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:**

1. Implementation of Ceaser Cipher on alphanumeric data.
2. 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:

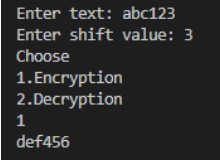
c=ord(i)-48

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:**



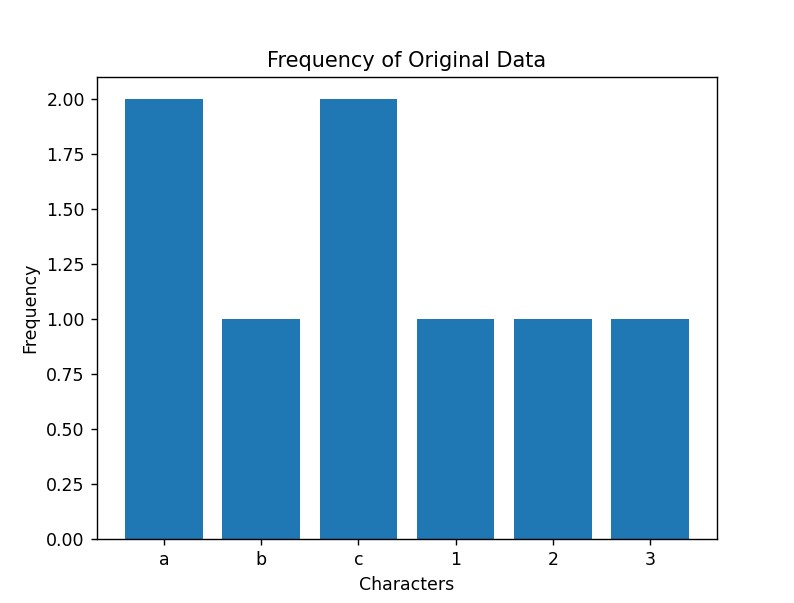
**QUESTIONS:**

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

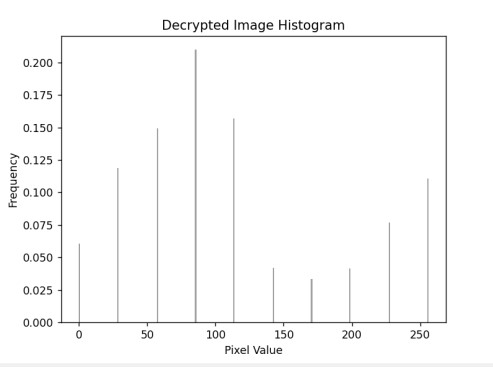
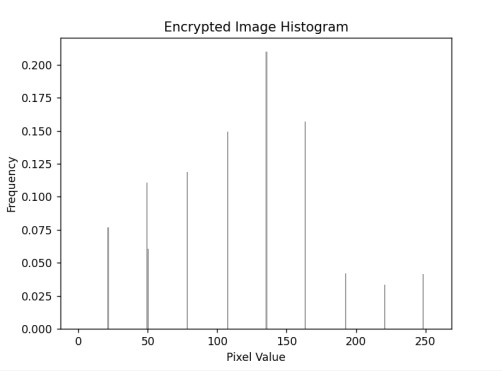
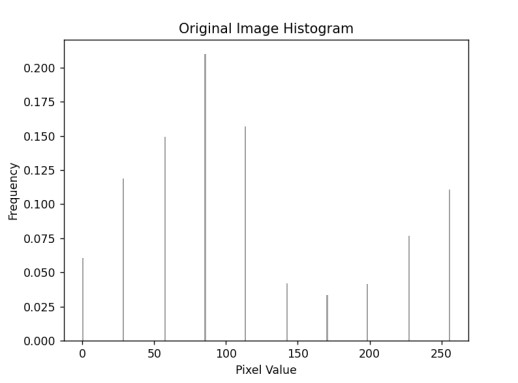
|  |
| --- |
| def encryp(text, key):  a = '' for char in text: if char.isalpha():  shift = 65 if char.isupper() else 97 a += chr((ord(char) - shift + key) % 26 + shift) elif char.isdigit():  a += chr((ord(char) - 48 + key) % 10 + 48) else: a += char return a  key = 3 ans="abcac123" final = encryp(ans,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





1. 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.

# DEPARTMENT OF INFORMATION TECHNOLOGY

**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 w](https://www.geeksforgeeks.org/caesar-cipher/)e 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.

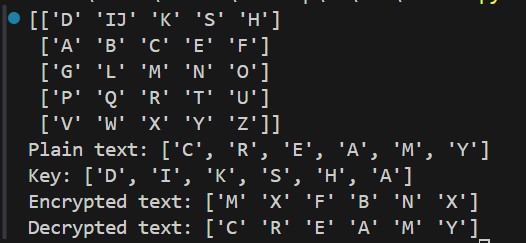
* **SOURCE CODE:**

|  |
| --- |
| 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:**



**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:**

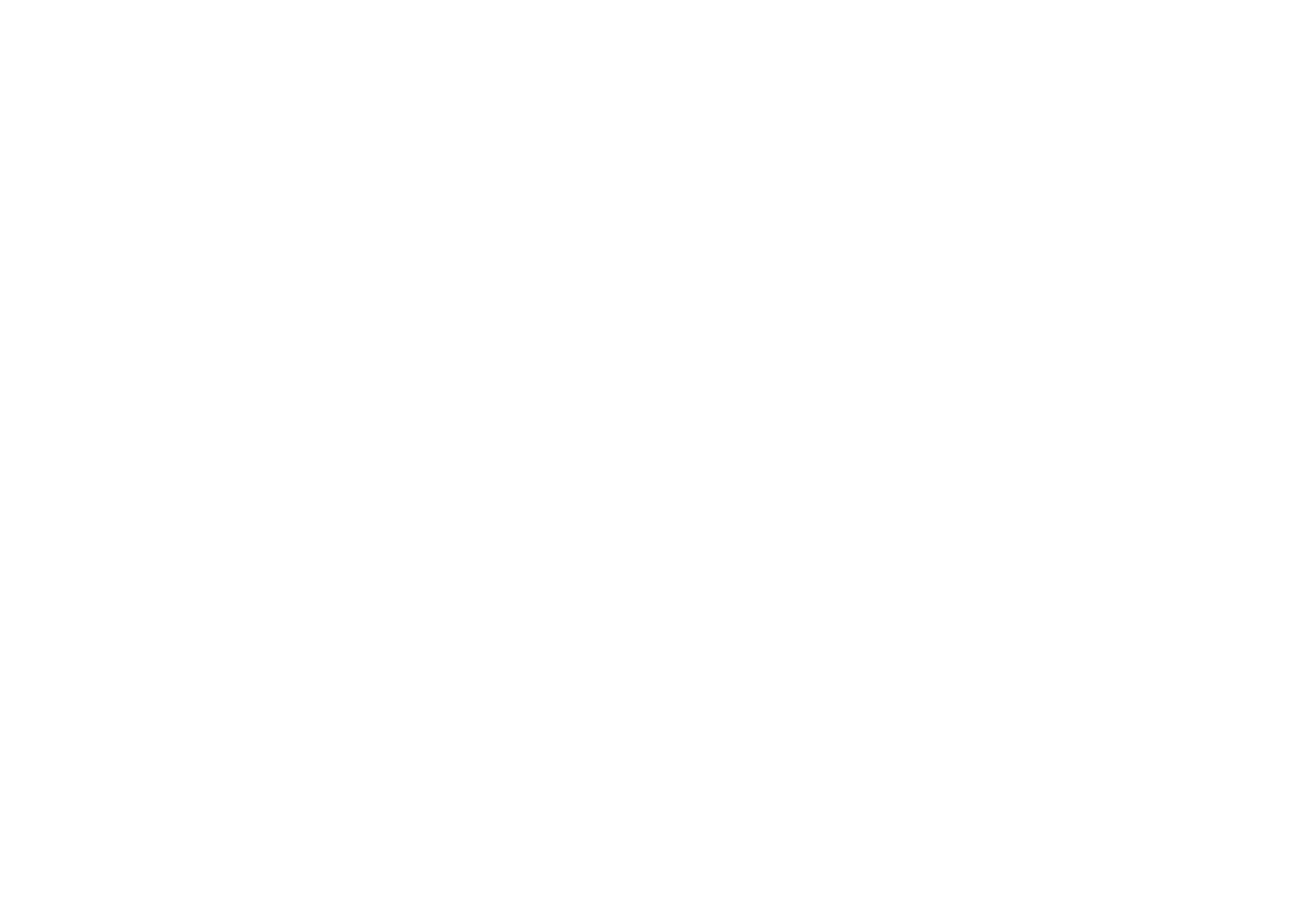
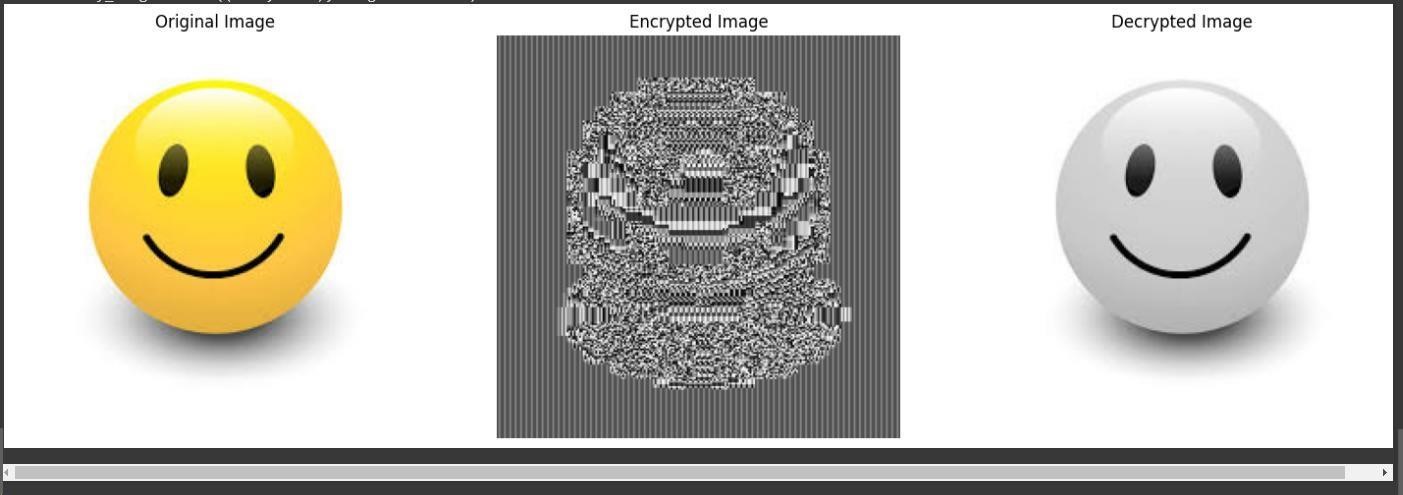
|  |
| --- |
| import numpy as np  from PIL import Image import matplotlib.pyplot as plt from google.colab import files import io  def text\_to\_matrix(text, size):  """Convert text to a matrix of numbers.""" text = text.upper().replace(" ", "") if len(text) != size \* size:  raise ValueError(f"Text 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()

1. **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.") |

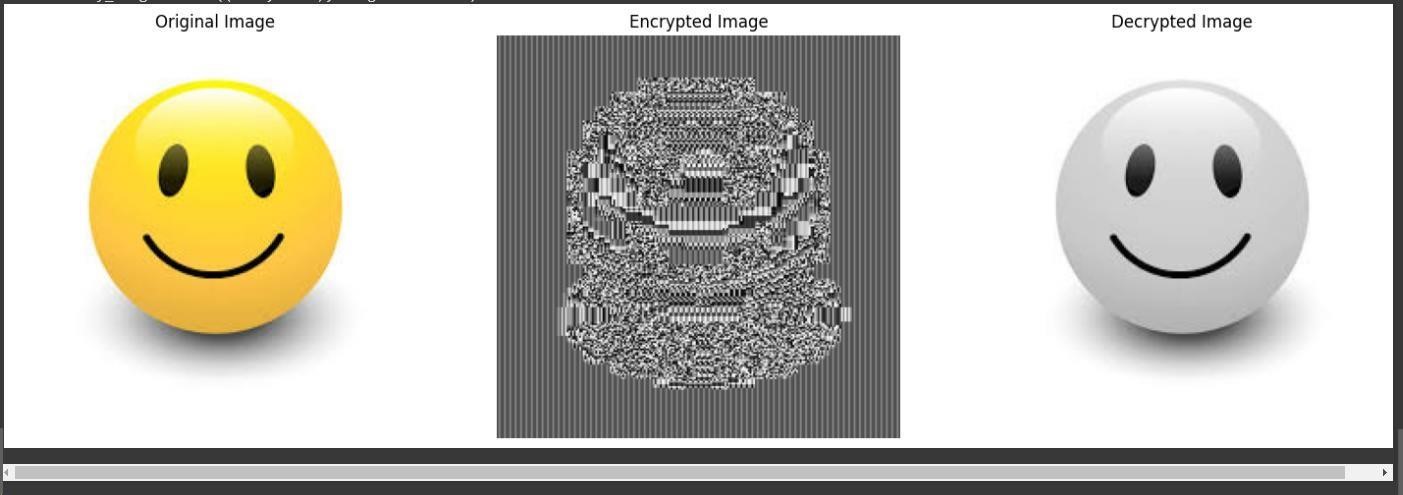
uploaded\_key\_image = files.upload()

# 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.**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**COURSE CODE: DJS22ITL504 DATE:3-9-24**

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

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

# 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 :**

|  |  |  |
| --- | --- | --- |
|  | | |
| [3, 1,  S1 = [[0, 1,  [2, 0,  [3, 0,  [2, 1, | 3, 2]]  2, 3],  1, 3],  1, 0],  0, 3]] | bits, |
| # Utility functions (same as before) def permute( 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 :**



1. **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] |

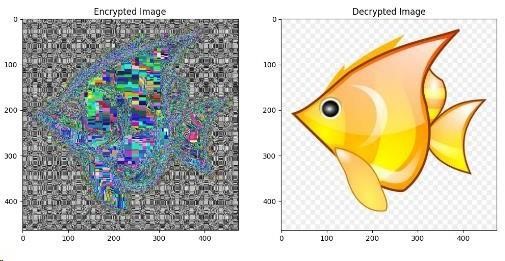
P4 = [2, 4, 3, 1]

|  |
| --- |
| 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.

# 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

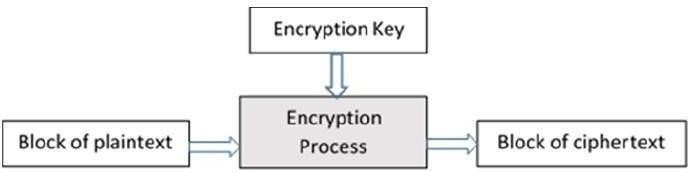
**CO/LO:** Design secure system using appropriate security mechanism

**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) |

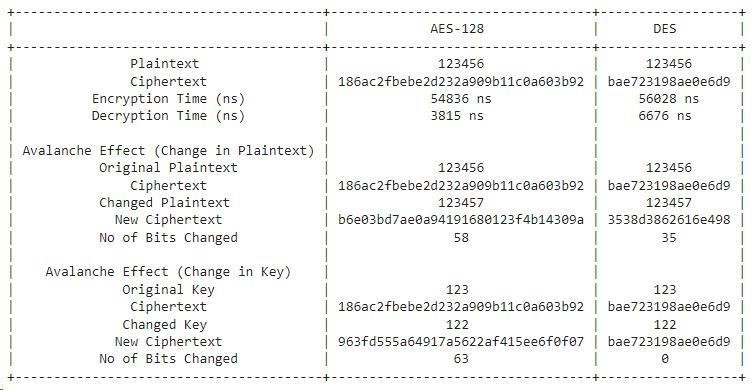
|  |
| --- |
| # AES encryption and decryption function def aes\_encrypt\_decrypt(plaintext, key):  key = key.ljust(16, '0').encode('utf-8') # Ensure key is 16 bytes cipher = Cipher(algorithms.AES(key), modes.ECB())  encryptor = cipher.encryptor() decryptor = cipher.decryptor()  # Padding plaintext to 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 :**



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 wiin 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

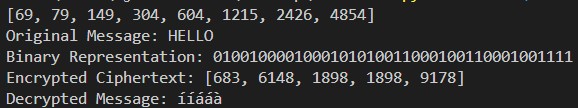
back

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 |

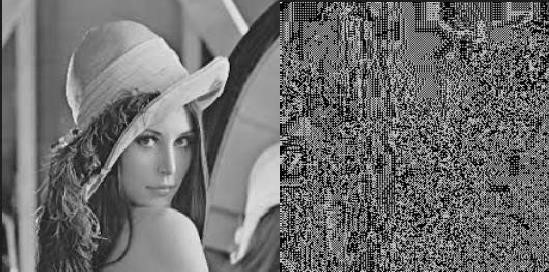
|  |
| --- |
| # Generate the public key and private key public\_key = generate\_super\_increasing\_sequence(n) private\_key = generate\_private\_key(public\_key, q, r)  # Plaintext message plaintext = "HELLO"  binary\_plaintext = string\_to\_binary(plaintext)  # Encrypt each byte of the binary plaintext 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) |



|  |
| --- |
| 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.

# 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:**

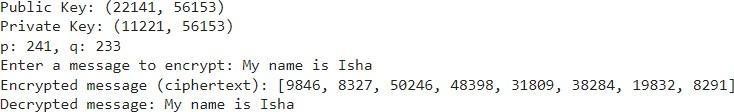
**Code for Text :**

|  |
| --- |
| 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,

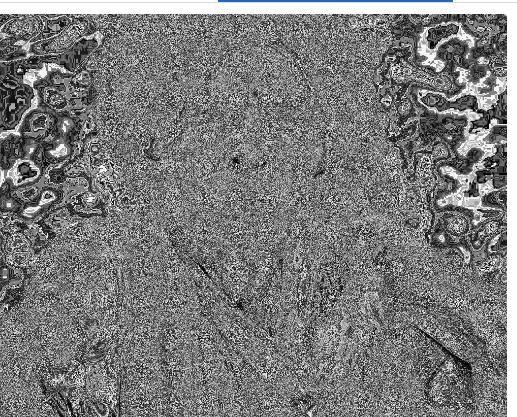


**Code for Image :**

|  |
| --- |
| 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

**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** |  | Varia ble Lengt h | Variab  le  Lengt h |
|  | **Output size** |  | 128 | 512 |
|  |  |  | Bits | Bits |
|  |  |  | (16 | (20 |
|  |  |  | Bytes | Bytes |
|  |  |  | ) | ) |
|  | **Initialization vector size** |  | 4 | 16 |
|  |  |  | Word | Words |
|  |  |  | s (32 | (32 |
|  |  |  | Bits | bits |
|  |  |  | Each) | each) |
|  | **Version available in market** |  | MD5 | SHA- 512 |
| **1** | **Time taken for short msg** |  | Fast, | Slightl |

|  |  |  |  |
| --- | --- | --- | --- |
|  | msg “Hi” | neglig able | y slower Than  MD5  But  still  Fast |
| **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 slowe  r 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 relativ ely fast but has  weak  er securi  ty 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.  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 ) |  |  |
|  | **Your Comments on avalanche effect(point 4)** | MD5 show s the avala nche  effect with  signifi cant chang es in  outpu  t | SHA- 512 also  shows a strong avalan  che  effect with  chang es in output |

|  |  |  |  |
| --- | --- | --- | --- |
| **5** | **Consider same messages from point 1,2, and 3 check length of** |  |  |
| **6** | **message digest generated** | 32- chara cters | 128- charac ters |
| **7** | **Your comments on length of msg** | MD5  is  vulner able to collisi  ons (weak  er  securi ty) | SHA-  512 is more secure but also  has  collisio n  vulner  abilitie s discov  ered later |
| **8** | **Can you find two different messages with same message digest ?** | MD5 has know n  collisi  ons | SHA- 512 does not have  practic al  collisio ns due  to its  strong  er  design s |
| **9** | **If message digest is given can you find original message** | Not possi  ble(  One way hash) | Not possib  le(  One way hash) |
| **10** | **How many operations are required for birthday attack ?** | 2^64 opera tions to find collisi  on | 2^256 operat ions to find a collisio n |
| **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}")

# 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

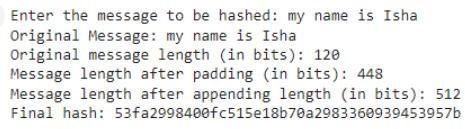
# Step 5: Output hash value h0 = (h0 + a) & 0xFFFFFFFF h1 = (h1 + b) & 0xFFFFFFFF h2 = (h2 + c) & 0xFFFFFFFF h3 = (h3 + d) & 0xFFFFFFFF h4 = (h4 + e) & 0xFFFFFFFF

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:**



**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):

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)

1. = 0x67452301
2. = 0xEFCDAB89
3. = 0x98BADCFE
4. = 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 \* 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

1. = (A + AA) & 0xFFFFFFFF
2. = (B + BB) & 0xFFFFFFFF
3. = (C + CC) & 0xFFFFFFFF
4. = (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:**



**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,

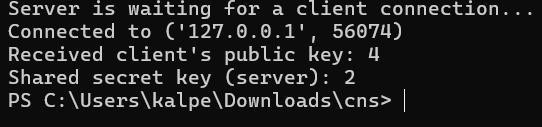
# Receive server's public key (A) server\_public\_key =

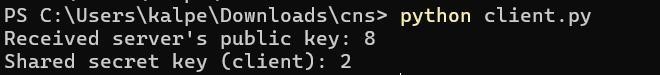
int(client\_socket.recv(1024).decode())

# 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()





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

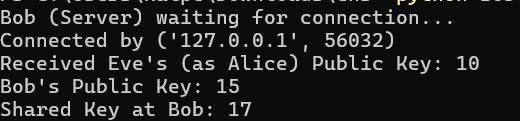
# Receive Eve's public key (pretending to be

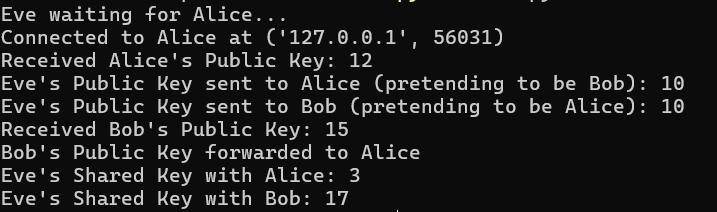
Bob) fake\_bob\_public\_key = int(client\_socket.recv(1024).decode())

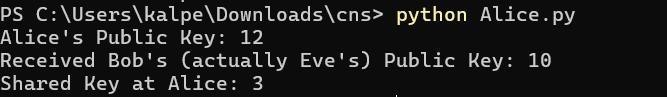
# Calculate shared secret key (thinking it's with Bob) shared\_key\_alice =

(fake\_bob\_public\_key \*\* alice\_private\_key) % q

client\_socket.close()







**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**