Shri Vile Parle Kelavani Mandal's

DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING

(Autonomous College Affiliated to the University of Mumbai)

NAAC Accredited with "A" Grade (CGPA : 3.18)

03

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE: 03-08-2024**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: TYBTech**

NAME: Anish Sharma **SAP:60003220045** **DIV:IT1-1** **ROLL:I011**

EXPERIMENT NO. 1

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

a. Implementation of Ceaser Cipher on alphanumeric data.
b. Implementation of Ceaser Cipher on gray scale image.

THEORY / CONCEPT / ALGORITHM:

The Caesar cipher is one of the simplest and most well-known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is shifted a certain number of places down or up the alphabet. The method is named after Julius Caesar, who is said to have used it to communicate with his officials.

SOURCE CODE:

1. Implementation of Ceaser Cipher on alphanumeric data.

```
a=list(input(&quot;Enter text: &quot;)) shift=int(input(&quot;Enter
shift value: &quot;))
ans=&quot;&quot;
z=int(input(&quot;Choose\n1.Encryption\n2.Decryption\n&quot;)
) if(z==1):     for i in a:         if i.isalpha():             c=ord(i)-97
ans=ans+chr(((c+shift)%26)+97)         else:
            c=ord(i)-48

ans=ans+chr(((c+shift)%10)+48)
print(ans) else:     b=list(a)
    ans1=&quot;&quot;
for i in b:
        if i.isalpha():



            c=ord(i)-97             ans1=ans1+chr(((c-
shift)%26)+97)         else:
```

```
            c=ord(i)-48
            ans1=ans1+chr(((c-shift)%10)+48)
    print(ans1)
```

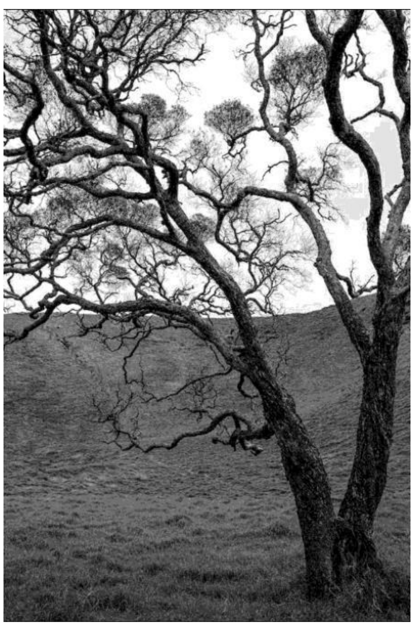
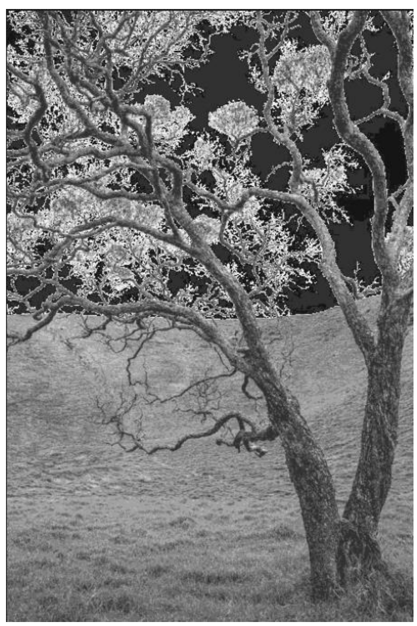
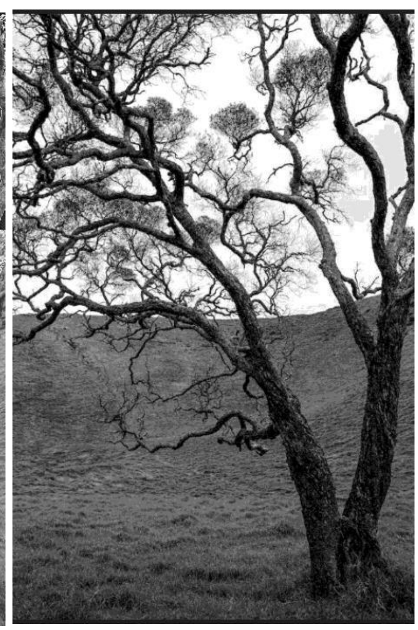
2. Implementation of Ceaser Cipher on gray scale image.

```
from PIL import Image import
numpy as np
 def caesar_cipher_encrypt(image,
key):     # Convert image to numpy
array     img_array = np.array(image)

    # Encrypt by adding the key and wrapping around
256     encrypted_array = (img_array + key) % 256
    # Convert back to image     encrypted_image =
Image.fromarray(encrypted_array.astype('uint8'))     return
encrypted_image
 def caesar_cipher_decrypt(image,
key):     # Convert image to numpy
array     img_array = np.array(image)

    # Decrypt by subtracting the key and wrapping around 256
decrypted_array = (img_array - key) % 256

    # Convert back to image     decrypted_image =
Image.fromarray(decrypted_array.astype('uint8'))     return
decrypted_image
# Load the grayscale image image_path = 'tree.png'  #
replace with your image path grayscale_image =
Image.open(image_path).convert('L')

# Define the key for the Caesar cipher key
= 50

# Encrypt the image encrypted_image =
caesar_cipher_encrypt(grayscale_image, key)
encrypted_image.save('encrypted_image.png')
```

```
# Decrypt the image decrypted_image =
caesar_cipher_decrypt(encrypted_image, key)
decrypted_image.save('decrypted_image.png')
 print("Encryption and decryption
completed.")
```

SAMPLE INPUT AND OUTPUT:

```
Enter text: abc123
Enter shift value: 3
Choose
1.Encryption
2.Decryption
1
def456
```

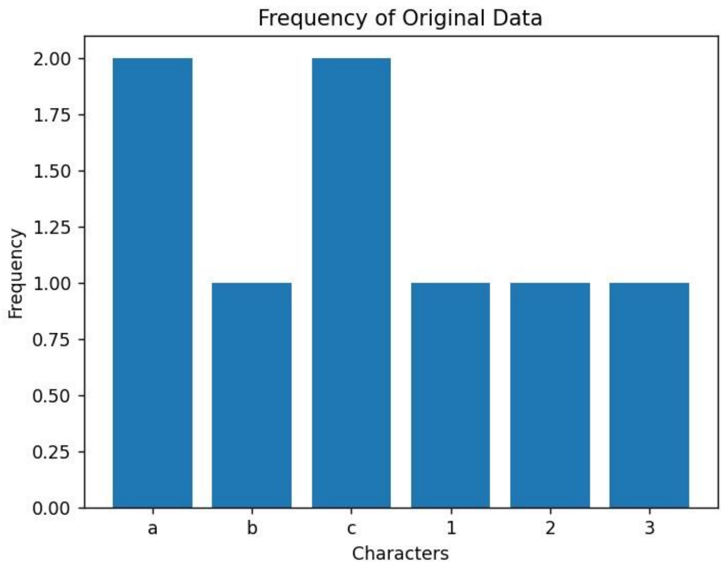
QUESTIONS:

1. Perform a frequency analysis of the encrypted alphanumeric data and compare it to the frequency of the original data.

```
from collections import Counter
import matplotlib.pyplot as plt
```

```
def encryp(text, key):
    a = ''      for char
in text:          if
char.isalpha():
            shift = 65 if char.isupper() else 97
a += chr((ord(char) - shift + key) % 26 + shift)
elif char.isdigit():
            a += chr((ord(char) - 48 + key) % 10 +
48)         else:             a += char     return a
 key = 3 ans="abcac123" final =
encryp(ans, key)
print(f"Original Data: {ans}")
print(f"Encrypted Data:
{final}")


def calculate_frequency(text):
    return Counter(text)
 ofreq = calculate_frequency(ans)
efreq =
calculate_frequency(final)
 def plot_frequency(frequency, title):
characters = list(frequency.keys())
counts = list(frequency.values())
plt.bar(characters, counts)
plt.title(title)
plt.xlabel('Characters')
plt.ylabel('Frequency')     plt.show()

plot_frequency(ofreq, "Frequency of Original Data")
plot_frequency(efreq, "Frequency of Encrypted Data")
```

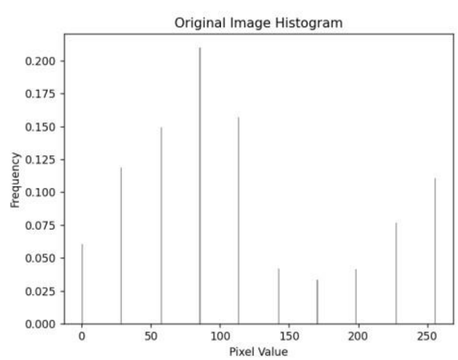
```
Original Data: abcac123
Encrypted Data: defdf456
```

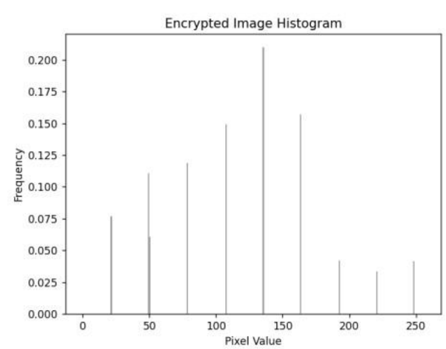


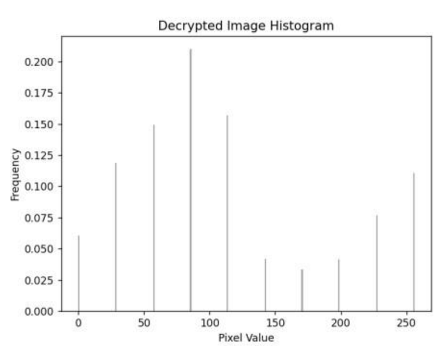
2. Generate histograms of the pixel values for the original, encrypted, and decrypted images.

```
import numpy as np import
matplotlib.pyplot as plt from
PIL import Image
 def
load_image(image_path):
    return Image.open(image_path).convert('L')
 def caesar_cipher_encrypt(image,
key):
    img_array = np.array(image)     encrypted_array = (img_array +
key) % 256     encrypted_image =
Image.fromarray(encrypted_array.astype('uint8'))     return
encrypted_image
 def caesar_cipher_decrypt(image,
key):
    img_array = np.array(image)
    decrypted_array = (img_array - key) % 256     decrypted_image =
Image.fromarray(decrypted_array.astype('uint8'))     return
decrypted_image
 def plot_histogram(image,
title):     img_array =
np.array(image)
    plt.hist(img_array.flatten(), bins=256, range=(0, 256),
density=True, color='gray', alpha=0.75)     plt.title(title)
plt.xlabel('Pixel Value')     plt.ylabel('Frequency')
```

```
    plt.show()

# Load the grayscale image image_path = 'tree.png'  #
replace with your image path original_image =
load_image(image_path)

# Define the key for the Caesar cipher key
= 50

# Encrypt the image encrypted_image =
caesar_cipher_encrypt(original_image, key)
# Decrypt the image decrypted_image =
caesar_cipher_decrypt(encrypted_image, key)
# Plot histograms plot_histogram(original_image, 'Original
Image Histogram') plot_histogram(encrypted_image,
'Encrypted Image Histogram')
plot_histogram(decrypted_image, 'Decrypted Image
Histogram')
```







c. **CONCLUSION:** In this experiment, I understood Implementation of Ceaser Cipher on alphanumeric data and on gray scale image as well as its frequency analysis.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

COURSE CODE: DJS22ITL504 **DATE: 13-08-2024**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: T. Y. BTech**

NAME: Diksha Velhal **SAP:60003220042** **ROLL: I045**

EXPERIMENT NO. 2

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

a. Implementation of Playfair Cipher on Alphanumeric data.

THEORY / CONCEPT / ALGORITHM:

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

- The Algorithm consists of 2 steps: **1. Generate the key Square (5×5):**
  The key square is a 5×5 grid of alphabets that acts as the key for encrypting the plaintext. Each of the 25 alphabets must be unique and one letter of the alphabet (usually J) is omitted from the table (as the table can hold only 25 alphabets). If the plaintext contains J, then it is replaced by I. The initial alphabets in the key square are the unique alphabets of the key in the order in which they appear followed by the remaining letters of the alphabet in order.
  2. **Algorithm to encrypt the plain text:** The plaintext is split into pairs of two letters (digraphs). If there is an odd number of letters, a Z is added to the last letter.
  Pair cannot be made with same letter. Break the letter in single and add a bogus letter to the previous letter.
  If the letter is standing alone in the process of pairing, then add an extra bogus letter with the alone letter

  Rules for Encryption:
- If both the letters are in the same column: Take the letter below each one (going back to the top if at the bottom).
- If both the letters are in the same row: Take the letter to the right of each one (going back to the leftmost if at the rightmost position).
- If neither of the above rules is true: Form a rectangle with the two letters and take the letters on the horizontal opposite corner of the rectangle.

- **SOURCE CODE:**

```
import  numpy  as  np
def ciphertext(key):
mat=[] for i in key:
        if i not in mat:
            if i=='I' or i=='J' and 'IJ':
                if 'IJ' not in mat:
                    mat.append('IJ')
            else:
                mat.append(i)

    a='A' for i in
    range(0,26):
        if a not in mat:
            if a=='I' or a=='J':
                if 'IJ' not in mat:
                    mat.append('IJ')
            else:
                mat.append(chr(ord(a)))

        a=chr(ord(a)+1)
    matrix=np.array(mat) return
    (np.reshape(matrix, (5, 5), order='C'))

def find_character(matrix, target):
    for i in range(len(matrix)):
        for j in range(len(matrix[i])):
            if matrix[i][j] == target:
                return (i, j) # Return the position as a tuple (row, column)
    return None
def find(arr,pt,i,j):
    ind1=find_character(arr,pt[i]) ind2=find_character(arr,pt[j])
    if(ind1[0]==ind2[0]):
    ans=[arr[ind1[0]][(ind1[1]+1)%5],arr[ind2[0]][(ind2[1]+1)%5]]
    elif(ind1[1]==ind2[1]):
        ans=[arr[(ind1[0]+1)%5][ind1[1]],arr[(ind2[0]+1)%5][ind2[1]]]
    else:
        if(ind1[0]<5-1):
            ans=[arr[ind1[0]][(ind1[1]+1)%5],arr[ind2[0]][(ind2[1]-1)%5]]
        else: ans=[arr[ind1[0]][(ind1[1]-1)%5],arr[ind2[0]][(ind2[1]+1)%5]]
    return ans
```

```
def finddec(arr,pt,i,j):
    ind1=find_character(arr,pt[i]) ind2=find_character(arr,pt[j])
    if(ind1[0]==ind2[0]): ans=[arr[ind1[0]][(ind1[1]-
    1)%5],arr[ind2[0]][(ind2[1]-1)%5]]
    elif(ind1[1]==ind2[1]):
        ans=[arr[(ind1[0]-1)%5][ind1[1]],arr[(ind2[0]-1)%5][ind2[1]]]
    else:
        if(ind1[0]<5-1):
            ans=[arr[ind1[0]][(ind1[1]-1)%5],arr[ind2[0]][(ind2[1]+1)%5]]
        else: ans=[arr[ind1[0]][(ind1[1]+1)%5],arr[ind2[0]][(ind2[1]-1)%5]]
    return ans
def encrypt(arr,pt):
    en=[] i=0 j=1 for k in
    range(int(len(pt)/2)):
        en.append(find(arr,pt,i,j))
        i+=2 j=i+1
    return en
def decrypt(arr,pt):
    en=[] i=0 j=1 for k in
    range(int(len(pt)/2)):
        en.append(finddec(arr,pt,i,j))
        i+=2 j=i+1
    return en

key=['D','I','K','S','H','A']
pt=['C','R','E','A','M','Y'] arr=ciphertext(key)
enc=np.array(encrypt(arr,pt)).flatten()
dec=np.array(decrypt(arr,enc)).flatten() print(arr)
print(f"Plain text: {pt}\nKey: {key}\nEncrypted text: {enc}\nDecrypted text: {dec}")
```

- **SAMPLE INPUT AND OUTPUT:**

```
[['D' 'IJ' 'K' 'S' 'H']
 ['A' 'B' 'C' 'E' 'F']
 ['G' 'L' 'M' 'N' 'O']
 ['P' 'Q' 'R' 'T' 'U']
 ['V' 'W' 'X' 'Y' 'Z']]
Plain text: ['C', 'R', 'E', 'A', 'M', 'Y']
Key: ['D', 'I', 'K', 'S', 'H', 'A']
Encrypted text: ['M' 'X' 'F' 'B' 'N' 'X']
Decrypted text: ['C' 'R' 'E' 'A' 'M' 'Y']
```

CONCLUSION: In this experiment we studied about playfair cipher which is a substitutional cipher and its implementation on alphanumeric data

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE: 20-08-24**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: TYBTech**

NAME: Diksha Velhal **SAP ID:60003220042**

EXPERIMENT NO. 3

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

Design and Implementation of a Hill cipher on Gray Scale Image / Color Image.

DESCRIPTION OF EXPERIMENT:

In this lab, we have to implement the Hill Cipher technique. Use two different images, one is covering image which act as key image which is shared by both sender and receiver and other is Informative image. As a first step, we add a cover image and informative image to obtained resultant image. The gray scale image is passed to the Hill Cipher algorithm to form encrypted image. The encrypted image is communicated over an unsecured channel. The encrypted image after receiving by receiver passed to Hill Cipher technique. Receiver first obtained inverse of Key image, K-1. The resulted image which is encrypted is passed to the Hill Cipher to obtain Informative Image. The cover image is subtracted from merged image to obtained informative image. The detail process is summarized in figure 1.

1. Hill cipher with key as text

SOURCE CODE:

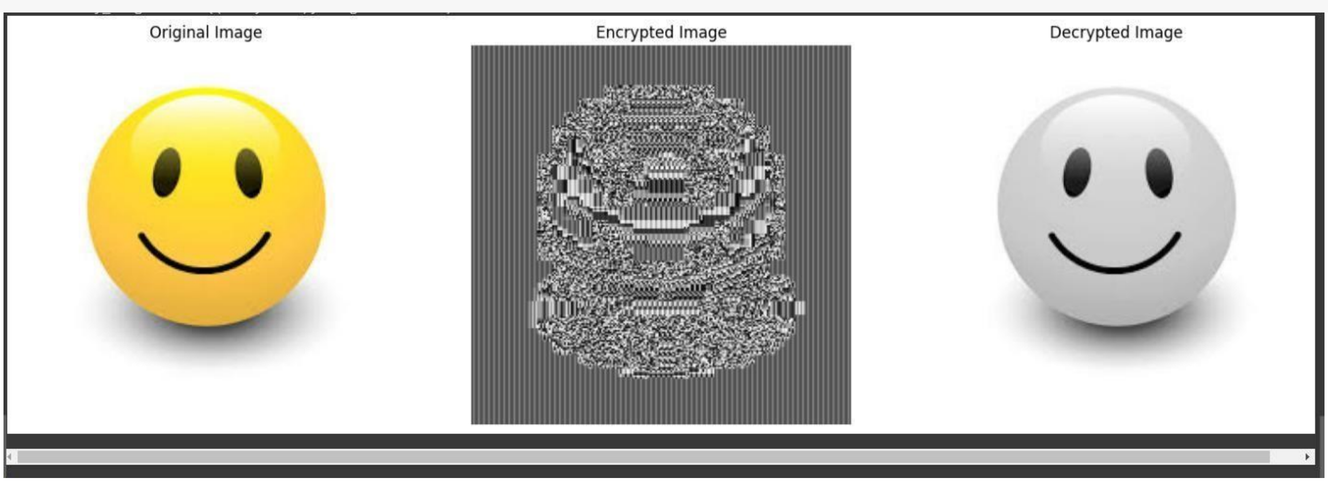
```
 import numpy as np

from   PIL   import   Image   import
matplotlib.pyplot   as   plt   from
google.colab import files import io
 def text_to_matrix(text,
size):
    """Convert text to a matrix of numbers.""" text =
    text.upper().replace(" ", "") if len(text) != size
    * size:
        raise ValueError(f"Text must be of length {size * size} for a
{size}x{size}  matrix.")  matrix  =  [ord(char)  -
    ord('A') for char in text]
```

```
    return np.array(matrix).reshape(size, size)

def matrix_to_text(matrix):
    """Convert matrix of numbers to text.""" return
    ''.join(chr(int(num) + ord('A')) for num in matrix.flatten())

def matrix_inverse(matrix, mod=256):
    """Compute the modular inverse of a matrix.""" det =
    int(round(np.linalg.det(matrix))) det_inv = pow(det, -1, mod) #
    Modular multiplicative inverse of det matrix_adj = np.round(det *
    np.linalg.inv(matrix)).astype(int) % mod return (det_inv *
    matrix_adj) % mod

def hill_cipher_encrypt_image(image, key_word, matrix_size=3):
    """Encrypt an image using the Hill cipher."""
    # Ensure key is of correct length if
    len(key_word) != matrix_size * matrix_size:
        raise ValueError(f"Key must be of length {matrix_size * matrix_size}
for a {matrix_size}x{matrix_size} matrix.")

    #    Create    key    matrix    key_matrix    =
    text_to_matrix(key_word, matrix_size)

    # Convert image data to grayscale image_array
    = np.array(image.convert('L'))

    # Flatten image data and ensure it's divisible by matrix size
    flat_data   =   image_array.flatten()   if   len(flat_data)   %
    matrix_size != 0:
        padding_length   =   matrix_size   -   (len(flat_data)   %   matrix_size)
        flat_data = np.append(flat_data, [0] * padding_length)

    #  Reshape  flat  data  into  matrix  form  data_matrix  =
    np.array(flat_data).reshape(-1, matrix_size)

    # Perform matrix multiplication and modulus
    encrypted_data_matrix   =   np.dot(data_matrix,   key_matrix)   %   256
    encrypted_flat_data = encrypted_data_matrix.flatten().astype(np.uint8)

    #  Reshape  to  original  image  dimensions  and  create  encrypted  image
    encrypted_image_array = encrypted_flat_data.reshape(image_array.shape)
    encrypted_image = Image.fromarray(encrypted_image_array)
    return encrypted_image, key_matrix
```

```
def hill_cipher_decrypt_image(encrypted_image, key_matrix,
matrix_size=3):
```

```
    """Decrypt an image using the Hill cipher."""
    # Compute the inverse key matrix inverse_key_matrix
    = matrix_inverse(key_matrix)

    # Convert encrypted image data to grayscale encrypted_array
    = np.array(encrypted_image.convert('L'))

    # Flatten encrypted data and ensure it's divisible by matrix size
    flat_data  =  encrypted_array.flatten()  if  len(flat_data)  %
    matrix_size != 0:
        padding_length = matrix_size - (len(flat_data) % matrix_size) flat_data
        = np.append(flat_data, [0] * padding_length)

    #  Reshape  flat  data  into  matrix  form  data_matrix  =
    np.array(flat_data).reshape(-1, matrix_size)

    #  Perform matrix multiplication and modulus decrypted_data_matrix =
    np.dot(data_matrix,  inverse_key_matrix)  %  256  decrypted_flat_data  =
    decrypted_data_matrix.flatten().astype(np.uint8)

    #  Reshape  to  original  image  dimensions  and  create  decrypted  image
    decrypted_image_array = decrypted_flat_data.reshape(encrypted_array.shape)
    decrypted_image     =     Image.fromarray(decrypted_image_array)     return
    decrypted_image

# File upload
uploaded = files.upload()

# Assume that the uploaded file is an image for
filename in uploaded.keys():
    image_data  =  uploaded[filename]  image  =
    Image.open(io.BytesIO(image_data))
    print(f'Uploaded file: {filename}')

# Example key key_word = "GYBNQKURP" # 9-letter key
for 3x3 matrix

#    Encrypt    the    image    encrypted_image,    key_matrix    =
hill_cipher_encrypt_image(image, key_word)
```

```
# Decrypt the image
decrypted_image=hill_cipher_decrypt_image(encrypted_image,key_matrix)

# Display original, encrypted, and decrypted images plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1) plt.title("Original Image")
plt.imshow(image,cmap='gray') plt.axis('off')

plt.subplot(1, 3, 2) plt.title("Encrypted Image")
plt.imshow(encrypted_image,cmap='gray') plt.axis('off')

plt.subplot(1, 3, 3) plt.title("Decrypted Image")
plt.imshow(decrypted_image,cmap='gray') plt.axis('off')

plt.show()
```



2. Hill Cipher with both input and key as image

SOURCE CODE:

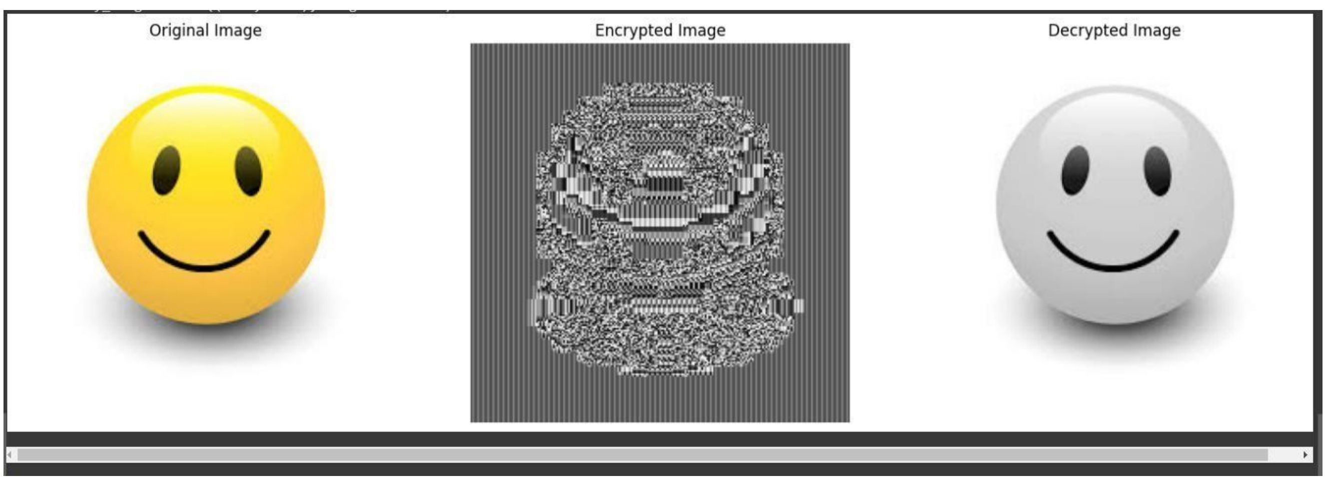
```
 import numpy as np   from PIL import Image import matplotlib.pyplot as plt from
google.colab import files

import io

def matrix_inverse(matrix, mod=256):
    """Compute the modular inverse of a matrix."""
    det = int(round(np.linalg.det(matrix))) try:
        det_inv = pow(det, -1, mod) # Modular multiplicative inverse of det
    except ValueError:
        raise ValueError("The determinant is not invertible for the given
modulus. Please use a different key.") matrix_adj = np.round(det *
    np.linalg.inv(matrix)).astype(int) % mod return (det_inv *
    matrix_adj) % mod

def hill_cipher_encrypt_image(image, key_image, matrix_size=3):
    """Encrypt an image using the Hill cipher."""
    # Convert key image to grayscale and use it as the key matrix key_matrix
    = np.array(key_image.convert('L'))

    # Ensure key matrix is square and matches the specified matrix_size
    if key_matrix.shape[0] != matrix_size or key_matrix.shape[1] !=
matrix_size: raise ValueError(f"Key image must be {matrix_size}x{matrix_size}
        for a
{matrix_size}x{matrix_size} matrix.")

    # Convert image data to grayscale image_array
    = np.array(image.convert('L'))

    # Flatten image data and ensure it's divisible by matrix size
    flat_data = image_array.flatten() if len(flat_data) %
    matrix_size != 0:
        padding_length = matrix_size - (len(flat_data) % matrix_size) flat_data
        = np.append(flat_data, [0] * padding_length)

    # Reshape flat data into matrix form
    data_matrix = np.array(flat_data).reshape(-1, matrix_size)

    # Perform matrix multiplication and modulus
    encrypted_data_matrix = np.dot(data_matrix, key_matrix) % 256
    encrypted_flat_data = encrypted_data_matrix.flatten().astype(np.uint8)
```

```
    # Reshape to original image dimensions and create encrypted image
    encrypted_image_array = encrypted_flat_data.reshape(image_array.shape)
    encrypted_image = Image.fromarray(encrypted_image_array) return
    encrypted_image, key_matrix

def hill_cipher_decrypt_image(encrypted_image, key_matrix, matrix_size=3):
    """Decrypt an image using the Hill cipher."""
    # Compute the inverse key matrix inverse_key_matrix
    = matrix_inverse(key_matrix)

    # Convert encrypted image data to grayscale encrypted_array
    = np.array(encrypted_image.convert('L'))

    # Flatten encrypted data and ensure it's divisible by matrix size
    flat_data = encrypted_array.flatten() if len(flat_data) %
    matrix_size != 0:
        padding_length = matrix_size - (len(flat_data) % matrix_size) flat_data
        = np.append(flat_data, [0] * padding_length)

    # Reshape flat data into matrix form data_matrix =
    np.array(flat_data).reshape(-1, matrix_size)

    # Perform matrix multiplication and modulus decrypted_data_matrix =
    np.dot(data_matrix, inverse_key_matrix) % 256 decrypted_flat_data =
    decrypted_data_matrix.flatten().astype(np.uint8)

    # Reshape to original image dimensions and create decrypted image
    decrypted_image_array = decrypted_flat_data.reshape(encrypted_array.shape)
    decrypted_image = Image.fromarray(decrypted_image_array) return
    decrypted_image

# Function to resize key image to a specific size def
resize_key_image(key_image, size):
    """Resize the key image to the required size.""" return
    key_image.resize((size, size), Image.ANTIALIAS)

# File upload print("Please upload the image to
be encrypted.") uploaded_image = files.upload()
print("Please upload the key image.")
```

```
uploaded_key_image = files.upload()

# Load the uploaded images
image_data = None
key_image_data = None

for filename in uploaded_image.keys():
    image_data      =      Image.open(io.BytesIO(uploaded_image[filename]))
    print(f'Uploaded image file: {filename}')

for filename in uploaded_key_image.keys():
    key_image_data  =  Image.open(io.BytesIO(uploaded_key_image[filename]))
    print(f'Uploaded key image file: {filename}')

#  Resize  key  image  to  required  dimensions
key_sizes = [2, 3, 4] # List of possible sizes
key_size = None key_matrix = None

for size in key_sizes:
    resized_key_image = resize_key_image(key_image_data, size) key_matrix
    = np.array(resized_key_image.convert('L'))

    #   Check   if   key   matrix   is   square   and   invertible   if
    key_matrix.shape[0] == size and key_matrix.shape[1] == size:
        try:
            encrypted_image, key_matrix =
hill_cipher_encrypt_image(image_data, resized_key_image, matrix_size=size)
            decrypted_image = hill_cipher_decrypt_image(encrypted_image,
key_matrix, matrix_size=size)
            key_size = size
            break
        except ValueError:
            continue

if key_size:
    #     Display     original,     encrypted,     and     decrypted     images
    plt.figure(figsize=(18, 6))

    plt.subplot(1,         3,         1)
    plt.title("Original         Image")
    plt.imshow(image_data,
    cmap='gray')
```

```
    plt.axis('off')
plt.subplot(1, 3, 2) plt.title("Encrypted
    Image") plt.imshow(encrypted_image,
    cmap='gray') plt.axis('off')
plt.subplot(1, 3, 3) plt.title("Decrypted
    Image") plt.imshow(decrypted_image,
    cmap='gray') plt.axis('off')
plt.show() else:
    print("No suitable key image found. Please upload a different key image.")
```

OUTPUT:



CONCLUSION: In this experiment we implemented Hill Climb Cipher.

EXPERIMENT NO. 4

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

Implementation and analysis of a S-DES on Plain Text / Image.

DESCRIPTION OF EXPERIMENT:

1. **Take 2 hex values. Use any SHA calculator to compute hash value of (Name+Sap) and apply S-DES on computed hash.**
   Code :

```
from PIL import Image import numpy
as np
import matplotlib.pyplot as plt

# SDES Key and Permutation functions (same as before) P10 = [3, 5, 2, 7, 4,
10, 1, 9, 8, 6]
P8 = [6, 3, 7, 4, 8, 5, 10, 9]
P4 = [2, 4, 3, 1]
IP = [2, 6, 3, 1, 4, 8, 5, 7]
IP_INV = [4, 1, 3, 5, 7, 2, 8, 6]
EP = [4, 1, 2, 3, 2, 3, 4, 1]
S0 = [[1, 0, 3, 2],
      [3, 2, 1, 0],
      [0, 2, 1, 3],
      [3, 1, 3, 2]]
S1 = [[0, 1, 2, 3],
      [2, 0, 1, 3],
      [3, 0, 1, 0],
      [2, 1, 0, 3]]

# Utility functions (same as before) def permute(
table): return [bits[i - 1] for i in table]

def left_shift(bits, n):
                                  ts,
```

```
    return bits[n:] + bits[:n]

def xor(bits1, bits2):
    return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]

def sbox_lookup(sbox, row, col):
    return sbox[row][col]

def generate_keys(key):
    key_p10 = permute(key, P10) left, right = left_shift(key_p10[:5],
    1), left_shift(key_p10[5:], 1) key1 = permute(left + right, P8)
    left, right = left_shift(left, 2), left_shift(right, 2) key2 =
    permute(left + right, P8) return key1, key2

def sdes_function(bits, key):
    expanded_bits = permute(bits, EP) xor_bits = xor(expanded_bits, key)
    left, right = xor_bits[:4], xor_bits[4:] row1, col1 = left[0] * 2 +
    left[3], left[1] * 2 + left[2] row2, col2 = right[0] * 2 + right[3],
    right[1] * 2 + right[2] sbox_output = sbox_lookup(S0, row1, col1) * 4
    + sbox_lookup(S1, row2,
col2) sbox_bits = [(sbox_output >> i) & 1 for i in reversed(range(4))]
    return permute(sbox_bits, P4)

def fk(bits, key):
    left,  right  =  bits[:4],  bits[4:]
    f_output = sdes_function(right, key)
    return xor(left, f_output) + right

def  sdes_encrypt(plaintext,  key):
    key1, key2 = generate_keys(key)
    ip_bits = permute(plaintext, IP)
    fk1_output = fk(ip_bits, key1)
    swapped_bits = fk1_output[4:] + fk1_output[:4]
    fk2_output = fk(swapped_bits, key2) ciphertext
    =    permute(fk2_output,    IP_INV)    return
    ciphertext

def sdes_decrypt(ciphertext, key): key1,
    key2 = generate_keys(key)
```

```
    ip_bits = permute(ciphertext, IP) fk1_output =
    fk(ip_bits,      key2)      swapped_bits      =
    fk1_output[4:] + fk1_output[:4] fk2_output =
    fk(swapped_bits,      key1)      plaintext      =
    permute(fk2_output, IP_INV) return plaintext

# Image Encryption and Decryption def bits_to_byte(bits): return
sum([bit * (1 << (7 - i)) for i, bit in enumerate(bits)])

def byte_to_bits(byte):
    return [(byte >> (7 - i)) & 1 for i in range(8)]

def encrypt_image(image_path, key):
    image = Image.open(image_path) pixels =
    np.array(image)    encrypted_pixels    =
    np.zeros_like(pixels)

    for i in range(pixels.shape[0]):
        for j in range(pixels.shape[1]):
            for  k  in  range(3):  #  For  RGB  channels  bits  =
                byte_to_bits(pixels[i,   j,   k])   encrypted_bits   =
                sdes_encrypt(bits,  key)  encrypted_pixels[i,  j,  k]  =
                bits_to_byte(encrypted_bits)

    encrypted_image = Image.fromarray(encrypted_pixels)
    encrypted_image.save("encrypted_image.png")  return
    encrypted_image

def decrypt_image(image_path, key):
    image = Image.open(image_path)
    pixels = np.array(image)
    decrypted_pixels = np.zeros_like(pixels)

    for i in range(pixels.shape[0]):
        for j in range(pixels.shape[1]):
            for  k  in  range(3):  #  For  RGB  channels  bits  =
                byte_to_bits(pixels[i,   j,   k])   decrypted_bits   =
                sdes_decrypt(bits,  key)  decrypted_pixels[i,  j,  k]  =
                bits_to_byte(decrypted_bits)

    decrypted_image = Image.fromarray(decrypted_pixels)
```

```
    decrypted_image.save("decrypted_image.png") return
    decrypted_image

# Example usage key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0] image_path = "lolipop.jpg"
# Provide the path to the image you want to encrypt

# Encrypt and display the image encrypted_image
= encrypt_image(image_path, key)

#  Decrypt  and  display  the  image  decrypted_image  =
decrypt_image("encrypted_image.png", key)

# Display side by side plt.figure(figsize=(12, 6))
      plt.subplot(1,          2,          1)
plt.imshow(encrypted_image)
plt.title("Encrypted Image")
      plt.subplot(1,          2,          2)
plt.imshow(decrypted_image)
plt.title("Decrypted Image")

plt.show()
```

Output :

```
Ciphertext: 10001101
Decrypted text: 10101010
```

2. **Apply S-DES on image.**

   Code :

```
from PIL import Image import numpy
as np
import matplotlib.pyplot as plt

# SDES Key and Permutation functions (same as before) P10 =
[3, 5, 2, 7, 4, 10, 1, 9, 8, 6]
P8 = [6, 3, 7, 4, 8, 5, 10, 9]
```

```
P4 = [2, 4, 3, 1]
```

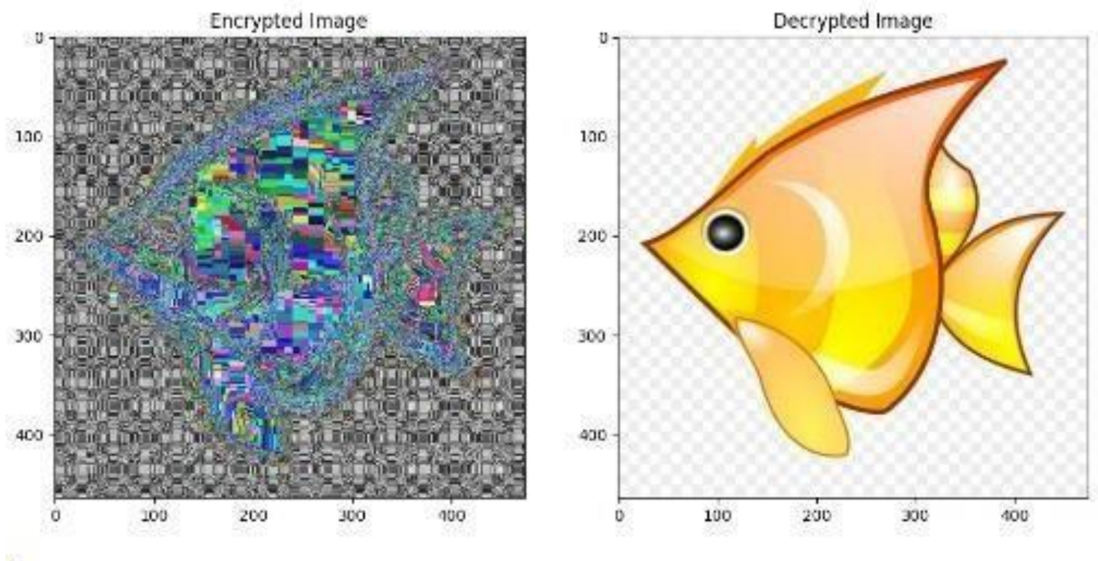
```
IP = [2, 6, 3, 1, 4, 8, 5, 7]
IP_INV = [4, 1, 3, 5, 7, 2, 8, 6]
EP = [4, 1, 2, 3, 2, 3, 4, 1]
S0 = [[1, 0, 3, 2],
      [3, 2, 1, 0],
      [0, 2, 1, 3],
      [3, 1, 3, 2]]

S1 = [[0, 1, 2, 3],
      [2, 0, 1, 3],
      [3, 0, 1, 0],
      [2, 1, 0, 3]]
```

```
# Utility functions (same as before) def
permute(bits, table): return [bits[i -
1] for i in table]

def left_shift(bits, n): return
    bits[n:] + bits[:n]

def xor(bits1, bits2):
    return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]

def sbox_lookup(sbox, row, col):
    return sbox[row][col]

def generate_keys(key):
    key_p10 = permute(key, P10) left, right = left_shift(key_p10[:5], 1),
    left_shift(key_p10[5:], 1) key1 = permute(left + right, P8)
    left, right = left_shift(left, 2), left_shift(right, 2)
    key2 = permute(left + right, P8) return key1, key2

def sdes_function(bits, key):
    expanded_bits   =   permute(bits,   EP)
    xor_bits = xor(expanded_bits, key) left,
    right = xor_bits[:4], xor_bits[4:]
    row1, col1 = left[0] * 2 + left[3], left[1] * 2 + left[2] row2, col2
    = right[0] * 2 + right[3], right[1] * 2 + right[2] sbox_output =
    sbox_lookup(S0, row1, col1) * 4 + sbox_lookup(S1, row2,
col2) sbox_bits = [(sbox_output >> i) & 1 for i in reversed(range(4))]
    return permute(sbox_bits, P4)
```

```
def fk(bits, key):
    left,  right  =  bits[:4],  bits[4:]
    f_output = sdes_function(right, key)
    return xor(left, f_output) + right

def  sdes_encrypt(plaintext,  key):  key1,  key2  =
    generate_keys(key)          ip_bits            =
    permute(plaintext,     IP)     fk1_output     =
    fk(ip_bits,      key1)      swapped_bits      =
    fk1_output[4:]  +  fk1_output[:4]  fk2_output  =
    fk(swapped_bits,     key2)     ciphertext     =
    permute(fk2_output, IP_INV) return ciphertext

def  sdes_decrypt(ciphertext,  key):  key1,  key2  =
    generate_keys(key)          ip_bits            =
    permute(ciphertext,     IP)     fk1_output     =
    fk(ip_bits,      key2)      swapped_bits      =
    fk1_output[4:]  +  fk1_output[:4]  fk2_output  =
    fk(swapped_bits,     key1)     plaintext      =
    permute(fk2_output, IP_INV) return plaintext

#  Image  Encryption  and  Decryption  def  bits_to_byte(bits):  return
sum([bit * (1 << (7 - i)) for i, bit in enumerate(bits)])

def byte_to_bits(byte):
    return [(byte >> (7 - i)) & 1 for i in range(8)]

def encrypt_image(image_path, key):
    image = Image.open(image_path)
    pixels = np.array(image)
    encrypted_pixels = np.zeros_like(pixels)

    for i in range(pixels.shape[0]):
        for j in range(pixels.shape[1]):
            for   k   in   range(3):   #   For   RGB   channels   bits   =
                byte_to_bits(pixels[i,    j,    k])    encrypted_bits    =
                sdes_encrypt(bits,   key)  encrypted_pixels[i,   j,   k]  =
                bits_to_byte(encrypted_bits)
```

```
    encrypted_image = Image.fromarray(encrypted_pixels)
    encrypted_image.save("encrypted_image.png")  return
    encrypted_image

def decrypt_image(image_path, key):
    image = Image.open(image_path) pixels =
    np.array(image)    decrypted_pixels    =
    np.zeros_like(pixels)

    for i in range(pixels.shape[0]):
        for j in range(pixels.shape[1]):
            for   k   in   range(3):   #   For   RGB   channels   bits   =
                byte_to_bits(pixels[i,    j,    k])    decrypted_bits    =
                sdes_decrypt(bits,   key)  decrypted_pixels[i,   j,   k]  =
                bits_to_byte(decrypted_bits)

    decrypted_image = Image.fromarray(decrypted_pixels)
    decrypted_image.save("decrypted_image.png")  return
    decrypted_image

# Example usage key = [1, 0, 1, 0, 0, 0, 0, 0, 1, 0] image_path =
"lolipop.jpg" # Provide the path to the image you want to encrypt

# Encrypt and display the image encrypted_image
= encrypt_image(image_path, key)

# Decrypt and display the image
decrypted_image = decrypt_image("encrypted_image.png", key)

# Display side by side plt.figure(figsize=(12,
6))

plt.subplot(1,     2,     1)
plt.imshow(encrypted_image)
plt.title("Encrypted
Image")

plt.subplot(1,     2,     2)
plt.imshow(decrypted_image)
plt.title("Decrypted
Image") plt.show()
```

Output :



CONCLUSION:

In this experiment , we learnt to implement S-DES on plaintext and image.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE:24-9-24**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: TYBTech**

Name : Diksha Velhal **Sap ID : 60003220042**

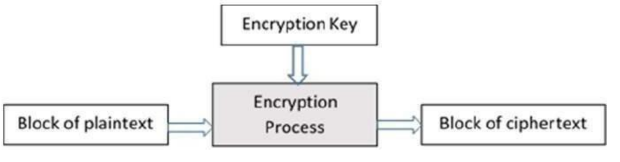
EXPERIMENT NO. 5

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:
Analysis of Modern Block Ciphers (use crypt APIs)

DESCRIPTION OF EXPERIMENT:

A block cipher takes a block of plaintext bits and generates a block of ciphertext bits, generally of same size. The size of block is fixed in the given scheme. The choice of block size does not directly affect to the strength of encryption scheme. The strength of cipher depends up on the key length.



Analysis:

1. Use crypt API to encrypt/decrypt a plaintext block using AES, DES
2. Avalanche Effect: Change in Plaintext
3. Avalanche Effect: Change in key

SOURCE CODE:

```
 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes from
 Crypto.Cipher import DES from time import time
 from prettytable import PrettyTable

 # Helper function to count bit differences def
count_bits_changed(ct1, ct2): # Convert ciphertexts to binary
binary_ct1 = ''.join(format(byte, '08b') for byte in ct1)
binary_ct2 = ''.join(format(byte, '08b') for byte in ct2)
# Count the number of bits that differ bits_changed = sum(b1 != b2 for b1,
b2 in zip(binary_ct1, binary_ct2)) return bits_changed, len(binary_ct1)
```

```
# AES encryption and decryption function def
aes_encrypt_decrypt(plaintext, key):
    key = key.ljust(16, '0').encode('utf-8') # Ensure key is 16 bytes
    cipher = Cipher(algorithms.AES(key), modes.ECB())

    encryptor = cipher.encryptor() decryptor
    = cipher.decryptor()

    # Padding plaintext to 16 bytes
    padded_plaintext = plaintext.ljust(16, ' ')

    #     Encrypt     start_enc     =     time()     ciphertext     =
    encryptor.update(padded_plaintext.encode('utf-8')) +
encryptor.finalize()
    end_enc = time()

    # Decrypt start_dec = time() decrypted_text =
    decryptor.update(ciphertext) + decryptor.finalize() end_dec =
    time()

    # Convert time to nanoseconds enc_time_ns =
    (end_enc - start_enc) * 1e9 dec_time_ns =
    (end_dec   -   start_dec)   *   1e9   return
    ciphertext, enc_time_ns, dec_time_ns

#     DES     encryption     and     decryption     function     def
des_encrypt_decrypt(plaintext,    key):    key    =    key.ljust(8,
'0').encode('utf-8') # Ensure key is 8 bytes cipher = DES.new(key,
DES.MODE_ECB)

    # Padding plaintext to 8 bytes
    padded_plaintext = plaintext.ljust(8, ' ')

    # Encrypt start_enc
    = time()
    ciphertext = cipher.encrypt(padded_plaintext.encode('utf-8'))
    end_enc = time()

    # Decrypt start_dec = time()
    decrypted_text =
    cipher.decrypt(ciphertext) end_dec =
    time()

    # Convert time to nanoseconds
    enc_time_ns = (end_enc - start_enc) * 1e9
    dec_time_ns = (end_dec - start_dec) * 1e9
    return ciphertext, enc_time_ns, dec_time_ns
```

```
# Main Experiment Code def
main_experiment():
    plaintext = "123456" # Sample plaintext
    key_aes = "123" # AES Key key_des = "123"
    # DES Key

    # AES Encryption/Decryption aes_ct, aes_enc_time, aes_dec_time =
    aes_encrypt_decrypt(plaintext,
key_aes)

    # DES Encryption/Decryption des_ct, des_enc_time, des_dec_time =
    des_encrypt_decrypt(plaintext,
key_des)

    # Avalanche Effect - Change in Plaintext new_plaintext
    = "123457"

    aes_new_ct, _, _ = aes_encrypt_decrypt(new_plaintext, key_aes)
    des_new_ct, _, _ = des_encrypt_decrypt(new_plaintext, key_des)

    aes_bits_changed, _ = count_bits_changed(aes_ct, aes_new_ct)
    des_bits_changed, _ = count_bits_changed(des_ct, des_new_ct)

    # Avalanche Effect - Change in Key
    new_key_aes = "122" new_key_des =
    "122"

    aes_new_ct_key, _, _ = aes_encrypt_decrypt(plaintext, new_key_aes)
    des_new_ct_key, _, _ = des_encrypt_decrypt(plaintext, new_key_des)

    aes_bits_changed_key, _ = count_bits_changed(aes_ct, aes_new_ct_key)
    des_bits_changed_key, _ = count_bits_changed(des_ct, des_new_ct_key)

    # Create a PrettyTable to display the results
    table = PrettyTable()

    # Adding columns to the table
    table.field_names = ["", "AES-128", "DES"]

    # Plaintext and ciphertext table.add_row(["Plaintext",
    plaintext, plaintext]) table.add_row(["Ciphertext",
    aes_ct.hex(), des_ct.hex()]) table.add_row(["Encryption Time
    (ns)", f"{aes_enc_time:.0f} ns",
f"{des_enc_time:.0f} ns"]) table.add_row(["Decryption Time (ns)",
    f"{aes_dec_time:.0f} ns",
f"{des_dec_time:.0f} ns"])

    # Avalanche Effect for change in Plaintext
    table.add_row(["\nAvalanche Effect (Change in Plaintext)", "", ""])
    table.add_row(["Original Plaintext", plaintext, plaintext])
```

```
table.add_row(["Ciphertext",         aes_ct.hex(),         des_ct.hex()])
table.add_row(["Changed   Plaintext",    new_plaintext,    new_plaintext])
table.add_row(["New   Ciphertext",   aes_new_ct.hex(),   des_new_ct.hex()])
table.add_row(["No of Bits Changed", aes_bits_changed, des_bits_changed])
# Avalanche Effect for change in Key table.add_row(["\nAvalanche
Effect (Change in Key)", "", ""]) table.add_row(["Original Key",
key_aes, key_des]) table.add_row(["Ciphertext", aes_ct.hex(),
des_ct.hex()]) table.add_row(["Changed Key", new_key_aes,
new_key_des]) table.add_row(["New Ciphertext", aes_new_ct_key.hex(),
 des_new_ct_key.hex()]) table.add_row(["No of Bits Changed",
      aes_bits_changed_key,
 des_bits_changed_key])

# Print the table
print(table)

 # Run the
 experiment


 main_experiment(
```

Output :

	AES-128	DES
Plaintext	123456	123456
Ciphertext	186ac2fbebe2d232a909b11c0a603b92	bae723198ae0e6d9
Encryption Time (ns)	54836 ns	56028 ns
Decryption Time (ns)	3815 ns	6676 ns
Avalanche Effect (Change in Plaintext)		
Original Plaintext	123456	123456
Ciphertext	186ac2fbebe2d232a909b11c0a603b92	bae723198ae0e6d9
Changed Plaintext	123457	123457
New Ciphertext	b6e03bd7ae0a94191680123f4b14309a	3538d3862616e498
No of Bits Changed	58	35
Avalanche Effect (Change in Key)		
Original Key	123	123
Ciphertext	186ac2fbebe2d232a909b11c0a603b92	bae723198ae0e6d9
Changed Key	122	122
New Ciphertext	963fd555a64917a5622af415ee6f0f07	bae723198ae0e6d9
No of Bits Changed	63	0

Based on amount of time taken for encryption/decryption comment wrt to performance ? AES generally takes longer than DES due to its larger block size (128 bits vs. 64 bits) and more complex encryption rounds. Thus, DES exhibits faster encryption and decryption times but at the cost of lower security.

Which algo exhibits better avalanche effect wrt to change in plaintext ?

AES shows a stronger avalanche **effect** when the plaintext is changed. Even a small change in the plaintext (like changing one character) causes a significant difference in the ciphertext, making AES more robust in this aspect compared to DES.
Which algo exhibits better avalanche effect wrt to change in key ?
AES also demonstrates a better avalanche effect when the key is changed. A small modification to the key results in a dramatically different ciphertext, making AES more secure and sensitive to key variations than DES.

Conclusion :
In this experiment , we compared DES and AES and we found out that AES is more effective than DES.

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE: 28-09-2024**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: TYBTech**

NAME: Diksha Velhal **SAP:60003220042** **ROLL: I045**

EXPERIMENT NO. 5

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

To implement the Knapsack algorithm for encrypting and decrypting text files and image files.

DESCRIPTION OF EXPERIMENT:

The Knapsack problem is a computational problem used in cryptography, where the task is to determine whether a subset of given integers sums to a particular value.

Procedure:

1. **Key Generation:**
   - Select a super-increasing sequence for the private key.
   - Choose a modulus `m` such that it is greater than the sum of all elements in the sequence.
   - Choose a multiplier `n` that is co-prime to the modulus `m`.
   - Generate the public key by computing `(wᵢ * n) % m` for each element `wᵢ` in the private key.
2. **Encryption:**
   - Convert the text to its binary representation.
   - Multiply the binary bits with corresponding values of the public key and sum the results to generate the ciphertext.
3. **Decryption:**
   - Calculate the modular inverse of the multiplier `n` with respect to `m`.

- Multiply the ciphertext by the inverse and reduce modulo m.
- Use the super-increasing sequence to find the binary bits of the original message and convert it back to text.

Analysis:

- Observe how the plaintext is converted into its binary equivalent and encrypted using the publi c key.

- Analy ze the struct ure of the public key, privat e key, and the transf ormati on proces s.
- Note how the decr yptio n retri eves the origi nal binar y sequ ence and conv erts it back into read able text.

SOURCE CODE:

```
import random

# Function to generate a super-increasing sequence for the public key
def generate_super_increasing_sequence(n):
    sequence = [random.randint(1, 100)]
    while len(sequence) < n:
        next_element = sum(sequence) + random.randint(1, 10)
        sequence.append(next_element)
    print(sequence)
    return sequence

# Function to generate the private key from the public key
def generate_private_key(public_key, q, r):
    private_key = [(r * element) % q for element in public_key] return
    private_key

# Function to encrypt the plaintext using the public key def
knapsack_encrypt(plaintext, public_key):
    encrypted_message = sum(public_key[i] for i in range(len(plaintext)) if
plaintext[i] == '1') return
    encrypted_message

# Function to decrypt the ciphertext using the private key def
knapsack_decrypt(ciphertext, private_key, q, r):
    r_inverse = pow(r, -1, q) # Modular multiplicative inverse of r
    decrypted_message = '' for element in reversed(private_key):
        if (ciphertext * r_inverse) % q >= element:
            decrypted_message = '1' + decrypted_message
            ciphertext -= element
        else:
            decrypted_message = '0' + decrypted_message
    return decrypted_message

# Function to convert a string to binary def
string_to_binary(text):
    return ''.join(format(ord(char), '08b') for char in text)

# Function to convert binary to string def binary_to_string(binary): chars
= [chr(int(binary[i:i + 8], 2)) for i in range(0, len(binary), 8)] return
''.join(chars)

# Example usage
if_name_== "_main_":
    n = 8 # Number of elements in the super-increasing sequence q = 103 #
    Modulus (should be greater than the sum of the super-increasing
sequence) r = 3 # Multiplier for generating
    private key
```

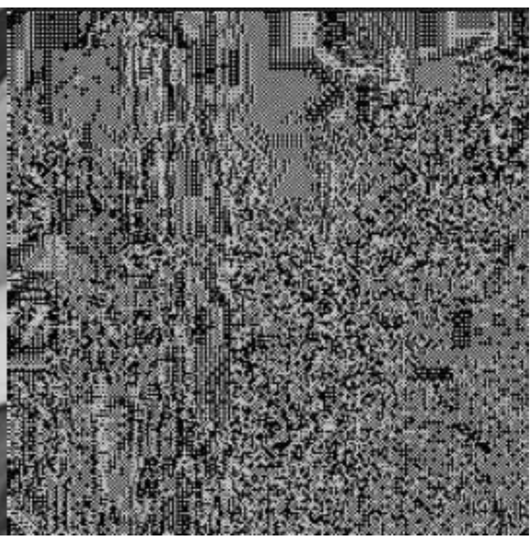
```
# Generate the public key and private key public_key
= generate_super_increasing_sequence(n) private_key
= generate_private_key(public_key, q, r)
# Plaintext message plaintext
= "HELLO"
binary_plaintext = string_to_binary(plaintext)
# Encrypt each byte of the binary plaintext
ciphertext = [] for i in range(0,
len(binary_plaintext), 8):
    byte = binary_plaintext[i:i + 8] encrypted_byte =
 knapsack_encrypt(byte, public_key)
 ciphertext.append(encrypted_byte)
# Decrypt each byte of the ciphertext decrypted_binary = '' for
encrypted_byte in ciphertext: decrypted_byte =
knapsack_decrypt(encrypted_byte, private_key, q, r) decrypted_binary +=
decrypted_byte
# Convert decrypted binary back to string
decrypted_message = binary_to_string(decrypted_binary)
  print("Original Message:", plaintext)
print("Binary Representation:",
binary_plaintext) print("Encrypted Ciphertext:",
ciphertext) print("Decrypted Message:",
decrypted_message)
```

```
[69, 79, 149, 304, 604, 1215, 2426, 4854]
Original Message: HELLO
Binary Representation: 0100100001000101010011000100110001001111
Encrypted Ciphertext: [683, 6148, 1898, 1898, 9178]
Decrypted Message: ííááà
```

```
import random from
PIL import Image
import numpy as np
# Function to generate a super-increasing sequence for the public key
def generate_super_increasing_sequence(n):
    sequence = [random.randint(1, 100)]
    while len(sequence) < n:
        next_element = sum(sequence) + random.randint(1, 10)
        sequence.append(next_element)
    return sequence

# Function to generate the private key from the public key
def generate_private_key(public_key, q, r):
```

```
    private_key = [(r * element) % q for element in public_key] return
    private_key

# Function to encrypt the plaintext using the public key def
knapsack_encrypt(plaintext, public_key):
    encrypted_message = sum(public_key[i] for i in range(len(plaintext)) if
plaintext[i] == '1') return
    encrypted_message

# Function to decrypt the ciphertext using the private key def
knapsack_decrypt(ciphertext, private_key, q, r):
    r_inverse = pow(r, -1, q) # Modular multiplicative inverse of r
    decrypted_message = '' for element in reversed(private_key):
        if (ciphertext * r_inverse) % q >= element:
            decrypted_message = '1' + decrypted_message
            ciphertext -= element
        else:
            decrypted_message = '0' + decrypted_message
    return decrypted_message

# Function to convert grayscale image to binary def
image_to_binary(image_path):
    img = Image.open(image_path).convert('L') # Convert to grayscale img =
    img.resize((225, 225))  # Resize to 225x225 img_array = np.array(img)
    binary_data = ''.join(format(pixel, '08b') for row in img_array for pixel
    in
row.flatten()) return binary_data,
    img_array.shape

# Function to convert binary back to grayscale image def
binary_to_image(binary_data, shape, output_path):
    total_pixels = shape[0] * shape[1] # Total number of pixels
    if len(binary_data) < total_pixels * 8:
        raise ValueError("Decrypted binary data is too short for the specified
shape.")

    img_array = np.array([int(binary_data[i:i + 8], 2) for i in range(0,
len(binary_data), 8)]) img_array = img_array[:total_pixels] # Trim to total
    number of pixels img_array = img_array.reshape(shape) img =
    Image.fromarray(img_array.astype('uint8'), 'L') # Create grayscale image
    img.save(output_path)

# Example usage
if_name_== "_main_":
    n = 16 # Number of elements in the super-increasing sequence q = 103 #
    Modulus (should be greater than the sum of the super-increasing
sequence) r = 3 # Multiplier for generating
    private key
```

```
# Generate the public key and private key public_key
= generate_super_increasing_sequence(n) private_key
= generate_private_key(public_key, q, r)
# Convert grayscale image to binary input_image_path = 'input_image.png' #
Specify your grayscale image file path binary_data, shape =
image_to_binary(input_image_path)
# Encrypt each byte of the binary data
ciphertext = [] for i in range(0,
len(binary_data), 8):
    byte = binary_data[i:i + 8] encrypted_byte =
 knapsack_encrypt(byte, public_key)
 ciphertext.append(encrypted_byte)
# Decrypt each byte of the ciphertext decrypted_binary = '' for
encrypted_byte in ciphertext: decrypted_byte =
knapsack_decrypt(encrypted_byte, private_key, q, r) decrypted_binary +=
decrypted_byte
# Convert decrypted binary back to grayscale image output_image_path =
'decrypted_image.png' # Specify the output image path
binary_to_image(decrypted_binary, shape, output_image_path)
  print("Encryption and decryption
completed.")
```

OBSERVATIONS AND CONCLUSION:

In this experiment, we learned how to implement the Knapsack algorithm for encrypting and decrypting text files and image files.

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE:12-10-24**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS: TYBTech**

Name : Diksha Velhal **Sap ID: 60003220042**

EXPERIMENT NO. 7

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

To implement the RSA algorithm for encrypting and decrypting text files and image files.

DESCRIPTION OF EXPERIMENT:

- Select two large prime numbers, p and q.
- Multiply these numbers to find n = p x q, where n is called the modulus for encryption and decryption.
- Choose a number e less than n, such that n is relatively prime to (p - 1) x (q -1). It means that e and (p - 1) x (q - 1) have no common factor except 1. Choose "e" such that 1<e < φ (n), e is prime to φ (n), gcd (e,d(n)) =1
- If n = p x q, then the public key is <e, n>. A plaintext message m is encrypted using public key <e, n>. To find ciphertext from the plain text following formula is used to get ciphertext C.
  C = m^e mod n
  Here, m must be less than n. A larger message (>n) is treated as a concatenation of messages, each of which is encrypted separately.
- To determine the private key, we use the following formula to calculate the d such that: de mod {(p - 1) x (q - 1)} = 1
  Or
  de mod φ (n) = 1
- The private key is <d, n>. A ciphertext message c is decrypted using private key <d, n>. To calculate plain text m from the ciphertext c following formula is used to get plain text m.
  m = c^d mod n

SOURCE CODE:

Code for Text :

```
import random
from sympy import isprime, mod_inverse
      def
generate_keys(p=241,
q=233): # Step 2:
   Calculate n = p * q n
   = p * q

   # Step 3: Calculate ϕ(n) = (p -
   1) * (q - 1) ϕ_n = (p - 1) * (q
   - 1)
```

```
    # Step 5: Calculate d such that de
    mod ϕ(n) = 1 d = mod_inverse(e,
    ϕ_n)

    # Return public and
    private keys return
    (e, n), (d, n), p, q

def gcd(a,
    b):
    while b:
        a, b = b, a %
    b return a

def encrypt(message,
    public_key): e, n =
    public_key ciphertext
    = []

    # Split message into chunks if necessary for i
    in range(0, len(message), 2): # Using chunks of
    2 characters chunk = message[i:i+2] m =
    int.from_bytes(chunk.encode('utf8'), 'big') if
    m >= n:
            raise ValueError("Chunk too long
        to encrypt.") C = pow(m, e, n)
    ciphertext.appen d(C) return
    ciphertext
```

def **Output:**decrypt (ciphertext,

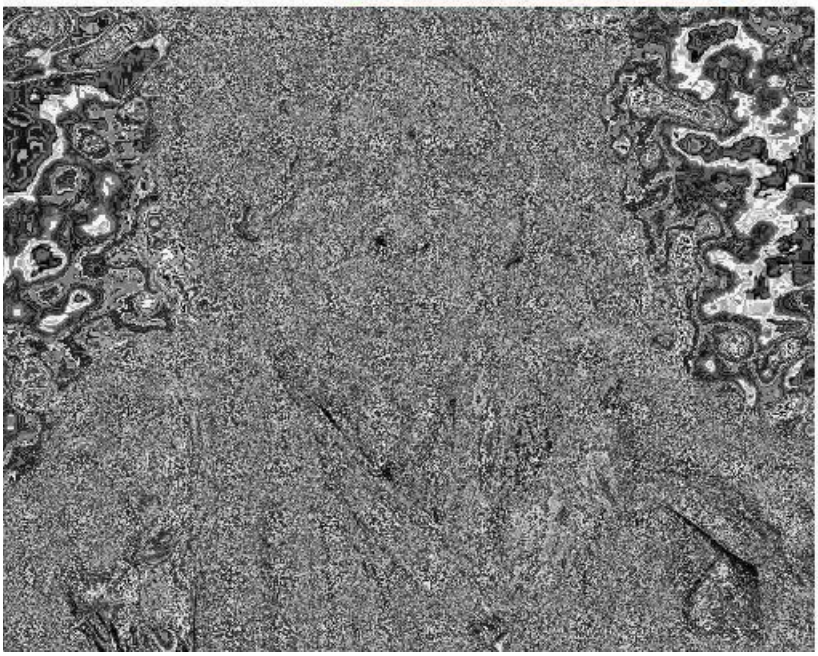
```
Public Key: (22141, 56153)
Private Key: (11221, 56153)
p: 241, q: 233
Enter a message to encrypt: My name is Isha
Encrypted message (ciphertext): [9846, 8327, 50246, 48398, 31809, 38284, 19832, 8291]
Decrypted message: My name is Isha
```

Code for Image :

```
import  random  from
sympy         import
mod_inverse     from
PIL   import   Image
import numpy as np
from IPython.display import display
      def
generate_keys(
```

```
    d = mod_inverse(e, φ_n)
    return (e, n), (d, n)

def gcd(a, b):
    while b: a, b = b,
        a % b
    return a

def encrypt_image(image_bytes, public_key):
    e, n = public_key encrypted_data
    = []

    # Encrypt each byte chunk using RSA for
    byte in image_bytes:
        encrypted_byte = pow(byte, e, n)

    encrypted_data.append(encrypted_byte)

    return encrypted_data

def decrypt_image(encrypted_data, private_key):
    d, n = private_key decrypted_data
    = []

    # Decrypt each byte chunk using RSA for
    encrypted_byte in encrypted_data:
    decrypted_byte = pow(encrypted_byte, d, n)
    decrypted_data.append(decrypted_byte) return
    decrypted_data

def save_encrypted_image(encrypted_data, image_shape, output_filename=None):
    # Convert encrypted data to mod 256 for grayscale image visualization
    encrypted_mod_data = [x % 256 for x in encrypted_data]

    # Handle cases where the data length does not exactly match the original image shape
    num_pixels = np.prod(image_shape) # Total number of pixels in the original image if
    len(encrypted_mod_data) > num_pixels: encrypted_mod_data = encrypted_mod_data[:num_pixels]
    # Truncate if there's excess data
    elif len(encrypted_mod_data) < num_pixels:
        # Pad with random noise if there's insufficient data to prevent structure visibility
        encrypted_mod_data += [random.randint(0, 255) for _ in range(num_pixels - len(encrypted_mod_data))]

    # Reshape to the original image dimensions
    encrypted_array = np.array(encrypted_mod_data, dtype=np.uint8).reshape(image_shape)
    encrypted_image = Image.fromarray(encrypted_array)

    if output_filename:

    encrypted_image.save(output_filename)

    return encrypted_image

def load_image_as_bytes(image_path): image = Image.open(image_path).convert('L') # Convert
    image to grayscale for simplicity image_data = np.array(image)
    image_bytes = image_data.flatten().tolist() return
    image_bytes, image_data.shape

def save_decrypted_image(decrypted_data, image_shape, output_filename=None):
    decrypted_array = np.array(decrypted_data, dtype=np.uint8).reshape(image_shape) decrypted_image
    = Image.fromarray(decrypted_array)

    if output_filename:

    decrypted_image.save(output_filename)

    return decrypted_image

# Main program execution
p = 101 # Choose large primes for real-world security
q = 103 public_key, private_key = generate_keys(p, q)

print(f"Public Key: {public_key}") print(f"Private
Key: {private_key}")

# Load the image and get bytes
```

```
image_path = "girl.PNG" # Replace with your image path image_bytes, image_shape =
load_image_as_bytes(image_path)

# Encrypt the image encrypted_data =
encrypt_image(image_bytes, public_key)

# Save encrypted image for visualization (optional)
encrypted_image = save_encrypted_image(encrypted_data, image_shape) print("Encrypted
image generated.")

# Decrypt the image
decrypted_data = decrypt_image(encrypted_data, private_key)

# Save decrypted image
decrypted_image = save_decrypted_image(decrypted_data, image_shape) print("Decrypted
image generated.")

# Load and display images using PIL to visualize original_image =
Image.open(image_path)

# Display Original Image print("Displaying Original Image:")
display(original_image)

# Display Encrypted Image print("Displaying
Encrypted Image:") display(encrypted_image)

# Display Decrypted Image print("Displaying
Decrypted Image:") display(decrypted_image)
```



Output :

Compare the impact of different key sizes on the security and performance of the RSA algorithm. At what point does increasing key size yield diminishing returns in terms of security?

Impact on Security: Larger key sizes (e.g., 2048-bit or 4096-bit) provide greater security, making it harder for attackers to break the encryption through brute force or factorization attacks.

Impact on Performance: Larger keys slow down encryption and decryption processes, as both operations require more computational resources.

Diminishing Returns: Beyond 4096-bit keys, the increase in security is minimal, while performance degradation becomes more noticeable. For most applications, 2048-bit keys are secure enough, and going larger only benefits high-security environments.

Evaluate the benefits and drawbacks of using RSA in hybrid cryptosystems. How does combining RSA with symmetric encryption enhance overall security?

Benefits: Combining RSA with symmetric encryption (e.g., AES) creates a hybrid cryptosystem that leverages RSA for secure key exchange and AES for faster encryption and decryption of bulk data. This offers a good balance between security and efficiency.

Drawbacks: RSA alone is computationally expensive for large data encryption, and the combination adds complexity in terms of managing both keys (asymmetric and symmetric). However, RSA's strength in securely exchanging symmetric keys mitigates its inefficiency for bulk data encryption, providing a more secure solution.

Evaluate the effectiveness of RSA in real-world applications. In what scenarios does RSA excel, and where does it fall short compared to other cryptographic algorithms?

Where RSA Excels: RSA is excellent for secure key exchange, digital signatures, and authentication processes in applications like HTTPS, SSL/TLS, and email encryption (PGP).

Where RSA Falls Short: RSA is less efficient for encrypting large data compared to symmetric algorithms like AES, which are faster and more suited for bulk encryption. In such cases, RSA is typically used in hybrid systems for key exchange rather than direct data encryption.

CONCLUSION:

In this experiment we learnt to implement RSA algorithm

Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Information Technology

COURSE CODE: DJ19ITL501 **DATE:21-10-24**

COURSE NAME: Cryptography and Network Security Laboratory **CLASS:IT-1**

NAME: Diksha Velhal
SapID:60003220042

Experiment No. 8

CO/LO: CO1

Aim : For varying message sizes, test integrity of message using MD-5, SHA-1, and analyse the performance of the two protocols. Use crypt APIs

DESCRIPTION OF EXPERIMENT:

- Implement MD5 and SHA512 and analyse the performance of the two protocols and fill the comparison table.

	Criterias	**MD5**	**SHA 512**
	Input Size	Variable Length	Variable Length
	Output size	128 Bits (16 Bytes	512 Bits (20 Bytes

Department of Information Technology

		)	)
	Initialization vector size	4 Words (32 Bits Each)	16 Words (32 bits each)
	Version available in market	MD5	SHA-512
1	**Time taken for short msg** msg "Hi"	Fast, negligable	Slightly slower Than MD5 But still Fast

Department of Information Technology

2	**Time taken for moderate length msg one paragraph**"**Security Threats** Computer systems face a number of security threats. One of the basic threats is data loss, which means that parts of a database can no longer be retrieved. This could be the result of physical damage to the storage medium (like fire or water damage), human error or hardware failures.Another security threat is unauthorized access. Many computer systems contain sensitive information, and it could be very harmful if it were to fall in the wrong hands. Imagine someone getting a hold of your social security number, date of birth, address and bank information. Getting unauthorized access to computer systems is known as hacking. Computer hackers have developed sophisticated methods to obtain data from databases, which they may use for personal gain or to harm others.A third category of security threats consists of viruses and other harmful programs. A computer virus is a computer program that can cause damage to a computer's software, hardware or data. It is referred to as a virus because it has the capability to replicate itself and hide inside other computer files. "	MD5 is faster for short texts but slower for larger texts	SHA-512 is slower than MD5 but more secure for larger texts
3	**Time taken for long length one page msg**"Instructor: *Paul Zandbergen* Paul has a PhD from the University of British Columbia and has taught Geographic Information Systems, statistics and computer programming for 15 years.Computer systems face a number of security threats. Learn about different approaches to system security, including firewalls, data encryption, passwords and biometrics. Security Threats Computer systems face a number of security threats. One of the basic threats is data loss, which means that parts of a database can no longer be retrieved. This could be the result of physical damage to the storage medium (like fire or water damage), human error or hardware failures. Another security threat is unauthorized access. Many computer systems contain sensitive information, and it could be very harmful if it were to fall in the wrong hands. Imagine someone getting a hold of your social security number, date of birth, address and bank information. Getting unauthorized access to computer systems is known as hacking. Computer hackers have developed sophisticated methods to obtain data from databases, which they may use for personal gain or to harm others. A third category of security threats consists of viruses and other harmful programs. A computer virus is a computer program that can cause damage to a computer's	MD5 is relatively fast but has weaker security for large texts	SHA-512 is slower but more secure for larger texts

software, hardware or data. It is referred to as a virus because it has the capability to replicate itself and hide inside other computer files.

System Security

The objective of system security is the protection of information and property from theft, corruption and other types of damage, while allowing the information and property to remain accessible and productive. System security includes the development and implementation of security countermeasures. There are a number of different approaches to computer system security, including the use of a firewall, data encryption, passwords and biometrics.

Firewall

One widely used strategy to improve system security is to use a firewall. A firewall consists of software and hardware set up between an internal computer network and the Internet. A computer network manager sets up the rules for the firewall to filter out unwanted intrusions. These rules are set up in such a way that unauthorized access is much more difficult.

A system administrator can decide, for example, that only users within the firewall can access particular files, or that those outside the firewall have limited capabilities to modify the files. You can also set up a firewall for your own computer, and on many computer systems, this is built into the operating system.

Encryption

One way to keep files and data safe is to use encryption. This is often used when data is transferred over the Internet, where it could potentially be seen by others. Encryption is the process of encoding messages so that it can only be viewed by authorized individuals. An encryption key is used to make the message unreadable, and a secret decryption key is used to decipher the message.

Encryption is widely used in systems like e-commerce and Internet banking, where the databases contain very sensitive information. If you have made purchases online using a credit card, it is very likely that you've used encryption to do this.

Passwords

The most widely used method to prevent unauthorized access is to use passwords. A password is a string of characters used to authenticate a user to access a system. The password needs to be kept secret and is only intended for the specific user. In computer systems, each password is associated with a specific username since many individuals may be accessing the same system.

Department of Information Technology

	Good passwords are essential to keeping computer systems secure. Unfortunately, many computer users don't use very secure passwords, such as the name of a family member or important dates - things that would be relatively easy to guess by a hacker. One of the most widely used passwords - you guessed it - 'password.' Definitely not a good password to use.		

	So what makes for a strong password? • Longer is better - A long password is much harder to break. The minimum length should be 8 characters, but many security experts have started recommending 12 characters or more. • Avoid the obvious - A string like '0123456789' is too easy for a hacker, and so is 'LaDyGaGa'. You should also avoid all words from the dictionary. • Mix it up - Use a combination of upper and lowercase and add special characters to make a password much stronger. A password like 'hybq4' is not very strong, but 'Hy%Bq&4$' is very strong. Remembering strong passwords can be challenging. One tip from security experts is to come up with a sentence that is easy to remember and to turn that into a password by using abbreviations and substitutions. For example, 'My favorite hobby is to play tennis' could become something like Mf#Hi$2Pt%. Regular users of computer systems have numerous user accounts. Just consider how many accounts you use on a regular basis: email, social networking sites, financial institutions, online shopping sites and so on. A regular user of various computer systems and web sites will have dozens of different accounts, each with a username and password. To make things a little bit easier on computer users, a number of different approaches have been developed.		
	Your Comments on time taken for different length of msgs (point 1,2,3)	MD5 is faster	SHA-512 is slower
4	Msg “Hi” and msg “Ho” Msg “CSS” and Msg “DSS” (analyse one character change in input affects how many places in output)		

Department of Information Technology

	Your Comments on avalanche effect(point 4)	MD5 shows the avalanche effect with significant changes in output	SHA-512 also shows a strong avalanche effect with changes in output

5	**Consider same messages from point 1,2, and 3 check length of**		
6	**message digest generated**	32-characters	128-characters
7	**Your comments on length of msg**	MD5 is vulnerable to collisions (weaker security)	SHA-512 is more secure but also has collision vulnerabilities discovered later

Department of Information Technology

8	**Can you find two different messages with same message digest ?**	MD5 has known collisions	SHA-512 does not have practical collisions due to its stronger designs
9	**If message digest is given can you find original message**	Not possible(One way hash)	Not possible(One way hash)
10	**How many operations are required for birthday attack ?**	2^64 operations to find collision	2^256 operations to find a collision
11	**Which hash function you find it strong and why?** **SHA-512 is stronger in terms of fewer collisions but is slower.SHA-512 is generally preferred due to its higher security level over MD5**		

SOURCE CODE:
SHA-512 Code:

```
import hashlib


def sha1_hash(message):
    print(f"Original Message: {message}")
```

Department of Information Technology

```
# Step 1: Pad the message to make it 448 mod 512
message_bytes = bytearray(message.encode('utf-8')) # Convert message to bytearray
original_length = len(message_bytes) * 8
print(f"Original message length (in bits): {original_length}")

# Append padding bits message_bytes.append(0x80) # Append a single "1" bit (0x80
is 10000000 in binary) while (len(message_bytes) * 8) % 512 != 448:
    message_bytes.append(0) # Pad with "0" bits
print(f"Message length after padding (in bits): {len(message_bytes) * 8}")

#  Step  2:  Append  the  original  message  length  as  a  64-bit  big-endian  integer
message_bytes += original_length.to_bytes(8, byteorder='big')
print(f"Message length after appending length (in bits): {len(message_bytes) * 8}")

# Step 3: Initialize buffer
h0 = 0x67452301 h1 =
0xEFCDAB89 h2 = 0x98BADCFE
h3 = 0x10325476 h4 =
0xC3D2E1F0

# Step 4: Process the message in 512-bit chunks for
i in range(0, len(message_bytes), 64):
    chunk = message_bytes[i:i+64] w = [0] * 80 for j in
    range(16): w[j] = int.from_bytes(chunk[4 * j:4 * j + 4],
    'big')
    for j in range(16, 80):
        w[j] = (w[j-3] ^ w[j-8] ^ w[j-14] ^ w[j-16]) & 0xFFFFFFFF

    a, b, c, d, e = h0, h1, h2, h3, h4
    for j in range(80): if 0 <= j <=
    19:

            f = (b & c) | (~b & d)
            k = 0x5A827999
        elif 20 <= j <=
            39: f = b ^ c
            ^ d k =
            0x6ED9EBA1
        elif 40 <= j <= 59: f = (b & c) |
            (b & d) | (c & d) k =
            0x8F1BBCDC
        else: f = b ^ c ^
            d      k      =
            0xCA62C1D6
```

```
            temp = (a << 5 | a >> 27) + f + e + k + w[j] &
            0xFFFFFFFF e = d d = c c = b << 30 | b >> 2 b = a a =
            temp

    # Step 5: Output hash value
    h0 = (h0 + a) & 0xFFFFFFFF
    h1 = (h1 + b) & 0xFFFFFFFF
    h2 = (h2 + c) & 0xFFFFFFFF
    h3 = (h3 + d) & 0xFFFFFFFF
    h4 = (h4 + e) & 0xFFFFFFFF

    hash_value = (h0 << 128) | (h1 << 96) | (h2 << 64) | (h3 << 32) | h4
    print(f"Final hash: {hash_value.to_bytes(20, 'big').hex()}")

# Example usage with user input
message = input("Enter the message to be hashed: ")
sha1_hash(message)
```

Output:

```
Enter the message to be hashed: my name is Isha
Original Message: my name is Isha
Original message length (in bits): 120
Message length after padding (in bits): 448
Message length after appending length (in bits): 512
Final hash: 53fa2998400fc515e18b70a2983360939453957b
```

MD5 Code:

```
import struct
from math import sin # Import sin function from math module

# Define MD5 helper functions and constants def
left_rotate(x, amount):
    """Left rotates x by amount bits""" x
    &= 0xFFFFFFFF
    return ((x << amount) | (x >> (32 - amount))) & 0xFFFFFFFF

# Define MD5 auxiliary functions def
F(x, y, z):
    return (x & y) | (~x & z)



def G(x, y, z):
    return (x & z) | (y & ~z)

def H(x, y, z):
```

Department of Information Technology

```
    return x ^ y ^ z

def I(x, y, z):
    return y ^ (x | ~z)

# MD5 main function def
md5(message):
    #   Step   1:   Padding   the   message
    original_byte_len     =     len(message)
    original_bit_len = original_byte_len * 8

    # Add padding
    message = bytearray(message.encode('utf-8'))
    message.append(0x80) # Append '1' bit (10000000 in binary)

    while (len(message) * 8) % 512 != 448:
        message.append(0) # Append '0' bits

    # Step 2: Append length of the original message (in bits) as a 64-bit little-endian
integer message += struct.pack('<Q', original_bit_len)

    # Step 3: Initialize MD buffer (A, B, C, D)
    A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476

    # Constants for MD5 (T values) derived from the sine function
    T = [int(4294967296 * abs(sin(i + 1))) & 0xFFFFFFFF for i in range(64)]

    # Step 4: Process the message in 512-bit chunks for
    chunk_offset in range(0, len(message), 64):
        chunk = message[chunk_offset:chunk_offset + 64]
        M = [struct.unpack('<I', chunk[i:i + 4])[0] for i in range(0, 64, 4)]

        AA, BB, CC, DD = A, B, C, D

        # Main MD5 loop for
        i in range(64):
            if 0 <= i <= 15: f =
                F(B, C, D) g = i s
                = [7, 12, 17, 22]
            elif 16 <= i <= 31: f =
                G(B, C, D) g = (5 *
```

Department of Information Technology

```
            i + 1) % 16 s = [5,
            9, 14, 20]
        elif 32 <= i <= 47: f =
            H(B, C, D) g = (3 *
            i + 5) % 16


            s = [4, 11, 16, 23]
        elif 48 <= i <= 63:
            f = I(B, C, D)
            g = (7 * i) %
            16
            s = [6, 10, 15, 21]


        f = (f + A + T[i] + M[g]) & 0xFFFFFFFF
        A, B, C, D = D, (B + left_rotate(f, s[i % 4])) & 0xFFFFFFFF, B, C

    # Add the chunk's hash to the current result
    A = (A + AA) & 0xFFFFFFFF
    B = (B + BB) & 0xFFFFFFFF
    C = (C + CC) & 0xFFFFFFFF
    D = (D + DD) & 0xFFFFFFFF

  # Step 5: Output the final MD5 hash (digest)
  md5_digest = struct.pack('<4I', A, B, C, D)
  print(f"Final MD5 hash: {md5_digest.hex()}")

# Example usage with user input
message = input("Enter the message to be hashed:
") md5(message)
```

Output:

```
Enter the message to be hashed: my name is Isha
Final MD5 hash: 70a28364498a64ac860301237ca780fe
```

OBSERVATIONS AND CONCLUSION:

In this experiment , we compared MD5 and SHA-512 on various parameters

References:

[1] Behrouz A. Forouzan., “Cryptography and Network Security”, 2nd Edition, McGraw Hill Education 2010.

[2] Willam Stallings, “Cryptography and Network Security”, 5th Edition, Pearson

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJS22ITL504 **DATE:20-10-24**

COURSE NAME: Cryptography and Network Security Laboratory CLASS: TYBTech
Name : Diksha Velhal
SAP: 60003220042

EXPERIMENT NO. 9

CO/LO: Design secure system using appropriate security mechanism

AIM / OBJECTIVE:

To implement Diffie Hellman (client server)

DESCRIPTION OF EXPERIMENT:

- Implement 1 client and 1 server and show successful key sharing between both parties
- Implement 2 client and 1 server where 1 client is Alice, 1 is eve and server is bob. Simulate man in the middle attack and show key sharing between alice-eve and eve-bob

SOURCE CODE:

```
import socket
import
secrets
```

```
# Diffie-Hellman parameters
     (public values) p = 23 # A   prime
number
```

```
# Server's private key
(randomly generated)
```

```
# Server's public key (A = g^a % p)
server_public_key = mod_exp(g,
```

```
#  Create  server  socket  server_socket  =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

```
print("Server is waiting for a
client connection...") connection,
```

```
  try:
print(f"Connected to {client_address}")

# Send public values p and g

# Send server's public key (A) to client
connection.sendall(str(server_public_key).encode())

# Receive client's public key (B) client_public_key
=                 int(connection.recv(1024).decode())
print(f"Received      client's      public      key:
{client_public_key}")

# Compute shared secret key (s = B^a % p)
shared_secret_key = mod_exp(client_public_key,
server_private_key, p) print(f"Shared secret key
(server): {shared_secret_key}")
```

```
import socket
import
secrets

# Create client socket
client_socket    =    socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

# Receive public values p and g from server
p_g_data =
client_socket.recv(1024).decode().sp

# Client's private key
(randomly generated)

# Client's public key (B = g^b % p)
client_public_key = mod_exp(g,

# Receive server's public key (A)
server_public_key =
int(client_socket.recv(1024).decode())

# Send client's public key (B) to server

# Compute shared secret key (s = A^b % p)
shared_secret_key = mod_exp(server_public_key,
client_private_key, p) print(f"Shared secret key
```

```
client_socket.close()
```

```
Server is waiting for a client connection...
Connected to ('127.0.0.1', 56074)
Received client's public key: 4
Shared secret key (server): 2
PS C:\Users\kalpe\Downloads\cns>
```

```
PS C:\Users\kalpe\Downloads\cns> python client.py
Received server's public key: 8
Shared secret key (client): 2
```

```
import socket

import random

# Diffie-
Hellman

# Generate Bob's private key
bob_private_key =

# Calculate Bob's public key bob_public_key =
(alpha ** bob_private_key) % q

# Setup Bob's server
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
server_socket.bind(('localhost', 8081)) # Use port

print("Bob (Server) waiting for connection...")

conn, addr = server_socket.accept()

# Receive Eve's (pretending to be
    Alice) public key eve_public_key =
  int(conn.recv(1024).decode())
# Send Bob's public key to Eve (pretending to be
    Alice) conn.send(str(bob_public_key).encode())
print(f"Bob's Public Key: {bob_public_key}")
# Calculate shared secret key with Eve
(thinking it's Alice) shared_key_bob =
(eve_public_key ** bob_private_key) % q
```

```
conn.close()
```

```
import socket
import
random
```

```
# Generate Eve's private key
eve_private_key =
```

```
# Calculate Eve's public key eve_public_key =
(alpha ** eve_private_key) % q
```

```
# Setup Eve as middleman eve_socket_alice =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM) eve_socket_bob =
```

```
# Connect to Alice (client 1)
eve_socket_alice.bind(('localhost', 8080)) # Use
port 8080 for Eve-Alice eve_socket_alice.listen(1)
```

```
conn_alice,                addr_alice           =
eve_socket_alice.accept()
```

```
# Receive Alice's public key
    alice_public_key =
    int(conn_alice.recv(1024).decode())


    # Send Eve's public key (pretending it's Bob's) to
    Alice conn_alice.send(str(eve_public_key).encode())
    print(f"Eve's Public Key sent to Alice (pretending to be Bob):
    # Connect to Bob (server)

     eve_socket_bob.connect(('localhost', 8081)) # Connect to Bob's server on port
    # Send Eve's public key (pretending it's Alice's)
    to Bob

    eve_socket_bob.send(str(eve_public_key).encode())

    # Receive Bob's public key bob_public_key
    =
    int(eve_socket_bob.recv(1024).decode())


    # Send Bob's public key to Alice
    (pretending it's from Bob)
    conn_alice.send(str(bob_public_key).enco
```

```
    # Calculate shared keys
    shared_key_with_alice = (alice_public_key **
    eve_private_key) % q shared_key_with_bob =

    print(f"Eve's Shared Key with Alice:
    {shared_key_with_alice}") print(f"Eve's Shared Key

    # Close
    connections
    conn_alice.clos
```

```
import socket
import
random

# Generate Alice's private key alice_private_key
=

# Calculate Alice's public key
alice_public_key = (alpha ** alice_private_key) % q

# Setup Alice as a client
client_socket    =    socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

# Send Alice's public key to Eve
(thinking it's Bob)
client_socket.send(str(alice_public

# Receive Eve's public key (pretending to be
      Bob)             fake_bob_public_key          =
   int(client_socket.recv(1024).decode())
# Calculate shared secret key (thinking it's with
Bob) shared_key_alice =
(fake_bob_public_key ** alice_private_key) % q

client_socket.close()
```

```
Bob (Server) waiting for connection...
Connected by ('127.0.0.1', 56032)
Received Eve's (as Alice) Public Key: 10
Bob's Public Key: 15
Shared Key at Bob: 17
```

```
Eve waiting for Alice...
Connected to Alice at ('127.0.0.1', 56031)
Received Alice's Public Key: 12
Eve's Public Key sent to Alice (pretending to be Bob): 10
Eve's Public Key sent to Bob (pretending to be Alice): 10
Received Bob's Public Key: 15
Bob's Public Key forwarded to Alice
Eve's Shared Key with Alice: 3
Eve's Shared Key with Bob: 17
```

```
PS C:\Users\kalpe\Downloads\cns> python Alice.py
Alice's Public Key: 12
Received Bob's (actually Eve's) Public Key: 10
Shared Key at Alice: 3
```

CONCLUSION: Thus we learnt to implement Diffie Hellman (client server)

QUESTIONS:

1. Show the working of Diffie Hellman with the help of an example
2. Why is diffie-hellman vulnerable to man in the middle attack? What are the countermeasures.

REFERENCES