**Assignment -4**

**G SAI DEEPAK**

**192311432**

**CSA 5196- Cryptography and Network**

**Security with Quantum Computing**

**1.**

import numpy as np

def initialize\_state():

"""Initialize the SHA-3 state matrix with 1600 bits (5x5 lanes of 64 bits)."""

return np.zeros((5, 5), dtype=np.uint64) # 5x5 lanes, each 64 bits

def inject\_message\_block(state, block\_size=1024):

"""Inject a message block ensuring at least one nonzero bit per lane in 1024-bit portion."""

lanes\_to\_fill = block\_size // 64 # Number of lanes covered by the message block

for i in range(lanes\_to\_fill):

x, y = divmod(i, 5) # Assign lane positions in a 5x5 matrix

state[x, y] = np.random.randint(1, 2\*\*64, dtype=np.uint64) # Ensure nonzero bits

return state

def count\_empty\_lanes(state):

"""Count the lanes in the capacity portion that are still zero."""

return np.sum(state == 0) # Count lanes still at zero

def track\_zero\_lanes():

"""Track iterations needed until all capacity lanes have at least one nonzero bit."""

state = initialize\_state()

rounds = 0

while count\_empty\_lanes(state) > 9: # 1600 - 1024 = 576 bits (9 lanes capacity)

state = inject\_message\_block(state)

rounds += 1

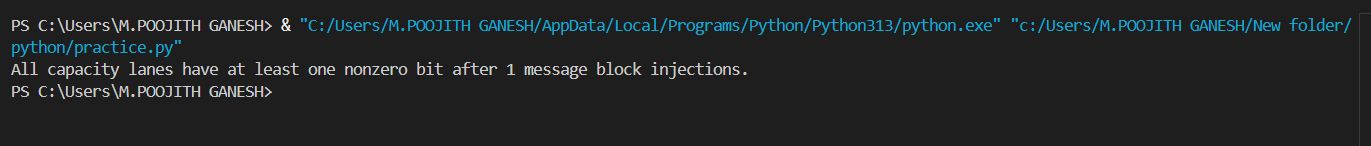
return rounds

# Run the tracking simulation

rounds\_needed = track\_zero\_lanes()

print(f"All capacity lanes have at least one nonzero bit after {rounds\_needed} message block injections.")

**Output:**



**2.**

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}")

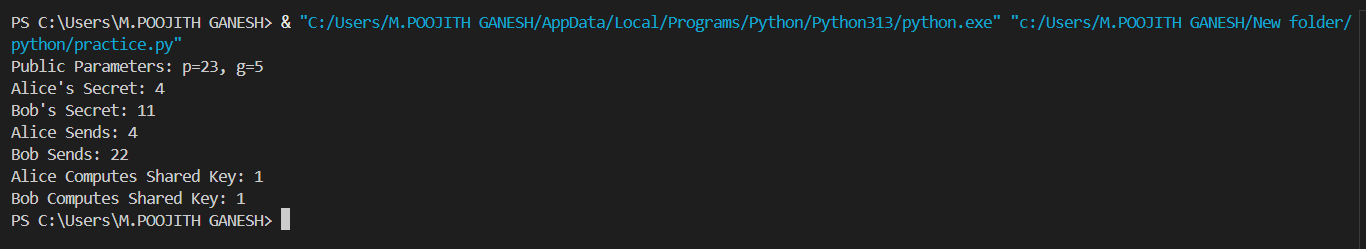
print(f"Bob Sends: {B}")

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:**



**3.**

from Crypto.Cipher import AES, DES

import binascii

def left\_shift(data):

    """Left shift a byte sequence by 1 bit, handling overflow correctly."""

    num = int.from\_bytes(data, 'big') << 1

    num &= (1 << (len(data) \* 8)) - 1

    return num.to\_bytes(len(data), 'big')

def generate\_cmac\_subkeys(cipher, key, block\_size):

    """Generate two CMAC subkeys (K1, K2)."""

    zero\_block = bytes(block\_size)

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

    L = cipher\_obj.encrypt(zero\_block)

    Rb = {16: 0x87, 8: 0x1B}[block\_size]

    K1 = left\_shift(L)

    if L[0] & 0x80:  # If MSB is 1, XOR with Rb

        K1 = (int.from\_bytes(K1, 'big') ^ Rb).to\_bytes(block\_size, 'big')

    K2 = left\_shift(K1)

    if K1[0] & 0x80:

        K2 = (int.from\_bytes(K2, 'big') ^ Rb).to\_bytes(block\_size, 'big')

    return K1, K2

aes\_key = b'\x2b\x7e\x15\x16\x28\xae\xd2\xa6\xab\xf7\xcf\x15\x88\x09\xcf\x4f'  # 16-byte AES key

des\_key = b'\x13\x34\x57\x79\x9B\xBC\xDF\xF1'  # 8-byte DES key

K1\_aes, K2\_aes = generate\_cmac\_subkeys(AES, aes\_key, 16)

print("AES CMAC Subkeys:")

print("K1:", binascii.hexlify(K1\_aes))

print("K2:", binascii.hexlify(K2\_aes))

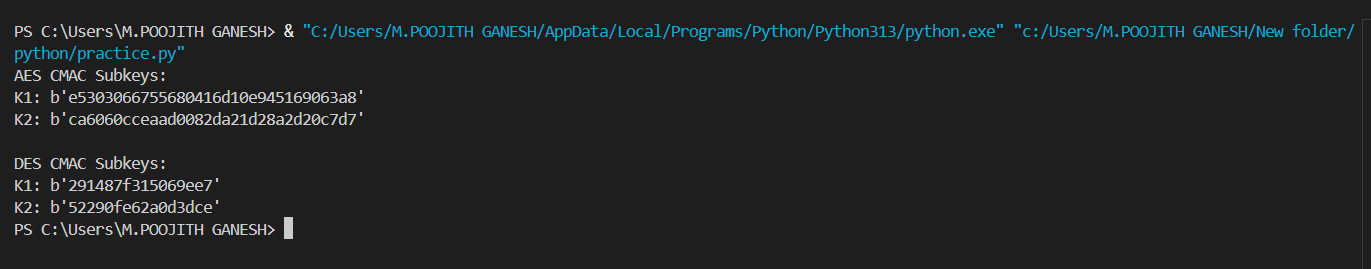
K1\_des, K2\_des = generate\_cmac\_subkeys(DES, des\_key, 8)

print("\nDES CMAC Subkeys:")

print("K1:", binascii.hexlify(K1\_des))

print("K2:", binascii.hexlify(K2\_des))

**Output:**



**4.**

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

key = get\_random\_bytes(16)

iv = get\_random\_bytes(16)

block\_size = 16

plaintext = b"HELLO WORLD AES!"

padded\_plaintext = pad(plaintext, block\_size)

ecb\_cipher = AES.new(key, AES.MODE\_ECB)

ecb\_ciphertext = ecb\_cipher.encrypt(padded\_plaintext)

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

cbc\_ciphertext = cbc\_cipher.encrypt(padded\_plaintext)

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

cfb\_ciphertext = cfb\_cipher.encrypt(plaintext)

ecb\_decipher = AES.new(key, AES.MODE\_ECB)

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

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

decrypted\_ecb = unpad(ecb\_decipher.decrypt(ecb\_ciphertext), block\_size)

decrypted\_cbc = unpad(cbc\_decipher.decrypt(cbc\_ciphertext), block\_size)

decrypted\_cfb = cfb\_decipher.decrypt(cfb\_ciphertext)

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

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

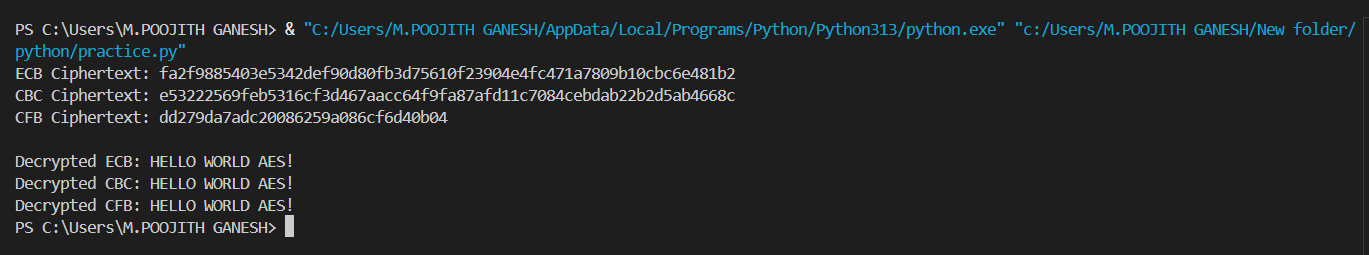
print("CFB Ciphertext:", cfb\_ciphertext.hex())

print("\nDecrypted ECB:", decrypted\_ecb.decode())

print("Decrypted CBC:", decrypted\_cbc.decode())

print("Decrypted CFB:", decrypted\_cfb.decode())

**Output:**



**5.**

from sympy import mod\_inverse

def affine\_encrypt(text, a, b):

    if gcd(a, 26) != 1:

        raise ValueError("a must be coprime with 26 for the cipher to be one-to-one.")

    cipher = ""

    for char in text.upper():

        if char.isalpha():

            p = ord(char) - ord('A')

            C = (a \* p + b) % 26

            cipher += chr(C + ord('A'))

        else:

            cipher += char

    return cipher

def affine\_decrypt(cipher, a, b):

    a\_inv = mod\_inverse(a, 26)

    plain = ""

    for char in cipher:

        if char.isalpha():

            C = ord(char) - ord('A')

            p = (a\_inv \* (C - b)) % 26

            plain += chr(p + ord('A'))

        else:

            plain += char

    return plain

def gcd(x, y):

    while y:

        x, y = y, x % y

    return x

a, b = 5, 8

plaintext = "HELLO WORLD"

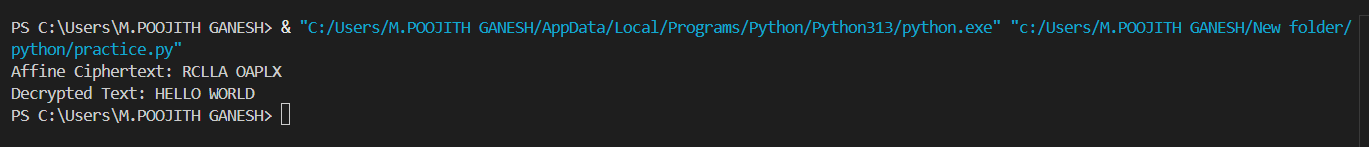
ciphertext = affine\_encrypt(plaintext, a, b)

decrypted\_text = affine\_decrypt(ciphertext, a, b)

print("Affine Ciphertext:", ciphertext)

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

**Output:**

****

**6.**

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad

import os

def xor\_bytes(a, b):

    """XOR two byte sequences."""

    return bytes(x ^ y for x, y in zip(a, b))

def generate\_cbc\_mac(key, message, block\_size=16):

    """Compute CBC-MAC of a one-block message."""

    cipher = AES.new(key, AES.MODE\_CBC, iv=b'\x00' \* block\_size)

    padded\_msg = pad(message, block\_size)

    mac = cipher.encrypt(padded\_msg)[-block\_size:]

    return mac

key = os.urandom(16)

message = b"HELLO BLOCK!"

T = generate\_cbc\_mac(key, message)

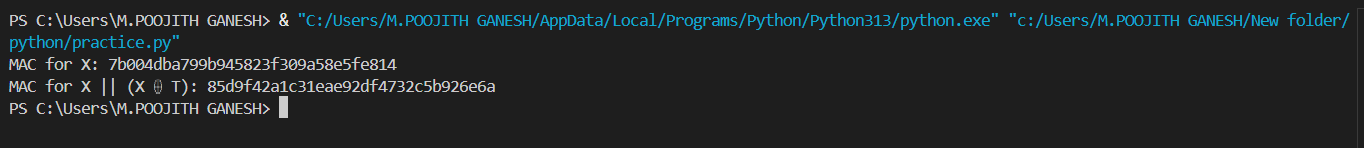
new\_message = message + xor\_bytes(message, T)

new\_mac = generate\_cbc\_mac(key, new\_message)

print("MAC for X:", T.hex())

print("MAC for X || (X ⊕ T):", new\_mac.hex())

**Output:**

****