Write a program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used in reverse order. Design a key-generation scheme with the appropriate shift schedule for the decryption process.

Sol:

from struct import pack, unpack

# Define the key schedules, permutations, etc.

# Initial Permutation (IP) table

IP = [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]

# Inverse of the Initial Permutation (IP^-1)

IP\_INV = [40, 8, 48, 16, 56, 24, 64, 32,

39, 7, 47, 15, 55, 23, 62, 30,

38, 6, 46, 14, 54, 22, 61, 29,

37, 5, 45, 13, 53, 21, 60, 28,

36, 4, 44, 12, 52, 20, 59, 27,

35, 3, 43, 11, 51, 19, 58, 26,

34, 2, 42, 10, 50, 18, 57, 25,

33, 1, 41, 9, 49, 17, 56, 24]

# Permutation Choice 1 (PC1)

PC1 = [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, 28, 20, 12, 4,

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

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

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

# Permutation Choice 2 (PC2)

PC2 = [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]

# Key shift schedule for each round

SHIFT\_SCHEDULE = [1, 1, 2, 2, 2, 2, 1, 2,

2, 2, 2, 2, 1, 2, 2, 1]

# Define the Feistel function (F function)

def feistel\_function(R, K):

# Apply expansion permutation (E) to R, then XOR with key K

# Implement the S-boxes and P4 permutation

pass # Implement the Feistel function

# Apply the initial permutation to the input block

def apply\_initial\_permutation(block):

return [block[i - 1] for i in IP]

# Apply the final permutation to the block

def apply\_final\_permutation(block):

return [block[i - 1] for i in IP\_INV]

# Apply permutation choice 1 (PC1) to the key

def apply\_pc1(key):

return [key[i - 1] for i in PC1]

# Apply permutation choice 2 (PC2) to the key

def apply\_pc2(key):

return [key[i - 1] for i in PC2]

# Perform left shift on the key halves

def left\_shift(half, shift\_count):

return half[shift\_count:] + half[:shift\_count]

# Generate the 16 round keys using PC1, shifts, and PC2

def generate\_keys(key):

key = apply\_pc1(key)

C = key[:28] # Left half of the key

D = key[28:] # Right half of the key

round\_keys = []

for i in range(16):

C = left\_shift(C, SHIFT\_SCHEDULE[i])

D = left\_shift(D, SHIFT\_SCHEDULE[i])

combined\_key = C + D

round\_keys.append(apply\_pc2(combined\_key))

return round\_keys[::-1] # Reverse the key order for decryption

# DES decryption function

def des\_decrypt(ciphertext, key):

# Step 1: Apply Initial Permutation (IP) on ciphertext

ciphertext = apply\_initial\_permutation(ciphertext)

# Step 2: Generate the 16 round keys in reverse order

round\_keys = generate\_keys(key)

# Step 3: Split the block into two halves

L = ciphertext[:32]

R = ciphertext[32:]

# Step 4: Perform 16 rounds of Feistel function with round keys in reverse order

for i in range(16):

temp = R

R = [L[j] ^ feistel\_function(R, round\_keys[i])[j] for j in range(32)]

L = temp

# Step 5: Swap halves and apply final permutation (IP^-1)

L, R = R, L

decrypted\_block = apply\_final\_permutation(L + R)

return decrypted\_block

# Example usage

key = [0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0,

1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0] # Sample 56-bit key (after parity bits)

ciphertext = [0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0,

0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0] # Example ciphertext

decrypted = des\_decrypt(ciphertext, key)

print(f'Decrypted Message: {decrypted}')

2) Write a program for encryption in the cipher block chaining (CBC) mode using an algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the definition of CBC. Which of the two would you choose:

Sol:

from Crypto.Cipher import DES3

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

import binascii

# Function to encrypt using 3DES in CBC mode

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

# Generate a random IV for CBC mode (8 bytes for 3DES)

iv = get\_random\_bytes(8)

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

# Pad the plaintext to make its length a multiple of the block size (8 bytes)

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

# Encrypt the data

ciphertext = cipher.encrypt(padded\_data)

# Return the IV and ciphertext concatenated (IV is needed for decryption)

return iv + ciphertext

# Function to decrypt using 3DES in CBC mode

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

# Extract the IV from the first 8 bytes of the ciphertext

iv = ciphertext[:8]

ciphertext = ciphertext[8:]

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

# Decrypt the data

padded\_data = cipher.decrypt(ciphertext)

# Unpad the decrypted data

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

return plaintext

# Example Usage

key = get\_random\_bytes(24) # 3DES uses a 24-byte key (168 bits)

plaintext = "This is a secure message."

# Encrypt

ciphertext = encrypt\_3des\_cbc(plaintext, key)

print(f"Ciphertext (in hex): {binascii.hexlify(ciphertext)}")

# Decrypt

decrypted\_message = decrypt\_3des\_cbc(ciphertext, key)

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

3) Write a program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1l bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed"

Sol:

from Crypto.Cipher import DES3

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

import binascii

# Function to apply padding (1 bit followed by zeroes to complete the block)

def apply\_custom\_padding(data):

block\_size = 8 # 3DES operates on 8-byte blocks

padding\_len = block\_size - len(data) % block\_size

return data + bytes([0x80] + [0] \* (padding\_len - 1))

# Function to encrypt using 3DES in ECB mode

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

cipher = DES3.new(key, DES3.MODE\_ECB)

padded\_data = apply\_custom\_padding(plaintext.encode())

ciphertext = cipher.encrypt(padded\_data)

return ciphertext

# Function to decrypt using 3DES in ECB mode

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

cipher = DES3.new(key, DES3.MODE\_ECB)

padded\_data = cipher.decrypt(ciphertext)

return unpad(padded\_data, DES3.block\_size).decode()

# Function to encrypt using 3DES in CBC mode

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

iv = get\_random\_bytes(8) # 8-byte IV for CBC

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

padded\_data = apply\_custom\_padding(plaintext.encode())

ciphertext = cipher.encrypt(padded\_data)

return iv + ciphertext # Return IV + ciphertext for decryption

# Function to decrypt using 3DES in CBC mode

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

iv = ciphertext[:8]

ciphertext = ciphertext[8:]

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

padded\_data = cipher.decrypt(ciphertext)

return unpad(padded\_data, DES3.block\_size).decode()

# Function to encrypt using 3DES in CFB mode

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

iv = get\_random\_bytes(8) # 8-byte IV for CFB

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

padded\_data = apply\_custom\_padding(plaintext.encode())

ciphertext = cipher.encrypt(padded\_data)

return iv + ciphertext # Return IV + ciphertext for decryption

# Function to decrypt using 3DES in CFB mode

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

iv = ciphertext[:8]

ciphertext = ciphertext[8:]

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

padded\_data = cipher.decrypt(ciphertext)

return unpad(padded\_data, DES3.block\_size).decode()

# Example Usage

key = get\_random\_bytes(24) # 3DES uses a 24-byte key

plaintext = "This is a test message."

# Encrypt and Decrypt in ECB mode

ciphertext\_ecb = encrypt\_3des\_ecb(plaintext, key)

decrypted\_message\_ecb = decrypt\_3des\_ecb(ciphertext\_ecb, key)

print(f"ECB - Ciphertext (hex): {binascii.hexlify(ciphertext\_ecb)}")

print(f"ECB - Decrypted message: {decrypted\_message\_ecb}")

# Encrypt and Decrypt in CBC mode

ciphertext\_cbc = encrypt\_3des\_cbc(plaintext, key)

decrypted\_message\_cbc = decrypt\_3des\_cbc(ciphertext\_cbc, key)

print(f"CBC - Ciphertext (hex): {binascii.hexlify(ciphertext\_cbc)}")

print(f"CBC - Decrypted message: {decrypted\_message\_cbc}")

# Encrypt and Decrypt in CFB mode

ciphertext\_cfb = encrypt\_3des\_cfb(plaintext, key)

decrypted\_message\_cfb = decrypt\_3des\_cfb(ciphertext\_cfb, key)

print(f"CFB - Ciphertext (hex): {binascii.hexlify(ciphertext\_cfb)}")

print(f"CFB - Decrypted message: {decrypted\_message\_cfb}")

Write a program for Encrypt and decrypt in cipher block chaining mode using one of the following ciphers affine modulo 256. Hill modulo 256, S-DES, DES. Test data for S-DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001 0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of 1111 0100 0000 1011. Decryption should work correspondingly.

Sol:

def sdes\_permutation(p, table):

"""Permutation or Initial/Final permutation"""

return ''.join(p[i - 1] for i in table)

def sdes\_key\_generation(key):

"""Generate the subkeys for S-DES"""

# P10 permutation table

p10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]

# P8 permutation table

p8 = [6, 3, 7, 4, 8, 5, 10, 9]

# Left and Right shifts

shifts = [1, 2]

key = sdes\_permutation(key, p10) # Initial P10 permutation

left = key[:5]

right = key[5:]

subkeys = []

for shift in shifts:

left = left[shift:] + left[:shift]

right = right[shift:] + right[:shift]

subkey = sdes\_permutation(left + right, p8) # Combine and permute for the subkey

subkeys.append(subkey)

return subkeys

def sdes\_f\_function(data, subkey):

"""The F function (S-Box and permutation)"""

# P4 permutation table

p4 = [2, 4, 3, 1]

# S-Boxes

s1 = [

[1, 0, 3, 2],

[3, 2, 1, 0],

[0, 2, 1, 3],

[3, 1, 0, 2]

]

s2 = [

[0, 1, 2, 3],

[2, 0, 3, 1],

[3, 0, 1, 2],

[2, 1, 0, 3]

]

left, right = data[:4], data[4:]

# Apply subkey to right side

combined = bin(int(left, 2) ^ int(subkey, 2))[2:].zfill(4)

# S-Box substitution

row1, col1 = int(combined[0] + combined[3], 2), int(combined[1] + combined[2], 2)

row2, col2 = int(combined[4] + combined[7], 2), int(combined[5] + combined[6], 2)

s1\_result = s1[row1][col1]

s2\_result = s2[row2][col2]

return bin(int(s1\_result, 2))[2:].zfill(2)

def sdes\_encrypt(plaintext, key):

"""Encrypt the 8-bit plaintext with the S-DES algorithm"""

subkeys = sdes\_key\_generation(key)

# Initial permutation table (IP)

ip = [2, 6, 3, 1, 4, 8, 5, 7]

ip\_reversed = [4, 1, 3, 5, 7, 2, 8, 6]

# Perform Initial permutation (IP)

text = sdes\_permutation(plaintext, ip)

left, right = text[:4], text[4:]

# First round (using subkey1)

f\_output = sdes\_f\_function(left, subkeys[0])

left = f\_output

# Swap the parts (1st round)

text = left + right

# Final permutation (FP)

result = sdes\_permutation(text, ip\_reversed)

return result

def xor\_bits(bits1, bits2):

"""XOR two binary strings"""

return ''.join(str(int(a) ^ int(b)) for a, b in zip(bits1, bits2))

# CBC mode encryption and decryption

def cbc\_encrypt(plaintext, key, iv):

"""Encrypt plaintext in CBC mode using S-DES"""

ciphertext = []

previous\_block = iv

for i in range(0, len(plaintext), 8):

block = plaintext[i:i+8]

# XOR plaintext block with previous ciphertext block (or IV for the first block)

xored = xor\_bits(block, previous\_block)

encrypted\_block = sdes\_encrypt(xored, key)

ciphertext.append(encrypted\_block)

previous\_block = encrypted\_block

return ''.join(ciphertext)

def cbc\_decrypt(ciphertext, key, iv):

"""Decrypt ciphertext in CBC mode using S-DES"""

plaintext = []

previous\_block = iv

for i in range(0, len(ciphertext), 8):

block = ciphertext[i:i+8]

decrypted\_block = sdes\_encrypt(block, key)

# XOR with the previous ciphertext block (or IV for the first block)

decrypted = xor\_bits(decrypted\_block, previous\_block)

plaintext.append(decrypted)

previous\_block = block

return ''.join(plaintext)

# Example usage

plaintext = "0000000100100011" # Example plaintext (16 bits, 2 blocks)

key = "0111111101" # Example key (10 bits)

iv = "10101010" # Binary initialization vector (IV)

# Encryption in CBC mode

ciphertext = cbc\_encrypt(plaintext, key, iv)

print(f"Ciphertext: {ciphertext}")

# Decryption in CBC mode

decrypted\_message = cbc\_decrypt(ciphertext, key, iv)

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

Write a program for RSA system, the public key of a given user is e31, n-3599. What is the private key of this user? Hint. First use trial-and-error to determine p and q. then use the extended Euclidean algorithm to find the multiplicative invetse of 31 modulo f(n).

Sol:

def extended\_euclidean(a, b):

"""Extended Euclidean algorithm to find the gcd and the coefficients (x, y) such that ax + by = gcd(a, b)"""

if b == 0:

return (a, 1, 0)

else:

gcd, x1, y1 = extended\_euclidean(b, a % b)

x = y1

y = x1 - (a // b) \* y1

return (gcd, x, y)

def mod\_inverse(e, phi\_n):

"""Find the modular inverse of e mod phi\_n using the extended Euclidean algorithm"""

gcd, x, y = extended\_euclidean(e, phi\_n)

if gcd != 1:

raise ValueError(f"Modular inverse does not exist for e = {e} and phi\_n = {phi\_n}")

else:

return x % phi\_n

# Step 1: Factor n = 3599 to find p and q

n = 3599

# Trial and error to find factors of n

for p in range(2, int(n \*\* 0.5) + 1):

if n % p == 0:

q = n // p

break

# Now we have p and q

print(f"p = {p}, q = {q}")

# Step 2: Calculate Euler's totient function, phi(n)

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

print(f"phi(n) = {phi\_n}")

# Step 3: Find the modular inverse of e = 31 modulo phi(n)

e = 31

d = mod\_inverse(e, phi\_n)

print(f"Private key d = {d}")

rite a program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?

Sol:

def mod\_exp(base, exponent, modulus):

"""Modular exponentiation: (base^exponent) % modulus"""

return pow(base, exponent, modulus)

def diffie\_hellman\_protocol(q, a, x\_A, x\_B):

"""Perform Diffie-Hellman key exchange protocol."""

# Alice computes A = a^x\_A % q and sends it to Bob

A = mod\_exp(a, x\_A, q)

# Bob computes B = a^x\_B % q and sends it to Alice

B = mod\_exp(a, x\_B, q)

# Alice computes the shared secret S\_A = B^x\_A % q

S\_A = mod\_exp(B, x\_A, q)

# Bob computes the shared secret S\_B = A^x\_B % q

S\_B = mod\_exp(A, x\_B, q)

# Verify both computed secrets are equal

if S\_A == S\_B:

print(f"Shared secret established: {S\_A}")

else:

print("Error in key exchange.")

# Example usage

q = 23 # A small prime number for simplicity

a = 5 # A primitive root modulo q

# Alice and Bob's secret private keys

x\_A = 6

x\_B = 15

# Run the Diffie-Hellman protocol

diffie\_hellman\_protocol(q, a, x\_A, x\_B)