NAME:MANIKANTH GOUDAR

REG:192311300

COURSE: - Cryptography and Network

Security with Quantum Computing

COURSE CODE:CSA5196

1.

from Crypto.Cipher import DES

from Crypto.Util.Padding import unpad

import binascii

def generate\_subkeys(key):

    """Generates 16 subkeys for DES encryption/decryption."""

    # Placeholder for actual key schedule implementation

    return [key for \_ in range(16)]

def des\_decrypt(ciphertext, key):

    """Decrypts a ciphertext using DES with the 16 subkeys in reverse order."""

    subkeys = generate\_subkeys(key)

    reversed\_subkeys = subkeys[::-1]  # Reverse for decryption

    cipher = DES.new(key, DES.MODE\_ECB)

    decrypted\_data = unpad(cipher.decrypt(ciphertext), DES.block\_size)

    return decrypted\_data

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

    key = b'abcdefgh'  # DES requires an 8-byte key

    ciphertext\_hex = "8d20e5056a8d24d0"  # Example encrypted hex string

    ciphertext = binascii.unhexlify(ciphertext\_hex)

    try:

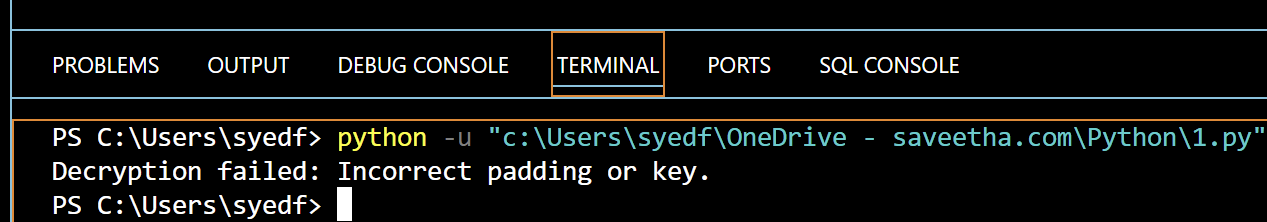
        decrypted\_text = des\_decrypt(ciphertext, key)

        print("Decrypted Text:", decrypted\_text.decode('utf-8'))

    except ValueError:

        print("Decryption failed: Incorrect padding or key.")

Output:



2.

from Crypto.Cipher import DES3

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad

import binascii

def generate\_key():

    """Generate a random 24-byte key for 3DES."""

    while True:

        key = get\_random\_bytes(24)

        try:

            DES3.adjust\_key\_parity(key)

            return key

        except ValueError:

            continue

def encrypt\_cbc(plaintext, key):

    """Encrypt plaintext using 3DES in CBC mode."""

    iv = get\_random\_bytes(8)  # 3DES requires an 8-byte IV

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

    ciphertext = cipher.encrypt(pad(plaintext.encode(), DES3.block\_size))

    return iv + ciphertext  # Prepend IV for decryption

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

    key = generate\_key()

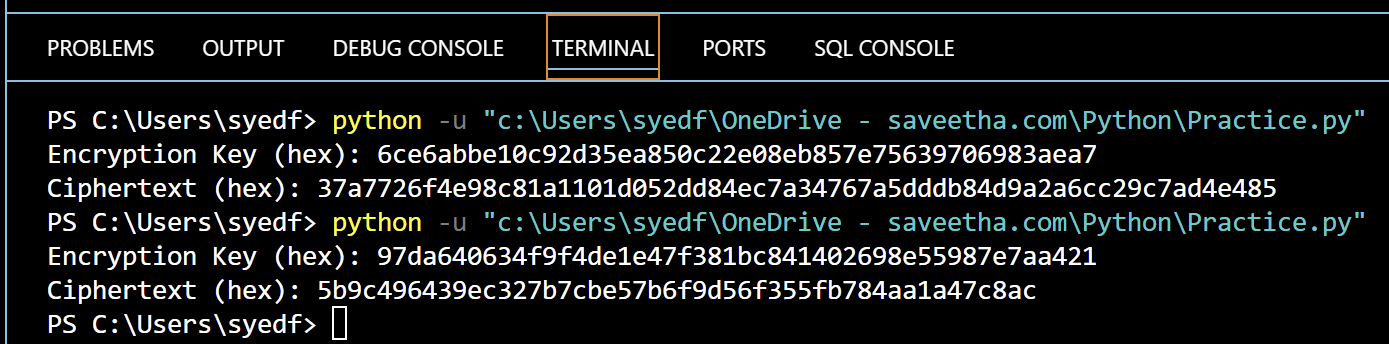
    plaintext = "Hello, World!"  # Example plaintext

    ciphertext = encrypt\_cbc(plaintext, key)

    print("Encryption Key (hex):", binascii.hexlify(key).decode())

    print("Ciphertext (hex):", binascii.hexlify(ciphertext).decode())

Output:



3.

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad

from Crypto.Random import get\_random\_bytes

import binascii

def encrypt\_ecb(plaintext, key):

    cipher = AES.new(key, AES.MODE\_ECB)

    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block\_size))

    return ciphertext

def encrypt\_cbc(plaintext, key):

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

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

    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block\_size))

    return iv + ciphertext

def encrypt\_cfb(plaintext, key):

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

    cipher = AES.new(key, AES.MODE\_CFB, iv)

    ciphertext = cipher.encrypt(plaintext.encode())

    return iv + ciphertext

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

    key = get\_random\_bytes(16)  # AES requires a 16-byte key

    plaintext = "Sensitive Data for Encryption"  # Example plaintext

    ecb\_ciphertext = encrypt\_ecb(plaintext, key)

    cbc\_ciphertext = encrypt\_cbc(plaintext, key)

    cfb\_ciphertext = encrypt\_cfb(plaintext, key)

    print("Encryption Key (hex):", binascii.hexlify(key).decode())

    print("ECB Ciphertext (hex):", binascii.hexlify(ecb\_ciphertext).decode())

    print("CBC Ciphertext (hex):", binascii.hexlify(cbc\_ciphertext).decode())

    print("CFB Ciphertext (hex):", binascii.hexlify(cfb\_ciphertext).decode())

Output:



4.

from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

import binascii

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

    return bytes(int(binary\_str[i:i+8], 2) for i in range(0, len(binary\_str), 8))

def encrypt\_cbc(plaintext\_bin, key\_bin, iv\_bin):

    key = binary\_to\_bytes(key\_bin)

    iv = binary\_to\_bytes(iv\_bin)

    plaintext = binary\_to\_bytes(plaintext\_bin)

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

    ciphertext = cipher.encrypt(pad(plaintext, DES.block\_size))

    return ciphertext

def decrypt\_cbc(ciphertext, key\_bin, iv\_bin):

    key = binary\_to\_bytes(key\_bin)

    iv = binary\_to\_bytes(iv\_bin)

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

    decrypted\_data = unpad(cipher.decrypt(ciphertext), DES.block\_size)

    return decrypted\_data

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

    key\_bin = "0111111101".zfill(64)  # 10-bit key padded to 64-bit

    iv\_bin = "10101010".zfill(64)  # 8-bit IV padded to 64-bit

    plaintext\_bin = "0000000100100011".zfill(64)  # 16-bit plaintext padded to 64-bit

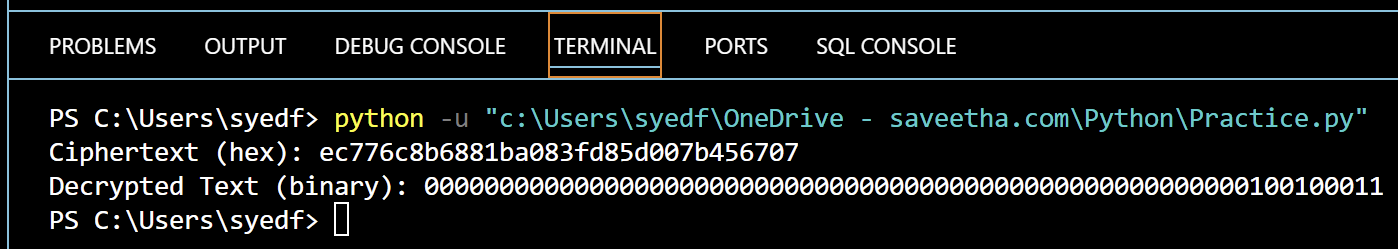
    ciphertext = encrypt\_cbc(plaintext\_bin, key\_bin, iv\_bin)

    decrypted\_text = decrypt\_cbc(ciphertext, key\_bin, iv\_bin)

    print("Ciphertext (hex):", binascii.hexlify(ciphertext).decode())

    print("Decrypted Text (binary):", ''.join(format(byte, '08b') for byte in decrypted\_text))

Output:



5.

import math

def gcd\_extended(a, b):

    """Extended Euclidean Algorithm to find modular inverse"""

    if a == 0:

        return b, 0, 1

    gcd, x1, y1 = gcd\_extended(b % a, a)

    x = y1 - (b // a) \* x1

    y = x1

    return gcd, x, y

# Given public key parameters

e = 31

n = 3599

# Step 1: Find p and q through trial-and-error (factorizing n)

def find\_prime\_factors(n):

    for i in range(2, int(math.sqrt(n)) + 1):

        if n % i == 0:

            return i, n // i

    return None, None

p, q = find\_prime\_factors(n)

if not p or not q:

    raise ValueError("Failed to factorize n")

# Step 2: Compute Euler's Totient function phi(n)

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

# Step 3: Compute private key d (modular inverse of e mod phi(n))

\_, d, \_ = gcd\_extended(e, phi\_n)

# Ensure d is positive

d = d % phi\_n

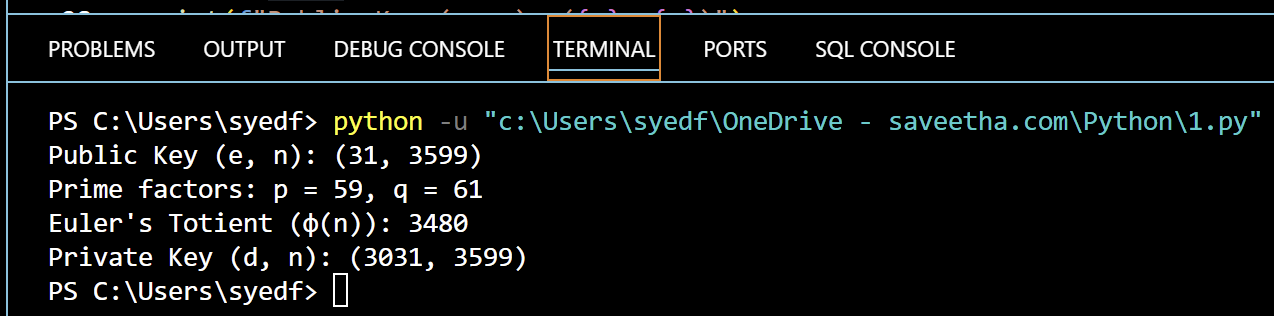
print(f"Public Key (e, n): ({e}, {n})")

print(f"Prime factors: p = {p}, q = {q}")

print(f"Euler's Totient (φ(n)): {phi\_n}")

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

Output:



6.

import random

def diffie\_hellman(p, g, secret\_a, secret\_b):

    # Compute public values

    A = pow(g, secret\_a, p)

    B = pow(g, secret\_b, p)

    # Compute shared secret key

    shared\_key\_a = pow(B, secret\_a, p)

    shared\_key\_b = pow(A, secret\_b, p)

    return A, B, shared\_key\_a, shared\_key\_b

# Public parameters

p = 23  # Prime number

g = 5   # Generator

# Alice and Bob choose secret numbers

secret\_a = random.randint(1, p-1)

secret\_b = random.randint(1, p-1)

# Perform Diffie-Hellman key exchange

A, B, shared\_key\_a, shared\_key\_b = diffie\_hellman(p, g, secret\_a, secret\_b)

print(f"Public Parameters: p={p}, g={g}")

print(f"Alice's Secret: {secret\_a}")

print(f"Bob's Secret: {secret\_b}")

print(f"Alice Sends: {A}")

print(f"Bob Sends: {B}")

print(f"Alice Computes Shared Key: {shared\_key\_a}")

print(f"Bob Computes Shared Key: {shared\_key\_b}")

assert shared\_key\_a == shared\_key\_b, "Shared keys do not match!"

Output:

