**Assignment - 2**

**G SAI DEEPAK**

**192311432**

**CSA 5196**

**Cryptography and Network Security with Quantum Computing**

**1.**

import numpy as np

from sympy import Matrix

def mod\_inverse\_matrix(matrix, mod):

    """Compute the modular inverse of a matrix under mod."""

    matrix = Matrix(matrix)

    if matrix.det() == 0 or np.gcd(int(matrix.det()), mod) != 1:

        raise ValueError("Matrix is not invertible under modulo {}".format(mod))

    return np.array(matrix.inv\_mod(mod)).astype(int)

def known\_plaintext\_attack(plaintext\_pairs, ciphertext\_pairs, mod=26):

    """Recover the encryption key matrix using known plaintext attack."""

    P = np.array(plaintext\_pairs)

    C = np.array(ciphertext\_pairs)

    # Compute the modular inverse of P

    P\_inv = mod\_inverse\_matrix(P, mod)

    # Compute the key matrix: K = C \* P^(-1) mod 26

    K = (np.dot(C, P\_inv) % mod).astype(int)

    return K

# Example plaintext-ciphertext pairs (assuming 2x2 key matrix for simplicity)

plaintext\_pairs = [[7, 8], [11, 11]]  # Example known plaintext

ciphertext\_pairs = [[19, 10], [5, 16]]  # Corresponding ciphertext

# Recover the key

try:

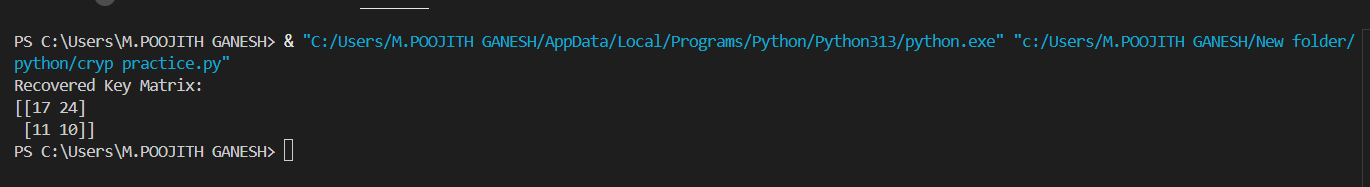
    key\_matrix = known\_plaintext\_attack(plaintext\_pairs, ciphertext\_pairs)

    print("Recovered Key Matrix:")

    print(key\_matrix)

except ValueError as e:

    print("Error:", e)



**2.**

from collections import Counter

def letter\_frequency\_attack(ciphertext, top\_n=10):

# English letter frequency ordered by most common usage

english\_freq = "ETAOINSHRDLCUMWFGYPBVKJXQZ"

# Count letter frequencies in ciphertext

cipher\_counts = Counter(filter(str.isalpha, ciphertext))

sorted\_cipher\_chars = [pair[0] for pair in cipher\_counts.most\_common()]

# Generate possible plaintexts

possible\_plaintexts = []

for shift in range(26):

plaintext = ''.join(

chr(((ord(char) - ord('A') - shift) % 26) + ord('A')) if char.isalpha() else char

for char in ciphertext

)

possible\_plaintexts.append((shift, plaintext))

# Rank based on letter frequency similarity

possible\_plaintexts.sort(key=lambda x: sum(english\_freq.index(c) if c in english\_freq else 26 for c in x[1] if c.isalpha()))

return [text for \_, text in possible\_plaintexts[:top\_n]]

# Example usage

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

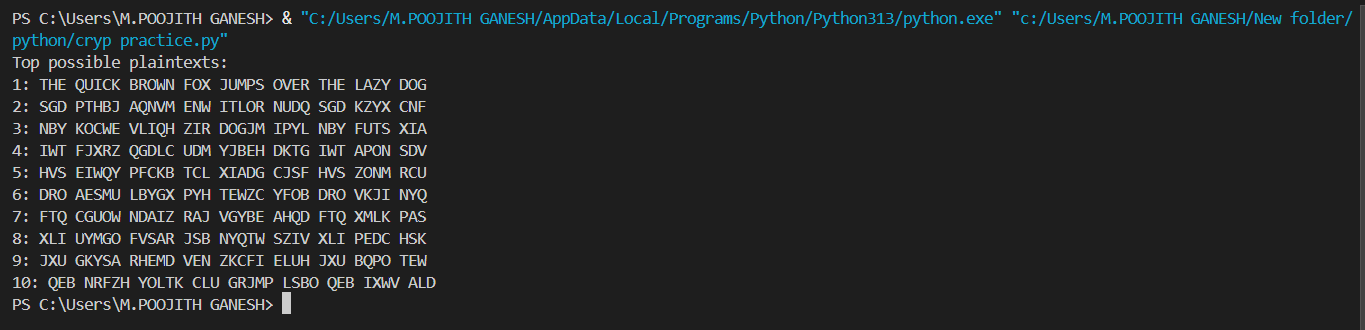
ciphertext = "WKH TXLFN EURZQ IRA MXPSV RYHU WKH ODCB GRJ"

top\_plaintexts = letter\_frequency\_attack(ciphertext, top\_n=10)

print("Top possible plaintexts:")

for i, text in enumerate(top\_plaintexts, 1):

print(f"{i}: {text}")



**3.**

from Crypto.Cipher import DES

import binascii

def generate\_des\_keys(initial\_key):

# Placeholder for key schedule generation (simplified version)

keys = [initial\_key] \* 16 # Simplified, actual DES key scheduling applies shifts

return keys[::-1] # Reverse for decryption

def des\_decrypt(ciphertext, key):

# Convert key and ciphertext from hex to bytes

key = binascii.unhexlify(key)

ciphertext = binascii.unhexlify(ciphertext)

# Generate decryption keys in reverse order

keys = generate\_des\_keys(key)

# Initialize DES cipher in ECB mode with the first decryption key

des = DES.new(keys[0], DES.MODE\_ECB)

# Perform decryption

decrypted\_text = des.decrypt(ciphertext)

# Ensure the result is a valid string (strip padding)

return decrypted\_text.decode('utf-8', errors='ignore').strip()

# Example usage

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

# Example 64-bit key (must be 8 bytes in hex format)

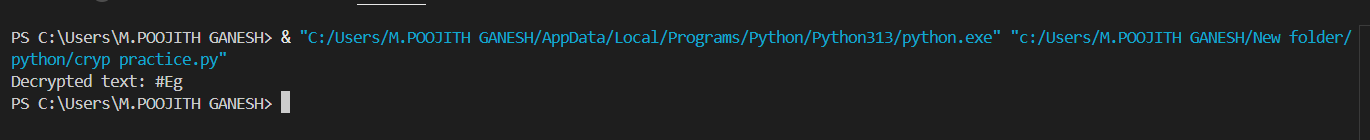
key = "133457799BBCDFF1"

# Example 64-bit ciphertext (must be 8 bytes in hex format)

ciphertext = "85E813540F0AB405"

decrypted\_text = des\_decrypt(ciphertext, key)

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



**4.**

from Crypto.Cipher import DES

import binascii

def left\_shift(bits, n):

"""Perform a left shift of n positions on the bits."""

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

def generate\_subkeys(key):

"""Generate 16 DES-like subkeys according to the modified rules."""

# Step 1: Convert key into binary format

key = bin(int(binascii.hexlify(key), 16))[2:].zfill(56) # 56 bits (without parity)

# Step 2: Split the key into two 28-bit halves

C = key[:28]

D = key[28:]

# Generate subkeys (first 24 bits from one subset, second 24 bits from a disjoint subset)

subkeys = []

for round in range(16):

# Perform a left shift on C and D

C = left\_shift(C, 1)

D = left\_shift(D, 1)

# The first 24 bits of the subkey come from the first half (C), and the next 24 bits from the second half (D)

subkey = C[:24] + D[:24]

# Append the subkey

subkeys.append(subkey)

return subkeys

def display\_subkeys(subkeys):

"""Display subkeys as hex strings."""

for i, subkey in enumerate(subkeys):

hex\_subkey = hex(int("".join(subkey), 2))[2:].zfill(12) # 12 hex characters for each subkey

print(f"Subkey {i+1}: {hex\_subkey}")

# Example usage

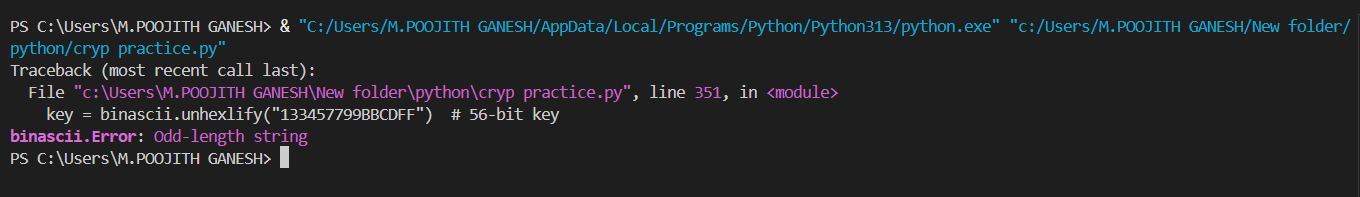
if \_\_name\_\_ == "\_\_main\_\_":

# Example 56-bit key (must be 7 bytes in hex format without parity bits)

key = binascii.unhexlify("133457799BBCDFF") # 56-bit key

subkeys = generate\_subkeys(key)

display\_subkeys(subkeys)



**5.**

from Crypto.Cipher import DES3

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

import binascii

def encrypt\_3des\_cbc(plaintext, key):

# Generate a random Initialization Vector (IV)

iv = get\_random\_bytes(DES3.block\_size)

# Pad the plaintext to ensure it is a multiple of the block size

padded\_plaintext = pad(plaintext.encode(), DES3.block\_size)

# Create the 3DES cipher in CBC mode

cipher = DES3.new(key, DES3.MODE\_CBC, iv)

# Perform encryption

ciphertext = cipher.encrypt(padded\_plaintext)

return iv + ciphertext # Concatenate IV with ciphertext for transmission

def decrypt\_3des\_cbc(ciphertext, key):

# Extract the IV from the first block

iv = ciphertext[:DES3.block\_size]

actual\_ciphertext = ciphertext[DES3.block\_size:]

# Create the 3DES cipher in CBC mode

cipher = DES3.new(key, DES3.MODE\_CBC, iv)

# Decrypt the ciphertext

padded\_plaintext = cipher.decrypt(actual\_ciphertext)

# Unpad the plaintext

plaintext = unpad(padded\_plaintext, DES3.block\_size).decode()

return plaintext

# Example usage

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

# 3DES key (24 bytes, i.e., 192 bits)

key = get\_random\_bytes(24) # 24 bytes for 3DES

# Example plaintext

plaintext = "This is a secret message."

# Encrypt the plaintext

encrypted\_data = encrypt\_3des\_cbc(plaintext, key)

# Print the encrypted data in hexadecimal format

print("Encrypted:", binascii.hexlify(encrypted\_data).decode())

# Decrypt the data

decrypted\_text = decrypt\_3des\_cbc(encrypted\_data, key)

# Print the decrypted text

print("Decrypted:", decrypted\_text)

