CSA5103 Cryptography

**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.**

# Import required module

from typing import List

# Initial permutation 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]

# Final permutation (IP-1)

FP = [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]

# Expansion table

E = [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-boxes

S\_BOXES = [

# S1

[[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]],

# S2

[[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]],

# S3

[[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]],

# S4

[[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]],

# S5

[[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]],

# S6

[[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]],

# S7

[[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]],

# S8

[[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]]

]

# Permutation P

P = [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]

# PC-1 and PC-2

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,

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]

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]

# Shift schedule

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

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

def permute(bits, table):

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

def xor(a, b):

return ''.join(str(int(x)^int(y)) for x, y in zip(a, b))

def sbox\_substitution(bits):

output = ''

for i in range(8):

block = bits[i\*6:(i+1)\*6]

row = int(block[0] + block[5], 2)

col = int(block[1:5], 2)

val = S\_BOXES[i][row][col]

output += format(val, '04b')

return output

def feistel(right, key):

expanded = permute(right, E)

xored = xor(expanded, key)

substituted = sbox\_substitution(xored)

return permute(substituted, P)

def generate\_keys(key64: str) -> List[str]:

key56 = permute(key64, PC1)

C, D = key56[:28], key56[28:]

keys = []

for shift in SHIFT\_SCHEDULE:

C = C[shift:] + C[:shift]

D = D[shift:] + D[:shift]

keys.append(permute(C + D, PC2))

return keys

def des\_decrypt(ciphertext64: str, key64: str) -> str:

keys = generate\_keys(key64)

keys.reverse() # Use in reverse for decryption

permuted = permute(ciphertext64, IP)

L, R = permuted[:32], permuted[32:]

for k in keys:

temp = R

R = xor(L, feistel(R, k))

L = temp

combined = R + L # Note the swap after last round

final = permute(combined, FP)

return final

# Example

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

cipher = '1010101010111011000010010001100000100111001101101100110011011101' # example 64-bit binary ciphertext

key = '0001001100110100010101110111100110011011101111001101111111110001' # example 64-bit binary key

decrypted = des\_decrypt(cipher, key)

print(f"Decrypted binary: {decrypted}")

**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:**

1. **For security? b. For performance?**

def xor(a, b):

return ''.join(str(int(x)^int(y)) for x, y in zip(a, b))

def text\_to\_blocks(text):

binary = ''.join(format(ord(c), '08b') for c in text)

while len(binary) % 64 != 0:

binary += '00000000'

return [binary[i:i+64] for i in range(0, len(binary), 64)]

def binary\_to\_text(binary):

chars = [chr(int(binary[i:i+8], 2)) for i in range(0, len(binary), 8)]

return ''.join(chars).rstrip('\x00')

def simple\_des\_encrypt(block, key):

return xor(block, key[:64])

def triple\_des\_encrypt\_block(block, k1, k2, k3):

block = simple\_des\_encrypt(block, k1)

block = simple\_des\_encrypt(block, k2)

block = simple\_des\_encrypt(block, k3)

return block

def encrypt\_3des\_cbc(plaintext, k1, k2, k3, iv):

blocks = text\_to\_blocks(plaintext)

ciphertext\_blocks = []

prev = iv

for block in blocks:

xored = xor(block, prev)

encrypted = triple\_des\_encrypt\_block(xored, k1, k2, k3)

ciphertext\_blocks.append(encrypted)

prev = encrypted

return ciphertext\_blocks

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

plaintext = "HELLO 3DES CBC MODE"

k1 = '0101010101010101010101010101010101010101010101010101010101010101'

k2 = '0011001100110011001100110011001100110011001100110011001100110011'

k3 = '1111000011110000111100001111000011110000111100001111000011110000'

iv = '0000000000000000000000000000000000000000000000000000000000000000'

cipher\_blocks = encrypt\_3des\_cbc(plaintext, k1, k2, k3, iv)

ciphertext = ''.join(cipher\_blocks)

print("Encrypted (binary):", ciphertext)

**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 1 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?**

BLOCK\_SIZE = 8

def pad(data):

pad\_len = BLOCK\_SIZE - (len(data) % BLOCK\_SIZE)

return data + bytes([0x80]) + bytes([0x00] \* (pad\_len - 1))

def xor\_bytes(a, b):

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

def block\_encrypt(block, key):

return xor\_bytes(block, key[:BLOCK\_SIZE])

def encrypt\_ecb(plaintext, key):

plaintext = pad(plaintext)

blocks = [plaintext[i:i+BLOCK\_SIZE] for i in range(0, len(plaintext), BLOCK\_SIZE)]

return b''.join(block\_encrypt(block, key) for block in blocks)

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

plaintext = pad(plaintext)

blocks = [plaintext[i:i+BLOCK\_SIZE] for i in range(0, len(plaintext), BLOCK\_SIZE)]

ciphertext = []

prev = iv

for block in blocks:

xored = xor\_bytes(block, prev)

encrypted = block\_encrypt(xored, key)

ciphertext.append(encrypted)

prev = encrypted

return b''.join(ciphertext)

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

blocks = [plaintext[i:i+BLOCK\_SIZE] for i in range(0, len(plaintext), BLOCK\_SIZE)]

ciphertext = []

prev = iv

for block in blocks:

encrypted = block\_encrypt(prev, key)

cipher\_block = xor\_bytes(block, encrypted[:len(block)])

ciphertext.append(cipher\_block)

prev = cipher\_block

return b''.join(ciphertext)

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

iv = b"InitVect"

key = b"K3yM0ck!"

plaintext = b"Hello CBC ECB CFB modes in Python!"

print("ECB:", encrypt\_ecb(plaintext, key).hex())

print("CBC:", encrypt\_cbc(plaintext, key, iv).hex())

print("CFB:", encrypt\_cfb(plaintext, key, iv).hex())

**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.**

P10 = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]

P8 = [6, 3, 7, 4, 8, 5, 10, 9]

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

IP\_inv = [4, 1, 3, 5, 7, 2, 8, 6]

EP = [4, 1, 2, 3, 2, 3, 4, 1]

P4 = [2, 4, 3, 1]

S0 = [[1, 0, 3, 2],

[3, 2, 1, 0],

[0, 2, 1, 3],

[3, 1, 3, 2]]

S1 = [[0, 1, 2, 3],

[2, 0, 1, 3],

[3, 0, 1, 0],

[2, 1, 0, 3]]

def permute(bits, table):

return [bits[i - 1] for i in table]

def left\_shift(bits, count):

return bits[count:] + bits[:count]

def xor(bits1, bits2):

return [b1 ^ b2 for b1, b2 in zip(bits1, bits2)]

def sbox\_lookup(bits, sbox):

row = (bits[0] << 1) | bits[3]

col = (bits[1] << 1) | bits[2]

val = sbox[row][col]

return [int(b) for b in format(val, '02b')]

def generate\_keys(key):

key = permute(key, P10)

left, right = key[:5], key[5:]

left, right = left\_shift(left, 1), left\_shift(right, 1)

k1 = permute(left + right, P8)

left, right = left\_shift(left, 2), left\_shift(right, 2)

k2 = permute(left + right, P8)

return k1, k2

def fk(bits, key):

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

expanded = permute(right, EP)

temp = xor(expanded, key)

left\_sbox = sbox\_lookup(temp[:4], S0)

right\_sbox = sbox\_lookup(temp[4:], S1)

p4 = permute(left\_sbox + right\_sbox, P4)

return xor(left, p4) + right

def sdes\_encrypt(plaintext, k1, k2):

bits = permute(plaintext, IP)

bits = fk(bits, k1)

bits = bits[4:] + bits[:4]

bits = fk(bits, k2)

return permute(bits, IP\_inv)

def sdes\_decrypt(ciphertext, k1, k2):

bits = permute(ciphertext, IP)

bits = fk(bits, k2)

bits = bits[4:] + bits[:4]

bits = fk(bits, k1)

return permute(bits, IP\_inv)

def str\_to\_bits(s):

return [int(b) for b in s]

def bits\_to\_str(b):

return ''.join(str(bit) for bit in b)

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

k1, k2 = generate\_keys(key\_bits)

ciphertext = []

prev = iv\_bits

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

block = plaintext\_bits[i:i+8]

xored = xor(block, prev)

encrypted = sdes\_encrypt(xored, k1, k2)

ciphertext.extend(encrypted)

prev = encrypted

return ciphertext

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

k1, k2 = generate\_keys(key\_bits)

plaintext = []

prev = iv\_bits

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

block = ciphertext\_bits[i:i+8]

decrypted = sdes\_decrypt(block, k1, k2)

xored = xor(decrypted, prev)

plaintext.extend(xored)

prev = block

return plaintext

plaintext\_bin = "0000000100100011"

key\_bin = "0111111101"

iv\_bin = "10101010"

plaintext\_bits = str\_to\_bits(plaintext\_bin)

key\_bits = str\_to\_bits(key\_bin)

iv\_bits = str\_to\_bits(iv\_bin)

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

print("Ciphertext:", bits\_to\_str(ciphertext))

expected = "1111010000001011"

print("Matches expected:", bits\_to\_str(ciphertext) == expected)

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

print("Decrypted:", bits\_to\_str(decrypted))

print("Matches original:", bits\_to\_str(decrypted) == plaintext\_bin)

**Write a program for RSA system, the public key of a given user is e = 31, 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 inverse of 31 modulo f(n).**

def gcd(a, b):

while b:

a, b = b, a % b

return a

def modinv(e, phi):

old\_r, r = e, phi

old\_s, s = 1, 0

while r != 0:

q = old\_r // r

old\_r, r = r, old\_r - q \* r

old\_s, s = s, old\_s - q \* s

if old\_r != 1:

raise Exception("No modular inverse")

else:

return old\_s % phi

def factor\_n(n):

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

if n % i == 0:

return i, n // i

return None, None

def rsa\_keygen(n, e):

p, q = factor\_n(n)

if not p or not q:

raise ValueError("Failed to factor n")

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

d = modinv(e, phi)

print(f"Found primes: p = {p}, q = {q}")

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

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

return d

def encrypt(m, e, n):

return pow(m, e, n)

def decrypt(c, d, n):

return pow(c, d, n)

n = 3599

e = 31

d = rsa\_keygen(n, e)

message = 42

cipher = encrypt(message, e, n)

decrypted = decrypt(cipher, d, n)

print(f"Message: {message}")

print(f"Encrypted: {cipher}")

print(f"Decrypted: {decrypted}")

**Write 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?**

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

result = 1

while exponent > 0:

if exponent % 2 == 1:

result = (result \* base) % modulus

base = (base \* base) % modulus

exponent //= 2

return result

def diffie\_hellman():

p = 23

a = 5

x\_A = 6

x\_B = 15

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

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

K\_A = mod\_exp(B, x\_A, p)

K\_B = mod\_exp(A, x\_B, p)

print(f"Public parameters: p = {p}, a = {a}")

print(f"Alice's private key: {x\_A}")

print(f"Bob's private key: {x\_B}")

print(f"Alice's public key (A): {A}")

print(f"Bob's public key (B): {B}")

print(f"Alice's shared secret (K\_A): {K\_A}")

print(f"Bob's shared secret (K\_B): {K\_B}")

print("Shared secret match:", K\_A == K\_B)

diffie\_hellman()