EXPERIMENT – 01

AIM:

Perform Encryption and Decryption using the following Substitutiontechniques: (i)Caesar cipher (ii) Playfair cipher (iii) Hill Cipher (iv) Vigenère cipher.

(i) CAESAR CIPHER

DESCRIPTION:

The Caesar cipher is a simple encryption technique that was used by Julius Caesar to send secret messages to his allies. It works by shifting the letters in the plaintext message by a certain number of positions, known as the "shift" or "key". The Caesar Cipher technique is one of the earliest and simplest methods of encryption techniques.
The Caesar cipher shifts each letter in the plaintext by a fixed number of positions down the alphabet. For example, with a shift of 3, 'A' becomes 'D'.
Encryption: $E(x) = (x + k) \bmod 26$
Decryption: $D(x) = (x - k) \bmod 26$

EXAMPLE:

Text : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Shift: 23
Cipher: XYZABCDEFGHIJKLMNOPQRSTUVW

Text : ATTACKATONCE
Shift: 4
Cipher: EXXEGOEXSRGI

ALGORITHM:

1. Choose a Shift Value: Select any integer n as the shift value. The shift value can be positive, negative, or zero.
2. Write the Alphabet: Write down the alphabet in order from A to Z.
3. Create a Shifted Alphabet:
   - Calculate the effective shift by using the modulus operation with the alphabet size (26 letters): Effective shift= n mod 26
   - Shift each letter of the original alphabet by the effective shift value.
   - For each letter in the original alphabet, find its corresponding letter in the shifted alphabet by moving n positions forward if n is positive or |n| positions backward if n is negative.
   - If the shift moves past 'Z', wrap around to the beginning of the alphabet, or if it moves before 'A', wrap around to the end of the alphabet.
4. Encrypt the Message:
   - Replace each letter of the original message with its corresponding letter from the shifted alphabet.

CODE:

```
def caesar_cipher_encrypt(plaintext, shift):
    cipher_text = ''
    for i in range(len(plaintext)):
        char = plaintext[i]

        if char.isupper():
            cipher_text += chr((ord(char) + shift - 65) % 26 + 65)
        elif char.islower():
            cipher_text += chr((ord(char) + shift - 97) % 26 + 97)
        else:
            cipher_text += char
    return cipher_text

def caesar_cipher_decrypt(ciphertext, shift):
    plaintext = ''
    for i in range(len(ciphertext)):
        char = ciphertext[i]

        if char.isupper():
            plaintext += chr((ord(char) - shift - 65) % 26 + 65)
        elif char.islower():
            plaintext += chr((ord(char) - shift - 97) % 26 + 97)
        else:
            plaintext += char
    return plaintext

# Input from the user
plaintext = input("Enter the Plaintext: ")
shift = int(input("Enter the shift value: "))

# Encryption
ciphertext = caesar_cipher_encrypt(plaintext, shift)
print("Cipher Text = " + ciphertext)

# Decryption
decrypted_text = caesar_cipher_decrypt(ciphertext, shift)
print("Decrypted Text = " + decrypted_text)
```

OUTPUT:

Enter the Plaintext: Caesar Cipher
Enter the shift value: 14
Cipher Text = Qosgof Qwdvsf
Decrypted Text = Caesar Cipher

OUTPUT ANALYSIS:

Encryption Time Complexity: O(n)
Decryption Time Complexity: O(n)
Space Complexity: O(n)

Where, n refers to the length of the plaintext or ciphertext being processed.

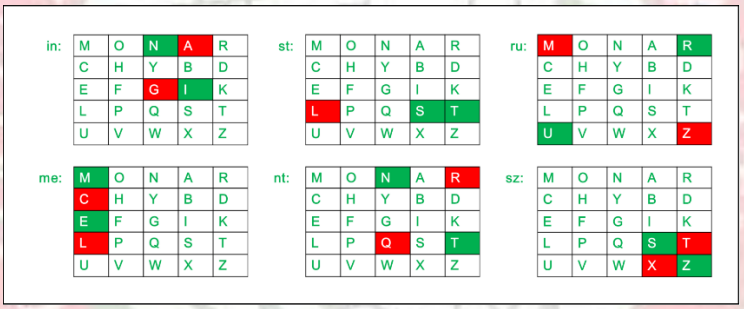
(ii) PLAYFAIR CIPHER

DESCRIPTION:

Playfair cipher is an encryption algorithm to encrypt or encode a message. It is the same as a traditional cipher. The only difference is that it encrypts a **digraph** (a pair of two letters) instead of a single letter.
The Playfair cipher encrypts digraphs (pairs of letters) using a 5x5 matrix of letters. Duplicate letters are replaced, and 'J' is typically combined with 'I'.
Encryption: Substitute each pair using the matrix rules.
Decryption: Reverse the substitution process using the same matrix.

EXAMPLE:

Plain Text: "gatlmzclrqtx"
Decrypted Text: instrumentsz
Decryption:



(red)-> (green)
ga -> in
tl -> st
mz -> ru
cl -> me
rq -> nt
tx -> sz

ALGORITHM:

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.
- 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.

CODE:

```
def create_matrix(keyword):
    keyword = ''.join(sorted(set(keyword), key=keyword.index))
    alphabet = 'ABCDEFGHIKLMNOPQRSTUVWXYZ'
    matrix = [c for c in keyword if c in alphabet]

    for char in alphabet:
        if char not in matrix:
            matrix.append(char)
    return [matrix[i:i+5] for i in range(0, 25, 5)]

def preprocess_text(text):
    text = text.upper().replace('J', 'I')
    digraphs = []
    i = 0
    while i < len(text):
        a = text[i]
        b = text[i+1] if i+1 < len(text) else 'X'
        if a == b:
            digraphs.append(a + 'X')
            i += 1
        else:
            digraphs.append(a + b)
            i += 2
    return digraphs

def find_position(matrix, char):
    for i, row in enumerate(matrix):
        if char in row:
            return i, row.index(char)
    return None

def encrypt_digraph(digraph, matrix):
    r1, c1 = find_position(matrix, digraph[0])
    r2, c2 = find_position(matrix, digraph[1])

    if r1 == r2:
        return matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]
    elif c1 == c2:
        return matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]
    else:
        return matrix[r1][c2] + matrix[r2][c1]

def decrypt_digraph(digraph, matrix):
    r1, c1 = find_position(matrix, digraph[0])
```

```
    r2, c2 = find_position(matrix, digraph[1])
    if r1 == r2:
        return matrix[r1][(c1 - 1) % 5] + matrix[r2][(c2 - 1) % 5]
    elif c1 == c2:
        return matrix[(r1 - 1) % 5][c1] + matrix[(r2 - 1) % 5][c2]
    else:
        return matrix[r1][c2] + matrix[r2][c1]

def playfair_cipher(text, keyword, mode='encrypt'):
    matrix = create_matrix(keyword)
    digraphs = preprocess_text(text)
    if mode == 'encrypt':
        return ''.join(encrypt_digraph(d, matrix) for d in digraphs)
    elif mode == 'decrypt':
        return ''.join(decrypt_digraph(d, matrix) for d in digraphs)

keyword = input("Enter the keyword: ").upper().replace('J', 'I')
plaintext = input("Enter the plaintext to encrypt: ").upper()

encrypted = playfair_cipher(plaintext, keyword, mode='encrypt')
print("Encrypted:", encrypted)

decrypted = playfair_cipher(encrypted, keyword, mode='decrypt')
print("Decrypted:", decrypted)
```

OUTPUT:

Enter the keyword: SECRETKEY
Enter the plaintext to encrypt: welldonemyfriends
Encrypted: VCIZQLPOSNKGBPYVKT
Decrypted: WELXLDONEMYFRIENDS

OUTPUT ANALYSIS:

Encryption Time Complexity: O(n)
Decryption Time Complexity: O(n)
Space Complexity: O(1)

Where, n refers to the length of the plaintext or ciphertext being processed.

(iii) HILL CIPHER

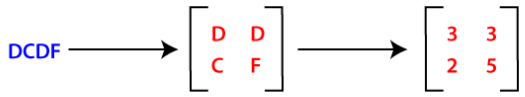
DESCRIPTION:

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.
Encryption: C = K*P mod 26
Decryption: P = K-1*C mod 26

EXAMPLE:

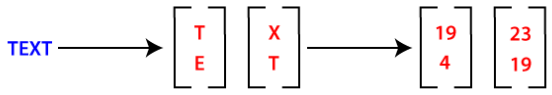
Steps For Encryption

Step 1: Let's say our key text (2x2) is **DCDF**. Convert this key using a substitution scheme into a 2x2 key matrix as shown below:

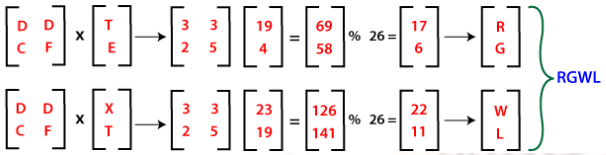


Step 2: Now, we will convert our plain text into vector form. Since the key matrix is 2x2, the vector must be 2x1 for matrix multiplication. (Suppose the key matrix is 3x3, a vector will be a 3x1 matrix.)

In our case, plain text is **TEXT** that is four letters long word; thus we can put in a 2x1 vector and then substitute as:

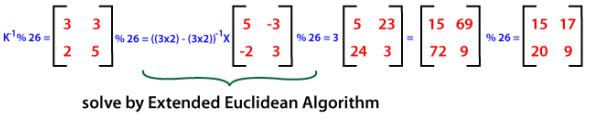


Step 3: Multiply the key matrix with each 2x1 plain text vector, and take the modulo of result (2x1 vectors) by 26. Then concatenate the results, and we get the encrypted or ciphertext as **RGWL**.



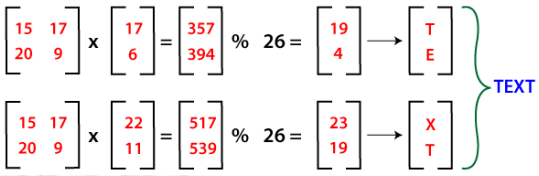
Steps For Decryption

Step 1: Calculate the inverse of the key matrix. First, we need to find the determinant of the key matrix (must be between 0-25). Here the Extended Euclidean algorithm is used to get modulo multiplicative inverse of key matrix determinant



solve by Extended Euclidean Algorithm

Step 2: Now, we multiply the 2x1 blocks of ciphertext and the inverse of the key matrix. The resultant block after concatenation is the plain text that we have encrypted i.e., **TEXT**.



ALGORITHM:

1. Input:

- **Plaintext**: The text to encrypt.
- **Key Matrix**: An n x n matrix (mod 26).
- **Block Size (n)**: The size of the key matrix and the block of plaintext.

2. Preprocessing:

- **Convert Plaintext to Numbers**: Map each letter to a number (A=0, B=1, ..., Z=25).
- **Padding**: If the plaintext length isn't a multiple of n, pad with 'X' or another letter.

3. Encryption:

- **Divide Plaintext into Blocks**: Split the plaintext into blocks of size n.
- **Matrix Multiplication**: For each block, multiply the block vector by the key matrix (mod 26).
- **Convert to Ciphertext**: Convert the resulting numbers back to letters.

4. Output:

- **Ciphertext**: The encrypted text formed by concatenating the resulting letters from each block.

CODE:

```
def createMatrix(size):
    return [[0] * size for _ in range(size)]

def getKeyMatrix(key, size, keyMatrix):
    k = 0
    for i in range(size):
        for j in range(size):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

def encrypt(messageVector, keyMatrix, size, cipherMatrix):
    for i in range(size):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(size):
                cipherMatrix[i][j] += (keyMatrix[i][x] * messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key, size):
    keyMatrix = createMatrix(size)
    messageVector = [[0] for _ in range(size)]
    cipherMatrix = [[0] for _ in range(size)]
    getKeyMatrix(key, size, keyMatrix)
    for i in range(size):
        messageVector[i][0] = ord(message[i]) % 65
    encrypt(messageVector, keyMatrix, size, cipherMatrix)
    CipherText = []
    for i in range(size):
        CipherText.append(chr(cipherMatrix[i][0] + 65))
    print("Ciphertext: ", "".join(CipherText))

size = int(input("Enter the size of the matrix (e.g., 3 for 3x3): "))
message = input(f"Enter the {size}-letter Plaintext: ").upper()
key = input(f"Enter the {size*size}-letter key: ").upper()
HillCipher(message, key, size)
```

OUTPUT:

Enter the size of the matrix (e.g., 3 for 3x3): 3
Enter the 3-letter Plaintext: BYE
Enter the 9-letter key: HILLMAGIC
Ciphertext: JNY

OUTPUT ANALYSIS:

Encryption Time Complexity: $O(k^2)$
Decryption Time Complexity: $O(k^3)$
Space Complexity: $O(k^2)$
Where, k refers to the size of the square matrix used for encryption and decryption

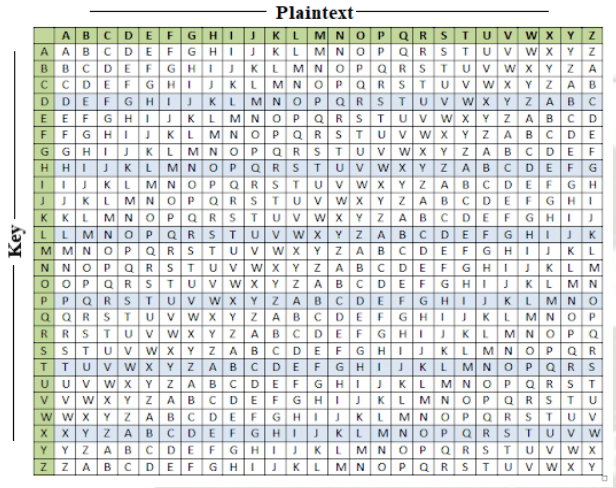
(iv) VIGENERE CIPHER

DESCRIPTION:

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 Vigenere Square or Vigenere 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 Cipher.
Encryption: $E_i = (P_i + K_i) \bmod 26$
Decryption: $D_i = (E_i - K_i) \bmod 26$

EXAMPLE:

Input : Plaintext : VIGENERECIPHER
Keyword : KEY
Output : Ciphertext : FMEORCBIASTFOV
For generating key, the given keyword is repeated in a circular manner until it matches the length of the plain text.
The keyword "KEY" generates the key "KEYKEYKEYKEYKE"



ALGORITHM:

- Write the plaintext.
- Use the plaintext and the key letter to select a row and a column in the Vigenere table.
- The first letter of the plaintext is the first row and the key is the first column. For example, if the plaintext is VIGENERECIPHER and the key is KEY, then the first row will be the one that starts with V, and the column will be the one that starts with K.
- The first letter of the ciphertext will be the letter where the first row and column intersect. In the case of our example that will be the letter F.
- Now, this process is continued till the entire plaintext is turned into a ciphertext. For VIGENERECIPHER, that will be FMEORCBIASTFOV.

CODE:

```
def generate_key(msg, key):
    key = list(key)
```

```
    if len(msg) == len(key):
        return key
    else:
        for i in range(len(msg) - len(key)):
            key.append(key[i % len(key)])
    return "".join(key)

def encrypt_vigenere(msg, key):
    encrypted_text = []
    key = generate_key(msg, key)
    for i in range(len(msg)):
        char = msg[i]
        if char.isupper():
            encrypted_char = chr((ord(char) + ord(key[i]) - 2 * ord('A')) % 26 + ord('A'))
        elif char.islower():
            encrypted_char = chr((ord(char) + ord(key[i]) - 2 * ord('a')) % 26 + ord('a'))
        else:
            encrypted_char = char
        encrypted_text.append(encrypted_char)
    return "".join(encrypted_text)

def decrypt_vigenere(msg, key):
    decrypted_text = []
    key = generate_key(msg, key)
    for i in range(len(msg)):
        char = msg[i]
        if char.isupper():
            decrypted_char = chr((ord(char) - ord(key[i]) + 26) % 26 + ord('A'))
        elif char.islower():
            decrypted_char = chr((ord(char) - ord(key[i]) + 26) % 26 + ord('a'))
        else:
            decrypted_char = char
        decrypted_text.append(decrypted_char)
    return "".join(decrypted_text)

text_to_encrypt = input("Enter the Plaintext to encrypt: ")
key = input("Enter the encryption key: ")
encrypted_text = encrypt_vigenere(text_to_encrypt, key)
print(f"Encrypted Text: {encrypted_text}")
decrypted_text = decrypt_vigenere(encrypted_text, key)
print(f"Decrypted Text: {decrypted_text}")
```

OUTPUT:

Enter the Plaintext to encrypt: VIGENERECIPHER
Enter the encryption key: KEY
Encrypted Text: FMEORCBIASTFOV
Decrypted Text: VIGENERECIPHER

OUTPUT ANALYSIS:

Encryption Time Complexity: O(n)
Decryption Time Complexity: O(n)
Space Complexity: O(n)
Where, n refers to the length of the plaintext or ciphertext being processed

EXPERIMENT – 02

AIM: Implement Transposition Cipher using Rail Fence Technique.

DESCRIPTION:

The Rail Fence Cipher is a type of transposition cipher used to encrypt plaintext by arranging it in a zigzag pattern across multiple "rails" and thenreading it row by row to produce the ciphertext. In the encryption process, characters of the plaintext are placed in a rail matrix following a zigzag pattern determined by the key, which represents the number of rails. Once the entire plaintext is written in this pattern, the ciphertext is formed by reading characters row by row from the top rail to the bottom.

For decryption, the process is reversed. First, the zigzag pattern is recreated in a matrix of the same dimensions used during encryption. The positions where characters should be placed are marked, and then the ciphertext characters are placed back into these marked positions. Finally, the plaintextis reconstructed by reading the characters in the original zigzag pattern.

This cipher is relatively simple to implement and provides a basic level of security by scrambling the text, making it less recognizable. However, it is vulnerable to various forms of cryptanalysis and is not suitable for securingsensitive information. The Rail Fence Cipher is primarily of historical and educational interest, demonstrating fundamental concepts in the study of cryptography.

EXAMPLE:

Encryption
Let us start by considering "RAILFENCE" as a plaintext. Let us now assume that there are three rails or fences, which is also known as a key. The zigzag pattern's height will be determined by the key. The message can then be written diagonally, from left to right, in a zigzag pattern –

R				F				E
	A		L		E		C	
		I				N		

In order to create the ciphertext we will merge distinct rows, which in this case is "RFEALECIN."

Decryption
The number of rows and columns in the cipher text needs to be determined before we can start the decryption process. The length of the ciphertext is equal to the number of columns. After that, we need to determine how many rows-which function as the key-were encrypted.
Now that we know how many rows and columns there are, we can build the table and figure out where the letters should go because the rail fence cipher zigzags to encrypt the text diagonally from left to right –

*				*				*
	*		*		*		*	
		*				*		

The points where letters from the ciphertext are inserted to create the plaintext are indicated by the *(asterisk). Beginning from the top row, which is the first "rail," we fill in the letters going left to right. Up until all of the asterisk spots are filled with letters from the ciphertext, we then carry on with this pattern on the following rail and so on −

R				F				E
	*		*		*		*	
		*				*		

Let us finish the table above −

R				F				E
	A		L		E		C	
		I				N		

Finally, we are able to combine the characters from left to right and top to bottom to get the Plaintext, "RAILFENCE."

ALGORITHM:

1. **Input:**
   - plaintext: The message to be encrypted.
   - num_rails: The number of rails (or rows) to use.
2. **Initialize:**
   - Create an array of strings, rails, of size num_rails to store the characters for each rail.
   - Set direction to down to track the direction of movement across the rails.
   - Initialize rail_index to 0 to keep track of the current rail.
3. **Process Each Character:**
   - For each character in the plaintext:
     - Append the character to rails[rail_index].
     - If rail_index is 0, set direction to down.
     - If rail_index is num_rails - 1, set direction to up.
     - If direction is down, increment rail_index.

- If direction is up, decrement rail_index.

4. **Generate Ciphertext:**
   - Concatenate all strings in the rails array to form the ciphertext.
5. **Output:**
   - Return the ciphertext.

CODE:

```
def encrypt_rail_fence(text, key):
    rail = [['\n' for i in range(len(text))] for j in range(key)]
    direction_down = False
    row, col = 0, 0

    for i in range(len(text)):
        rail[row][col] = text[i]
        col += 1

        if row == 0 or row == key - 1:
            direction_down = not direction_down
        row += 1 if direction_down else -1

    ciphertext = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                ciphertext.append(rail[i][j])

    return "".join(ciphertext)

def decrypt_rail_fence(cipher, key):
    rail = [['\n' for i in range(len(cipher))] for j in range(key)]
    direction_down = None
    row, col = 0, 0

    for i in range(len(cipher)):
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False

        rail[row][col] = '*'
        col += 1
        row += 1 if direction_down else -1

    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if rail[i][j] == '*' and index < len(cipher):
                rail[i][j] = cipher[index]
                index += 1

    plaintext = []
    row, col = 0, 0
    for i in range(len(cipher)):
        if row == 0:
            direction_down = True
```

```
        if row == key - 1:
            direction_down = False

        if rail[row][col] != '*':
            plaintext.append(rail[row][col])
            col += 1
        row += 1 if direction_down else -1

    return "".join(plaintext)

plaintext = input("Enter the plaintext: ")
key = int(input("Enter the key (number of rails): "))

ciphertext = encrypt_rail_fence(plaintext, key)
print(f"Ciphertext: {ciphertext}")

decrypted_text = decrypt_rail_fence(ciphertext, key)
print(f"Decrypted Text: {decrypted_text}")
```

OUTPUT:

Enter the plaintext: RAILFENCE
Enter the key (number of rails): 3
Ciphertext: RFEALECIN
Decrypted Text: RAILFENCE

OUTPUT ANALYSIS:

OPERATION	TIME COMPLEXITY	SPACE COMPLEXITY
Encryption:		
1. Initializing and filling matrix	O(k * n)	O(k * n)
2. Constructing ciphertext	O(n)	O(n)
Total (Encryption):	O(k * n)	O(k * n)
Decryption:		
1. Initializing and marking matrix	O(k * n)	O(k * n)
2. Filling matrix with ciphertext	O(n)	O(n)
3. Constructing plaintext	O(n)	O(n)
Total (Decryption):	O(k * n)	O(k * n)

- **Time Complexity:** Both encryption and decryption have a time complexity of O(k * n).
- **Space Complexity:** The space complexity is also O(k * n) for both operations.

The complexity scales with the key **k** and the length of the ciphertext **n**.

CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినే భవ
1979

EXPERIMENT – 03

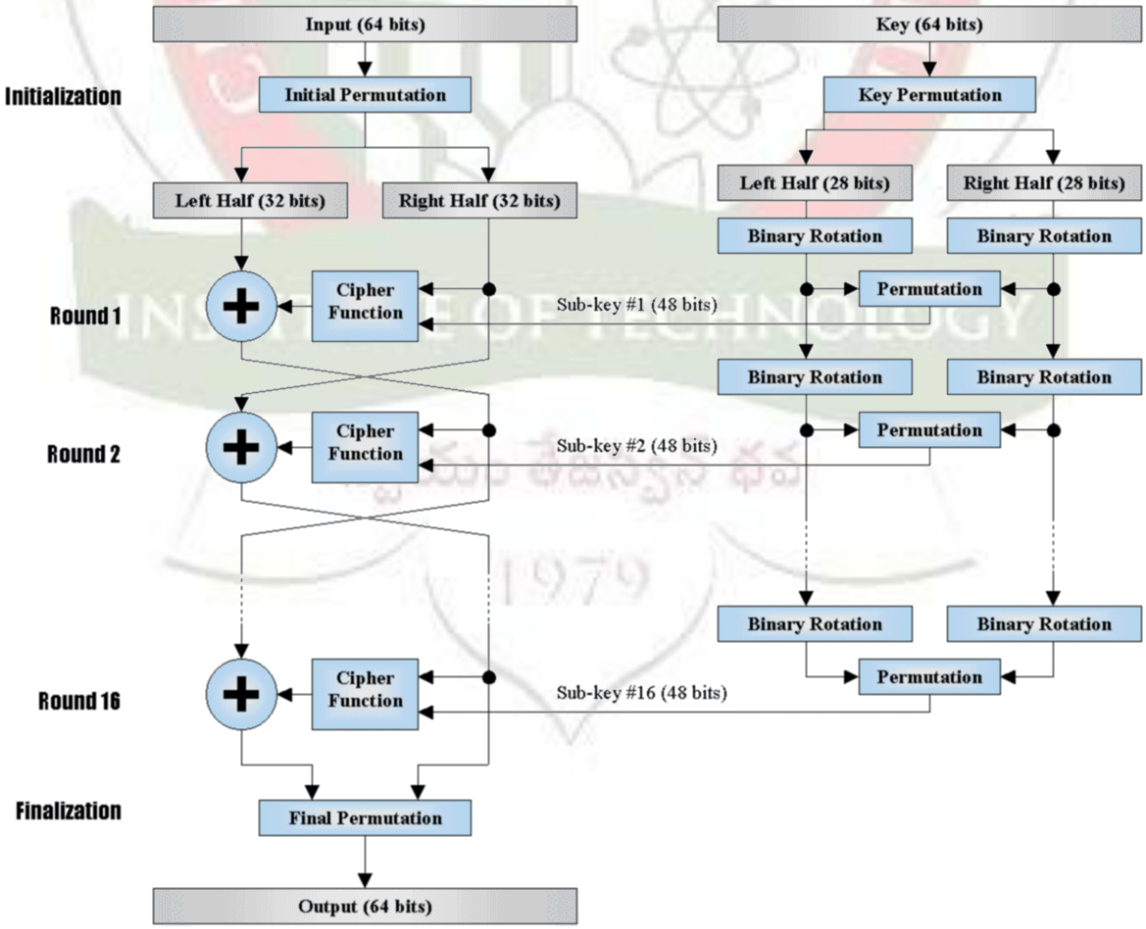
AIM: Implement Data Encryption Standard (DES) Algorithm for Symmetric Key Encryption.

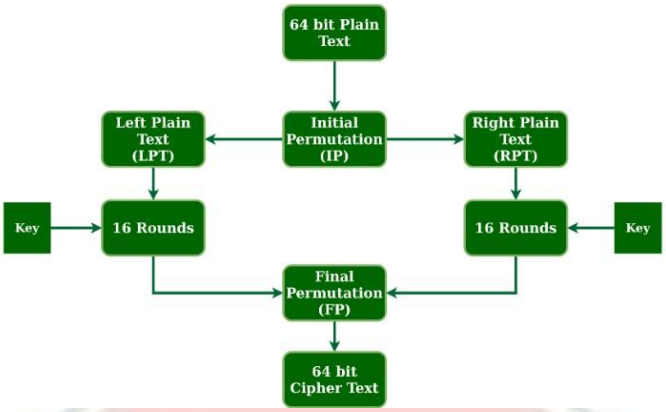
DESCRIPTION:

The **Data Encryption Standard (DES)** is a symmetric-key algorithm for the encryption of digital data. It operates on blocks of 64 bits using a 56-bit key. DES encrypts data in 16 rounds of processing using a combination of permutation and substitution techniques.

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.

Key Steps in DES:





1. **Initial Key Transformation**:
   - The original 64-bit key is reduced to 56 bits by discarding every 8th bit (bit positions 8, 16, 24, etc.).
   - In each round, a 48-bit subkey is derived from this 56-bit key through circular shifts and compression.
2. **Initial Permutation (IP)**:
   - The 64-bit plaintext block is rearranged according to a predefined table.
   - The permuted block is split into two 32-bit halves: Left Plain Text (LPT) and Right Plain Text (RPT).
3. **Rounds of Processing (16 Rounds)**:
   - In each round, the RPT is expanded to 48 bits using **Expansion Permutation** and XORed with the subkey.
   - The result is substituted using **S-boxes** and combined with the LPT.
   - The halves are swapped after each round.
4. **Final Permutation (FP)**:
   - After 16 rounds, the two halves are rejoined and subjected to a final permutation, resulting in a 64-bit ciphertext.

Key Processes in Each Round:

1. **Key Transformation**: Generates a unique 48-bit subkey per round by circularly shifting parts of the 56-bit key and applying compression.
2. **Expansion Permutation**: Expands the 32-bit RPT to 48 bits by splitting it into 4-bit blocks and expanding each into 6-bit blocks.
3. **Substitution and Permutation**: The expanded RPT is substituted and permuted using predefined tables (S-boxes and P-boxes).

This combination of substitutions, permutations, and XOR operations makes DES hard to break, though advances in computing have revealed vulnerabilities to brute-force and other cryptographic attacks.

CODE:

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
import binascii

def pad(text):
    while len(text) % 8 != 0:
        text += ' '
    return text

def encrypt_DES(key, message):
    des = DES.new(key, DES.MODE_ECB)
    padded_message = pad(message)
    encrypted_message = des.encrypt(padded_message.encode('utf-8'))
    return binascii.hexlify(encrypted_message).decode('utf-8')

def decrypt_DES(key, encrypted_message):
    des = DES.new(key, DES.MODE_ECB)
    decrypted_message = des.decrypt(binascii.unhexlify(encrypted_message))
    return decrypted_message.decode('utf-8').rstrip()

key = get_random_bytes(8)
message = input("Enter a Message: ")
print(f"Original Message: {message}")
encrypted_message = encrypt_DES(key, message)
print(f"Encrypted Message (Hex): {encrypted_message}")
decrypted_message = decrypt_DES(key, encrypted_message)
print(f"Decrypted Message: {decrypted_message}")
```

OUTPUT:

Enter a Message: This Experiment was done in the lab
Original Message: This Experiment was done in the lab
Encrypted Message (Hex):
90a2ff186d873bc8543e0512554a973d2b637996f294713390572ca6a7d5966b42bcc8fc1960287c
Decrypted Message: This Experiment was done in the lab

OUTPUT ANALYSIS:

Time Complexity

- **Encryption/Decryption:** DES performs 16 rounds of processing for each 64-bit block, involving constant-time operations per round. Thus, the time complexity is O(1) for each block.

Space Complexity

- **Key Storage:** Requires storage for a 56-bit key (plus 8 parity bits), so space complexity is O(1).
- **Data Storage:** Operates on 64-bit blocks, and intermediate states require constant space, so space complexity is O(1).


CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY
స్వయం తేజస్వినా భవ
1979

EXPERIMENT – 04

AIM: Implement Advanced Encryption Standard (AES) Algorithm for Symmetric Key Encryption.

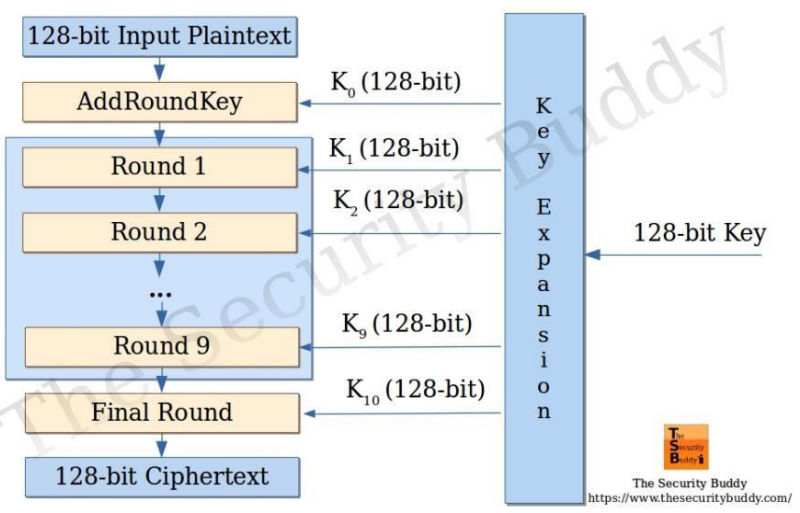
DESCRIPTION:

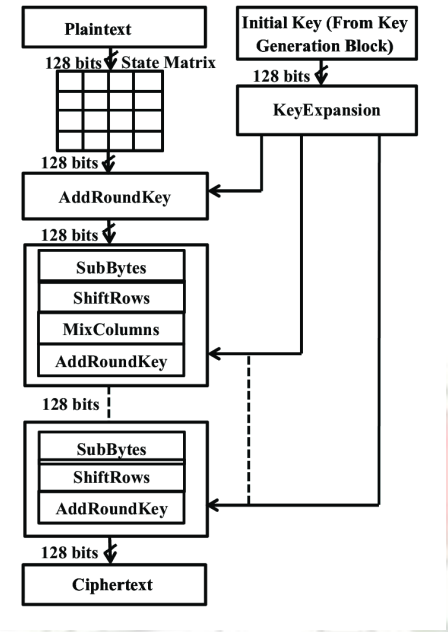
Advanced Encryption Standard (AES) is a highly trusted **encryption algorithm** used to secure data by converting it into an unreadable format without the proper key. Developed by the National Institute of Standards and Technology (NIST), **AES encryption** uses various **key lengths** (128, 192, or 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.

Key Features of AES:

1. **Symmetric Key Algorithm**: AES uses the same key for both encryption and decryption. Both the sender and the receiver must have the same secret key.
2. **Block Cipher**: AES operates on fixed-size blocks of data. Specifically, it processes blocks of 128 bits (16 bytes) at a time. If the input is not a multiple of 128 bits, padding is added to the plaintext.
3. **Variable Key Length**: AES supports three key lengths:
   - **128-bit** key (most common)
   - **192-bit** key
   - **256-bit** key (most secure)
4. **Iterative Algorithm**: AES encrypts data through multiple rounds of transformation (also known as rounds). The number of rounds depends on the key size:
   - 10 rounds for 128-bit keys
   - 12 rounds for 192-bit keys
   - 14 rounds for 256-bit keys

Key Steps in AES:





1. Input Block and Key Expansion:

- AES operates on a 128-bit (16-byte) block of plaintext.
- The key (128-bit, 192-bit, or 256-bit) is expanded into multiple round keys using a process called **key expansion**.
- The original key is used in the first round, and new keys are derived for subsequent rounds.

2. Initial Round:

Before entering the main rounds, AES performs an **initial round**:

- **AddRoundKey**: The plaintext block is XORed with the initial round key (derived from the original key).

3. Main Rounds:

Each main round consists of four operations:

- **SubBytes**: Each byte of the input block is replaced with a corresponding value from a predefined substitution box (S-box). This step provides non-linearity to the algorithm.
- **ShiftRows**: The rows of the 4x4 matrix formed from the input block are shifted. Row 1 remains unchanged, Row 2 is shifted by one byte to the left, Row 3 by two bytes, and Row 4 by three bytes.
- **MixColumns**: Each column of the 4x4 matrix is transformed by multiplying it with a fixed polynomial in GF(2^8) (Galois Field). This step combines bytes in each column and provides diffusion (spreading out of the plaintext bits across the ciphertext).
- **AddRoundKey**: The current block is XORed with the round key.

For the final round, the **MixColumns** step is omitted.

4. Final Round:

After completing the main rounds, AES performs a final round, which consists of:

- **SubBytes**
- **ShiftRows**
- **AddRoundKey**

The result of the final round is the ciphertext.

CODE:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

key = get_random_bytes(16)
```

```
iv = get_random_bytes(16)

def aes_encrypt(plain_text):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    cipher_text = cipher.encrypt(pad(plain_text.encode('utf-8'), AES.block_size))
    return cipher_text

def aes_decrypt(cipher_text):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plain_text = unpad(cipher.decrypt(cipher_text), AES.block_size).decode('utf-8')
    return plain_text

plain_text = input("Enter a Plaintext: ")
print("Original Message:", plain_text)
cipher_text = aes_encrypt(plain_text)
print("Encrypted Message (in bytes):", cipher_text)
decrypted_text = aes_decrypt(cipher_text)
print("Decrypted Message:", decrypted_text)
```

OUTPUT:

Enter a Plaintext: We are CBITians
Original Message: We are CBITians
Encrypted Message (in bytes): b'ZD\x1b\xf9a\xbf\x96\x87\xb4\x8cVGU\xc5\x08\xda'
Decrypted Message: We are CBITians

OUTPUT ANALYSIS:

Time Complexity:

- **Encryption/Decryption:** The time complexity for AES is O(n) for each block of nnn bits. AES processes data in blocks of 128 bits using a variable number of rounds (10 for 128-bit keys, 12 for 192-bit keys, and 14 for 256-bit keys). Therefore, the time complexity is effectively constant: O(10) = O(1) for 128-bit AES.
- In practice, AES is significantly faster than DES due to its simpler round function and is optimized in hardware implementations.

Space Complexity:

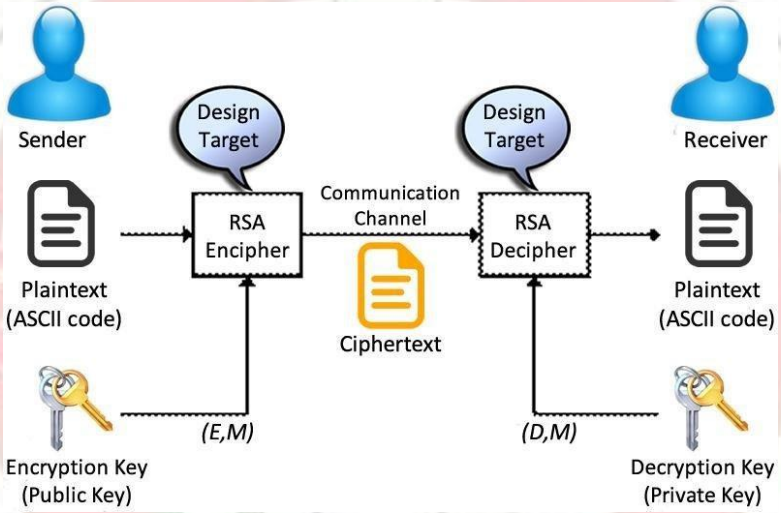
- AES also operates on fixed block sizes of 128 bits, and the key sizes can be 128, 192, or 256 bits.
- The space complexity is O(1) because it does not depend on the size of the input but is determined by fixed storage requirements for the state and key schedules.

EXPERIMENT – 05

AIM: Implement RSA Asymmetric Key Encryption Algorithm.

DESCRIPTION:

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric means that it works on two different keys i.e. **Public Key** and **Private Key.** As the name describes the Public Key is given to everyone and the Private key is kept private.



EXAMPLE:

For example, suppose the receiver selected the primes p=11 and q=17, along with e=3.

1. The receiver calculates n = p x q = (11)(17) = 187, which is half of the public key.
2. The receiver also calculates $\phi(n)=(p-1)(q-1)$ = 10x16=160. e=3 was also chosen.
3. The receiver calculates d =107, since then $de \equiv 1 \pmod{\phi(n)}$.
4. The receiver distributes his public key: n =187 and e=3.

Now suppose the sender wanted to send the message "HELLO". Since n is so small, the sender will have to send his message character by character.

1. 'H' is 72 in ASCII, so the message text is m=72m=72.
2. The sender calculates $me=72^3 \equiv 183 \pmod{187}$, making the ciphertext c=183. Again, this is the only information an attacker can get, since the attacker does not have the private key.
3. The receiver calculates $cd=183^{107} \equiv 72 \pmod{187}$, thus getting the message of m=72.
4. The receiver translates 72 into 'H'.

ALGORITHM:

1. **Choose Two Prime Numbers:**
   - Select two distinct prime numbers p and q.

2. **Compute n:**
   - Calculate n = p × q. This value n is used as the modulus for both the public and private keys.
3. **Compute ϕ(n):**
   - Calculate the totient ϕ(n) = (p−1) (q−1).
4. **Choose Public Exponent e:**
   - Select an integer e such that 1<e<ϕ(n) and gcd(e, ϕ(n)) = 1.
5. **Compute Private Exponent d:**
   - Compute d such that d × e ≡ 1 mod ϕ(n). This can be done using the Extended Euclidean Algorithm.
6. **Public and Private Key Pair:**
   - The public key is (e, n).
   - The private key is (d, n).

Encryption:

1. **Convert Message to Integer:**
   - Represent the plaintext message as an integer mmm such that 0 ≤ m < n.
2. **Encrypt the Message:**
   - Compute the ciphertext c using the public key: $c \equiv m^e \bmod n$

Decryption:

1. **Decrypt the Ciphertext:**
   - Compute the original message m using the private key: $m \equiv c^d \bmod n$

CODE:

```
import math

def gcd(a, h):
    temp = 0
    while(1):
        temp = a % h
        if (temp == 0):
            return h
        a = h
        h = temp

p = int(input("Enter a prime number p: "))
q = int(input("Enter another prime number q: "))
n = p * q
e = 2
phi = (p - 1) * (q - 1)

while (e < phi):
    if gcd(e, phi) == 1:
        break
    else:
        e = e + 1

k = 2
d = (1 + (k * phi)) // e
```

```
msg = int(input("Enter a message to encrypt (as an integer): "))
print("Message data = ", msg)

c = pow(msg, e) % n
print("Encrypted data = ", c)

m = pow(c, d) % n
print("Original Message Sent = ", m)
```

OUTPUT:

```
Enter a prime number p: 11
Enter another prime number q: 17
Enter a message to encrypt (as an integer): 72
Message data = 72
Encrypted data = 183
Original Message Sent = 72
```

OUTPUT ANALYSIS:

Time Complexity

1. **Key Generation:**
   - **Finding gcd(e,ϕ):** The gcd function is implemented using the Euclidean algorithm, which runs in O(log(min(a,h))). In the worst case, we need to compute this until e is co-prime with ϕ. The maximum number of iterations can be limited by the size of ϕ.
   - **Overall Time Complexity for Key Generation:** O(log(ϕ))
2. **Encryption:**
   - The encryption process involves computing $c=m^e \bmod n$. The time complexity for modular exponentiation can be done in O(log(e)), which is efficient when using the method of exponentiation by squaring.
   - Overall, for encryption, the time complexity is O(log(e)).
3. **Decryption:**
   - The decryption process involves computing $m=c^d \bmod$. Similarly, the time complexity for this modular exponentiation is also O(log(d)).
   - Overall, for decryption, the time complexity is O(log(d)).

Space Complexity

1. **Key Generation:**
   - The space used for variables p, q, n, e, ϕ, d is constant, i.e., O(1).
2. **Encryption and Decryption:**
   - The space used for storing the ciphertext c and the decrypted message m is also constant, i.e., O(1).

EXPERIMENT – 06

AIM: Demonstrate how two parties can securely exchange secret keys over an insecure Communication Channel using the Diffie-Hellman Key Exchange Algorithm.

DESCRIPTION:

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.

Diffie-Hellman Key Exchange is a cryptographic protocol used to securely exchange cryptographic keys over a public channel. The main goal of Diffie-Hellman is to allow two parties to establish a shared secret key, even if they are communicating over an insecure, public network.

Key Features:

- **Key Exchange**: It doesn't encrypt messages itself but generates a shared key that can be used for symmetric encryption.
- **Public-Private Key System**: It uses mathematical properties of prime numbers and modular arithmetic to create a shared key.
- **No prior shared secret**: Both parties can generate the same secret key without having exchanged any secret information beforehand.

EXAMPLE:

Consider a prime number $q = 7$. Select a primitive root α such that $\alpha < q$ and α is a primitive root of q. A primitive root of $q = 7$ is $\alpha = 5$.

Now, assume $x_A = 3$ as the private key of party A. Calculate the public key of A using the formula $Y_A = \alpha^{x_A} \mod q$. So, $Y_A = 5^3 \mod 7 = 125 \mod 7 = 6$.

Next, assume $x_B = 4$ as the private key of party B. Calculate the public key of B using the formula $Y_B = \alpha^{x_B} \mod q$. So, $Y_B = 5^4 \mod 7 = 625 \mod 7 = 2$.

To calculate the shared secret key, party A computes $K_1 = Y_B^{x_A} \mod q$. So, $K_1 = 2^3 \mod 7 = 8 \mod 7 = 1$.

Similarly, party B computes $K_2 = Y_A^{x_B} \mod q$. So, $K_2 = 6^4 \mod 7 = 1296 \mod 7 = 1$.

Since $K_1 = K_2$, the key exchange is successful, and both parties share the same secret key $K = 1$.

The shared secret key $K = 1$ can now be used for secure communication over an insecure channel.

ALGORITHM:

Input:

- A prime number q
- A primitive root α of q

Step 1:

- Party A selects a private key x_A such that $1 < x_A < q$
- Party B selects a private key x_B such that $1 < x_B < q$

Step 2:

- Party A computes public key $Y_A = \alpha^{x_A} \mod q$
- Party B computes public key $Y_B = \alpha^{x_B} \mod q$

Step 3:

- Party A sends Y_A to Party B
- Party B sends Y_B to Party A

Step 4:

- Party A computes shared secret $K_A = Y_B^{x_A} \mod q$
- Party B computes shared secret $K_B = Y_A^{x_B} \mod q$

Step 5:

- If $K_A = K_B$, the key exchange is successful
- Shared secret key $K = K_A = K_B$

Output:

- Both parties share the same secret key K

CODE:

```
from sympy import isprime

def is_primitive_root(alpha, q):
    required_set = set(range(1, q))  # Set of integers from 1 to q-1
    return set(pow(alpha, power, q) for power in range(1, q)) == required_set

def find_primitive_root(q):
    if not isprime(q):
```

```
        raise ValueError(f"{q} is not a prime number, primitive root does not exist.")
    for alpha in range(2, q):
        if is_primitive_root(alpha, q):
            return alpha
    return None

def diffie_hellman(prime_q, primitive_root, private_a, private_b):
    public_a = pow(primitive_root, private_a, prime_q)
    public_b = pow(primitive_root, private_b, prime_q)
    shared_secret_a = pow(public_b, private_a, prime_q)
    shared_secret_b = pow(public_a, private_b, prime_q)
    return public_a, public_b, shared_secret_a, shared_secret_b

q = int(input("Enter the a Large Prime 'q': "))
alpha = find_primitive_root(q)
x_A = int(input("Enter the a Private key of A': "))
x_B = int(input("Enter the a Private key of B': "))

if x_A >= q or x_B >= q:
    raise ValueError(f"Private keys must be less than {q}")
public_a, public_b, shared_secret_a, shared_secret_b = diffie_hellman(q, alpha, x_A, x_B)

print("Alpha: ", alpha)
print("y_A (Public key of A): ", public_a)
print("y_B (Public key of B): ", public_b)
print(f"Shared Secret computed by A: {shared_secret_a}")
print(f"Shared Secret computed by B: {shared_secret_b}")

if shared_secret_a == shared_secret_b:
    print("Key Exchange successful! Shared Secret is the Same.")
else:
    print("Key Exchange Failed!")
```

OUTPUT:

```
Enter the a Large Prime 'q': 7
Enter the a Private key of A': 3
Enter the a Private key of B': 4
Alpha:  3
y_A (Public key of A):  6
y_B (Public key of B):  4
Shared Secret computed by A: 1
Shared Secret computed by B: 1
Key Exchange successful! Shared Secret is the Same.
```

OUTPUT ANALYSIS:

Time Complexity:

1. **Finding Primitive Root:**
   - Outer loop runs O(q), and checking if a number is a primitive root also takes O(q).
   - Overall: O(q^2).
2. **Public Key Calculations:**
   - Each public key and shared secret calculation involves modular exponentiation, which is O(log x).
   - **Overall:** O(log x) for both calculations.

Overall Time Complexity:

- Dominated by finding the primitive root: O(q^2).

Space Complexity:

- Uses constant space for variables and O(q) space for the set in is primitive_root.
- Overall Space Complexity: O(q).

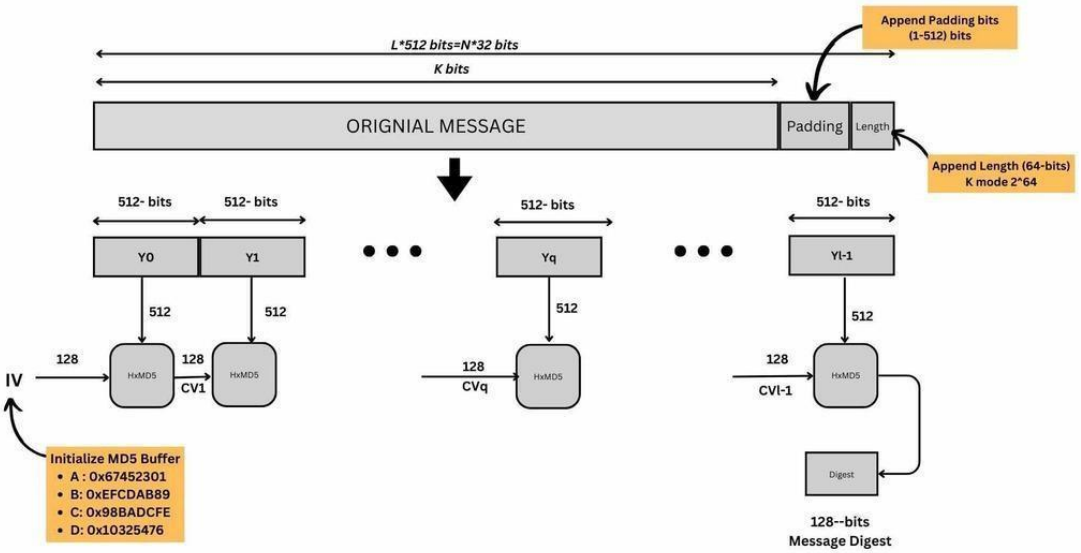
EXPERIMENT – 07

AIM: To Implement MD5 Cryptographic Hash Function

DESCRIPTION:

In Cryptography, the MD5 message-digest algorithm is a cryptographic hash function designed to convert a message into a 128-bit hash value.
MD5 is a cryptographic hash function algorithm that takes the message as input of any length and changes it into a fixed-length message of 16 bytes. MD5 algorithm stands for the **message-digest algorithm**. MD5 was developed in 1991 by **Ronald Rivest** as an improvement of MD4, with advanced security purposes. The output of MD5 (Digest size) is always **128 bits.**

Steps in the MD5 Algorithm



1. **Append Padding Bits**:
   - The original message is padded with a 1 bit followed by 0s until the message length is 64 bits shy of a multiple of 512.
   - For example, if the message is 1000 bits, 472 padding bits are added to reach 1472 bits.
2. **Append Length Bits**:
   - A 64-bit representation of the original message length is appended to the end of the padded message, making the total length a multiple of 512 bits.
3. **Initialize MD Buffer**:
   - MD5 initializes four buffer values, A, B, C, and D, to specific constants.
   - These values are then used to perform operations on 512-bit chunks of the message.
   - Four 32-bit buffers (A, B, C, and D) are initialized with specific constants:

- A = 0x67452301
- B = 0xEDFCBA45
- C = 0x98CBADFE
- D = 0x13DCE476

4. **Process Each 512-bit Block**:
   - The padded message is divided into 512-bit blocks, and each block goes through **64 operations** split into four rounds:
     - **Round 1**: Applies function F using buffers B, C, and D.
     - **Round 2**: Applies function G.
     - **Round 3**: Applies function H.
     - **Round 4**: Applies function I.
   - Each round uses bitwise operations (AND, OR, XOR, NOT) to mix data.
   - Operations involve addition modulo 2322^{32}232, rotating bits, and adding constants to produce non-reversible transformations.

Rounds	Process P
1	(b AND c) OR ((NOT b) AND (d))
2	(b AND d) OR (c AND (NOT d))
3	b OR c OR d
4	c OR (b OR (NOT d))

5. **Final Output**:
   - After processing all blocks, the buffers A, B, C, and D are concatenated to form the final 128-bit MD5 hash, output as a hexadecimal string.

CODE:

```
import hashlib

def generate_md5_hash(inputString):
    hash = hashlib.md5(inputString.encode())
    output = hash.hexdigest()
    return output

inputString = input("Enter a Message to Hash: ")
print("MD5 Hash of the input string:", end="")
print(generate_md5_hash(inputString))
```

OUTPUT:

```
Enter a Message to Hash: I Can Do This All Day
MD5 Hash of the input string:4a88f411b4ac892d7bc6ae35093347be
```

OUTPUT ANALYSIS:

The primary analysis involves checking for the fixed 128-bit output length, observing the avalanche effect, ensuring uniqueness (to a practical extent), and verifying consistency. These analyses confirm that the MD5 algorithm functions as expected, but it's also important to note MD5's limitations in secure applications due to its susceptibility to collision and pre-image attacks. For security-critical uses, consider SHA-256 or other more robust algorithms.

Time Complexity:
The time complexity of the MD5 algorithm is **O(n)**, where **n** is the length of the input message in bits. Here's how it breaks down:

1. **Padding and Length Appending**: Padding the message to make its length a multiple of 512 bits, and appending the 64-bit length of the original message takes **O(n)** time.
2. **Processing Each 512-bit Block**: The message is divided into 512-bit blocks. Each block undergoes 64 operations. Since there are **n/512** blocks, the processing time for this step is:
    - **O(n/512 × 64) = O(n)**

Thus, the overall time complexity of MD5 is **O(n)**. This means MD5 can hash messages quickly, even if they are large.

Space Complexity:
The space complexity of the MD5 algorithm is also **O(n)**, which includes:

1. **Message Storage**: The space needed to store the input message is **O(n)**.
2. **Padding and Length Bits**: Padding can add up to 512 bits (64 bytes) to the message, plus an additional 64 bits (8 bytes) for the length. This contributes negligibly to space complexity but still results in **O(n)** overall.
3. **Internal Buffer Storage**: MD5 uses four 32-bit buffers (A, B, C, D), totaling 128 bits (16 bytes).
4. **Constants and Intermediate Variables**: A few constants and variables are used, but these are of fixed size.

Therefore, the overall space complexity of MD5 is **O(n)**, mainly due to the storage of the input message and any padding data.

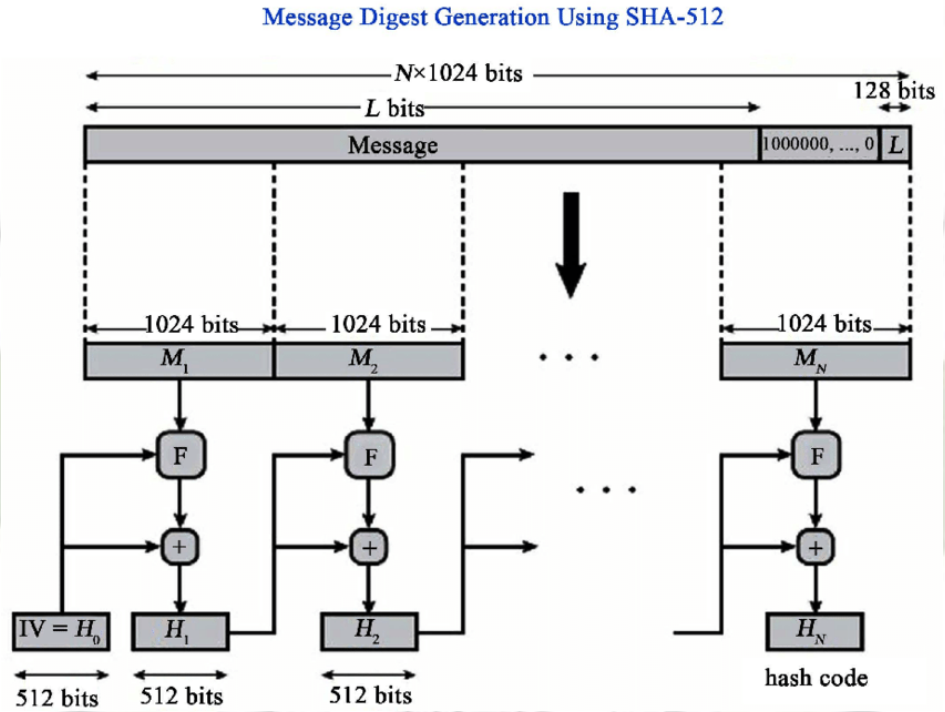
EXPERIMENT – 08

AIM: Implement SHA-512 Cryptographic Hash Function

DESCRIPTION:

SHA-512 is a cryptographic hash function that belongs to the SHA-2 family. It takes an input of any size and produces a fixed-size output, also known as a hash or digest. SHA-512 is widely used in various applications to ensure data integrity and security.
In the digital world, where data security is paramount, SHA-512 plays a crucial role in ensuring the integrity and authenticity of information. By creating a unique hash for any given input, it enables secure verification processes that are fundamental to online transactions, secure communications, and data storage.

Steps in the MD5 Algorithm



Steps in the SHA-512 Algorithm

1. **Append Padding Bits:**
   - The original message is padded with a '1' bit followed by enough '0' bits to make the length congruent to 896 bits modulo 1024. This means that the padded message will be 64 bits shy of a multiple of 1024 bits.
   - For example, if the original message is 1000 bits, 488 padding bits (1 followed by 487 zeros) are added, resulting in a total of 1488 bits.
2. **Append Length Bits:**
   - A 128-bit representation of the original message length (in bits) is appended to the end of the padded message. This ensures that the total length of the message is a multiple of 1024 bits.

3. **Initialize SHA-512 Buffer:**
   - Eight 64-bit buffers (H0 to H7) are initialized with specific constants:
     - H0 = 0x6a09e667f3bcc908
     - H1 = 0xbb67ae8584caa73b
     - H2 = 0x3c6ef372fe94f82b
     - H3 = 0xa54ff53a5f1d36f1
     - H4 = 0x510e527fade682d1
     - H5 = 0x9b05688c2b3e6c1f
     - H6 = 0x1f83d9abfb41bd6b
     - H7 = 0x5be0cd19137e2179
4. **Process Each 1024-bit Block:**
   - The padded message is divided into 1024-bit blocks. Each block is processed through a series of operations:
     - Each 1024-bit block is divided into sixteen 64-bit words.
     - The algorithm performs 80 rounds of computation on these words, divided into 10 rounds with specific mathematical operations.
5. **Perform Operations:**
   - For each round, specific functions (Σ, σ, Ch, and Maj) are applied using the buffer values. The operations involve:
     - **Addition modulo 2642^{64}264**: Each operation uses modulo 2642^{64}264 addition to prevent overflow.
     - **Bitwise Operations**: Operations such as AND, OR, XOR, and NOT are used to mix data and ensure diffusion.
     - **Constant Values**: Each round uses predetermined constant values to further mix the data.
6. **Update Buffer Values:**
   - After processing all rounds for a block, the buffer values (H0 to H7) are updated with the results of the current block's computations.
7. **Final Output:**
   - After processing all blocks, the final values of the buffers H0 to H7 are concatenated to form the final 512-bit SHA-512 hash.
   - The resulting hash is typically output as a hexadecimal string.

CODE:

```
import hashlib

def generate_sha512_hash(inputString):
    hash = hashlib.sha512(inputString.encode())
    output = hash.hexdigest()
    return output

inputString = input("Enter a Message to Hash: ")
print("SHA-512 Hash of the Input String: ", end="")
print(generate_sha512_hash(inputString))
```

OUTPUT:

Enter a Message to Hash: Avengers Assemble
SHA-512 Hash of the Input String:
38ec625415a7a2c4be8fb45e0a0a64fd8d858ca2b05037f0c07e704b1f3def9488fcacbc0735480a0476b3061a2a6e390a1eb2680c20fd4a1fb802547abb0417

OUTPUT ANALYSIS:

The SHA-512 hash function provides a secure and reliable way to represent data uniquely and irreversibly. Understanding its output and characteristics is crucial for its effective application in cybersecurity, data integrity, and cryptographic systems.

- **Time Complexity: O(n)**
  The time complexity is linear with respect to the input size. This means that the time it takes to compute the hash is proportional to the length of the input message.
- **Space Complexity: O(1)**
  The space complexity is constant. SHA-512 uses a fixed amount of memory regardless of the input size. It maintains a fixed-size hash value and uses a constant amount of additional memory for processing.

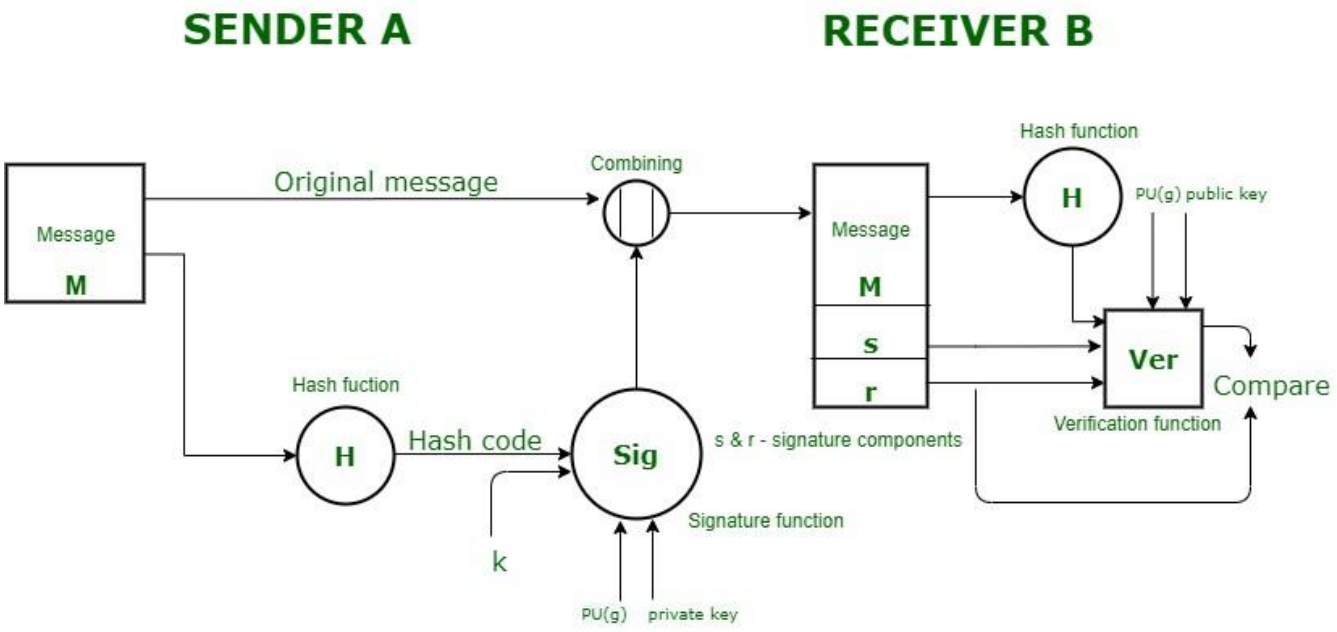
In summary, SHA-512 is efficient in terms of time and memory usage, processing input in linear time while using a constant amount of space.

EXPERIMENT – 09

AIM: Implement Digital Signature Standard (DSS) Algorithm.

DESCRIPTION:

Digital Signature Standard (DSS) is a Federal Information Processing Standard(FIPS) which defines algorithms that are used to generate digital signatures with the help of Secure Hash Algorithm(SHA) for the authentication of electronic documents. DSS only provides us with the digital signature function and not with any encryption or key exchanging strategies.



Sender Side:
In DSS Approach, a hash code is generated out of the message and following inputs are given to the signature function –

1. The hash code.
2. The random number 'k' generated for that particular signature.
3. The private key of the sender i.e., PR(a).
4. A global public key (which is a set of parameters for the communicating principles) i.e., PU(g).

These input to the function will provide us with the output signature containing two components – 's' and 'r'. Therefore, the original message concatenated with the signature is sent to the receiver.

Receiver Side:
At the receiver end, verification of the sender is done. The hash code of the sent message is generated. There is a verification function which takes the following inputs –

1. The hash code generated by the receiver.
2. Signature components 's' and 'r'.
3. Public key of the sender.
4. Global public key.

The output of the verification function is compared with the signature component 'r'. Both the values will match if the sent signature is valid because only the sender with the help of it private key can generate a valid signature.

CODE:

```
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dsa
from cryptography.exceptions import InvalidSignature

private_key = dsa.generate_private_key(key_size=2048)
public_key = private_key.public_key()

def sign_message(msg):
    return private_key.sign(msg, hashes.SHA256())

def verify_signature(msg, sig):
    try:
        public_key.verify(sig, msg, hashes.SHA256())
        return "Signature is valid."
    except InvalidSignature:
        return "Signature is invalid."

message = input("Enter the message to sign: ").encode()
signature = sign_message(message)
print(f"Signature: {signature.hex()}")
print(verify_signature(message, signature))
```

OUTPUT:

```
Enter the message to sign: Fine I will do it myself
Signature:
3046022100d55dca8e497e97bfce17a3bd135f464fb76b634b41485bd4011755c507ecd1d602210
09d75b88870c84793494bd13164ff59e08f7fe9560acde9ca3a34a5d9a323d38b
Signature is valid.
```

OUTPUT ANALYSIS:

Time Complexity

1. **Key Generation**:
   - **Time Complexity**: $O(k^3)$
   - **Explanation**: Generating the DSA key pair (private and public keys) involves complex calculations with prime numbers and modular arithmetic, where k is the key size in bits (e.g., 2048 bits). This process is computationally intensive but typically done only once.
2. **Signature Generation**:
   - **Time Complexity**: $O(k^2)$
   - **Explanation**: Signing a message involves hashing it (commonly with SHA-256) and performing arithmetic with the private key. These operations depend on the key size, k, and involve modular exponentiation and inversion, which generally have O(k^2) complexity.
3. **Signature Verification**:
   - **Time Complexity**: $O(k^2)$
   - **Explanation**: Verifying a DSA signature requires modular exponentiation with the public key, similar to the signing process. Verification is generally in O(k^2), though it may be slightly faster than signing due to simpler calculations involved.

Space Complexity

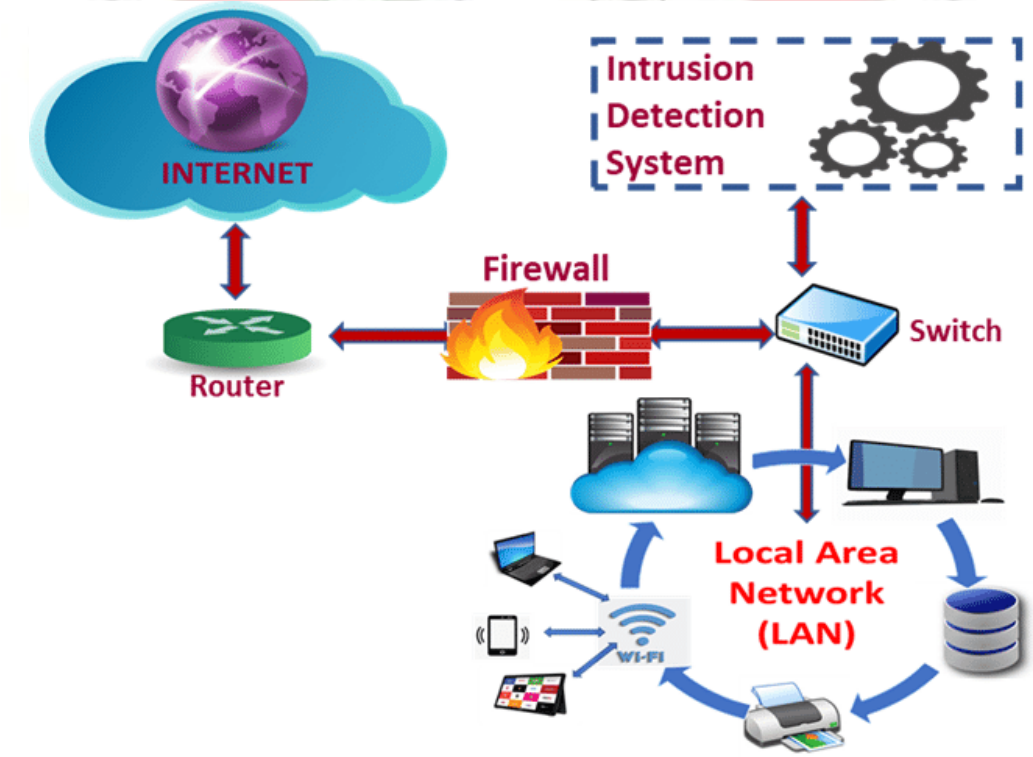
1. **Key Storage**:
   - **Space Complexity**: O(k)
   - **Explanation**: The private and public keys require O(k) space, where k is the key size in bits (e.g., 2048 bits). This space is typically small, given current memory capacities.
2. **Signature Storage**:
   - **Space Complexity**: O(k)
   - **Explanation**: A DSA signature consists of two values (r and s), together occupying O(k) space. For a 2048-bit key, the signature requires about 512 bytes of storage.
3. **Message Hashing**:
   - **Space Complexity**: O(1)
   - **Explanation**: Regardless of the message length, the hash output size is fixed (for example, 256 bits or 32 bytes with SHA-256), so the space complexity for storing the hash is constant, or O(1).

EXPERIMENT – 10

AIM: Demonstrate Intrusion Detection System (ids) using any tool eg. Snort or any other s/w.

DESCRIPTION:

An **Intrusion Detection System (IDS)** is a security tool that monitors network traffic and system activities for malicious actions or policy violations. One popular open-source IDS tool is Snort, which works by analyzing network packets in real time and comparing them against predefined rules to detect potential intrusions. To demonstrate Snort, first, install it on a Linux-based system such as Ubuntu. You need to configure the network interface in promiscuous mode to capture all network traffic. Next, create or modify Snort rules that define specific patterns to detect threats like port scanning, denial-of-service (DoS) attacks, or malware traffic. Snort operates in different modes: sniffer mode (monitoring traffic), packet logging mode (logging network packets), and network intrusion detection mode (real-time monitoring and alerting based on rules). Once Snort is running, it examines every packet traversing the network, looking for suspicious activity based on the rules. If a match is found, Snort triggers an alert, which is logged for further investigation. This allows network administrators to detect, analyze, and respond to potential security breaches, making IDS a critical component of network defence. The logs can also be integrated with other security systems for comprehensive analysis and reporting.



Snort Setup and ICMP Ping Detection

Prerequisites:

Ensure Snort is installed on your Linux system. If it's not, you can install it by running:

```
sudo apt-get update
sudo apt-get install snort
```

Step 1: Configure Network Interface in Promiscuous Mode

To capture all network traffic, set your network interface to promiscuous mode:

```
sudo ip link set eth0 promisc on
```

Replace eth0 with the correct network interface name (use ip a to find the interface if needed).

Step 2: Create a Basic Snort Rule

You'll create a rule to detect ICMP ping requests, which can be indicative of a ping sweep.

1. Open the Snort rules file, typically found at:
   ```
   sudo nano /etc/snort/rules/local.rules
   ```
2. Add this rule to detect ICMP ping requests:
   ```
   alert icmp any any -> any any (msg:"ICMP Ping detected"; itype:8; sid:1000001; rev:1;)
   ```

Explanation:

- alert: Triggers an alert action when this rule matches.
- icmp: Sets the rule to inspect ICMP traffic.
- any any -> any any: Matches traffic from any IP/port to any IP/port.
- msg:"ICMP Ping detected": Custom message displayed when the rule matches.
- itype:8: Filters ICMP packets for echo requests (ping requests).
- sid:1000001: Assigns a unique ID to this rule.
- rev:1: Specifies the rule version.

After editing, save the file and exit the editor.

Step 3: Run Snort in Network Intrusion Detection System (NIDS) Mode

To activate Snort and monitor for alerts in the console:

```
sudo snort -A console -q -c /etc/snort/snort.conf -i eth0
```

Explanation of Options:

- -A console: Outputs alerts directly to the console.
- -q: Runs Snort in quiet mode, reducing non-essential output.
- -c: Specifies the Snort configuration file to use.
- -i eth0: Sets the network interface Snort will monitor (replace eth0 if needed).

Step 4: Test the IDS

Initiate an ICMP ping request from a different device to test the rule:

```
ping <target-ip>
```

Replace <target-ip> with the IP of the machine running Snort.

Example Output:

If the rule works, Snort will output an alert similar to:

```
[**] [1:1000001:1] ICMP Ping detected [**]
```

This indicates that Snort successfully detected and alerted on the ICMP ping request based on the rule.

OUTPUT ANALYSIS:

In the output, Snort logs an alert when the ICMP Ping rule is triggered. The message [**] [1:1000001:1] ICMP Ping detected [**] indicates that a network packet matching the rule was detected. The numbers [1:1000001:1] represent the Generator ID, Signature ID(sid), and Revision ID, respectively. These help in uniquely identifying the rule and tracking its version. Snort captures the packet metadata, including the source and destination IP addresses, time, and protocol details. This output allows network administrators to monitor suspicious traffic and take appropriate security measures based on the alerts generated.