**Cryptography and Network Security**

**Practical List**

|  |  |
| --- | --- |
| **Sr. No.** | **Name of Particles** |
| 1 | Overview of Cryptography and Network Security. |
| 2 | To write a program to implement Playfair cipher for encryption and decryption. |
| 3 | To write a program to implement Hill cipher for encryption and decryption. |
| 4 | To write a program to implement DES algorithm. |
| 5 | To write a program to implement Blowfish Algorithm. |
| 6 | To write a program to implement AES algorithm. |
| 7 | To write a program to implement Diffie-Hellman key exchange technique for symmetric Cryptography. |
| 8 | To write a program to implement RSA Algorithm. |
| 9 | To write a program to implement signature and digital signature technique. |
| 10 | To write a program to implement SHA algorithm. |

**Practical 1**

**Aim – Overview of Cryptography and Network Security**

**Theory –**

**Cryptography**

Cryptography is the practice and study of techniques for secure communication in the presence of third parties called adversaries. It involves developing and analyzing protocols that prevent malicious third parties from retrieving information being shared between two entities.

**Principles of Cryptography**

The primary goals of cryptography are:

* **Confidentiality:** Ensuring that only authorized parties can access and understand data.
* **Integrity:** Protecting data from unauthorized modification or corruption.
* **Availability:** Guaranteeing that data and systems are accessible when needed.
* **Authentication:** Verifying the identity of communicating parties.
* **Non-repudiation:** Preventing parties from denying previous actions or commitments.

**Diagram and Explanation**

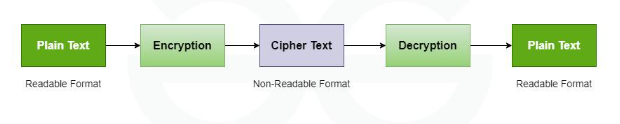


Fig: Basic Cryptography Process

The diagram illustrates the fundamental process of cryptography:

* **Plaintext:** The original, readable data.
* **Encryption:** The process of converting plaintext into ciphertext using an encryption algorithm and a key.
* **Ciphertext:** The encrypted, unreadable data.
* **Decryption:** The process of converting ciphertext back into plaintext using the decryption algorithm and the key.

**Importance of Cryptography**

Cryptography is essential for protecting sensitive information in today's digital world. Its applications include:

* **Secure communication:** Protecting data transmitted over networks.
* **Data protection:** Safeguarding data at rest (e.g., in databases).
* **Digital signatures:** Verifying the authenticity and integrity of digital documents.
* **Authentication:** Verifying the identity of users and systems.

**Cryptography Attacks**

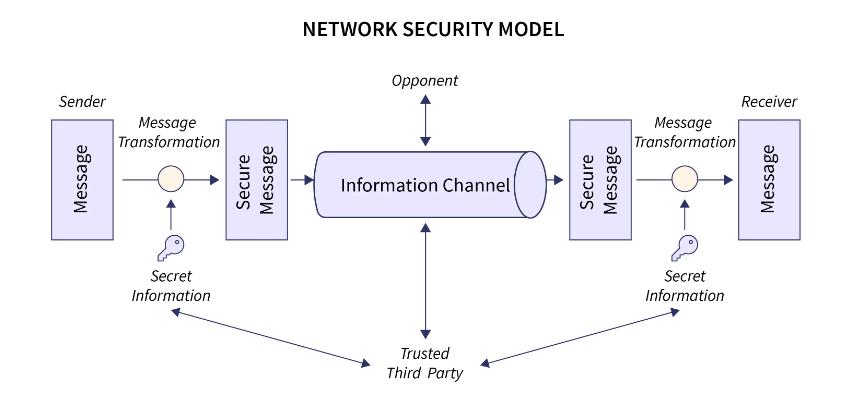
Cryptography attacks aim to compromise the security of cryptographic systems. Common types include:

* **Ciphertext-only attack:** The attacker has access only to the ciphertext.
* **Known-plaintext attack:** The attacker has access to both plaintext and corresponding ciphertext.
* **Chosen-plaintext attack:** The attacker can choose plaintexts and obtain the corresponding ciphertexts.
* **Man-in-the-middle attack:** An attacker intercepts and modifies communication between two parties.

**Network Security**

Network security involves protecting a computer network and its data from unauthorized access, use, disclosure, disruption, modification, or destruction. It encompasses hardware, software, and procedural measures.

**Network Security Model**

****

The network security model represents the secure communication between sender and receiver. This model depicts how the security service has been implemented over the network to prevent the opponent from causing a threat to the authenticity or confidentiality of the data that is being communicated through the network.

* Network security covers a huge amount of technologies, devices, and processes
* In simple words, it is a set of rules and regulations designed for protecting and securing the integrity, confidentiality, and accessibility of data and computer networks.
* The most common example of network security is password protection which was chosen by itself.

**Security Attacks**

Network security attacks can be classified into:

* **Passive attacks:** A Passive attack attempts to learn or make use of information from the system but does not affect system resources.
  + **Eavesdropping:** Unauthorized interception of network traffic to capture sensitive data.
  + **Traffic analysis:** Analyzing network traffic patterns to infer information.
* **Active attacks:** Active attacks are a type of cybersecurity attack in which an attacker attempts to alter, destroy, or disrupt the normal operation of a system or network.
* **Masquerade:** Impersonating a legitimate entity to gain unauthorized access.
* **Replay attack:** Reusing previously captured data to gain unauthorized access.
* **Denial of Service (DoS) attack:** Overwhelming a system or network with excessive traffic to render it unavailable.

**Importance of Network Security**

Network security is crucial for protecting sensitive data, maintaining business operations, and safeguarding against cyber threats. It helps build trust with customers and partners.

**Applications of Network Security**

Network security is applied in various domains:

* **E-commerce:** Protecting online transactions.
* **Online banking:** Securing financial information.
* **Healthcare:** Protecting patient data.
* **Government:** Safeguarding national security information.

**Conclusion –**

Cryptography and network security are fundamental to protecting digital assets and ensuring secure communication. They are essential components of modern information systems and require ongoing attention to address evolving threats.

**Practical 2**

**Aim – To write a program to implement Playfair cipher for encryption and decryption.**

**Theory –**

The Playfair cipher is a polygraphic substitution cipher that encrypts pairs of letters (digraphs) instead of single letters.

It was invented by Charles Wheatstone in 1854 and was used by the British military during the Boer War.

1. **Key Generation:**
   * Create a 5x5 square grid (matrix) filled with the letters of the alphabet, excluding the letter 'J' (which is considered equivalent to 'I').
   * Fill the remaining empty spaces with the remaining letters of the alphabet, starting with the letter 'I'.
   * The key is the 5x5 square grid.
2. **Encryption:**
   * Break the plaintext into pairs of letters. If there is an odd number of letters, add a 'X' to the end.
   * For each pair of letters:
     + If the two letters are in the same row of the key, replace each letter with the letter to its right (wrapping around if necessary).
     + If the two letters are in the same column of the key, replace each letter with the letter below it (wrapping around if necessary).
     + If the two letters are not in the same row or column, find the rectangle formed by the two letters. Replace each letter with the letter diagonally opposite it in the rectangle.
3. **Decryption:**
   * The decryption process is the reverse of the encryption process.
   * For each pair of ciphertext letters:
     + If the two letters are in the same row of the key, replace each letter with the letter to its left (wrapping around if necessary).
     + If the two letters are in the same column of the key, replace each letter with the letter above it (wrapping around if necessary).
     + If the two letters are not in the same row or column, find the rectangle formed by the two letters. Replace each letter with the letter diagonally opposite it in the rectangle.
     + Remove any 'X' letters added during encryption.

**Program –**

# Function to generate the Playfair cipher key matrix

def generate\_key\_matrix(key):

key = key.upper().replace("J", "I") # Replace 'J' with 'I' as Playfair usually uses a 5x5 matrix

key\_matrix = []

seen = set()

# Add key characters to the matrix

for char in key:

if char not in seen and char.isalpha():

seen.add(char)

key\_matrix.append(char)

# Add remaining letters of the alphabet to the matrix

for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ": # 'J' is omitted

if char not in seen:

seen.add(char)

key\_matrix.append(char)

# Convert the list into a 5x5 matrix

return [key\_matrix[i:i+5] for i in range(0, 25, 5)]

# Function to find the position of a letter in the key matrix

def find\_position(matrix, char):

for i, row in enumerate(matrix):

if char in row:

return i, row.index(char)

return None

# Function to preprocess the plaintext (handle repeating characters and add filler)

def preprocess\_text(plaintext):

plaintext = plaintext.upper().replace("J", "I") # Replace 'J' with 'I'

processed = ""

i = 0

while i < len(plaintext):

char1 = plaintext[i]

if i + 1 < len(plaintext):

char2 = plaintext[i + 1]

else:

char2 = 'X' # Add filler if it's the last letter

if char1 == char2:

processed += char1 + 'X' # Insert 'X' between repeated letters

i += 1

else:

processed += char1 + char2

i += 2

if len(processed) % 2 != 0:

processed += 'X' # Add filler if the text length is odd

return processed

# Function to encrypt/decrypt a digraph (pair of letters)

def process\_digraph(matrix, char1, char2, encrypt=True):

row1, col1 = find\_position(matrix, char1)

row2, col2 = find\_position(matrix, char2)

if row1 == row2: # Same row

if encrypt:

return matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]

else:

return matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]

elif col1 == col2: # Same column

if encrypt:

return matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]

else:

return matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]

else: # Rectangle case

return matrix[row1][col2] + matrix[row2][col1]

# Function to encrypt plaintext using Playfair cipher

def encrypt(plaintext, key):

key\_matrix = generate\_key\_matrix(key)

plaintext = preprocess\_text(plaintext)

ciphertext = ""

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

ciphertext += process\_digraph(key\_matrix, plaintext[i], plaintext[i + 1], encrypt=True)

return ciphertext

# Function to decrypt ciphertext using Playfair cipher

def decrypt(ciphertext, key):

key\_matrix = generate\_key\_matrix(key)

plaintext = ""

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

plaintext += process\_digraph(key\_matrix, ciphertext[i], ciphertext[i + 1], encrypt=False)

return plaintext

# Main program

def main():

print("Playfair Cipher")

key = input("Enter the key: ")

choice = input("Encrypt or Decrypt (e/d): ").lower()

if choice == 'e':

plaintext = input("Enter the plaintext: ")

ciphertext = encrypt(plaintext, key)

print(f"Ciphertext: {ciphertext}")

elif choice == 'd':

ciphertext = input("Enter the ciphertext: ")

plaintext = decrypt(ciphertext, key)

print(f"Plaintext: {plaintext}")

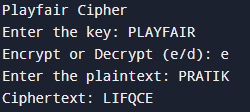
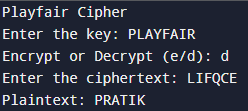
else:

print("Invalid choice. Please enter 'e' to encrypt or 'd' to decrypt.")

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

main()

**Output –**

**Conclusion –**

Hence, we have performed the Playfair cipher implementation for encryption and decryption successfully.

**Practical 3**

**Aim – To write a program to implement Hill cipher for encryption and decryption.**

**Theory –**

The Hill Cipher uses a polygraphic substitution cipher, which means homogeneous substitution over many levels of blocks.

This polygraphic substitution cipher allows Hill Cipher to function easily with digraphs (two-letter blocks), trigraphs (three-letter blocks), or any other multiple-sized blocks to create a uniform cipher.

Hill Cipher is based on linear algebra, advanced matrices (matrix multiplication and matrix inverses), and modulo arithmetic principles. Obviously, it is a more mathematical cipher than others.

Hill Cipher is also a block cipher. A block cipher uses a deterministic algorithm and a symmetric key to encrypt a block of text.

Unlike stream ciphers, it does not require encrypting one bit at a time. Hill Cipher is a block cipher, which means it can function with any block size.

While Hill Cipher is digraphic in nature, it can grow to multiply any letter size, adding complexity and reliability for improved usage.

Because most of Hill Ciphers' problems and solutions are mathematical in nature, it is simple to hide letters with precision.

**Encryption Process:**

1. **Convert to Numerical Values:** Convert each plaintext letter to a numerical value (e.g., A=0, B=1, ..., Z=25).
2. **Form Plaintext Vectors:** Divide the plaintext into groups of letters and represent each group as a column vector.
3. **Matrix Multiplication:** Multiply each plaintext vector by the key matrix. The result is the corresponding ciphertext vector.
4. **Convert to Ciphertext:** Convert the numerical values in the ciphertext vector back to letters.
5. Encrypting using the Hill cipher depends on the following operations –

E(K, P) = (K\*P) mod 26

Here K is our key matrix, and P is the vectorized plaintext.

**Decryption Process:**

1. **Convert to Numerical Values:** Convert each ciphertext letter to a numerical value.
2. **Form Ciphertext Vectors:** Divide the ciphertext into groups of letters and represent each group as a column vector.
3. **Inverse Matrix Multiplication:** Multiply each ciphertext vector by the inverse of the key matrix. The result is the corresponding plaintext vector.
4. **Convert to Plaintext:** Convert the numerical values in the plaintext vector back to letters.
5. The Hill cipher decryption process is based on the following operation −

D(K, C) = (K-1 \*C) mod 26

Here C is the vectorized ciphertext and K is our key matrix.

**Program –**

import math

def get\_key\_matrix\_from\_word(word, n):

word = word.upper().replace(" ", "")

key\_numbers = [ord(char) - ord('A') for char in word if char.isalpha()]

if len(key\_numbers) < n \* n:

print(f"Key too short, padding with 'A'.")

key\_numbers += [0] \* (n \* n - len(key\_numbers)) # Pad with 'A' (i.e., 0)

elif len(key\_numbers) > n \* n:

print(f"Key too long, truncating.")

key\_numbers = key\_numbers[:n \* n] # Truncate extra characters

key\_matrix = []

for i in range(n):

key\_matrix.append(key\_numbers[i \* n: (i + 1) \* n])

return key\_matrix

def text\_to\_numbers(text):

text = text.upper()

numbers = [ord(char)-ord('A') for char in text if char.isalpha()]

return numbers

def numbers\_to\_text(numbers):

text = ''.join([chr(num%26 + ord('A')) for num in numbers])

return text

def matrix\_multiply\_modulo(matrix, vector, modulus):

n = len(matrix)

result = []

for i in range(n):

s = 0

for j in range(n):

s += matrix[i][j] \* vector[j]

result.append(s % modulus)

return result

def matrix\_determinant(matrix):

n = len(matrix)

if n == 2:

a, b = matrix[0]

c, d = matrix[1]

det = a\*d - b\*c

elif n == 3:

a, b, c = matrix[0]

d, e, f = matrix[1]

g, h, i = matrix[2]

det = (a\*e\*i + b\*f\*g + c\*d\*h) - (c\*e\*g + b\*d\*i + a\*f\*h)

else:

print("Determinant not implemented for matrices larger than 3x3.")

return None

return det % 26

def modular\_inverse(a, modulus):

a = a % modulus

for x in range(1, modulus):

if (a \* x) % modulus == 1:

return x

return None

def matrix\_inverse\_modulo(matrix, modulus):

n = len(matrix)

det = matrix\_determinant(matrix)

det\_inv = modular\_inverse(det, modulus)

if det\_inv is None:

print(f"Determinant {det} has no inverse modulo {modulus}.")

return None

if n == 2:

a, b = matrix[0]

c, d = matrix[1]

# Compute adjugate matrix

adjugate = [[d, -b], [-c, a]]

# Bring adjugate elements into the range [0, modulus)

for i in range(n):

for j in range(n):

adjugate[i][j] = adjugate[i][j] % modulus

# Multiply adjugate by determinant inverse modulo modulus

inverse = []

for row in adjugate:

inverse\_row = [(det\_inv \* element) % modulus for element in row]

inverse.append(inverse\_row)

return inverse

else:

print("Matrix inverse not implemented for matrices larger than 2x2.")

return None

def prepare\_text(text, block\_size):

numbers = text\_to\_numbers(text)

if len(numbers) % block\_size != 0:

padding\_length = block\_size - (len(numbers) % block\_size)

numbers += [text\_to\_numbers('X')[0]] \* padding\_length

return numbers

def encrypt(plaintext\_numbers, key\_matrix, modulus):

n = len(key\_matrix)

ciphertext\_numbers = []

for i in range(0, len(plaintext\_numbers), n):

block = plaintext\_numbers[i:i+n]

cipher\_block = matrix\_multiply\_modulo(key\_matrix, block, modulus)

ciphertext\_numbers.extend(cipher\_block)

return ciphertext\_numbers

def decrypt(ciphertext\_numbers, inverse\_key\_matrix, modulus):

n = len(inverse\_key\_matrix)

plaintext\_numbers = []

for i in range(0, len(ciphertext\_numbers), n):

block = ciphertext\_numbers[i:i+n]

plain\_block = matrix\_multiply\_modulo(inverse\_key\_matrix, block, modulus)

plaintext\_numbers.extend(plain\_block)

return plaintext\_numbers

def main():

modulus = 26

print("Hill Cipher Encryption and Decryption")

word\_key = input("Enter the word key: ")

n = 2 # Only support 2x2 matrices for now

key\_matrix = get\_key\_matrix\_from\_word(word\_key, n)

print("Key Matrix:")

for row in key\_matrix:

print(row)

det = matrix\_determinant(key\_matrix)

det\_inv = modular\_inverse(det, modulus)

if det\_inv is None:

print(f"The determinant ({det}) is not invertible modulo {modulus}. Cannot proceed.")

return

plaintext = input("Enter the plaintext: ")

plaintext\_numbers = prepare\_text(plaintext, n)

ciphertext\_numbers = encrypt(plaintext\_numbers, key\_matrix, modulus)

ciphertext = numbers\_to\_text(ciphertext\_numbers)

print("Ciphertext:", ciphertext)

# Now, decrypt

inverse\_key\_matrix = matrix\_inverse\_modulo(key\_matrix, modulus)

if inverse\_key\_matrix is None:

print("Cannot compute inverse key matrix. Decryption not possible.")

return

decrypted\_numbers = decrypt(ciphertext\_numbers, inverse\_key\_matrix, modulus)

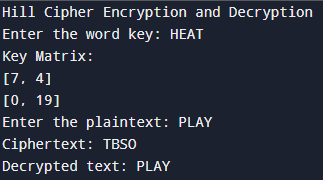
decrypted\_text = numbers\_to\_text(decrypted\_numbers)

print("Decrypted text:", decrypted\_text)

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

main()

**Output –**



**Conclusion –**

Hence, we have performed the Hill cipher implementation for encryption and decryption successfully.

**Practical 4**

**Aim – To write a program to implement DES algorithm.**

**Theory –**

The Data Encryption Standard algorithm is a block cipher algorithm that takes in 64-bit blocks of plaintext at a time as input and produces 64-bit blocks of cipher text at a time, using a 48-bit key for each input.

In block cipher algorithms, the text to be encrypted is broken into ‘blocks’ of text, and each block is encrypted separately using the key.

The decryption process is the exact opposite of the encryption. It takes in a 64 bit block of ciphertext, and produces the 64 bit block of plaintext using the same 48 bit key during encryption.

The encryptor and the decryptor need to use the same key, otherwise, they will not be able to communicate together.

The DES algorithm was successful in the early days of the internet, but its short key length of 56 bits makes it too insecure for today’s applications.

With the evolution of technology and the increase in computing power, an attacker with sufficient computing resources can break the key within a few minutes.

It has however been highly influential in the development and advancement of cryptography.

**Program –**

#initail permutation

ip\_table = [

58, 50, 42, 34, 26, 18, 10, 2,

60, 52, 44, 36, 28, 20, 12, 4,

62, 54, 46, 38, 30, 22, 14, 6,

64, 56, 48, 40, 32, 24, 16, 8,

57, 49, 41, 33, 25, 17, 9, 1,

59, 51, 43, 35, 27, 19, 11, 3,

61, 53, 45, 37, 29, 21, 13, 5,

63, 55, 47, 39, 31, 23, 15, 7

]

# PC1 permutation table

pc1\_table = [

57, 49, 41, 33, 25, 17, 9, 1,

58, 50, 42, 34, 26, 18, 10, 2,

59, 51, 43, 35, 27, 19, 11, 3,

60, 52, 44, 36, 63, 55, 47, 39,

31, 23, 15, 7, 62, 54, 46, 38,

30, 22, 14, 6, 61, 53, 45, 37,

29, 21, 13, 5, 28, 20, 12, 4

]

# Define the left shift schedule for each round

shift\_schedule = [1, 1, 2, 2,

2, 2, 2, 2,

1, 2, 2, 2,

2, 2, 2, 1]

# PC2 permutation table

pc2\_table = [

14, 17, 11, 24, 1, 5, 3, 28,

15, 6, 21, 10, 23, 19, 12, 4,

26, 8, 16, 7, 27, 20, 13, 2,

41, 52, 31, 37, 47, 55, 30, 40,

51, 45, 33, 48, 44, 49, 39, 56,

34, 53, 46, 42, 50, 36, 29, 32

]

#expension

e\_box\_table = [

32, 1, 2, 3, 4, 5,

4, 5, 6, 7, 8, 9,

8, 9, 10, 11, 12, 13,

12, 13, 14, 15, 16, 17,

16, 17, 18, 19, 20, 21,

20, 21, 22, 23, 24, 25,

24, 25, 26, 27, 28, 29,

28, 29, 30, 31, 32, 1

]

# S-box tables for DES

s\_boxes = [

# S-box 1

[

[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],

[0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],

[4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],

[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]

],

# S-box 2

[

[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],

[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],

[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],

[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]

],

# S-box 3

[

[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],

[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],

[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]

],

# S-box 4

[

[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],

[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],

[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],

[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]

],

# S-box 5

[

[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],

[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],

[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],

[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]

],

# S-box 6

[

[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]

],

# S-box 7

[

[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]

],

# S-box 8

[

[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]

]

]

p\_box\_table = [

16, 7, 20, 21, 29, 12, 28, 17,

1, 15, 23, 26, 5, 18, 31, 10,

2, 8, 24, 14, 32, 27, 3, 9,

19, 13, 30, 6, 22, 11, 4, 25

]

ip\_inverse\_table = [

40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 63, 31,

38, 6, 46, 14, 54, 22, 62, 30,

37, 5, 45, 13, 53, 21, 61, 29,

36, 4, 44, 12, 52, 20, 60, 28,

35, 3, 43, 11, 51, 19, 59, 27,

34, 2, 42, 10, 50, 18, 58, 26,

33, 1, 41, 9, 49, 17, 57, 25

]

def str\_to\_bin(user\_input):

binary\_representation = ''.join(format(ord(char), '08b') for char in user\_input)

binary\_representation = binary\_representation[:64].ljust(64, '0') # Ensure it's 64 bits long

return binary\_representation

def binary\_to\_ascii(binary\_str):

ascii\_str = ''.join([chr(int(binary\_str[i:i+8], 2)) for i in range(0, len(binary\_str), 8)])

return ascii\_str.rstrip() # Remove any padding

def binary\_to\_hex(binary\_str):

return hex(int(binary\_str, 2))[2:].upper().zfill(16) # Convert binary to hex

def hex\_to\_binary(hex\_str):

return bin(int(hex\_str, 16))[2:].zfill(64) # Convert hex to binary

def ip\_on\_binary\_rep(binary\_representation):

ip\_result = ''.join(binary\_representation[ip\_table[i] - 1] for i in range(64))

return ip\_result

def key\_in\_binary\_conv(hex\_key):

binary\_representation\_key = ''.join(format(int(hex\_key[i:i+2], 16), '08b') for i in range(0, len(hex\_key), 2))

return binary\_representation\_key

def generate\_round\_keys(hex\_key):

binary\_representation\_key = key\_in\_binary\_conv(hex\_key)

pc1\_key\_str = ''.join(binary\_representation\_key[bit - 1] for bit in pc1\_table)

c0, d0 = pc1\_key\_str[:28], pc1\_key\_str[28:]

round\_keys = []

for round\_num in range(16):

c0 = c0[shift\_schedule[round\_num]:] + c0[:shift\_schedule[round\_num]]

d0 = d0[shift\_schedule[round\_num]:] + d0[:shift\_schedule[round\_num]]

cd\_concatenated = c0 + d0

round\_key = ''.join(cd\_concatenated[bit - 1] for bit in pc2\_table)

round\_keys.append(round\_key)

return round\_keys

def encryption(user\_input, hex\_key):

binary\_rep\_of\_input = str\_to\_bin(user\_input)

round\_keys = generate\_round\_keys(hex\_key)

ip\_result\_str = ip\_on\_binary\_rep(binary\_rep\_of\_input)

lpt, rpt = ip\_result\_str[:32], ip\_result\_str[32:]

for round\_num in range(16):

expanded\_result\_str = ''.join(rpt[e\_box\_table[i] - 1] for i in range(48))

round\_key\_str = round\_keys[round\_num]

xor\_result\_str = ''.join(str(int(expanded\_result\_str[i]) ^ int(round\_key\_str[i])) for i in range(48))

six\_bit\_groups = [xor\_result\_str[i:i + 6] for i in range(0, 48, 6)]

s\_box\_substituted = ''

for i in range(8):

row\_bits = int(six\_bit\_groups[i][0] + six\_bit\_groups[i][-1], 2)

col\_bits = int(six\_bit\_groups[i][1:-1], 2)

s\_box\_value = s\_boxes[i][row\_bits][col\_bits]

s\_box\_substituted += format(s\_box\_value, '04b')

p\_box\_result = ''.join(s\_box\_substituted[p\_box\_table[i] - 1] for i in range(32))

new\_rpt\_str = ''.join(str(int(lpt[i]) ^ int(p\_box\_result[i])) for i in range(32))

lpt, rpt = rpt, new\_rpt\_str

final\_result = rpt + lpt

final\_cipher = ''.join(final\_result[ip\_inverse\_table[i] - 1] for i in range(64))

final\_cipher\_hex = binary\_to\_hex(final\_cipher)

print("Final Ciphertext (Hex):", final\_cipher\_hex)

return final\_cipher\_hex

def decryption(final\_cipher\_hex, hex\_key):

round\_keys = generate\_round\_keys(hex\_key)

final\_cipher\_bin = hex\_to\_binary(final\_cipher\_hex)

ip\_dec\_result\_str = ip\_on\_binary\_rep(final\_cipher\_bin)

lpt, rpt = ip\_dec\_result\_str[:32], ip\_dec\_result\_str[32:]

for round\_num in range(16):

expanded\_result\_str = ''.join(rpt[e\_box\_table[i] - 1] for i in range(48))

round\_key\_str = round\_keys[15 - round\_num]

xor\_result\_str = ''.join(str(int(expanded\_result\_str[i]) ^ int(round\_key\_str[i])) for i in range(48))

six\_bit\_groups = [xor\_result\_str[i:i + 6] for i in range(0, 48, 6)]

s\_box\_substituted = ''

for i in range(8):

row\_bits = int(six\_bit\_groups[i][0] + six\_bit\_groups[i][-1], 2)

col\_bits = int(six\_bit\_groups[i][1:-1], 2)

s\_box\_value = s\_boxes[i][row\_bits][col\_bits]

s\_box\_substituted += format(s\_box\_value, '04b')

p\_box\_result = ''.join(s\_box\_substituted[p\_box\_table[i] - 1] for i in range(32))

new\_rpt\_str = ''.join(str(int(lpt[i]) ^ int(p\_box\_result[i])) for i in range(32))

lpt, rpt = rpt, new\_rpt\_str

final\_result = rpt + lpt

final\_plain\_bin = ''.join(final\_result[ip\_inverse\_table[i] - 1] for i in range(64))

final\_plain\_str = binary\_to\_ascii(final\_plain\_bin)

print("Decrypted Text:", final\_plain\_str)

return final\_plain\_str

# User Interaction

operation = input("Do you want to (E)ncrypt or (D)ecrypt? ").strip().upper()

if operation == 'E':

plaintext = input("Enter the plaintext (max 8 characters): ").strip()

hex\_key = input("Enter the key (in Hex, 16 characters): ").strip()

cipher = encryption(plaintext, hex\_key)

elif operation == 'D':

ciphertext = input("Enter the ciphertext (in Hex): ").strip()

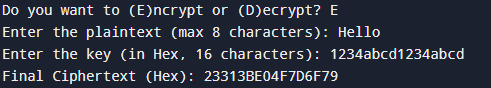
hex\_key = input("Enter the key (in Hex, 16 characters): ").strip()

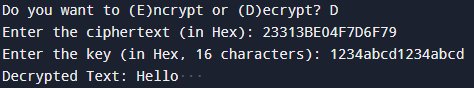
decrypted\_text = decryption(ciphertext, hex\_key)

else:

print("Invalid operation selected.")

**Output –**





**Conclusion –**

Hence, we have successfully implemented DES algorithm.

**Practical 5**

**Aim – To write a program to implement Blowfish Algorithm.**

**Theory –**

Blowfish is a symmetric block cipher designed by Bruce Schneier in 1993.

It was intended as a replacement for DES (Data Encryption Standard) and IDEA (International Data Encryption Algorithm).

Blowfish is known for its speed, simplicity, and security, making it a popular choice for various cryptographic applications.

**Key Points:**

* **Block Size:** Blowfish operates on 64-bit blocks of data.
* **Key Length:** The key can be of any length between 32 and 448 bits.
* **Structure:** It uses a combination of Feistel network and substitution-permutation network (SPN) structures.
* **Key Expansion:** The key expansion process generates 18 subkeys of 32 bits each.

**Blowfish Algorithm:**

1. **Key Expansion:**
   * The key is divided into 16 32-bit subkeys.
   * An initialization vector (IV) is used to initialize the subkeys.
   * A series of iterations is performed to generate the remaining subkeys.
2. **Encryption:**
   * The plaintext is divided into 64-bit blocks.
   * Each block is processed in 16 rounds.
   * In each round:
     + The block is divided into two 32-bit halves, L and R.
     + The left half (L) is XORed with a subkey.
     + The result is passed through a substitution function (S-box) and XORed with the right half (R).
     + The right half (R) becomes the new left half (L) for the next round.
   * After 16 rounds, the halves are swapped, and the final ciphertext is obtained.
3. **Decryption:**
   * The ciphertext is divided into 64-bit blocks.
   * The same key expansion process is used.
   * The decryption process involves reversing the encryption steps, using the same subkeys in reverse order.

**Program –**

class Blowfish:

def \_\_init\_\_(self, key):

self.P = [0x243F6A88, 0x85A308D3, 0x13198A2E, 0x03707344,

0xA4093822, 0x299F31D0, 0x082EFA98, 0xEC4E6C89,

0x452821E6, 0x38D01377, 0xBE5466CF, 0x34E90C6C,

0xC0AC29B7, 0xC97C50DD, 0x3F84D5B5, 0xB5470917,

0x9216D5D9, 0x8979FB1B]

self.S = [0xD1310BA6, 0x98DFB5AC, 0x2FFD72DB, 0xD01ADFB7,

0xB8E1AFED, 0x6A267E96, 0xBA7C9045, 0xF12C7F99,

0x24A19947, 0xB3916CF7, 0x0801F2E2, 0x858EFC16,

0x636920D8, 0x71574E69, 0xA458FEA3, 0xF4933D7E]

def F(self, x):

return ((self.S[0] + x) \* self.S[1]) & 0xFFFFFFFF

def encrypt\_block(self, L, R):

for i in range(16):

L = L ^ self.P[i]

R = self.F(L) ^ R

L, R = R, L

L, R = R, L

R = R ^ self.P[16]

L = L ^ self.P[17]

return L, R

def decrypt\_block(self, L, R):

for i in range(17, 1, -1):

L = L ^ self.P[i]

R = self.F(L) ^ R

L, R = R, L

L, R = R, L

R = R ^ self.P[1]

L = L ^ self.P[0]

return L, R

def encrypt(self, plaintext):

L = int.from\_bytes(plaintext[:4].encode(), byteorder='big')

R = int.from\_bytes(plaintext[4:].encode(), byteorder='big')

L, R = self.encrypt\_block(L, R)

return (L.to\_bytes(4, byteorder='big') + R.to\_bytes(4,byteorder='big')).hex()

def decrypt(self, ciphertext):

ciphertext = bytes.fromhex(ciphertext)

L = int.from\_bytes(ciphertext[:4], byteorder='big')

R = int.from\_bytes(ciphertext[4:], byteorder='big')

L, R = self.decrypt\_block(L, R)

return (L.to\_bytes(4, byteorder='big') + R.to\_bytes(4,byteorder='big')).decode()

def main():

key = "simplekey"

plaintext = "hello123"

blowfish = Blowfish(key)

print(f"Plaintext: {plaintext}")

encrypted\_text = blowfish.encrypt(plaintext)

print(f"Encrypted text: {encrypted\_text}")

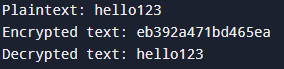
decrypted\_text = blowfish.decrypt(encrypted\_text)

print(f"Decrypted text: {decrypted\_text}")

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

main()

**Output –**



**Conclusion –**

Hence, we have successfully implemented Blowfish algorithm.

**Practical 6**

**Aim – To write a program to implement AES algorithm.**

**Theory –**

The Advanced Encryption Standard (AES) is a symmetric block cipher adopted by the U.S. government as a standard for encrypting sensitive but unclassified information.

It's considered to be highly secure due to its complexity and the absence of any known practical attacks against it.

**AES Algorithm Overview**

AES operates on 128-bit blocks of data, and uses a 128-, 192-, or 256-bit key. The encryption process involves a series of rounds, each consisting of four transformations:

1. **SubBytes:** Replaces each byte in the block with a new byte from a substitution table.
2. **ShiftRows:** Cyclically shifts each row of the block by a certain number of positions.
3. **MixColumns:** Performs a matrix multiplication on each column of the block.
4. **AddRoundKey:** XORs the block with a round key derived from the main key.

The number of rounds depends on the key length: 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

**Key Expansion**

Before the encryption process begins, the main key is expanded into a series of round keys. This is done using a key schedule algorithm that involves a combination of rotations, substitutions, and XOR operations.

**Implementation Considerations**

When implementing AES, several factors need to be considered:

* **Key Length:** Choose the appropriate key length based on the security requirements of your application.
* **Mode of Operation:** AES is a block cipher, so it needs to be used in a mode of operation like Electronic Codebook (ECB), Cipher Block Chaining (CBC), Counter (CTR), etc.
* **Padding:** If the plaintext length is not a multiple of the block size, padding is required. Common padding schemes include PKCS#7 and Zero Padding.
* **Performance:** The efficiency of the implementation depends on various factors, such as the programming language, the optimization techniques used, and the hardware capabilities of the system.

**Program –**

# Simplified S-box for AES

s\_box = [

0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5,

0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,

0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0,

0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,

0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc,

0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,

0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a,

0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,

0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0,

0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,

0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b,

0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,

0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85,

0x45, 0xf1, 0x7d, 0x22, 0xb6, 0xbf, 0x12, 0x98,

0x48, 0xa9, 0xa7, 0xa4, 0xf5, 0x4b, 0xa0, 0xe8

]

def substitute\_bytes(byte, s\_box):

"""Substitute bytes using the given S-box."""

if 0 <= byte < len(s\_box):

return s\_box[byte]

else:

raise IndexError("Byte value is out of range for substitution.")

def add\_round\_key(state, round\_key):

"""Add round key to state (XOR operation)."""

return [s ^ k for s, k in zip(state, round\_key)]

def aes\_encrypt(plaintext, key):

"""Encrypt plaintext using a simplified AES-like algorithm."""

key = [ord(c) for c in key]

plaintext = [ord(c) for c in plaintext]

# Ensure plaintext length is a multiple of 16

while len(plaintext) % 16 != 0:

plaintext.append(0) # Pad with zeroes

encrypted\_text = ""

# Process 16-byte blocks

for i in range(0, len(plaintext), 16):

block = plaintext[i:i+16]

state = block

# Step 1: Add round key (XOR with key)

state = add\_round\_key(state, key)

# Step 2: Substitute bytes using S-box

try:

state = [substitute\_bytes(b, s\_box) for b in state]

except IndexError as e:

print(f"ERROR! {e}")

return None

# Convert to encrypted characters

encrypted\_text += ''.join(chr(b) for b in state)

return encrypted\_text

def aes\_decrypt(ciphertext, key):

"""Decrypt ciphertext (reverse of simplified encryption)."""

key = [ord(c) for c in key]

ciphertext = [ord(c) for c in ciphertext]

decrypted\_text = ""

# Process 16-byte blocks

for i in range(0, len(ciphertext), 16):

block = ciphertext[i:i+16]

state = block

# Step 1: Reverse Substitute bytes using S-box (reverse process)

state = [s\_box.index(b) for b in state]

# Step 2: Reverse Add round key (XOR with key)

state = add\_round\_key(state, key)

# Convert to decrypted characters

decrypted\_text += ''.join(chr(b) for b in state)

return decrypted\_text.strip('\x00') # Remove padding

def main():

key = input("Enter a 16-character key: ")

if len(key) != 16:

print("Key must be exactly 16 characters!")

return

plaintext = input("Enter the message to encrypt (max 16 characters): ")

if len(plaintext) > 16:

print("Message should not exceed 16 characters.")

return

print(f"\nPlain text: {plaintext}")

# Encryption

encrypted\_text = aes\_encrypt(plaintext, key)

if encrypted\_text is None:

return

print(f"Encrypted text: {''.join(hex(ord(c))[2:] for c in encrypted\_text)}")

# Decryption

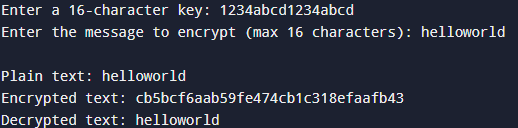
decrypted\_text = aes\_decrypt(encrypted\_text, key)

print(f"Decrypted text: {decrypted\_text}")

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

main()

**Output –**



**Conclusion –**

Hence, we have successfully implemented AES algorithm.

**Practical 7**

**Aim – To write a program to implement Diffie-Hellman key exchange technique for symmetric Cryptography.**

**Theory –**

The Diffie-Hellman key exchange is a cryptographic protocol used to establish a shared secret between two parties over an insecure channel.

This shared secret can then be used to encrypt subsequent communications using a symmetric encryption algorithm.

**Key Concepts**

* **Public Key Cryptography:** Unlike symmetric encryption, which uses the same key for both encryption and decryption, public key cryptography uses a pair of keys: a public key and a private key. The public key can be freely shared, while the private key must be kept secret.
* **Modular Arithmetic:** Diffie-Hellman relies on modular arithmetic, which involves performing arithmetic operations within a specific range (a modulus).

**The Diffie-Hellman Protocol**

1. **Agreement on Public Parameters:** Both parties agree on two publicly known values:
   * **Prime number:** A large prime number, denoted as p.
   * **Generator:** A number g that is a primitive root modulo p.
2. **Generation of Private Keys:** Each party generates a random private key:
   * **Alice:** Generates a random private key a.
   * **Bob:** Generates a random private key b.
3. **Calculation of Public Values:**
   * **Alice:** Calculates her public value A = g^a mod p and sends it to Bob.
   * **Bob:** Calculates his public value B = g^b mod p and sends it to Alice.
4. **Calculation of Shared Secret:**
   * **Alice:** Calculates the shared secret s = B^a mod p.
   * **Bob:** Calculates the shared secret s = A^b mod p.

**Security**

The security of Diffie-Hellman relies on the discrete logarithm problem, which is computationally difficult to solve. This problem involves finding the exponent x in the equation g^x = y mod p.

**Program –**

import random

# Function to generate a prime number

def generate\_prime():

# Using a simple method to generate a prime number

prime\_candidates = [17, 19, 23, 29, 31, 37, 41, 43, 47]

return random.choice(prime\_candidates)

# Function to generate a primitive root

def generate\_primitive\_root(prime):

roots = []

for g in range(2, prime):

if len(set(pow(g, powers, prime) for powers in range(1, prime))) == prime - 1:

roots.append(g)

return random.choice(roots)

# Function to compute the shared secret

def diffie\_hellman(private\_key\_a, private\_key\_b, prime, primitive\_root):

public\_key\_a = pow(primitive\_root, private\_key\_a, prime)

public\_key\_b = pow(primitive\_root, private\_key\_b, prime)

shared\_secret\_a = pow(public\_key\_b, private\_key\_a, prime)

shared\_secret\_b = pow(public\_key\_a, private\_key\_b, prime)

return shared\_secret\_a, shared\_secret\_b

# Simple XOR encryption/decryption function

def xor\_encrypt\_decrypt(message, key):

return ''.join(chr(ord(c) ^ key) for c in message)

def main():

# Generate prime and primitive root

prime = generate\_prime()

primitive\_root = generate\_primitive\_root(prime)

print(f"Prime: {prime}, Primitive Root: {primitive\_root}")

# User inputs for private keys

private\_key\_a = int(input("User A, enter your private key: "))

private\_key\_b = int(input("User B, enter your private key: "))

# Compute the shared secret

shared\_secret\_a, shared\_secret\_b = diffie\_hellman(private\_key\_a, private\_key\_b, prime, primitive\_root)

print(f"Shared Secret (User A): {shared\_secret\_a}")

print(f"Shared Secret (User B): {shared\_secret\_b}")

# Use the shared secret as a symmetric key (simplified for demonstration)

symmetric\_key = shared\_secret\_a % 256 # Use modulo to keep key in byte range

print(f"Symmetric Key: {symmetric\_key}")

# Encrypt and decrypt a message

message = input("Enter a message to encrypt: ")

encrypted\_message = xor\_encrypt\_decrypt(message, symmetric\_key)

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

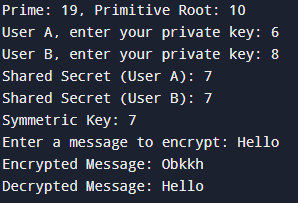
decrypted\_message = xor\_encrypt\_decrypt(encrypted\_message, symmetric\_key)

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

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

main()

**Output –**



**Conclusion –**

Hence, we have successfully implemented Diffie-Hellman key exchange technique for symmetric Cryptography.

**Practical 8**

**Aim – To write a program to implement RSA Algorithm.**

**Theory –**

RSA (Rivest-Shamir-Adleman) is one of the most widely used public-key cryptographic algorithms. It provides a method for secure communication by using a pair of keys: a public key and a private key. The public key is freely distributed, while the private key remains secret.

**Key Generation:**

1. **Choose two large prime numbers, p and q.** These numbers should be kept secret.
2. **Calculate the modulus, n:** n = p \* q
3. **Calculate Euler's totient function, φ(n):** φ(n) = (p-1) \* (q-1)
4. **Choose an integer e, 1 < e < φ(n), such that gcd(e, φ(n)) = 1.** This means e and φ(n) are relatively prime.
5. **Calculate the private key, d:** d ≡ e^-1 (mod φ(n)). This means d is the modular multiplicative inverse of e modulo φ(n).

**Encryption:**

To encrypt a message m:

1. Convert the message into a numerical representation (e.g., ASCII values).
2. Calculate the ciphertext c: c ≡ m^e (mod n)

**Decryption:**

To decrypt a ciphertext c:

1. Calculate the plaintext m: m ≡ c^d (mod n)
2. Convert the numerical representation back into the original message.

**Security:**

The security of RSA relies on the difficulty of factoring the modulus n into its prime factors p and q. This is a computationally intensive problem, even for large values of n. If an attacker could factor n, they could easily calculate the private key from the public key.

**Applications:**

RSA is used in various applications, including:

* **Secure communication:** For encrypting and decrypting messages.
* **Digital signatures:** For verifying the authenticity of digital documents.
* **Public key infrastructure (PKI):** For managing and distributing digital certificates.

**Program –**

import random

def generate\_prime(bits=8):

def is\_prime(num):

if num <= 1:

return False

for i in range(2, int(num\*\*0.5) + 1):

if num % i == 0:

return False

return True

while True:

num = random.getrandbits(bits)

if is\_prime(num):

return num

def gcd(a, b):

while b:

a, b = b, a % b

return a

def mod\_inverse(e, phi):

m0, x0, x1 = phi, 0, 1

if phi == 1:

return 0

while e > 1:

q = e // phi

t = phi

phi = e % phi

e = t

t = x0

x0 = x1 - q \* x0

x1 = t

# Make x1 positive

if x1 < 0:

x1 += m0

return x1

def generate\_keys():

p = generate\_prime()

q = generate\_prime()

n = p \* q

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

e = 65537

d = mod\_inverse(e, phi)

return (e, n), (d, n)

def encrypt(plaintext, public\_key):

e, n = public\_key

ciphertext = [pow(ord(char), e, n) for char in plaintext]

return ciphertext

def decrypt(ciphertext, private\_key):

d, n = private\_key

plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])

return plaintext

def main():

public\_key, private\_key = generate\_keys()

print(f"Public Key: {public\_key}")

print(f"Private Key: {private\_key}")

plaintext = input("Enter the Plaintext: ").strip()

print(f"Plaintext: {plaintext}")

ciphertext = encrypt(plaintext, public\_key)

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

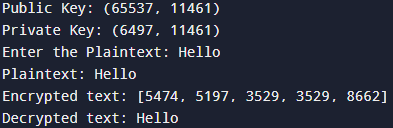
decrypted\_text = decrypt(ciphertext, private\_key)

print(f"Decrypted text: {decrypted\_text}")

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

main()

**Output –**



**Conclusion –**

Hence, we have successfully implemented RSA algorithm.

**Practical 9**

**Aim – To write a program to implement signature and digital signature technique.**

**Theory –**

A digital signature is a cryptographic technique that provides authenticity, integrity, and non-repudiation for digital data. It ensures that the data has not been altered since it was signed and that it originates from the claimed signer.

**Components of a Digital Signature:**

1. **Message:** The data to be signed.
2. **Hash function:** A cryptographic function that takes an input (the message) and produces a fixed-size output (the hash value).
3. **Private key:** The signer's private key, used to create the signature.
4. **Public key:** The signer's public key, used to verify the signature.

**Digital Signature Process:**

1. **Hashing:** The message is hashed using a cryptographically secure hash function (e.g., SHA-256).
2. **Signing:** The signer's private key is used to encrypt the hash value, creating the digital signature.
3. **Transmission:** The message and the digital signature are transmitted to the recipient.

**Verification Process:**

1. **Hashing:** The recipient hashes the received message using the same hash function.
2. **Decryption:** The recipient uses the signer's public key to decrypt the digital signature.
3. **Comparison:** The decrypted hash value is compared to the hash value calculated in step 1. If they match, the signature is valid.

**Security Properties:**

* **Authenticity:** The signature ensures that the message originated from the claimed signer.
* **Integrity:** The signature verifies that the message has not been altered since it was signed.
* **Non-repudiation:** The signer cannot deny having signed the message.

**Applications:**

* **Email:** To verify the authenticity of email messages.
* **Digital documents:** To ensure the integrity and authenticity of electronic documents.
* **Secure communication:** To protect the confidentiality and integrity of data transmitted over a network.
* **Digital certificates:** To authenticate individuals, organizations, and devices.

**Program –**

import hashlib

import random

def create\_signature(message, secret\_key):

message\_hash = hashlib.sha256(message.encode()).hexdigest()

signature = message\_hash + secret\_key

return signature

def verify\_signature(message, secret\_key, signature):

expected\_signature = create\_signature(message, secret\_key)

return expected\_signature == signature

def generate\_keys():

private\_key = random.randint(1, 100)

public\_key = private\_key + 2

return private\_key, public\_key

def rsa\_sign(message, private\_key):

message\_hash = hashlib.sha256(message.encode()).hexdigest()

return message\_hash + str(private\_key)

def rsa\_verify(message, signature, public\_key):

expected\_signature = rsa\_sign(message, public\_key - 2)

return expected\_signature == signature

message = input("Enter the message: ")

secret\_key = input("Enter your secret key: ")

simple\_signature = create\_signature(message, secret\_key)

print(f"Simple Signature: {simple\_signature}")

is\_verified = verify\_signature(message, secret\_key, simple\_signature)

print(f"Signature verified: {is\_verified}")

private\_key, public\_key = generate\_keys()

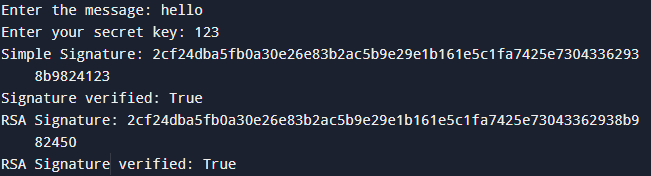
rsa\_signature = rsa\_sign(message, private\_key)

print(f"RSA Signature: {rsa\_signature}")

is\_rsa\_verified = rsa\_verify(message, rsa\_signature, public\_key)

print(f"RSA Signature verified: {is\_rsa\_verified}")

**Output –**



**Conclusion –**

Hence, we have successfully implemented signature and digital signature technique.

**Practical 10**

**Aim – To write a program to implement SHA algorithm.**

**Theory –**

The Secure Hash Algorithm (SHA) is a family of cryptographic hash functions developed by the National Security Agency (NSA) of the United States.

These functions are designed to produce a fixed-size output (hash value) from any given input message.

The hash value is used to verify the integrity of the message, ensuring that it has not been altered or tampered with.

The SHA family includes several variants, each with a different output size:

* SHA-1: Produces a 160-bit hash value.
* SHA-224: Produces a 224-bit hash value.
* SHA-256: Produces a 256-bit hash value.
* SHA-384: Produces a 384-bit hash value.
* SHA-512: Produces a 512-bit hash value.

All SHA algorithms are based on the same underlying principle: they break the input message into 512-bit blocks, and then apply a series of mathematical operations to each block to produce a hash value.

These operations include:

* **Message Padding**: If the input message is not a multiple of 512 bits, it is padded with zeros and a single bit set to 1.
* **Block Processing**: Each 512-bit block is processed through a series of rounds. In each round, the block is combined with a chaining variable, which is updated with the output of the round.
* **Output Generation**: After all blocks have been processed, the final chaining variable is the hash value.

The specific mathematical operations used in SHA algorithms are complex and involve various bitwise operations, arithmetic operations, and logical operations.

The goal of these operations is to create a function that is resistant to collisions, meaning that it is unlikely that two different input messages will produce the same hash value.

SHA algorithms are widely used in various applications, including digital signatures, password hashing, and data integrity verification.

They are considered to be secure, but it is important to use the appropriate variant for the specific application.

For example, SHA-1 is no longer considered secure due to the discovery of collision attacks, and it is recommended to use SHA-2 or SHA-3 instead.

**Program –**

import hashlib

def calculate\_sha256(input\_string):

# Create a SHA-256 hash object

sha256\_hash = hashlib.sha256()

# Update the hash object with the bytes of the input string

sha256\_hash.update(input\_string.encode('utf-8'))

# Return the hexadecimal representation of the hash

return sha256\_hash.hexdigest()

# Get user input

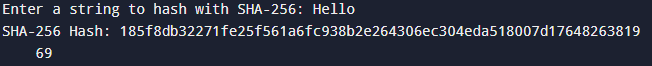
user\_input = input("Enter a string to hash with SHA-256: ")

# Calculate and display the SHA-256 hash

hash\_result = calculate\_sha256(user\_input)

print(f"SHA-256 Hash: {hash\_result}")

**Output –**



**Conclusion –**

Hence, we have successfully implemented SHA algorithm.