Lab#1 Implement Caesar cipher encryption-decryption. Perform in PYTHON as well as in Virtual lab.

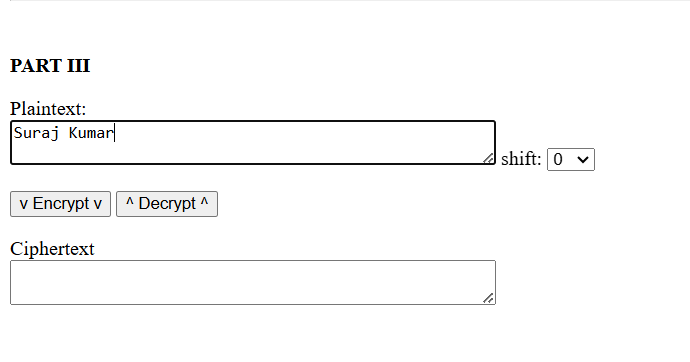
**Virtual Lab Simulator:**

https://virtual-labs.github.io/exp-digital-signatures-iiith/simulation.html

**Step 0:**



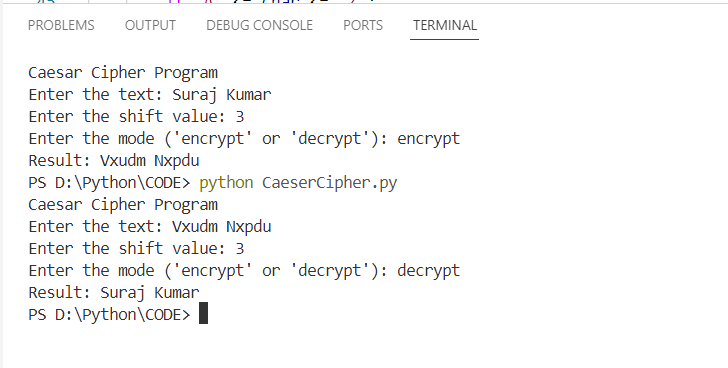
**Step 1: Enter the Plain Text.**



**Step 2: Generate the Hash Output for SHA - 1.**



**Outcome: -**



**Code Snippet:**

# Caesar Cipher Implementation in Python with User Input and Switch-Case

def caesar\_cipher(text, shift, mode):

    # Normalize shift to handle negative values or values greater than 26

    shift = shift % 26

    if shift < 0:

        shift += 26

    def shift\_char(char, shift\_amount):

        if 'A' <= char <= 'Z':

            return chr(((ord(char) - 65 + shift\_amount) % 26) + 65)

        elif 'a' <= char <= 'z':

            return chr(((ord(char) - 97 + shift\_amount) % 26) + 97)

        else:

            return char  # Non-alphabetical characters remain unchanged

    match mode:  # Using Python's match-case for switch-case behavior

        case 'encrypt':

            return ''.join(shift\_char(char, shift) for char in text)

        case 'decrypt':

            return ''.join(shift\_char(char, -shift) for char in text)

        case \_:

            raise ValueError("Invalid mode. Use 'encrypt' or 'decrypt'.")

# Main program for user input

if \_\_name\_\_ == "\_\_main\_\_":

    print("Caesar Cipher Program")

    text = input("Enter the text: ")

    try:

        shift = int(input("Enter the shift value: "))

    except ValueError:

        print("Shift must be an integer.")

        exit()

    mode = input("Enter the mode ('encrypt' or 'decrypt'): ").strip().lower()

    try:

        result = caesar\_cipher(text, shift, mode)

        print(f"Result: {result}")

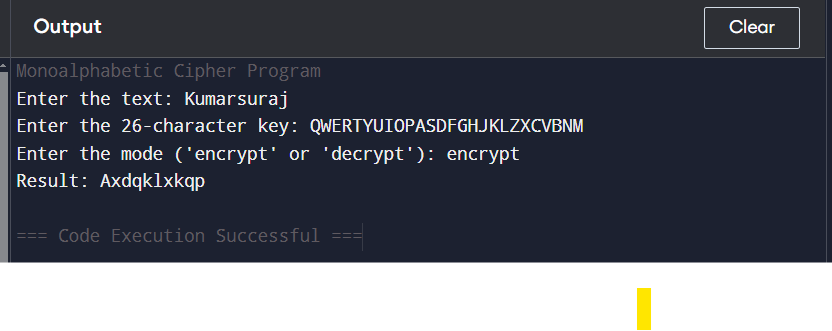
    except ValueError as e:

        print(e)

**Lab#2 Implement Monoalphabetic cipher encryption-decryption.**

Perform in PYTHON.

**Outcome: -**



# Monoalphabetic Cipher Implementation in Python with Switch-Case

def monoalphabetic\_cipher(text, key, mode):

# Define the alphabet

alphabet = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

key = key.upper()

if len(set(key)) != 26 or not key.isalpha():

raise ValueError("Key must be a 26-character unique alphabet string.")

def encrypt\_char(char):

if char.isalpha():

idx = alphabet.index(char.upper())

return key[idx] if char.isupper() else key[idx].lower()

else:

return char

def decrypt\_char(char):

if char.isalpha():

idx = key.index(char.upper())

return alphabet[idx] if char.isupper() else alphabet[idx].lower()

else:

return char

match mode: # Using Python's match-case for switch-case behavior

case 'encrypt':

return ''.join(encrypt\_char(char) for char in text)

case 'decrypt':

return ''.join(decrypt\_char(char) for char in text)

case \_:

raise ValueError("Invalid mode. Use 'encrypt' or 'decrypt'.")

# Main program for user input

if \_\_name\_\_ == "\_\_main\_\_":

print("Monoalphabetic Cipher Program")

text = input("Enter the text: ")

key = input("Enter the 26-character key: ").strip()

mode = input("Enter the mode ('encrypt' or 'decrypt'): ").strip().lower()

try:

result = monoalphabetic\_cipher(text, key, mode)

print(f"Result: {result}")

except ValueError as e:

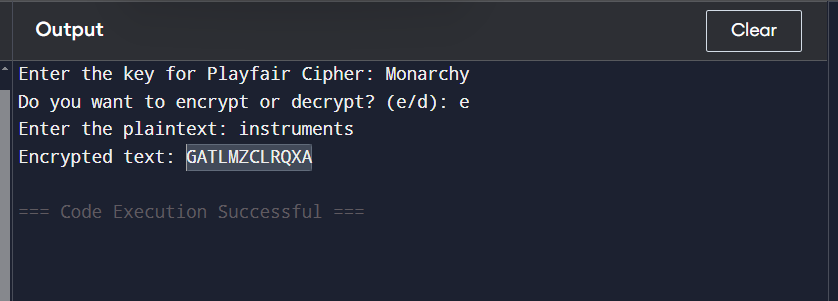
print(e)

**Lab#3 Implement Playfair cipher encryption-decryption.**

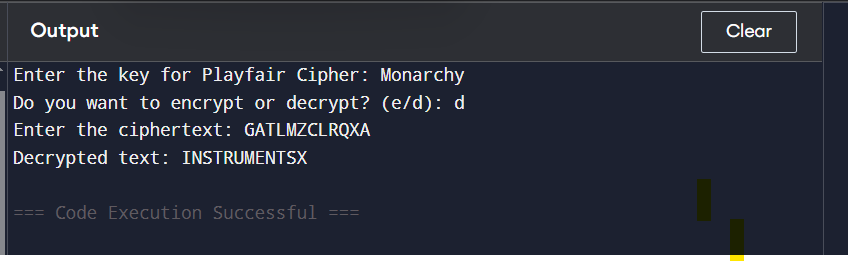
Perform in PYTHON.

**Outcome: -**

ENCRYPTION: -



DECRPTION: -



**CODE: -**

def create\_matrix(key):

    # Remove duplicates and replace J with I

    key = key.upper().replace('J', 'I')

    key = "".join(sorted(set(key), key=key.index))

    # Create the matrix

    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"

    matrix = [c for c in key if c in alphabet] + [c for c in alphabet if c not in key]

    return [matrix[i:i+5] for i in range(0, 25, 5)]

def find\_position(matrix, char):

    for row\_idx, row in enumerate(matrix):

        if char in row:

            return row\_idx, row.index(char)

    return None

def prepare\_text(text, for\_encryption=True):

    text = text.upper().replace("J", "I")

    result = ""

    i = 0

    while i < len(text):

        a = text[i]

        b = text[i+1] if i + 1 < len(text) else ""

        if for\_encryption and a == b:

            result += a + "X"

            i += 1

        elif b == "":

            result += a + "X"

            i += 1

        else:

            result += a + b

            i += 2

    return result

def encrypt\_pair(pair, matrix):

    r1, c1 = find\_position(matrix, pair[0])

    r2, c2 = find\_position(matrix, pair[1])

    if r1 == r2:  # Same row

        return matrix[r1][(c1 + 1) % 5] + matrix[r2][(c2 + 1) % 5]

    elif c1 == c2:  # Same column

        return matrix[(r1 + 1) % 5][c1] + matrix[(r2 + 1) % 5][c2]

    else:  # Rectangle

        return matrix[r1][c2] + matrix[r2][c1]

def decrypt\_pair(pair, matrix):

    r1, c1 = find\_position(matrix, pair[0])

    r2, c2 = find\_position(matrix, pair[1])

    if r1 == r2:  # Same row

        return matrix[r1][(c1 - 1) % 5] + matrix[r2][(c2 - 1) % 5]

    elif c1 == c2:  # Same column

        return matrix[(r1 - 1) % 5][c1] + matrix[(r2 - 1) % 5][c2]

    else:  # Rectangle

        return matrix[r1][c2] + matrix[r2][c1]

def playfair\_encrypt(plaintext, key):

    matrix = create\_matrix(key)

    plaintext = prepare\_text(plaintext, for\_encryption=True)

    ciphertext = ""

    for i in range(0, len(plaintext), 2):

        ciphertext += encrypt\_pair(plaintext[i:i+2], matrix)

    return ciphertext

def playfair\_decrypt(ciphertext, key):

    matrix = create\_matrix(key)

    plaintext = ""

    for i in range(0, len(ciphertext), 2):

        plaintext += decrypt\_pair(ciphertext[i:i+2], matrix)

    return plaintext

if \_\_name\_\_ == "\_\_main\_\_":

    key = input("Enter the key for Playfair Cipher: ")

    choice = input("Do you want to encrypt or decrypt? (e/d): ").lower()

    if choice == "e":

        plaintext = input("Enter the plaintext: ")

        ciphertext = playfair\_encrypt(plaintext, key)

        print(f"Encrypted text: {ciphertext}")

    elif choice == "d":

        ciphertext = input("Enter the ciphertext: ")

        plaintext = playfair\_decrypt(ciphertext, key)

        print(f"Decrypted text: {plaintext}")

    else:

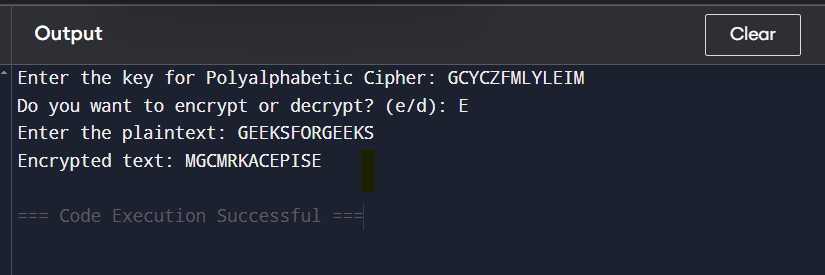
        print("Invalid choice!")

**Lab#4 Implement Polyalphabetic cipher encryption-decryption.**

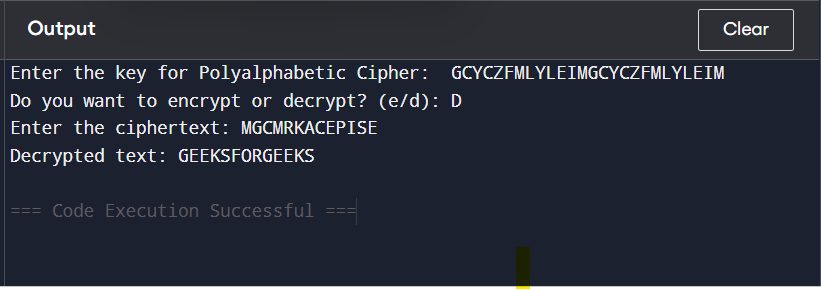
Perform in PYTHON.

**Outcome: -**

**ENCRYPTION: -**



**DECRYPTION: -**



**CODE:** -

def generate\_vigenere\_table():

table = []

for i in range(26):

row = [chr((i + j) % 26 + 65) for j in range(26)]

table.append(row)

return table

def vigenere\_encrypt(plaintext, key):

table = generate\_vigenere\_table()

plaintext = plaintext.upper()

key = key.upper()

ciphertext = ""

key\_index = 0

for char in plaintext:

if char.isalpha():

row = ord(key[key\_index]) - 65

col = ord(char) - 65

ciphertext += table[row][col]

key\_index = (key\_index + 1) % len(key)

else:

ciphertext += char

return ciphertext

def vigenere\_decrypt(ciphertext, key):

table = generate\_vigenere\_table()

ciphertext = ciphertext.upper()

key = key.upper()

plaintext = ""

key\_index = 0

for char in ciphertext:

if char.isalpha():

row = ord(key[key\_index]) - 65

col = table[row].index(char)

plaintext += chr(col + 65)

key\_index = (key\_index + 1) % len(key)

else:

plaintext += char

return plaintext

if \_\_name\_\_ == "\_\_main\_\_":

key = input("Enter the key for Polyalphabetic Cipher: ")

choice = input("Do you want to encrypt or decrypt? (e/d): ").lower()

if choice == "e":

plaintext = input("Enter the plaintext: ")

ciphertext = vigenere\_encrypt(plaintext, key)

print(f"Encrypted text: {ciphertext}")

elif choice == "d":

ciphertext = input("Enter the ciphertext: ")

plaintext = vigenere\_decrypt(ciphertext, key)

print(f"Decrypted text: {plaintext}")

else:

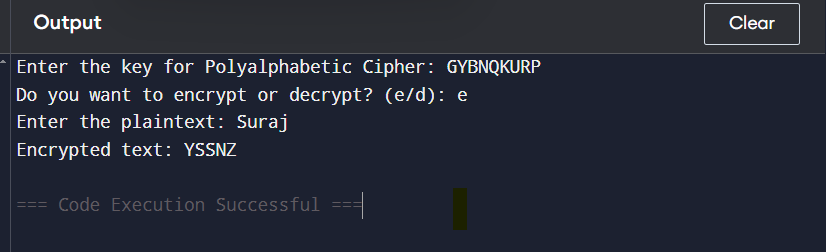
print("Invalid choice!")

**Lab#5 Implement Hill cipher encryption-decryption.**

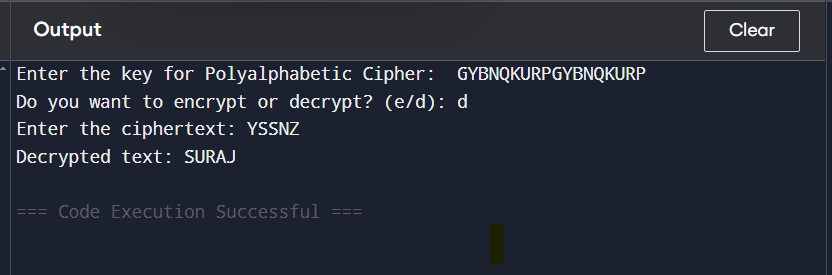
Perform in PYTHON.

**Outcome: -**

**ENCRYPTION: -**



**DECRYPTION: -**



**CODE:** -

def generate\_vigenere\_table():

table = []

for i in range(26):

row = [chr((i + j) % 26 + 65) for j in range(26)]

table.append(row)

return table

def vigenere\_encrypt(plaintext, key):

table = generate\_vigenere\_table()

plaintext = plaintext.upper()

key = key.upper()

ciphertext = ""

key\_index = 0

for char in plaintext:

if char.isalpha():

row = ord(key[key\_index]) - 65

col = ord(char) - 65

ciphertext += table[row][col]

key\_index = (key\_index + 1) % len(key)

else:

ciphertext += char

return ciphertext

def vigenere\_decrypt(ciphertext, key):

table = generate\_vigenere\_table()

ciphertext = ciphertext.upper()

key = key.upper()

plaintext = ""

key\_index = 0

for char in ciphertext:

if char.isalpha():

row = ord(key[key\_index]) - 65

col = table[row].index(char)

plaintext += chr(col + 65)

key\_index = (key\_index + 1) % len(key)

else:

plaintext += char

return plaintext

if \_\_name\_\_ == "\_\_main\_\_":

key = input("Enter the key for Polyalphabetic Cipher: ")

choice = input("Do you want to encrypt or decrypt? (e/d): ").lower()

if choice == "e":

plaintext = input("Enter the plaintext: ")

ciphertext = vigenere\_encrypt(plaintext, key)

print(f"Encrypted text: {ciphertext}")

elif choice == "d":

ciphertext = input("Enter the ciphertext: ")

plaintext = vigenere\_decrypt(ciphertext, key)

print(f"Decrypted text: {plaintext}")

else:

print("Invalid choice!")

**Lab#6 Case Study on: Simple DES prepare report**

1. **Introduction:**

The Data Encryption Standard (DES) has played a pivotal role in the history of symmetric-key cryptography. However, due to its complexity and security vulnerabilities, a simplified version—Simple DES (S-DES)—has been developed to help students and researchers understand the fundamentals of encryption without the overhead of industrial-grade systems. S-DES operates on 8-bit plaintext blocks using a 10-bit key, incorporating key generation, permutations, substitutions, and Feistel rounds. This lab report provides an in-depth analysis of S-DES, covering its design, operation, and educational benefits.

1. **Objectives:**

• Understand the DES Structure: Analyses the structure of DES and its simplified version, S-DES.

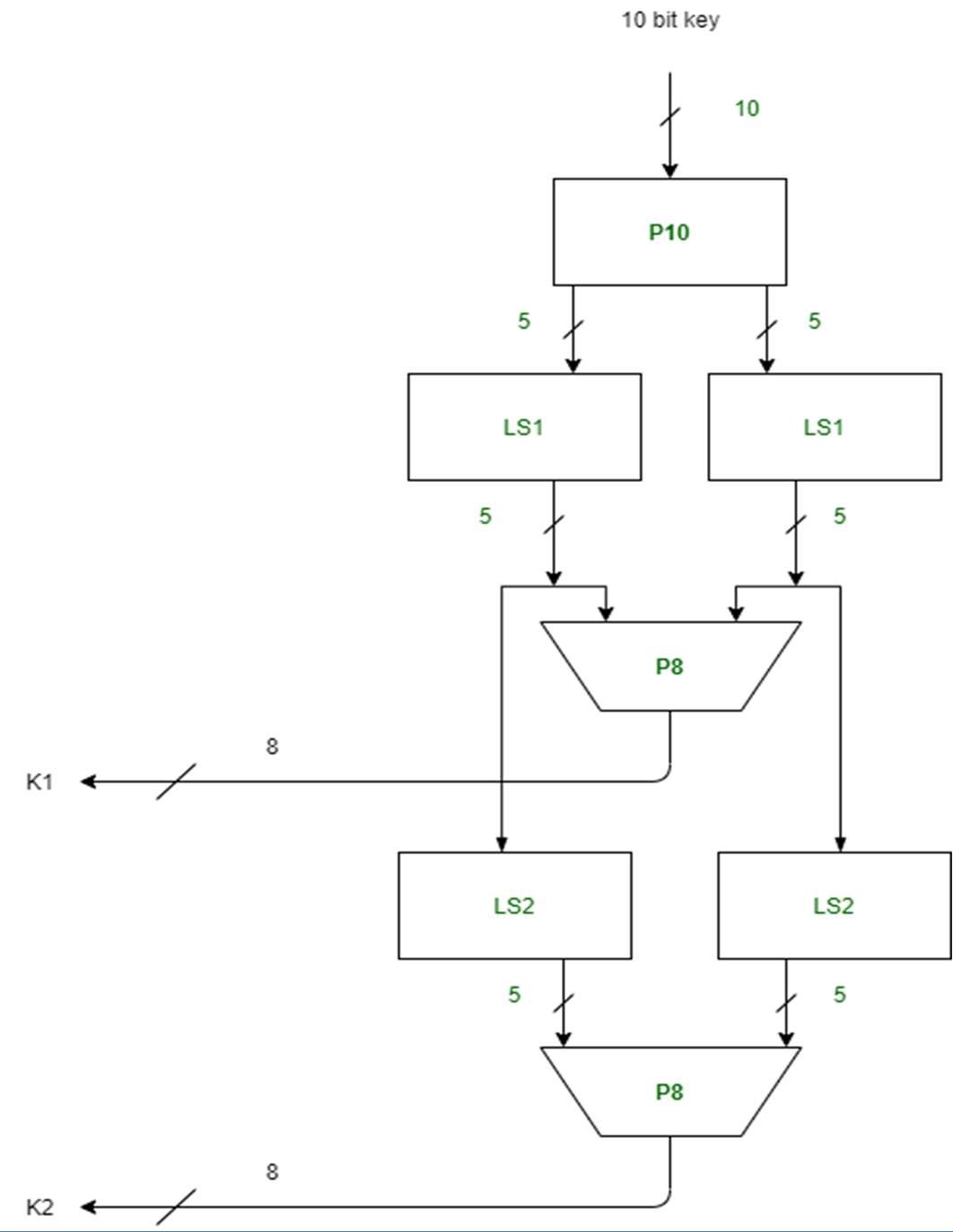
• Implement S-DES: Develop the key generation, encryption, and decryption routines.

• Examine Cryptographic Processes: Explore how permutations, substitutions, and XOR operations secure data.

• Validate Reversibility: Ensure that the encryption process can be reversed to recover the original plaintext.

• Discuss Educational Implications: Evaluate the benefits and limitations of S-DES as an instructional model.

1. **3. Methodology**
2. **3.1 Diagram of the S-DES Process**



**3.2OverView of DES:**

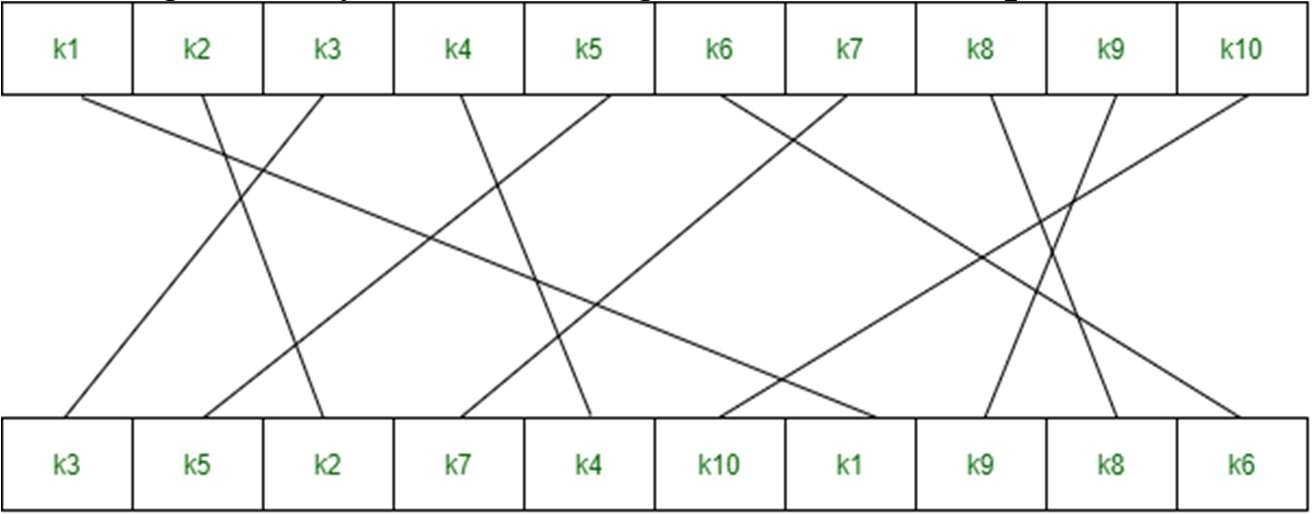
Simple DES (S-DES) is a compact block cipher that encrypts 8-bit plaintext using a 10-bit key. Its structure mirrors that of DES but with reduced complexity:

* + - Initial Permutation (IP): Rearranges the bits of the plaintext.
    - Feistel Rounds: Two rounds of processing using subkeys (K₁ and K₂) derived from the main key.
    - Switching Function: A swap operation between rounds that interleaves the data halves.
    - Inverse Initial Permutation (IP⁻¹): Reverses the initial permutation to produce the ciphertext.
  1. **Key Generation:**

The 10-bit key is transformed into two 8-bit subkeys (K₁ and K₂) through the following steps:

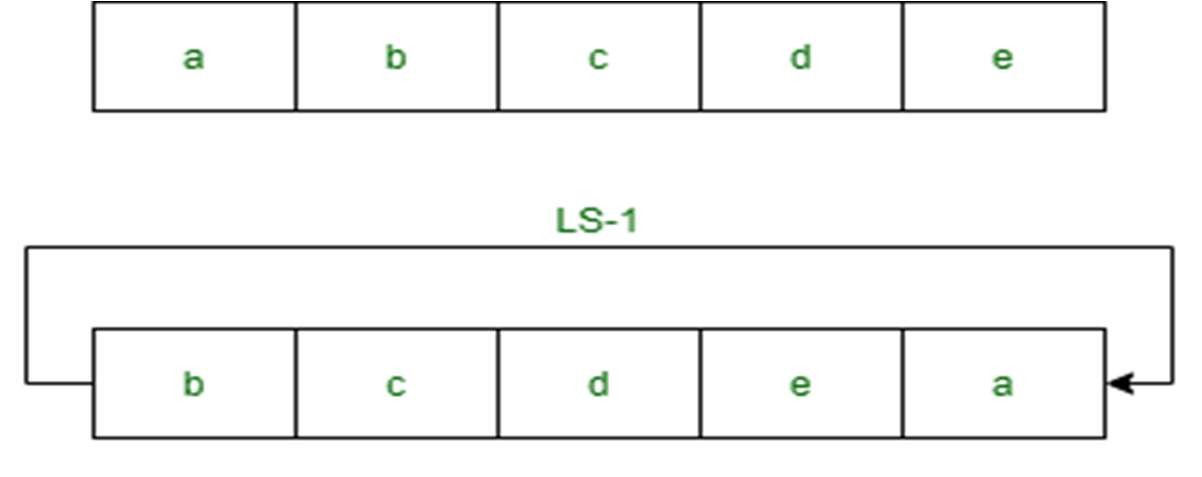
* + 1. **Permutation P10:**

Rearrange the key bits according to a fixed 10-bit permutation table.



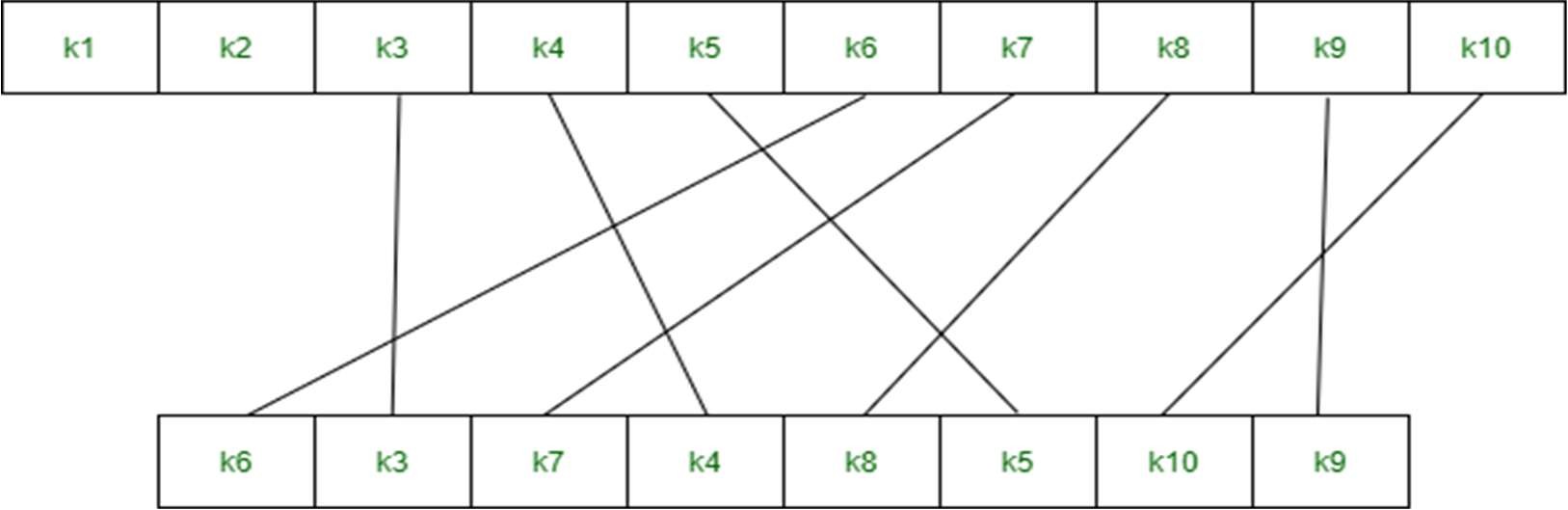
1. **Left Shift (LS-1):**

Split the permuted key into two 5-bit halves and perform a circular left shift by one position on each half.



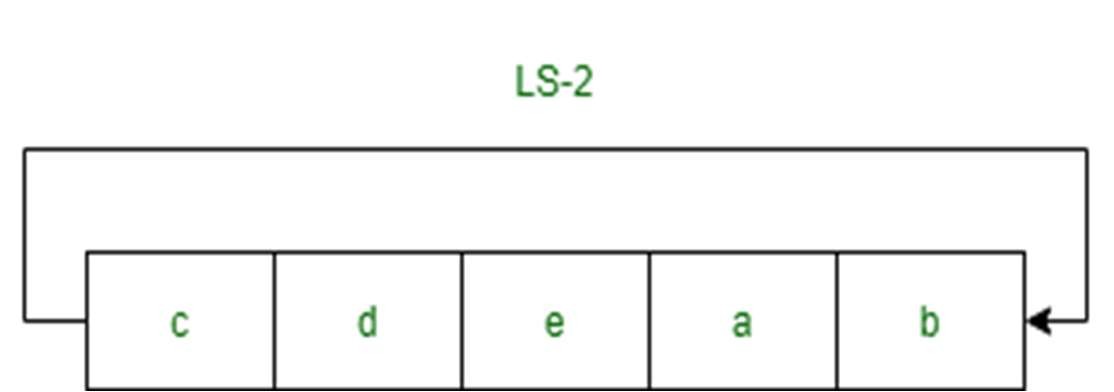
1. **Permutation P8 for K₁:**

Combine the shifted halves and permute them using a fixed 8-bit table to generate subkey K₁.



1. **Left Shift (LS-2):**

Shift each 5-bit half two positions to the left.



1. **Permutation P8 for K₂:**

Combine the shifted halves and apply the P8 permutation again to generate subkey K₂.

3.**4 The Encryption Process:**

The encryption process in S-DES consists of the following steps:

**1. Initial Permutation (IP):** The 8-bit plaintext is permuted using a predefined table.

**2. Splitting:** The permuted text is divided into two 4-bit halves, labelled L (left) and R (right).

**3. Round 1 (Using K₁):**

**Detailed Feistel Function Sub-steps:**

* **Divide the plaintext:** Split into L and R.
* **Expansion Permutation (EP):** Expand the 4-bit R to a 6-bit value by rearranging and duplicating certain bits.
* **Key Mixing:** OR the expanded R with subkey K₁.
* **Substitution using S-Boxes:**
* Divide the 6-bit result into two parts.
* Input the left part to S-Box S1 and the right part to S-Box S2.
* Each S-Box outputs 2 bits.
* **Combine and Permutation P4:**
* Combine the two 2-bit outputs to form a 4-bit value.
* Permute the 4-bit result using the P4 permutation table.
* **Final XOR:** XOR the P4 output with the left half L.
* **Swap:** Exchange the two halves to prepare for the next round.

**4. Round 2 (Using K₂):** Repeat the Feistel function using subkey K₂ on the new right half. In this

round, the final swap is omitted.

5. **Inverse Permutation (IP⁻¹):** The final combination of the two halves is passed through the nverse

initial permutation to produce the ciphertext.

**3.5 The Round Function fK:**

The round function fK plays a central role in the encryption process:

* Expansion/Permutation (EP):

Expand the 4-bit input (from R) to 6 bits.

* Key Mixing: XOR the expanded bits with the corresponding subkey.
* Substitution using S-Boxes: Split the result and process each half through its respective S-Box (S1 and S2) to obtain 2-bit outputs.
* Permutation P4: Combine and rearrange the outputs into a 4-bit value.
* Final XOR: XOR the P4 result with the left half of the data block.

3.6 **The Decryption Process**

Decryption mirrors the encryption process, with subkeys applied in reverse order:

1. Apply the initial permutation (IP) to the ciphertext.

2. Execute Round 1 using subkey K₂.

3. Swap the two halves.

4. Execute Round 2 using subkey K₁.

5. Apply the inverse initial permutation (IP⁻¹) to recover the original plaintext.

4**. Implementation**

**Pseudo-code Overview:**

function SDES\_Encrypt(plaintext, key):

K1, K2 = generateSubkeys(key)

IP\_text = initialPermutation(plaintext)

(L, R) = split(IP\_text)

// Round 1 using subkey K1

temp = fK(R, K1)

L\_new = L XOR temp

// Swap halves for round 2

(L, R) = (R, L\_new)

// Round 2 using subkey K2

temp = fK(R, K2)

L\_final = L XOR temp

// Combine halves and apply inverse IP

preoutput = combine(L\_final, R)

ciphertext = inverseInitialPermutation(preoutput)

return ciphertext

function fK(R, subkey):

expanded\_R = expansionPermutation(R)

xor\_result = expanded\_R XOR subkey

(left\_part, right\_part) = split(xor\_result)

s0\_output = S0(left\_part)

s1\_output = S1(right\_part)

combined = combine(s0\_output, s1\_output)

p4\_output = permutationP4(combined)

return p4\_output

Note: The decryption process follows the same steps as encryption, with the subkeys applied in reverse order (K₂ first, then

K₁).

**5. Test And Outcome**

**5.1 Test Case**

A sample test case was executed with the following parameters:

* Key: 1 0 1 0 0 0 0 0 1 0
* Plaintext: 11010111

**Step 1:** We accepted a 10-bit key and permuted the bits by putting them in the P10 table.

Key = 1 0 1 0 0 0 0 0 1 0

(k1, k2, k3, k4, k5, k6, k7, k8, k9, k10) = (1, 0, 1, 0, 0, 0, 0, 0, 1, 0)

P10 Permutation is: P10(k1, k2, k3, k4, k5, k6, k7, k8, k9, k10) = (k3, k5, k2, k7, k4, k10, k1, k9,

k8, k6)

After P10, we get 1 0 0 0 0 0 1 1 0 0

**Step 2:** We divide the key into 2 halves of 5-bit each.

l=1 0 0 0 0, r=0 1 1 0 0

**Step 3:** Now we apply one bit left-shift on each key.

l = 0 0 0 0 1, r = 1 1 0 0 0

**Step 4:** Combine both keys after step 3 and permute the bits by putting them in the P8 table. The

output of the given table is the first key K1.

After LS-1 combined, we get 0 0 0 0 1 1 1 0 0 0

P8 permutation is: P8(k1, k2, k3, k4, k5, k6, k7, k8, k9, k10) = (k6, k3, k7, k4, k8, k5, k10, k9)

After P8, we get Key-1: 1 0 1 0 0 1 0 0

**Step 5:** The output obtained from step 3 i.e. 2 halves after one-bit left shift should again undergo

the process of two-bit left shift.

Step 3 output - l = 0 0 0 0 1, r = 1 1 0 0 0

After two-bit shift - l = 0 0 1 0 0, r = 0 0 0 1 1

**Step 6:** Combine the 2 halves obtained from step 5 and permute them by putting them in the P8

table. The output of the given table is the second key K2.

After LS-2 combined = 0 0 1 0 0 0 0 0 1 1

P8 permutation is: P8(k1, k2, k3, k4, k5, k6, k7, k8, k9, k10) = (k6, k3, k7, k4, k8, k5, k10, k9)

After P8, we get Key-2: 0 1 0 0 0 0 1 1

**Final Outcome:**

Key-1 is: 1 0 1 0 0 1 0 0

Key-2 is: 0 1 0 0 0 0 1 1

The S-DES encryption algorithm produced a ciphertext which, when processed through the decryption routine (using the subkeys in reverse order), successfully recovered the original plaintext. This confirms the correctness and reversibility of the encryption/decryption process.

6. **Applications and Limitations**

**Applications:**

* Used in cryptographic education to understand DES principles.
* Serves as a basic model for more complex encryption schemes.

**Limitations:**

* Small key size (10-bit) makes it highly vulnerable to brute-force attacks.
* Not suitable for real-world encryption due to weak security.

**7. Conclusion**

This lab report provided a detailed case study of Simple DES (S-DES), demonstrating how basic

cryptographic operations are integrated to form a functional block cipher. Through the implementation of key generation, encryption, and decryption processes, we confirmed that S-DES reliably recovers plaintext from ciphertext, emphasizing the reversibility inherent in the Feistel structure. Although S-DES is not suitable for modern security needs, its simplicity makes it an invaluable educational tool for understanding the principles of symmetric-key cryptography.

8**. References**

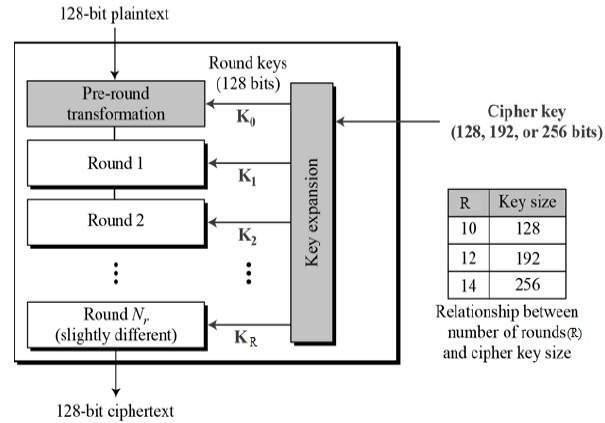
* Stallings, W. Cryptography and Network Security: Principles and Practice.
* Paar, C., & Pelzl, J. Understanding Cryptography: A Textbook for Students and Practitioners.
* Online resources and lecture notes on Simple DES (S-DES) implementations and tutorials.

**Lab#7 Case study on: Simple AES prepare report.**

**Explanation:**

The **Advanced Encryption Standard (AES)** is a **symmetric-key block cipher** adopted by the **U.S. National Institute of Standards and Technology (NIST)** in 2001 as a replacement for DES. It is based on the **Rijndael** algorithm and is widely used for secure data encryption.

**Diagram:**



**Working:**

AES operates on **fixed-size blocks of 128 bits** and supports key sizes of **128, 192, or 256 bits**. Unlike DES, AES does not use a Feistel structure but follows a **substitution-permutation network (SPN)** for encryption.

**1. Key Expansion**

* The input **key (128, 192, or 256 bits)** is expanded into multiple **round keys** using a process called the **Key Schedule**.
* The number of rounds depends on the key size:

o **10 rounds** for **128-bit key**

o **12 rounds** for **192-bit key**

o **14 rounds** for **256-bit key**

**2. Initial Round**

 The **plaintext (128-bit block)** undergoes an **AddRoundKey step**, where it is XORed with the first-round key.

**3. Main Rounds (9, 11, or 13 rounds depending on key size)**

Each round consists of the following steps:

1. **SubBytes (Byte Substitution)** o Each byte is replaced using a **Substitution Box (S-Box)**, adding confusion.
2. **ShiftRows** o Rows of the state matrix are **shifted cyclically** to the left, ensuring diffusion.
3. **MixColumns** o A mathematical transformation is applied to columns, further mixing the data.
4. **AddRoundKey** o The **current state** is XORed with a **round key** generated from the key schedule.

**4. Final Round**

The **final round** omits the **MixColumns step** and consists of:

* **SubBytes**
* **ShiftRows**
* **AddRound Key**
* The final **128-bit ciphertext** is generated.

**Advantages of DES:**

* **Highly Secure** – AES is resistant to known attacks like **brute-force, differential cryptanalysis, and linear cryptanalysis**.
* **Larger Key Size** – Supports **128, 192, and 256-bit** keys, making it far more secure than DES.
* **Efficient and Fast** – Performs well in both hardware and software implementations.
* **Wide Adoption** – Used in **banking, government, cloud security, and wireless encryption (WPA2, TLS, SSL, etc.)**.
* **Flexible and Scalable** – Works with different key lengths based on security requirements.

**Disadvantages of DES:**

* **Computational Complexity** – More complex than DES, making it **slightly slower** in low-power devices.
* **Vulnerable to Side-Channel Attacks** – If implemented poorly, AES can be attacked using **power analysis or timing attacks**.
* **Key Management Overhead** – Secure storage and exchange of AES keys can be a challenge.

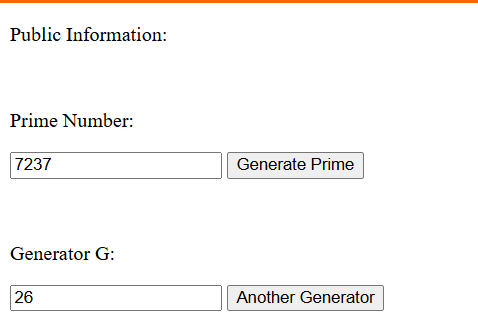
**Practical 8: Implement Diffi-Hellmen Key exchange Method.**

Perform in PYTHON as well as in Virtual lab.

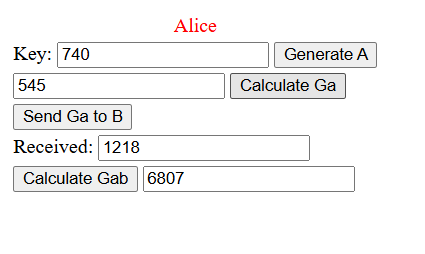
Link for Virtual Lab: - https://cse29-iiith.vlabs.ac.in/exp/diffie-hellman/simulation.html

**Virtual Lab Simulator:**

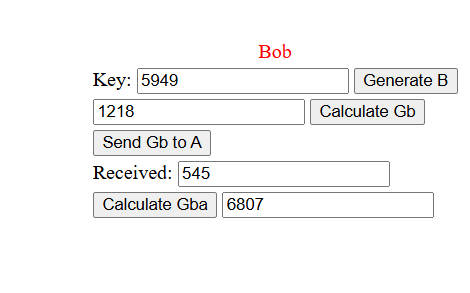
**Step 1: Generate Prime.**



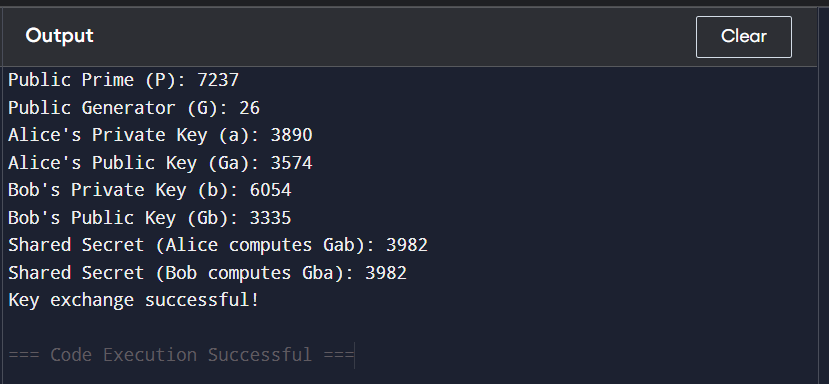
**Step 2: Sender Side.**



**Step 3: Receiver Side**



**OUTPUT:**



**CODE:**

import random

def mod\_exp(base, exponent, mod):

"""Efficiently computes (base^exponent) % mod using modular exponentiation."""

result = 1

base = base % mod # Ensure base is within mod range

while exponent > 0:

if exponent % 2 == 1:

result = (result \* base) % mod

exponent //= 2

base = (base \* base) % mod

return result

# Public parameters (from the images)

P = 7237 # Prime number

G = 26 # Generator

# Alice's private key

a = random.randint(2, P-2) # Randomly chosen private key

Ga = mod\_exp(G, a, P) # Compute public key

# Bob's private key

b = random.randint(2, P-2) # Randomly chosen private key

Gb = mod\_exp(G, b, P) # Compute public key

# Exchange Ga and Gb (simulated here)

received\_by\_bob = Ga

received\_by\_alice = Gb

# Compute shared secret

shared\_secret\_alice = mod\_exp(received\_by\_alice, a, P)

shared\_secret\_bob = mod\_exp(received\_by\_bob, b, P)

# Output results

print("Public Prime (P):", P)

print("Public Generator (G):", G)

print("Alice's Private Key (a):", a)

print("Alice's Public Key (Ga):", Ga)

print("Bob's Private Key (b):", b)

print("Bob's Public Key (Gb):", Gb)

print("Shared Secret (Alice computes Gab):", shared\_secret\_alice)

print("Shared Secret (Bob computes Gba):", shared\_secret\_bob)

# Verify both shared secrets are equal

if shared\_secret\_alice == shared\_secret\_bob:

print("Key exchange successful!")

else:

print("Error: Shared secrets do not match!")

**Practical 9:** **Implement RSA encryption-decryption algorithm.**

**OUTPUT:**



**CODE:**

import math

def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

def mod\_inverse(e, phi):

t, new\_t = 0, 1

r, new\_r = phi, e

while new\_r != 0:

quotient = r // new\_r

t, new\_t = new\_t, t - quotient \* new\_t

r, new\_r = new\_r, r - quotient \* new\_r

if r > 1:

return -1 # No modular inverse exists

if t < 0:

t += phi

return t

def mod\_exp(base, exp, mod):

result = 1

base = base % mod

while exp > 0:

if exp % 2 == 1:

result = (result \* base) % mod

exp = exp >> 1

base = (base \* base) % mod

return result

def main():

# Two prime numbers (for real implementation, use large primes)

p, q = 61, 53

n = p \* q

phi = (p - 1) \* (q - 1)

e = 17

while gcd(e, phi) != 1:

e += 1

d = mod\_inverse(e, phi)

print(f"Public Key: (e = {e}, n = {n})")

print(f"Private Key: (d = {d}, n = {n})\n")

message = int(input(f"Enter a number to encrypt (must be < {n}): "))

if message >= n:

print(f"Message must be smaller than {n}")

return

encrypted = mod\_exp(message, e, n)

print(f"Encrypted Message: {encrypted}")

decrypted = mod\_exp(encrypted, d, n)

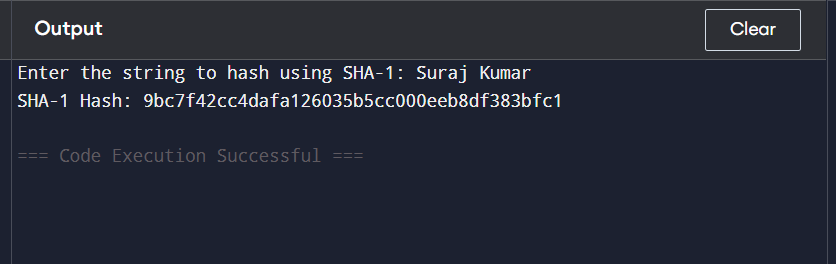
print(f"Decrypted Message: {decrypted}")

if \_\_name\_\_ == "\_\_main\_\_":

main()miz.pro")

**Practical 10:** **Write a program to generate SHA-1 hash.**

**OUTPUT:**



**CODE:**

import hashlib

# Take input from the user

user\_input = input("Enter the string to hash using SHA-1: ")

# Encode the input string

encoded\_input = user\_input.encode()

# Generate SHA-1 hash

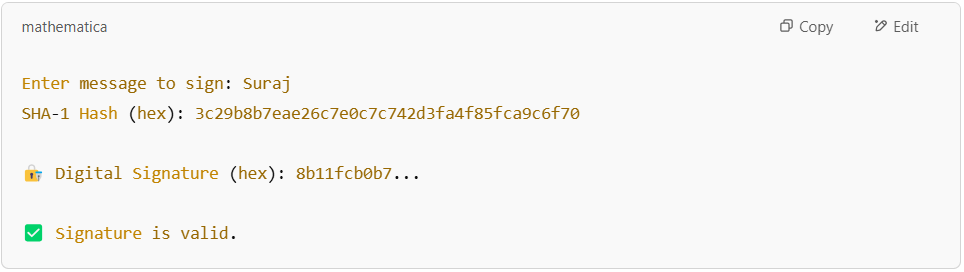
sha1\_hash = hashlib.sha1(encoded\_input).hexdigest()

# Print the resulting hash

print("SHA-1 Hash:", sha1\_hash)

**Practical 11:** **Implement a digital signature algorithm.**

**OUTPUT:**

****

**CODE:**

**Note:** First, install the required library (if not already installed): (pip install cryptography).

from cryptography.hazmat.primitives.asymmetric import rsa, padding

from cryptography.hazmat.primitives import hashes

import binascii

# Step 1: Generate RSA key pair

private\_key = rsa.generate\_private\_key(public\_exponent=65537, key\_size=2048)

public\_key = private\_key.public\_key()

# Step 2: Get input message from user

message = input("Enter message to sign: ").encode('utf-8')

# Step 3: Hash the message using SHA-1

digest = hashes.Hash(hashes.SHA1())

digest.update(message)

hash\_bytes = digest.finalize()

print("SHA-1 Hash (hex):", binascii.hexlify(hash\_bytes).decode())

# Step 4: Sign the hash

signature = private\_key.sign(

hash\_bytes,

padding.PSS(

mgf=padding.MGF1(hashes.SHA1()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA1()

)

# Show signature in hex format

signature\_hex = binascii.hexlify(signature).decode()

print("\n🔏 Digital Signature (hex):", signature\_hex)

# Step 5: Verify the signature

try:

public\_key.verify(

signature,

hash\_bytes,

padding.PSS(

mgf=padding.MGF1(hashes.SHA1()),

salt\_length=padding.PSS.MAX\_LENGTH

),

hashes.SHA1()

)

print("✅ Signature is valid.")

except Exception as e:

print("❌ Signature verification failed.")

**Lab#12 Study and use the Wireshark for the various network protocols.**

1. **Introduction to Wireshark**

Wireshark is one of the most powerful and widely-used network protocol analyzers. It is open-source and allows users to capture and interactively browse the traffic running on a computer network. Originally named **Ethereal**, it was later renamed Wireshark in 2006 due to trademark issues.

Wireshark helps network administrators, cybersecurity experts, and developers analyze and troubleshoot network-related issues, security breaches, and performance bottlenecks.

1. **Key Features of Wireshark**

Wireshark comes with a rich set of features that make it the go-to tool for network analysis:

* 1. ***Live Packet Capture + Offline Analysis***

Capture real-time traffic from interfaces like Ethernet, Wi-Fi, USB, loopback, etc.

Save captures in .pcap/.pcapng formats for offline analysis—ideal for forensics, debugging, and reporting.

**Use Cases:**

* + - Diagnose slow network or dropped connections
    - Monitor suspicious or malicious activity
    - Review historical network traffic

* 1. ***Deep Protocol Inspection***

Wireshark supports deep analysis of 1000+ protocols: TCP/IP, HTTP, DNS, ARP, ICMP, DHCP, FTP, TLS, and more.

**How It Works:**

* Dissects each packet across layers (L2 to L7)
* Shows field names, values, interpretations
* Highlights anomalies or malformed packets

1. ***Advanced Display & Capture Filters*** Filter specific traffic using powerful expressions.

**Examples:**

* + http → Only HTTP packets
  + ip.addr == 192.168.1.1 → Traffic from/to an IP
  + tcp.port == 443 → HTTPS traffic
  + frame contains "password" → Find unencrypted credentials

Auto-suggestions, syntax validation, and logical operators (and, or, not)

1. ***Color Coding for Fast Identification***

Default and custom rules make key traffic stand out.

**Color Examples:**

* + Light purple → TCP
  + Light blue → DNS
  + Black/Red → Malformed/dropped packets

Quickly identify retransmissions, unauthorized traffic, or errors.

1. ***Three-Pane Interface for Packet Analysis*** 
   * + **Packet List:** Timestamp, source, destination, protocol, length, info
     + **Packet Details:** Protocol breakdown (Ethernet > IP > TCP > HTTP)
     + **Packet Bytes:** Raw data in hex + ASCII Enables forensic-style deep inspection.

1. ***Export & Share Data***

Export captured traffic in multiple formats:

* + - * .pcap/.pcapng → Standard format
      * .csv, .json, .xml, .txt → Reports, logs, automation

Share with security teams, use for documentation or analysis scripts.

1. ***Decryption Support (SSL/TLS, WPA2, etc.)***

Analyze encrypted traffic if you have access to keys or credentials.

**Supported Protocols:**

* + - * SSL/TLS (via SSLKEYLOGFILE or private key)
      * WPA/WPA2 (with passphrase)
      * IPsec, SNMPv3, Kerberos

Decryption only works if keys or credentials are available.

1. ***Packet Reassembly***

Reconstruct large data split across multiple packets.

**Examples:**

* + - * Full HTTP responses
      * FTP file transfers
      * Chat sessions over IRC

3

1. ***VoIP & RTP Analysis***

Decode and analyze SIP/RTP voice traffic.

Features:

* Listen to RTP streams
* View call quality (jitter, loss, delay)
* Troubleshoot VoIP issues in real-time

1. ***Custom Profiles & Layouts***

Create task-specific profiles to streamline your workflow.

**Why Use It?**

* + Custom column views
  + Saved filters
  + Switch contexts quickly (e.g., web, DNS, security)

1. ***Multi-Platform Support***

Wireshark works on Windows, macOS, and Linux with consistent UI and feature sets.

1. ***Command-Line Version – tshark***

Wireshark includes tshark for terminal-based traffic capture and analysis.

**Use Cases:**

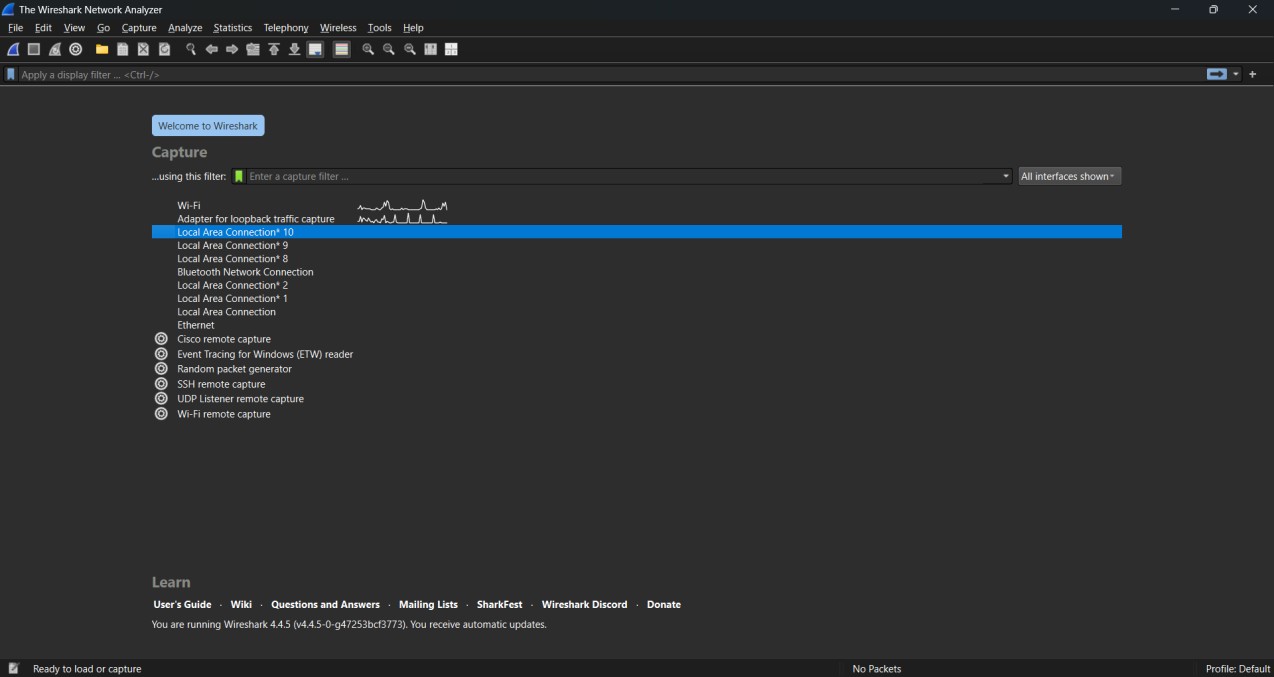
* + Automation scripts
  + Cron jobs
  + Headless server environments

1. ***Integration with Other Tools*** Works well with tools like:
   * + **Nmap** → Network scanning
     + **Snort/Suricata** → IDS alerts
     + **Metasploit** → Attack monitoring
     + **Tcpdump** → Initial captures for later Wireshark analysis
2. **Key Features of Wireshark**

***For Windows:***

* + 1. Download Wireshark from the official site: [https://www.wireshark.org](https://www.wireshark.org/)
    2. Follow the installation wizard.
    3. Make sure to install WinPcap or Npcap (required for packet capturing).

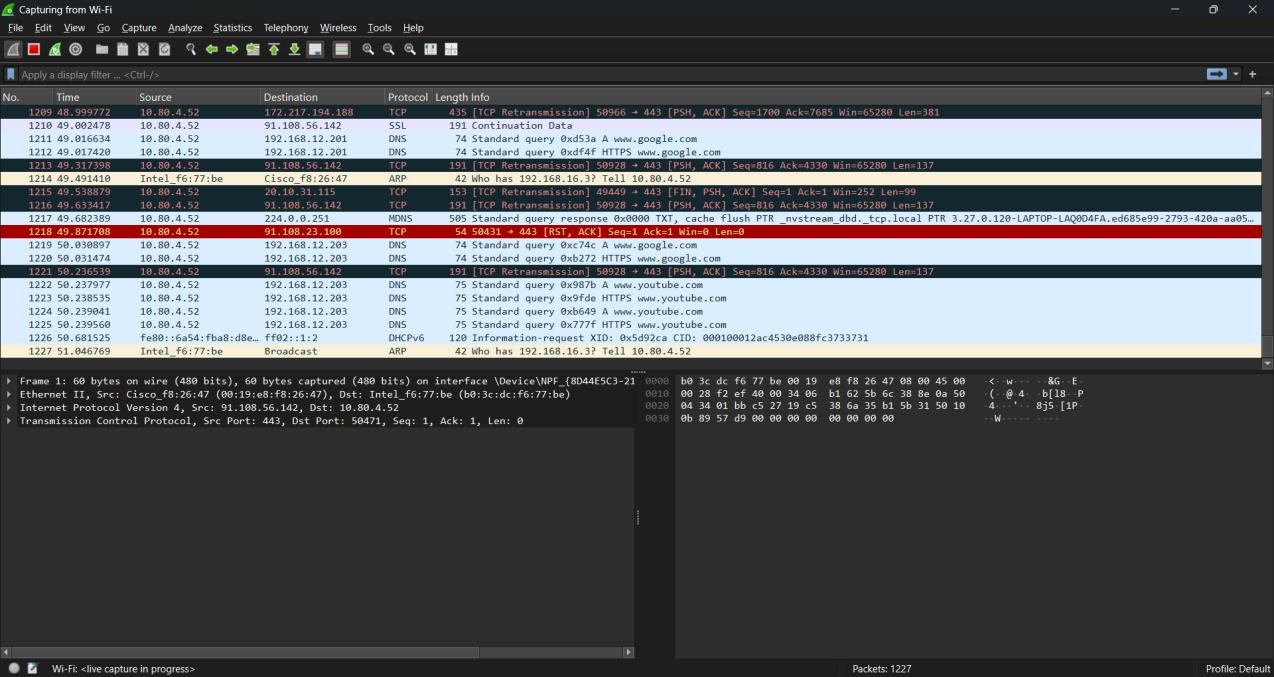
1. **Basic Terminologies in Wireshark** 
   * + Packet: A unit of data sent across a network.
     + Capture Filter: Used before starting the capture to limit what data gets collected.
     + Display Filter: Used after capturing data to filter the view.
     + Protocols: Define rules for data exchange (e.g., HTTP, TCP, DNS, ARP).
2. **Using Wireshark *Step 1: Starting a Capture*** 
   * Open Wireshark.
   * Select the appropriate network interface (e.g., Wi-Fi, Ethernet).
   * Click **Start Capturing Packets**.



***Step 2: Applying Filters***

Wireshark provides both **capture** and **display** filters. Examples:

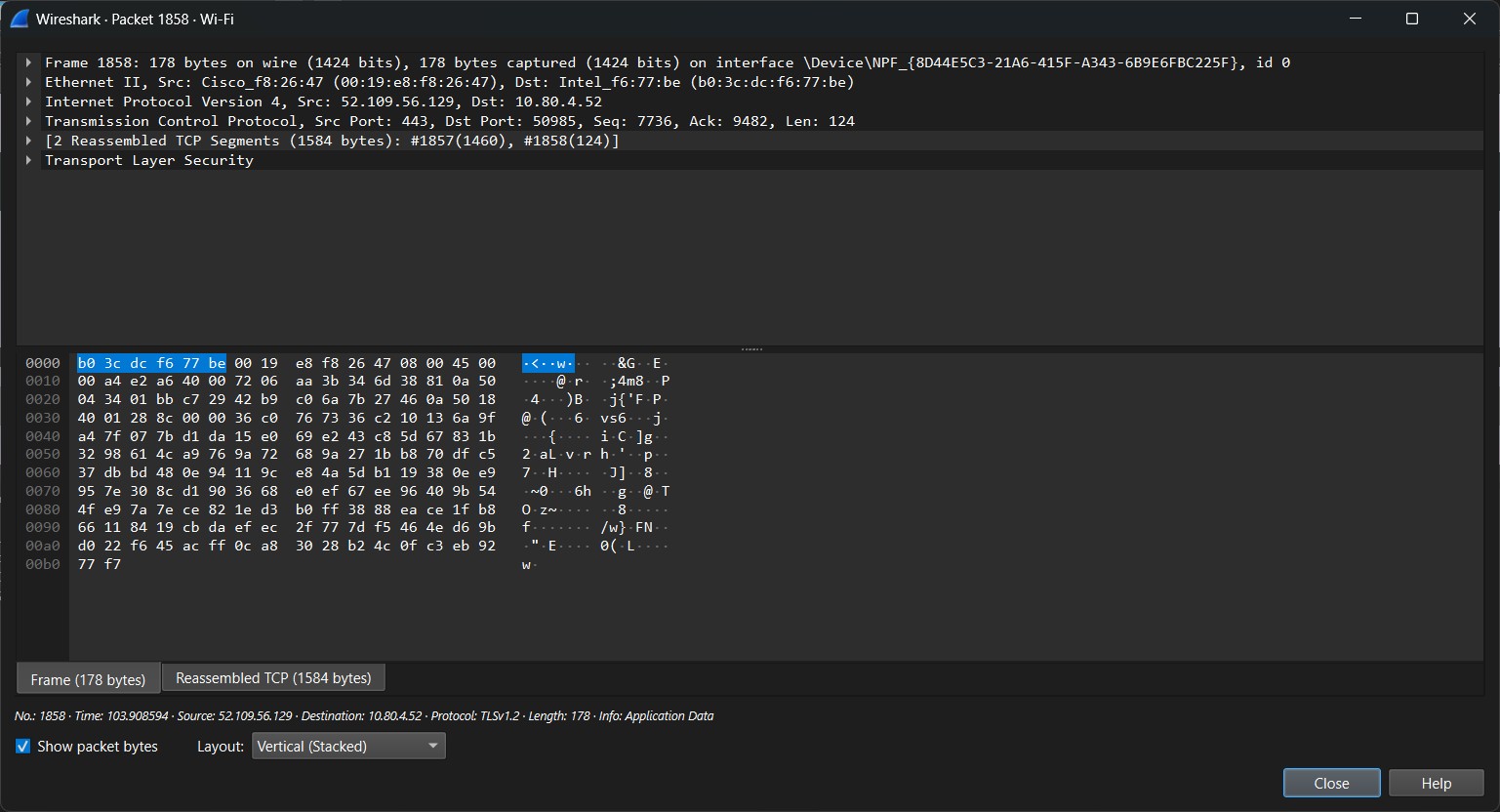
* + http → Show only HTTP traffic
  + ip.addr == 192.168.1.1 → Show packets to/from that IP
  + tcp.port == 443 → Show only HTTPS packets



***Step 3: Analyzing Packets***

Clicking on a packet reveals detailed information in the lower panes:

* + Frame details
  + Ethernet header
  + IP Header
  + TCP/UDP information
  + Application layer details



1. **Real-World Use Cases of Wireshark** 
   * Network troubleshooting: Identify dropped packets, latency issues, or misconfigured devices.
   * Security analysis: Detect malware, ARP spoofing, and intrusion attempts.
   * Protocol development: Validate and debug network-related code.
   * Learning and teaching tool: Perfect for students learning networking.