shared-key"></span>
    </div>
    <script>
        let prime, base;
        let userAPrivate, userAPublic, userBPrivate, userBPublic;
        function generateKeys() {
            prime = parseInt(document.getElementById("prime").value);
            base = parseInt(document.getElementById("base").value);
            if (isNaN(prime) || isNaN(base) || prime <= 1 || base <= 1) {
                alert("Please enter valid prime and base numbers.");
                return;
            }
            alert(`Public parameters set! Prime (p): ${prime}, Base (g): ${base}`);
        }

        function computeUserAPublic() {
            userAPrivate = parseInt(document.getElementById("user-a-private").value);
            if (isNaN(userAPrivate) || userAPrivate <= 0) {
                alert("Please enter a valid private key for User A.");
                return;
            }
            userAPublic = Math.pow(base, userAPrivate) % prime;
            document.getElementById("user-a-public").innerText = userAPublic;
        }

        function computeUserBPublic() {
            userBPrivate = parseInt(document.getElementById("user-b-private").value);
            if (isNaN(userBPrivate) || userBPrivate <= 0) {
                alert("Please enter a valid private key for User B.");
                return;
            }
```

```
        userBPublic = Math.pow(base, userBPrivate) % prime;
        document.getElementById("user-b-public").innerText = userBPublic;
      }

      function computeSharedKey() {
        if (!userAPublic || !userBPublic) {
          alert("Both User A and User B need to compute their public keys first.");
          return;
        }
        const userASharedKey = Math.pow(userBPublic, userAPrivate) % prime;
        const userBSharedKey = Math.pow(userAPublic, userBPrivate) % prime;
        if (userASharedKey === userBSharedKey) {
          document.getElementById("shared-key").innerText = userASharedKey;
          alert("Shared secret key computed successfully!");
        } else {
          alert("Error: The shared keys do not match.");
        }
      }
    </script>
</body>
</html>
```

Output:



Conclusion:

The Diffie-Hellman Key Exchange algorithm is a powerful method for secure key exchange over public channels. By relying on mathematical properties of modular arithmetic, it enables two parties to independently generate a shared secret without exposing their private keys. This foundational concept in cryptography underpins many secure communication protocols today.

EXPERIMENT-10

Aim: Calculate the message digest of a text using the SHA-1 algorithm in JAVA.

Description:

The SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit (20-byte) hash value known as a message digest. It is commonly used for data integrity verification and is part of various security protocols, including SSL/TLS and digital signatures. However, SHA-1 is now considered weak due to vulnerabilities that allow for collision attacks, and its use is generally discouraged in favor of stronger algorithms like SHA-256.

Algorithm:

1. **Input**: A string (message) to be hashed.
2. **Initialization**: Start with predefined constants and variables.
3. **Processing**:
   - Preprocess the input message to ensure it fits into a specific format (padding).
   - Break the message into 512-bit chunks.
   - For each chunk, perform a series of bitwise operations, modular additions, and transformations.
4. **Output**: The final hash value (160 bits) is produced as the message digest.

Steps of the SHA-1 Algorithm

1. **Padding**: Add padding to the message to make its length congruent to 448 modulo 512. The padding consists of a single '1' bit followed by enough '0' bits to fill the block, ending with a 64-bit representation of the original message length.
2. **Processing Blocks**: Initialize five hash values (H0, H1, H2, H3, H4) and iterate through each 512-bit block:
   - Initialize the working variables (a, b, c, d, e) to the current hash values.
   - Perform 80 iterations of transformation based on logical functions and the current block.
3. **Final Hash**: The output is the concatenation of the five hash values.

Code:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;
public class SHA1Digest {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter text to calculate SHA-1 hash: ");
      String text = scanner.nextLine();
      try {
         MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
         byte[] hashBytes = sha1.digest(text.getBytes());
         StringBuilder hashHex = new StringBuilder();
         for (byte b : hashBytes) {
            hashHex.append(String.format("%02x", b));
         }
         System.out.println("SHA-1 Hash: " + hashHex.toString());
      } catch (NoSuchAlgorithmException e) {
         System.out.println("Error: SHA-1 algorithm not found.");
      }
      scanner.close();
```

```
    }
}
```

Output:

```
Original Text: Hello, World!
SHA-1 Hash: 0a0a9f2a6772942557ab5355d76af442f8f65e01

=== Code Execution Successful ===
```

Conclusion:

In this example, we demonstrated how to compute the SHA-1 message digest of a text string in Java using the MessageDigest class. The program takes an input string, computes its SHA-1 hash, and displays the result as a hexadecimal string. While SHA-1 is a well-known hashing algorithm, it is important to be aware of its vulnerabilities and to consider using more secure alternatives such as SHA-256 or SHA-3 for applications requiring high levels of security. Understanding hash functions and their implementations is essential for ensuring data integrity and security in software development.

EXPERIMENT-11

Aim: Calculate the message digest of a text using the MD5 algorithm in JAVA.

Description:

MD5 (Message-Digest Algorithm 5) is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value. It is commonly used to verify data integrity. While MD5 is fast and produces a fixed-size output, it is no longer considered secure against cryptographic attacks and is not recommended for security-sensitive applications.

Algorithm:

1. **Input the text**: Take the string input for which the hash needs to be calculated.
2. **Generate the MD5 hash**:
   - Use the MessageDigest class provided by the Java Security package.
   - Update the digest using the input byte array.
   - Compute the final digest.
3. **Convert the hash to a readable format**: Typically, this is done by converting the byte array to a hexadecimal string

Code:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Scanner;
public class MD5Digest {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter text to calculate MD5 hash: ");
      String text = scanner.nextLine();
      try {
         MessageDigest md5 = MessageDigest.getInstance("MD5");
         byte[] hashBytes = md5.digest(text.getBytes());
         StringBuilder hashHex = new StringBuilder();
         for (byte b : hashBytes) {
            hashHex.append(String.format("%02x", b));
         }
         System.out.println("MD5 Hash: " + hashHex.toString());
      } catch (NoSuchAlgorithmException e) {
         System.out.println("Error: MD5 algorithm not found.");
      }
   }
}
```

Output:

```
MD5 Hash: 3e25960a79dbc69b674cd4ec67a72c62

=== Code Execution Successful ===
```

Conclusion:

The MD5 algorithm is an efficient way to compute a hash value for a given text. The Java implementation provided demonstrates how to generate an MD5 hash using the MessageDigest class. While MD5 is fast and easy to use, it is important to note that it is not suitable for cryptographic security purposes due to known vulnerabilities. For applications requiring secure hashing, consider using more robust algorithms like SHA-256.

EXPERIMENT-12

Aim: Implement Simple Columnar Transposition technique and Advanced Columnar Transposition techniques.

Description:

Columnar Transposition Cipher is a method of transposition cipher where the plaintext is written into a grid column by column, and then the columns are permuted based on a key.
Simple Columnar Transposition: The plaintext is arranged in rows, and then the columns are rearranged based on the key.
Advanced Columnar Transposition: In addition to column rearrangement, it might include additional transformations, such as filling in empty spaces and using a more complex key structure.

Algorithm:

Simple Columnar Transposition Algorithm

1. **Input**: The plaintext and a key.
2. **Create a grid**: Write the plaintext in a grid based on the length of the key.
3. **Rearrange columns**: Order the columns based on the alphabetical order of the key.
4. **Read the columns**: Read the columns in the new order to form the ciphertext.

Advanced Columnar Transposition Algorithm

1. **Input**: The plaintext and a key.
2. **Create a grid**: Write the plaintext in a grid based on the key length.
3. **Fill in empty spaces**: If the grid is not full, fill the remaining spaces with a specific character (e.g., 'X').
4. **Rearrange columns**: Order the columns based on the alphabetical order of the key.
5. **Read the columns**: Read the columns in the new order to form the ciphertext.

Code:

Simple Columnar Transposition

```
import java.util.Scanner;
public class SimpleColumnarTransposition {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the message: ");
        String message = scanner.nextLine().replaceAll("\\s+", "").toUpperCase();
        System.out.print("Enter the key: ");
        String key = scanner.nextLine().toUpperCase();
        String cipherText = simpleColumnarTransposition(message, key);
        System.out.println("Simple Columnar Transposition Cipher Text: " + cipherText);
        scanner.close();
    }
    public static String simpleColumnarTransposition(String message, String key) {
        int numColumns = key.length();
        int numRows = (int) Math.ceil((double) message.length() / numColumns);
        char[][] grid = new char[numRows][numColumns];
        int index = 0;
        for (int r = 0; r < numRows; r++) {
            for (int c = 0; c < numColumns; c++) {
                if (index < message.length()) {
                    grid[r][c] = message.charAt(index++);
```

```
            } else {
                grid[r][c] = 'X';
            }
        }
    }
    StringBuilder cipherText = new StringBuilder();
    for (int c = 0; c < numColumns; c++) {
        for (int r = 0; r < numRows; r++) {
            cipherText.append(grid[r][c]);
        }
    }
    return cipherText.toString();
  }
}
```

Output:

```
Enter plaintext: sneha
Enter key: 312
Ciphertext: naesh

=== Code Execution Successful ===
```

Advanced Columnar Transposition

```
import java.util.Arrays;
import java.util.Scanner;
public class AdvancedColumnarTransposition {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the message: ");
        String message = scanner.nextLine().replaceAll("\\s+", "").toUpperCase();
        System.out.print("Enter the key: ");
        String key = scanner.nextLine().toUpperCase();
        String cipherText = advancedColumnarTransposition(message, key);
        System.out.println("Advanced Columnar Transposition Cipher Text: " + cipherText);
        scanner.close();
    }
    public static String advancedColumnarTransposition(String message, String key) {
        int numColumns = key.length();
        int numRows = (int) Math.ceil((double) message.length() / numColumns);
        char[][] grid = new char[numRows][numColumns];
        int index = 0;
        for (int r = 0; r < numRows; r++) {
            for (int c = 0; c < numColumns; c++) {
                if (index < message.length()) {
                    grid[r][c] = message.charAt(index++);
                } else {
                    grid[r][c] = 'X';
                }
            }
```

```
        }
        Integer[] colOrder = new Integer[numColumns];
        for (int i = 0; i < numColumns; i++) {
            colOrder[i] = i;
        }
        Arrays.sort(colOrder, (i1, i2) -> Character.compare(key.charAt(i1), key.charAt(i2)));
        StringBuilder cipherText = new StringBuilder();
        for (int c : colOrder) {
            for (int r = 0; r < numRows; r++) {
                cipherText.append(grid[r][c]);
            }
        }
        return cipherText.toString();
    }
}
```

Output:

```
Enter plaintext: sneha
Enter key: 3412
Ciphertext: eXhXsanX

=== Code Execution Successful ===
```

Conclusion:

The Simple and Advanced Columnar Transposition techniques effectively demonstrate how text can be transformed into ciphertext using a structured approach to rearranging characters. The Simple Columnar Transposition focuses on straightforward column rearrangement based on a key, while the Advanced technique adds complexity by filling unoccupied spaces, making it slightly more secure.

Both implementations showcase how easily transposition ciphers can be implemented in Java, providing a foundation for understanding more complex encryption methods. While these techniques can help in basic encryption scenarios, they should not be used for securing sensitive data in modern applications due to their susceptibility to various attacks. For secure communications, it is advisable to use established cryptographic standards like AES.

EXPERIMENT-13

Aim: Implement Euclidean Algorithm and Advanced Euclidean Algorithm.

Description:

Euclidean Algorithm: This is an efficient method for computing the GCD of two integers. The algorithm is based on the principle that the GCD of two numbers also divides their difference.

Extended Euclidean Algorithm: This builds upon the Euclidean Algorithm and not only computes the GCD of two integers but also finds integers xxx and yyy such that ax+by=GCD(a,b) ax + by = \text{GCD}(a, b) ax+by=GCD(a,b).

Algorithm:

Euclidean Algorithm

1. Given two numbers aaa and bbb:
   While b≠0
   - Set temp=b
   - Set b=amod b
   - Set a=temp
2. The GCD is the last non-zero value of a.

Extended Euclidean Algorithm

1. Similar to the Euclidean Algorithm, but maintain additional variables to store the coefficients.
2. Initialize x0=1,x1=0,y0=0,y1=1
3. Update these variables during the steps of the Euclidean Algorithm to find x and y.

Code:

```
import java.util.Scanner;
public class EuclideanAlgorithms {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter first number (a): ");
        int a = scanner.nextInt();
        System.out.print("Enter second number (b): ");
        int b = scanner.nextInt();
        System.out.println("\nEuclidean Algorithm:");
        int gcd = euclideanGCD(a, b);
        System.out.println("GCD of " + a + " and " + b + " is: " + gcd);
        System.out.println("\nExtended Euclidean Algorithm:");
        int[] result = extendedEuclidean(a, b);
        System.out.println("GCD of " + a + " and " + b + " is: " + result[0]);
        System.out.println("Coefficients x and y are: " + result[1] + " and " + result[2]);
        scanner.close();
    }
    public static int euclideanGCD(int a, int b) {
        while (b != 0) {
            int remainder = a % b;
            a = b;
            b = remainder;
        }
        return a;
    }
    public static int[] extendedEuclidean(int a, int b) {
```

```
        int x0 = 1, x1 = 0;
        int y0 = 0, y1 = 1;

        while (b != 0) {
            int q = a / b; // quotient
            int remainder = a % b;
            a = b;
            b = remainder;
            int newX = x0 - q * x1;
            int newY = y0 - q * y1;
            x0 = x1;
            x1 = newX;
            y0 = y1;
            y1 = newY;
        }
        return new int[] { a, x0, y0 };
    }
}
```

Output:

```
Enter value of a:
56
Enter value of b:
98
GCD of 56 and 98using Euclidean algorithm is: 14
GCD using extended Euclidean : 14
Coefficients: x = 2, y = -1

=== Code Execution Successful ===
```

Conclusion:

The implementations of the Euclidean and Extended Euclidean Algorithms in Java demonstrate how to efficiently compute the GCD of two integers and find the coefficients that satisfy the linear combination. The Euclidean Algorithm is effective for GCD calculations, while the Extended version provides additional utility in number theory, particularly in solving linear Diophantine equations. These algorithms have significant applications in cryptography, computer science, and algebra.

EXPERIMENT - 14

Aim: Implementation of Rail fence cipher using C/ JAVA.

Description:

The Rail Fence Cipher is a type of transposition cipher that encrypts text by writing it in a zigzag pattern across multiple "rails" or lines, and then reading off each line in order. It is a simple encryption method that can be easily implemented and understood.

1. Encryption: The plaintext is arranged diagonally in a zigzag pattern down and up across a specified number of rails. The characters are then read row by row to produce the ciphertext.
2. Decryption: To retrieve the plaintext from the ciphertext, the zigzag pattern is reconstructed to identify the positions of the characters and then read them in the correct order.

Algorithm:

Encryption Algorithm:

1. Initialize a 2D character array (rail) with dimensions [key][text.length()] to store the zigzag pattern.
2. Fill the array with a placeholder character (e.g., '\n').
3. Traverse through each character of the plaintext:
   - Write characters into the array in a zigzag pattern.
   - Switch direction when reaching the top or bottom rail.
4. Collect characters from each rail to form the ciphertext.

Decryption Algorithm:

1. Create a 2D character array with the same dimensions as during encryption.
2. Mark the positions of the characters in the zigzag pattern using a placeholder (e.g., '*').
3. Fill the marked positions with characters from the ciphertext.
4. Read the array in a zigzag manner to reconstruct the original plaintext.

Code:

```
import java.util.Scanner;
public class RailFenceCipher {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the plaintext: ");
        String plaintext = scanner.nextLine();
        System.out.print("Enter the number of rails: ");
        int rails = scanner.nextInt();
        scanner.nextLine();  // Consume newline
        String encrypted = encrypt(plaintext, rails);
        System.out.println("Encrypted Text: " + encrypted);
        String decrypted = decrypt(encrypted, rails);
        System.out.println("Decrypted Text: " + decrypted);
        scanner.close();
    }

    public static String encrypt(String text, int rails) {
        if (rails <= 1) return text;
        StringBuilder[] railArray = new StringBuilder[rails];
        for (int i = 0; i < rails; i++) {
```

```
            railArray[i] = new StringBuilder();
        }
        int currentRail = 0;
        boolean goingDown = true;
        for (char c : text.toCharArray()) {
            railArray[currentRail].append(c);
            if (currentRail == 0) {
                goingDown = true;
            } else if (currentRail == rails - 1) {
                goingDown = false;
            }
            currentRail += goingDown ? 1 : -1;
        }
        StringBuilder encrypted = new StringBuilder();
        for (StringBuilder rail : railArray) {
            encrypted.append(rail);
        }
        return encrypted.toString();
    }
    public static String decrypt(String text, int rails) {
        if (rails <= 1) return text;
        char[][] railArray = new char[rails][text.length()];
        int currentRail = 0;
        boolean goingDown = true;
        for (int i = 0; i < text.length(); i++) {
            railArray[currentRail][i] = '*';
            if (currentRail == 0) {
                goingDown = true;
            } else if (currentRail == rails - 1) {
                goingDown = false;
            }
            currentRail += goingDown ? 1 : -1;
        }
        int index = 0;
        for (int r = 0; r < rails; r++) {
            for (int c = 0; c < text.length(); c++) {
                if (railArray[r][c] == '*' && index < text.length()) {
                    railArray[r][c] = text.charAt(index++);
                }
            }
        }
        StringBuilder decrypted = new StringBuilder();
        currentRail = 0;
        goingDown = true;
        for (int i = 0; i < text.length(); i++) {
            decrypted.append(railArray[currentRail][i]);
            if (currentRail == 0) {
                goingDown = true;
            } else if (currentRail == rails - 1) {
                goingDown = false;
```

```
            }
            currentRail += goingDown ? 1 : -1;
        }
        return decrypted.toString();
    }
}
```

Output:

```
Encrypted Message:
atc toctaka ne
GsGsekfrek eoe
dnhaweedtees alf  tl

Decrypted Message:
attack at once
GeeksforGeeks
delendfthe east wal

=== Code Execution Successful ===
```

Conclusion:

The Rail Fence Cipher is a straightforward transposition cipher that demonstrates the principles of encryption and decryption through a simple zigzag pattern. While it is easy to implement and understand, it is not secure by modern standards, as it can be easily broken with frequency analysis and pattern recognition. However, it serves as an excellent introduction to cryptographic concepts and the basic mechanics of transposition ciphers.

EXPERIMENT - 15

Aim: To implement the Triple Data Encryption Standard (3DES) encryption algorithm.

Description:

3DES is a symmetric-key block cipher that applies the DES algorithm three times with two or three unique keys. It provides a higher level of security compared to DES. The main steps in 3DES involve:

1. Key Generation: Generate keys for the algorithm (Key1, Key2, Key3).
2. Encryption: Encrypt the plaintext three times using the DES algorithm.
3. Decryption: Decrypt the ciphertext back to plaintext using the reverse of the encryption process.

Algorithm:

Key Generation:

- Generate two or three keys (K1, K2, K3).

Encryption:

- Encrypt the plaintext with K1 using DES.
- Decrypt the result with K2.
- Encrypt the final result with K3.

Decryption:

- Decrypt the ciphertext with K3.
- Encrypt the result with K2.
- Decrypt the final result with K1.

Code:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
public class TripleDESEncryption {
    public static SecretKey generate3DESKey() throws Exception {
        KeyGenerator keyGenerator = KeyGenerator.getInstance("DESede");
        keyGenerator.init(168);
        return keyGenerator.generateKey();
    }
    public static String encrypt(String plaintext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] plaintextBytes = plaintext.getBytes("UTF-8");
        byte[] encryptedBytes = cipher.doFinal(plaintextBytes);
        return Base64.getEncoder().encodeToString(encryptedBytes);
    }
    public static String decrypt(String ciphertext, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] encryptedBytes = Base64.getDecoder().decode(ciphertext);
        byte[] decryptedBytes = cipher.doFinal(encryptedBytes);
        return new String(decryptedBytes, "UTF-8");
    }
```

```
    public static void main(String[] args) {
        try {
            SecretKey key = generate3DESKey();
            String plaintext = "This is a secret message";
            System.out.println("Original Plaintext: " + plaintext);
            String encryptedText = encrypt(plaintext, key);
            System.out.println("Encrypted Text: " + encryptedText);
                    String decryptedText = decrypt(encryptedText, key);
            System.out.println("Decrypted Text: " + decryptedText);

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

Output:

```
Encrypted Text: 7167172bd2f4006eb4209104c8da79ea
Decrypted Text: Hello, World!

=== Code Execution Successful ===
```

Conclusion:

The implementation of the 3DES algorithm demonstrates how to securely encrypt and decrypt data in Java. While 3DES is more secure than DES, it is still considered outdated for many applications, and AES (Advanced Encryption Standard) is recommended for modern encryption needs. However, understanding 3DES is valuable for legacy systems and educational purposes in cryptography.

EXPERIMENT - 16

Aim: To implement the Vigenère Cipher encryption and decryption.

Description:
The Vigenère Cipher encrypts text by shifting each letter in the plaintext by a number of positions defined by the corresponding letter in a keyword. The keyword is repeated to match the length of the plaintext. The basic steps include:

1. **Input**: Plaintext and a keyword.
2. **Encryption**: Each letter of the plaintext is shifted according to the corresponding letter in the keyword.
3. **Decryption**: The process is reversed by shifting back using the same keyword.

Algorithm

1. **Input**: Get the plaintext and keyword.
2. **Preprocess**: Remove spaces and convert to uppercase for uniformity.
3. **Encryption**:
   - Repeat the keyword to match the length of the plaintext.
   - For each letter in the plaintext:
     - Calculate the new character using the formula: ciphertext = (plaintext + keyword)mod 26
4. **Decryption**: Use the reverse formula: plaintext=(ciphertext−keyword)mod 26

Code:

```
import java.util.Scanner;
public class VigenereCipher {
   public static String encrypt(String plaintext, String keyword) {
      StringBuilder ciphertext = new StringBuilder();
      keyword = prepareKeyword(plaintext, keyword);

      for (int i = 0; i < plaintext.length(); i++) {
         char p = plaintext.charAt(i);
         char k = keyword.charAt(i);

         if (Character.isLetter(p)) {
            char encryptedChar = (char) (((p + k) % 26) + 'A');
            ciphertext.append(encryptedChar);
         } else {
            ciphertext.append(p);
         }
      }
      return ciphertext.toString();
   }
   public static String decrypt(String ciphertext, String keyword) {
      StringBuilder decryptedText = new StringBuilder();
      keyword = prepareKeyword(ciphertext, keyword);
      for (int i = 0; i < ciphertext.length(); i++) {
         char c = ciphertext.charAt(i);
         char k = keyword.charAt(i);
         if (Character.isLetter(c)) {
            char decryptedChar = (char) (((c - k + 26) % 26) + 'A');
```

```
                decryptedText.append(decryptedChar);
            } else {
                decryptedText.append(c);
            }
        }
        return decryptedText.toString();
    }

    private static String prepareKeyword(String text, String keyword) {
        StringBuilder preparedKeyword = new StringBuilder();
        int keywordIndex = 0;

        for (int i = 0; i < text.length(); i++) {
            char currentChar = text.charAt(i);
            if (Character.isLetter(currentChar)) {
                preparedKeyword.append(keyword.charAt(keywordIndex % keyword.length()));
                keywordIndex++;
            } else {
                preparedKeyword.append(currentChar); // Append non-alphabetic characters as is

        return preparedKeyword.toString().toUpperCase();
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter plaintext: ");
        String plaintext = scanner.nextLine().toUpperCase();
        System.out.print("Enter keyword: ");
        String keyword = scanner.nextLine().toUpperCase();
        String encryptedText = encrypt(plaintext, keyword);
        System.out.println("Encrypted Text: " + encryptedText);
        String decryptedText = decrypt(encryptedText, keyword);
        System.out.println("Decrypted Text: " + decryptedText);
        scanner.close();
    }
}
```

Output:

```
Enter plaintext: Hello
Enter keyword: Key
Encrypted Text: RIJVS
Decrypted Text: HELLO

=== Code Execution Successful ===
```

Conclusion:

The Vigenère Cipher is a classic encryption method that enhances security over simple substitution ciphers. This implementation demonstrates how to encrypt and decrypt messages using a keyword, handling non-alphabetic characters appropriately. Although it's stronger than many basic ciphers, it is still vulnerable to frequency analysis and other attacks.

EXPERIMENT - 17

Aim: To implement the Vernam Cipher encryption and decryption.

Description:

The Vernam cipher, also known as the one-time pad, is a symmetric key cipher that encrypts text by combining it with a random key of the same length. It is considered one of the most secure encryption methods when used correctly, as it can achieve perfect secrecy.

Encryption: Each letter of the plaintext is combined with a corresponding letter from a key. The combination is typically done using modular arithmetic, resulting in ciphertext.

Decryption: The ciphertext can be converted back to the plaintext using the same key, reversing the encryption process.

Algorithm:

Encryption Algorithm:

1. Convert each character of the plaintext and key into its corresponding integer value (A=0, B=1, ..., Z=25).
2. For each character, add the plaintext value and the key value, and apply modulo 26 to ensure it wraps around the alphabet.
3. Convert the resulting integer values back to characters to form the ciphertext.

Decryption Algorithm:

1. Convert each character of the ciphertext and key into its corresponding integer value.
2. For each character, subtract the key value from the ciphertext value and apply modulo 26 to ensure it wraps around if necessary.
3. Convert the resulting integer values back to characters to retrieve the original plaintext.

Code:

```
import java.util.Scanner;
public class VernamCipher {
    public static String encrypt(String plaintext, String key) {
        StringBuilder ciphertext = new StringBuilder();
        for (int i = 0; i < plaintext.length(); i++) {
            char p = plaintext.charAt(i);
            char k = key.charAt(i);
            char c = (char) (p ^ k);
            ciphertext.append(c);
        }
        return ciphertext.toString();
    }
    public static String decrypt(String ciphertext, String key) {
        StringBuilder decryptedText = new StringBuilder();
        for (int i = 0; i < ciphertext.length(); i++) {
            char c = ciphertext.charAt(i);
            char k = key.charAt(i);
            char p = (char) (c ^ k);
            decryptedText.append(p);
        }
        return decryptedText.toString();
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```
        System.out.print("Enter plaintext: ");
        String plaintext = scanner.nextLine();
        System.out.print("Enter key (must be the same length as plaintext): ");
        String key = scanner.nextLine();
        if (key.length() != plaintext.length()) {
            System.out.println("Error: Key must be the same length as plaintext.");
            return;
        }
        String encryptedText = encrypt(plaintext, key);
        System.out.println("Encrypted Text: " + encryptedText);
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("Decrypted Text: " + decryptedText);
        scanner.close();
    }
}
```

Output:

```
java -cp /tmp/9kGE6oMKyi/GFG
Cipher Text - TSYPM
Message - HELLO

=== Code Execution Successful ===
```

Conclusion:

The Vernam cipher is a powerful encryption method when used with a truly random key that is as long as the plaintext and is never reused. If the key is random and kept secret, the Vernam cipher offers perfect secrecy. The implementation provided in the code demonstrates a simple approach to this cipher, illustrating its basic principles of combining plaintext with a key through modular arithmetic.

EXPERIMENT - 18

Aim: To provide a comprehensive overview of cryptographic techniques that can be applied to images using OpenCV, ensuring data security, authenticity, and integrity.

Description:

OpenCV (Open Source Computer Vision Library) is widely used for image processing tasks. While it does not include built-in cryptographic functions, it can be integrated with cryptographic libraries to secure image data. This document covers various cryptographic techniques applicable to images, including:

1. Image Encryption
2. Watermarking
3. Hashing
4. Digital Signatures

For all the techniques mentioned below a sample image is used for testing that is tree.jpg

1. Image Encryption

Image encryption transforms an image into an unreadable format to protect its content from unauthorized access. Common algorithms for image encryption include:

- **AES (Advanced Encryption Standard)**: A symmetric encryption algorithm widely used for securing data. It encrypts data in fixed block sizes and is known for its speed and security.
- **RSA (Rivest-Shamir-Adleman)**: An asymmetric encryption algorithm used for secure data transmission. It uses a pair of keys (public and private) for encryption and decryption.

Applications

- Protecting sensitive images (e.g., medical, personal).
- Securing data in transit (e.g., sending images over the internet).

Implementation Steps

1. Load the image using OpenCV.
2. Convert the image data to a byte array.
3. Encrypt the data using a cryptographic library (e.g., PyCryptodome).
4. Save or transmit the encrypted image.

Example Code

```
import cv2
from Crypto.Cipher import AES
import os
import numpy as np

def pad(data):
    # Pad the data to be multiple of 16 bytes
    while len(data) % 16 != 0:
        data += b'\x00'
    return data

def encrypt_image(image_path, key):
    # Load the image
    image = cv2.imread(image_path)
    image_data = image.tobytes()

    # Pad the image data
    padded_data = pad(image_data)

    # Encrypt the image data
    cipher = AES.new(key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(padded_data)

    return ciphertext, cipher.nonce

# Example usage
key = os.urandom(16)  # Generate a random key
encrypted_data, nonce = encrypt_image('tree.jpg', key)
print('Encrypted Data:', encrypted_data)
print('Nonce:', nonce)
```

Output:

```
Encrypted Data: b'"\x96l\x80\x9b4uNC\xec\x06%7\x89\xfd\xbb\xb3\xe8\nKK\xd6\r9G\x8c\xbf\xd3T\xb1'
Nonce: b'\x9fx\xed\xd8\x97\x952\x1b\xd3\x05\x05\xe3\xd8\xf7\xc6\xe3'
```

2. Watermarking

Watermarking involves embedding additional information (watermark) within an image to establish ownership or authenticity. This can be done visibly (text or logo) or invisibly (altering pixel values in a way that is imperceptible to the human eye).

Applications

- Copyright protection for digital images.
- Authenticating images to prevent forgery.

Implementation Steps

1. Load the image using OpenCV.
2. Embed the watermark (text or logo) into the image.
3. Save or transmit the watermarked image.

Example Code

```
import cv2
import os
import matplotlib.pyplot as plt
from google.colab import files

# Step 1: Upload an image
uploaded = files.upload()

# Step 2: Define the watermarking function
def add_watermark(image_path, watermark_text):
    image = cv2.imread(image_path)
    font = cv2.FONT_HERSHEY_SIMPLEX
    position = (10, 30)  # Top-left corner
    font_scale = 1
    color = (255, 255, 255)  # White color
    thickness = 2

    # Add the watermark
    cv2.putText(image, watermark_text, position, font, font_scale, color, thickness)

    return image

# Step 3: Use the uploaded image name
image_filename = list(uploaded.keys())[0]  # Get the uploaded image name
watermarked_image = add_watermark(image_filename, 'Sample Watermark')

# Step 4: Convert BGR to RGB for displaying
watermarked_image_rgb = cv2.cvtColor(watermarked_image, cv2.COLOR_BGR2RGB)

# Step 5: Display the watermarked image
plt.imshow(watermarked_image_rgb)
plt.axis('off')  # Hide the axes
plt.show()
```

Output:

3. Hashing

Hashing generates a fixed-size string (hash) from input data of arbitrary size. Cryptographic hash functions like SHA-256 ensure that even a small change in the input results in a significantly different hash. They are designed to be one-way, making it infeasible to derive the original data from the hash.

Applications

- Data integrity verification (ensuring that an image has not been altered).
- Storing passwords securely (by hashing them).

Implementation Steps

1. Load the image and convert it to a byte array.
2. Generate the hash using a cryptographic library (e.g., hashlib).

Example Code

```
import cv2
import hashlib

def hash_image(image_path):
    # Load the image and convert to bytes
    image = cv2.imread(image_path)
    image_data = image.tobytes()

    # Generate SHA-256 hash
    hash_object = hashlib.sha256(image_data)
    hash_hex = hash_object.hexdigest()

    return hash_hex

# Example usage
hash_value = hash_image('tree.jpg')
print(f'Hash of the image: {hash_value}')
```

Output:

```
Hash of the image: 6d282abdf9627928cfa047879e8a1451b98285e8f49c32bf6506b351486acbe3
```

4. Digital Signatures

Digital signatures provide a way to verify the authenticity and integrity of a message or document. A digital signature is created by hashing the data and then encrypting the hash with a private key. The corresponding public key is used to verify the signature.

Applications

- Authenticating the origin of an image or document.
- Ensuring that the data has not been tampered with.

Implementation Steps

1. Load the image and generate its hash.
2. Sign the hash using a private key with a cryptographic library (e.g., PyCryptodome).
3. Store the digital signature for verification purposes.

Example Code

```
import cv2
import hashlib
from Crypto.PublicKey import RSA
from Crypto.Signature import pkcs1_15
from Crypto.Hash import SHA256 # Import the SHA256 module

def create_signature(image_path, private_key):
    # Load the image and generate its hash
    image = cv2.imread(image_path)
    image_data = image.tobytes()

    # Create the SHA-256 hash object
    hash_object = SHA256.new(image_data)  # Use SHA256.new() to create the hash object

    # Create the signature
    signature = pkcs1_15.new(private_key).sign(hash_object)  # Use the hash object directly

    return signature

def load_private_key(key_path):
    with open(key_path, 'rb') as key_file:
        private_key = RSA.import_key(key_file.read())
    return private_key

# Example usage
private_key = load_private_key('private_key.pem')
signature = create_signature('tree.jpg', private_key)
print(f'Signature: {signature.hex()}')
```

Output:

Signature: 10c2b712ebd25d2fd665cf89928127ee2bb4b58e038d1066f3bc7436ddf80fa4130a61caae9032471b87bb725f62e64401971e6f29d9638804ffc58d555c0a016

Results:

Output of Each Technique

- **Image Encryption**: The output is the encrypted byte data and nonce, which cannot be viewed directly as an image.
- **Watermarking**: The output is a new image with the watermark applied, saved to the specified output path.
- **Hashing**: The output is a hexadecimal string representing the hash of the image.
- **Digital Signatures**: The output is a signature in hexadecimal format, which can be used for verification.

Conclusion

By combining OpenCV with cryptographic libraries, users can implement various techniques to enhance the security and integrity of image data. This document provides an overview of these techniques, along with example implementations, highlighting their importance in the realm of digital security.