EXPERIMENT 4:

AIM: To implement to perform Cryptography using Transposition Technique

DESCRIPTION:

In this program, we implement the Simple and Advanced Columnar Transposition techniques for encrypting a message. Columnar Transposition is a classical encryption technique where the plaintext is written row by row in a matrix of fixed dimensions (rows and columns) and then read out column by column according to a key. The key is a permutation of column indices, which determines the order in which the columns are read during encryption.

The **Simple Columnar Transposition** technique involves filling the matrix with the message row by row and then reading the message column by column according to the key to produce the encrypted message.

The **Advanced Columnar Transposition** technique applies multiple rounds of the Simple Columnar Transposition, further scrambling the message by reapplying the transposition process multiple times.

ALGORITHM:

Simple Columnar Transposition

1. **Input:** A message, a key (list of column indices), number of rows, and number of columns.
2. **Step 1:** Calculate the dimensions of the matrix based on the message length and number of columns.
3. **Step 2:** Pad the message with spaces to fit the matrix dimensions if necessary.
4. **Step 3:** Fill the matrix row by row with the characters of the padded message.
5. **Step 4:** Initialize an empty list for the encrypted message.
6. **Step 5:** For each column index in the key:
    - Read the characters down the corresponding column.
    - Append the characters to the encrypted message list.
7. **Step 6:** Return the encrypted message as a string.

Advanced Columnar Transposition

1. **Input:** A message, a key, number of rows, number of columns, and the number of rounds.

2. **Step 1:** Initialize the transposed message as the original message.
3. **Step 2:** For each round, perform the Simple Columnar Transposition on the current transposed message.
4. **Step 3:** After completing all rounds, return the final transposed (encrypted) message.

PROGRAM:

```
import random
import math

def simple_columnar_transposition(message, key, num_rows, num_cols):
	# Pad the message with spaces if needed
	padded_message = message.ljust(num_rows * num_cols)  # Pad the message to fit the
matrix
	matrix = []
	x = 0

	# Fill the matrix row by row
	for i in range(num_rows):
	row = []
	for j in range(num_cols):
	row.append(padded_message[x])
	x += 1
	matrix.append(row)

	encrypted = []
	for i in key:
	for r in range(num_rows):
	if matrix[r][i] != " ":
		encrypted.append(matrix[r][i])  # No need for index correction since key starts
from 0
	# print(matrix, encrypted)
	return ''.join(encrypted)

def advanced_columnar_transposition(message, key, num_rows, num_cols, rounds):
	transposed_message = message
	for _ in range(rounds):
	transposed_message = simple_columnar_transposition(transposed_message, key,
num_rows, num_cols)
```

```
        return transposed_message

def generate_random_key(num_cols):
        key = list(range(num_cols))
        random.shuffle(key)
        return key

def main():
        # Input message
        message = input("Enter the message: ").replace(" ", "")

        # Generate random rows and columns
        num_cols = random.randint(2, 10)  # Random number of columns between 2 and 10
        num_rows = math.ceil(len(message) / num_cols)

        # Generate a random key
        key = generate_random_key(num_cols)

        # Simple Columnar Transposition
        encrypted_message = simple_columnar_transposition(message, key, num_rows,
num_cols)
        print(f"Simple Columnar Transposition: {encrypted_message}")

        # Advanced Columnar Transposition
        rounds = random.randint(1, 5)  # Random number of rounds between 1 and 5
        advanced_encrypted_message = advanced_columnar_transposition(message, key,
num_rows, num_cols, rounds)
        print(f"Advanced Columnar Transposition (Rounds: {rounds}):
{advanced_encrypted_message}")

if __name__ == "__main__":
        main()
```

OUTPUT:

```
Enter the message: helloletsseee
Simple Columnar Transposition: lelseseoehtle
Advanced Columnar Transposition (Rounds: 5): seleohetllese

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

CONCLUSION:

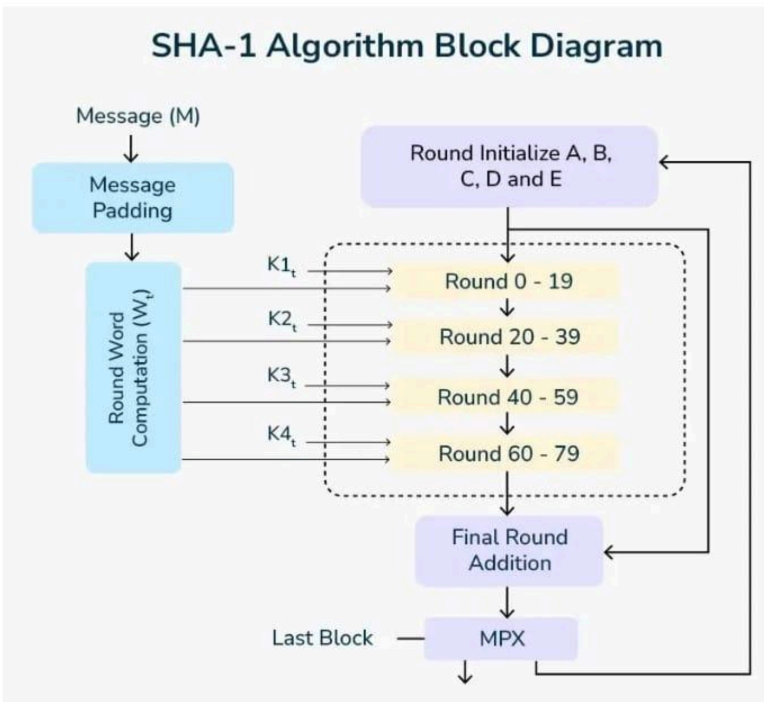
By executing the above Python program, we successfully performed encryption using both Simple and Advanced Columnar Transposition techniques.

EXPERIMENT - 10

AIM:- Write a program to implement SHA 1 algorithm

DESCRIPTION:-

The generateSHA1 method operates similarly but uses the SHA-1 algorithm, resulting in a 160-bit (40-character hexadecimal) hash. SHA-1 produces longer and generally more secure hashes than MD5, though it is still considered vulnerable to certain cryptographic attacks.



CODE:-

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

public class SHA1HashExample {
    public static String generateSHA1(String input) {
        try {
            MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
            byte[] hashBytes = sha1.digest(input.getBytes());

            StringBuilder sb = new StringBuilder();
            for (byte b : hashBytes) {
                sb.append(String.format("%02x", b));
            }
```

```
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
        String input = "Hello, World!";
        System.out.println("SHA-1 Hash: " + generateSHA1(input));
    }
}
```

OUTPUT:-

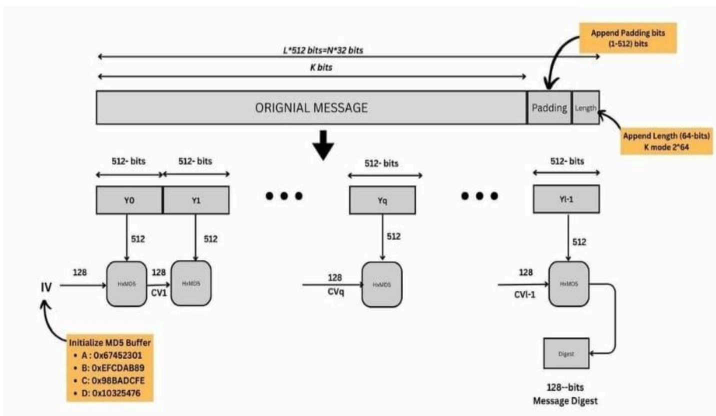
```
java -cp /tmp/nFpSDMYSVe/SHA1HashExample
SHA-1 Hash: 0a0a9f2a6772942557ab5355d76af442
    f8f65e01

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

EXPERIMENT - 11

AIM:- Write a program to implement MD 5 algorithm

DESCRIPTION:-

The generateMD5 method uses the MD5 algorithm, which produces a 128-bit (32-character hexadecimal) hash. This hash is generated by converting the input string to bytes, applying the MD5 hash function, and then formatting each byte as a two-character hexadecimal string

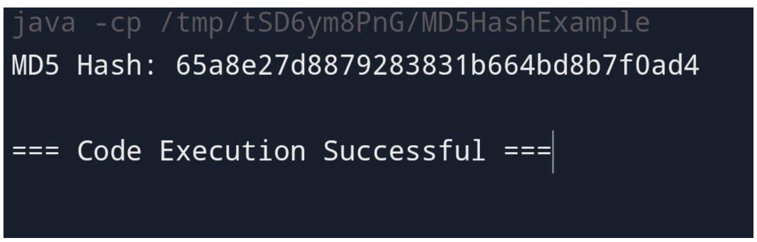


CODE:-

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

public class MD5HashExample {
    public static String generateMD5(String input) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            byte[] hashBytes = md.digest(input.getBytes());

            StringBuilder sb = new StringBuilder();
            for (byte b : hashBytes) {
                sb.append(String.format("%02x", b));
            }
            return sb.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static void main(String[] args) {
```

```
        String input = "Hello, World!";
        System.out.println("MD5 Hash: " + generateMD5(input));
    }
}
```

OUTPUT:-

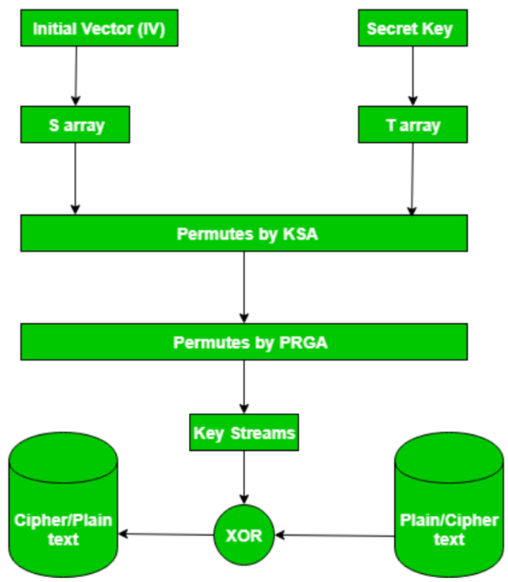


EXPERIMENT - 12

AIM:- Write a program to implement RC4 algorithm

DESCRIPTION:-

RC4 (Rivest Cipher 4) is a fast, symmetric stream cipher widely used in protocols like SSL/TLS and WEP for secure data transmission. It generates a pseudo-random keystream to encrypt or decrypt data by XORing each byte of the plaintext with the keystream byte. RC4's simplicity and speed made it popular, but vulnerabilities (like biased output and susceptibility to certain attacks) have led to its gradual phase-out from most modern encryption standards.



CODE:-

```
# Python3 program for the RC4 algorithm
# for encryption and decryption

# Initialize text and key
plain_text = "001010010010"
key = "101001000001"
n = 3  # No. of bits to consider at a time

# The initial state vector array
S = [i for i in range(0, 2**n)]
key_list = [int(key[i:i + n], 2) for i in range(0, len(key), n)]
pt = [int(plain_text[i:i + n], 2) for i in range(0, len(plain_text), n)]
```

```
# Adjust key_list length to match S
if len(S) != len(key_list):
    key_list += key_list[:len(S) - len(key_list)]

# Key Scheduling Algorithm (KSA)
def KSA(S, key_list):
    j = 0
    for i in range(len(S)):
        j = (j + S[i] + key_list[i]) % len(S)
        S[i], S[j] = S[j], S[i]
    return S

# Pseudo-Random Generation Algorithm (PRGA)
def PRGA(S, text_length):
    i = j = 0
    key_stream = []
    for _ in range(text_length):
        i = (i + 1) % len(S)
        j = (j + S[i]) % len(S)
        S[i], S[j] = S[j], S[i]
        t = (S[i] + S[j]) % len(S)
        key_stream.append(S[t])
    return key_stream

# XOR between generated key stream and input text
def XOR(input_text, key_stream):
    return [input_text[i] ^ key_stream[i] for i in range(len(input_text))]

# Encryption function
def encryption():
    print("Plain text:", plain_text)
    print("Key:", key)
    S_init = KSA(S[:], key_list)  # Initial permutation
    key_stream = PRGA(S_init, len(pt))
    cipher_text = XOR(pt, key_stream)
    encrypted_to_bits = ''.join(f"{bin(c)[2:]:0>{n}}" for c in cipher_text)
    print("Cipher text:", encrypted_to_bits)
    return cipher_text
```

```
# Decryption function
def decryption(cipher_text):
    S_init = KSA(S[:], key_list)
    key_stream = PRGA(S_init, len(pt))
    original_text = XOR(cipher_text, key_stream)
    decrypted_to_bits = ''.join(f"{bin(p)[2:]:0>{n}}" for p in original_text)
    print("Decrypted text:", decrypted_to_bits)

# Driver Code
cipher_text = encryption()
print("-----------------------------------------------------------")
decryption(cipher_text)
```

OUTPUT:-

```
Plain text :  001010010010
Key :  101001000001
n :  3

S :  [0, 1, 2, 3, 4, 5, 6, 7]
Plain text ( in array form ):  [1, 2, 2, 2]
Key list :  [5, 1, 0, 1, 5, 1, 0, 1]

KSA iterations :

0  [5, 1, 2, 3, 4, 0, 6, 7]
1  [5, 7, 2, 3, 4, 0, 6, 1]
2  [5, 2, 7, 3, 4, 0, 6, 1]
3  [5, 2, 7, 0, 4, 3, 6, 1]
4  [5, 2, 7, 0, 6, 3, 4, 1]
5  [5, 2, 3, 0, 6, 7, 4, 1]
6  [5, 2, 3, 0, 6, 7, 4, 1]
7  [1, 2, 3, 0, 6, 7, 4, 5]

The initial permutation array is :  [1, 2, 3, 0, 6, 7, 4, 5]

PGRA iterations :

0  [1, 3, 2, 0, 6, 7, 4, 5]
1  [1, 3, 6, 0, 2, 7, 4, 5]
2  [1, 3, 6, 2, 0, 7, 4, 5]
3  [1, 3, 6, 2, 0, 7, 4, 5]
Key stream :  [7, 1, 6, 1]


Cipher text :  110011100011
```

```
KSA iterations :

0  [5, 1, 2, 3, 4, 0, 6, 7]
1  [5, 7, 2, 3, 4, 0, 6, 1]
2  [5, 2, 7, 3, 4, 0, 6, 1]
3  [5, 2, 7, 0, 4, 3, 6, 1]
4  [5, 2, 7, 0, 6, 3, 4, 1]
5  [5, 2, 3, 0, 6, 7, 4, 1]
6  [5, 2, 3, 0, 6, 7, 4, 1]
7  [1, 2, 3, 0, 6, 7, 4, 5]

The initial permutation array is :  [1, 2, 3, 0, 6, 7, 4, 5]

Key stream :  [7, 1, 6, 1]

PGRA iterations :

0  [1, 3, 2, 0, 6, 7, 4, 5]
1  [1, 3, 6, 0, 2, 7, 4, 5]
2  [1, 3, 6, 2, 0, 7, 4, 5]
3  [1, 3, 6, 2, 0, 7, 4, 5]

Decrypted text :  001010010010
```

EXPERIMENT - 13

AIM:- Write a program to implement Euclidean Algorithm and Advanced Euclidean Algorithm

DESCRIPTION:-

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. The GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorise both numbers and multiply common prime factors.

Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the larger number, the algorithm stops when we find the remainder 0.

Extended Euclidean Algorithm:

Extended Euclidean algorithm also finds integer coefficients x and y such that:ax + by = gcd(a, b). The extended Euclidean algorithm updates the results of gcd(a, b) using the results calculated by the recursive call gcd(b%a, a).

CODE:

#Euclidean algorithm

```
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)

# Driver code
a, b = 10, 15
print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 35, 10
print(f"GCD({a}, {b}) = {gcd(a, b)}")

a, b = 31, 2
```

```
print(f"GCD({a}, {b}) = {gcd(a, b)}")
```

 # Advanced Euclidean Algorithm

```
def gcd_extended(a, b):
    if a == 0:
        return b, 0, 1
    gcd, x1, y1 = gcd_extended(b % a, a)
    x = y1 - (b // a) * x1
    y = x1
    return gcd, x, y

# Driver Program
a = 35
b = 15
gcd, x, y = gcd_extended(a, b)
print(f"gcd({a}, {b}) = {gcd}, x = {x}, y = {y}")
```

OUTPUT:

```
GCD(10, 15) = 5
GCD(35, 10) = 5
GCD(31, 2) = 1

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

```
gcd(35, 15) = 5, x = 1, y = -2

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

EXPERIMENT - 14

AIM:- To know about cryptographic tools.

DESCRIPTION:-

Cryptlib

- Cryptlib is an open-source, cross-platform software security toolkit designed for building cryptographic security into applications. It provides a wide range of cryptographic operations including encryption, digital signatures, secure data transport, and more.
- Cryptlib is distributed under the Sleepycat License, which is compatible with the GNU General Public License (GPL). It is also available under a commercial license for proprietary use.
- Recent updates have focused on expanding support for modern cryptographic algorithms like AES-GCM, Curve25519, and improved support for secure multi-party computations. Cryptlib offers APIs that make it suitable for embedding cryptographic functions in mobile, web, and server applications.

Crypto++

- Crypto++ (also known as CryptoPP, libcrypto++, and libcryptopp) is a widely-used, free and open-source C++ library of cryptographic algorithms, written by Wei Dai. It supports a comprehensive suite of cryptographic primitives and schemes, including block ciphers (AES, DES), stream ciphers, public-key cryptography (RSA, DSA, ECC), and various hash functions (SHA-256, SHA-3).
- Crypto++ is frequently used in research and industry projects. It is maintained actively, with recent enhancements including support for newer encryption standards like ChaCha20-Poly1305, BLAKE3 hashing, and the EdDSA signature algorithm.
- The library emphasizes high performance and portability, making it a preferred choice for C++ developers in security-sensitive applications.

LibreSSL

- LibreSSL is an open-source implementation of the Transport Layer Security (TLS) protocol, developed by the OpenBSD project. It was forked from OpenSSL in 2014 in response to the Heartbleed vulnerability, with the goal of modernizing the codebase, improving security, and reducing complexity.
- LibreSSL has focused on removing obsolete features and improving code quality. It no longer includes deprecated cryptographic algorithms like SSLv2/SSLv3 and MD5, focusing on secure, modern standards like TLS 1.3, SHA-3, and Ed25519.

- Recent updates have included enhancements in cryptographic algorithm support, better random number generation, and improved performance on modern CPU architectures. It remains popular among UNIX-like systems for providing secure communication channels.

OpenSSL

- **Description**: OpenSSL is one of the most widely used open-source libraries for implementing the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. It also provides a comprehensive suite of cryptographic functions, including public-key cryptography, symmetric encryption, and hashing algorithms.
- **Updates**: OpenSSL 3.0 introduced a new "Provider" architecture that allows for better flexibility in integrating cryptographic algorithms. It also includes improved support for post-quantum cryptography, ChaCha20-Poly1305, and updated TLS 1.3 support.
- **Use Cases**: Secure web communication (HTTPS), VPNs, and cryptographic operations in software applications.

PyCryptodome

- **Description**: PyCryptodome is a self-contained Python package of low-level cryptographic primitives. It is a fork of the old PyCrypto library and includes many improvements and additional features such as support for AES, RSA, DSA, and hash functions.
- **Updates**: The latest versions have improved support for AES-GCM, RSA-PSS signatures, and hardware-accelerated encryption on modern processors.
- **Use Cases**: Cryptographic operations in Python-based projects, such as secure file encryption and digital signatures.

Tink (by Google)

- **Description**: Tink is an open-source cryptographic library developed by Google. It provides easy-to-use and secure APIs for cryptographic operations such as encryption, digital signatures, and key management. Tink aims to make cryptography safer and simpler for developers.
- **Updates**: Tink now supports hybrid encryption schemes, better integration with cloud key management services (e.g., Google Cloud KMS, AWS KMS), and enhanced support for AES-SIV for misuse-resistant encryption.
- **Use Cases**: Secure application development, cloud-based encryption, and secure data storage.