Student Name: Anusha Bondili

Registration Number : AP24135010025

Course: Ph.D.

1. Implementation of Symmetric cipher algorithm (AES)

**Aim:** To implement a Symmetric Cipher Algorithm – AES encryption and decryption

**Source Code:** (Without Library Files)

*"""*

*AES-128 implementation – Python Code*

*-------------------------------------------------*

*Supports:*

*- Block encrypt/decrypt (16 bytes)*

*- CBC mode wrapper with PKCS7 padding*

*- User input for plaintext, key, IV*

*"""*

***# ---------- AES constants – Defining a S-Box ----------***

SBOX = [

    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,

    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,

    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,

    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,

    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,

    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,

    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,

    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,

    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,

    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,

    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,

    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,

    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,

    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,

    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,

    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16,

]

***# Inverse S-box (for decryption) – for constructing Inverse S-Box for Decrypting***

INV\_SBOX = [0]\*256

for i, v in enumerate(SBOX):

    INV\_SBOX[v] = i

***# Round constants for key expansion***

RCON = [0x00,0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80,0x1B,0x36]

***# ---------- finite field helpers ----------***

def mul(a, b):

*"""Multiply two numbers in GF(2^8) field"""*

    res = 0

    for \_ in range(8):

        if b & 1:

            res ^= a

        carry = a & 0x80

        a = ((a << 1) & 0xFF)

        if carry:

            a ^= 0x1B

        b >>= 1

    return res

***# ---------- AES transformations ----------***

def sub\_bytes(state): return [SBOX[b] for b in state]

def inv\_sub\_bytes(state): return [INV\_SBOX[b] for b in state]

def shift\_rows(state):

*"""Rotate each row by different offsets"""*

    s = state[:]

    s[1], s[5], s[9], s[13] = state[5], state[9], state[13], state[1]

    s[2], s[6], s[10], s[14] = state[10], state[14], state[2], state[6]

    s[3], s[7], s[11], s[15] = state[15], state[3], state[7], state[11]

    return s

def inv\_shift\_rows(state):

*"""Inverse row rotations"""*

    s = state[:]

    s[1], s[5], s[9], s[13] = state[13], state[1], state[5], state[9]

    s[2], s[6], s[10], s[14] = state[10], state[14], state[2], state[6]

    s[3], s[7], s[11], s[15] = state[7], state[11], state[15], state[3]

    return s

def mix\_columns(state):

*"""Mix columns using GF(2^8) matrix multiplication"""*

    s = state[:]

    for c in range(4):

        i = 4\*c

        a0, a1, a2, a3 = s[i], s[i+1], s[i+2], s[i+3]

        s[i+0] = mul(0x02, a0) ^ mul(0x03, a1) ^ a2 ^ a3

        s[i+1] = a0 ^ mul(0x02, a1) ^ mul(0x03, a2) ^ a3

        s[i+2] = a0 ^ a1 ^ mul(0x02, a2) ^ mul(0x03, a3)

        s[i+3] = mul(0x03, a0) ^ a1 ^ a2 ^ mul(0x02, a3)

    return s

def inv\_mix\_columns(state):

*"""Inverse MixColumns"""*

    s = state[:]

    for c in range(4):

        i = 4\*c

        a0, a1, a2, a3 = s[i], s[i+1], s[i+2], s[i+3]

        s[i+0] = mul(0x0e, a0) ^ mul(0x0b, a1) ^ mul(0x0d, a2) ^ mul(0x09, a3)

        s[i+1] = mul(0x09, a0) ^ mul(0x0e, a1) ^ mul(0x0b, a2) ^ mul(0x0d, a3)

        s[i+2] = mul(0x0d, a0) ^ mul(0x09, a1) ^ mul(0x0e, a2) ^ mul(0x0b, a3)

        s[i+3] = mul(0x0b, a0) ^ mul(0x0d, a1) ^ mul(0x09, a2) ^ mul(0x0e, a3)

    return s

def add\_round\_key(state, round\_key):

*"""XOR state with round key"""*

    return [b ^ rk for b, rk in zip(state, round\_key)]

***# ---------- key expansion ----------***

def key\_expansion(key16):

*"""Expand 16-byte key into 176-byte round key schedule"""*

    assert len(key16) == 16

    expanded = list(key16)

    i = 16

    rcon\_iter = 1

    while len(expanded) < 176:

        t = expanded[-4:]

        if i % 16 == 0:

            t = t[1:] + t[:1]         # RotWord

            t = [SBOX[b] for b in t]  # SubWord

            t[0] ^= RCON[rcon\_iter]   # Rcon

            rcon\_iter += 1

        for j in range(4):

            expanded.append(expanded[i-16] ^ t[j])

            i += 1

    return expanded

***# ---------- block encrypt/decrypt ----------***

def encrypt\_block(block16, expanded\_key):

*"""Encrypt a single 16-byte block"""*

    state = list(block16)

    state = add\_round\_key(state, expanded\_key[0:16])

    for round in range(1, 10):  *# 9 main rounds*

        state = sub\_bytes(state)

        state = shift\_rows(state)

        state = mix\_columns(state)

        state = add\_round\_key(state, expanded\_key[16\*round : 16\*(round+1)])

*# final round*

    state = sub\_bytes(state)

    state = shift\_rows(state)

    state = add\_round\_key(state, expanded\_key[160:176])

    return bytes(state)

def decrypt\_block(block16, expanded\_key):

*"""Decrypt a single 16-byte block"""*

    state = list(block16)

    state = add\_round\_key(state, expanded\_key[160:176])

    for round in range(9, 0, -1):

        state = inv\_shift\_rows(state)

        state = inv\_sub\_bytes(state)

        state = add\_round\_key(state, expanded\_key[16\*round : 16\*(round+1)])

        state = inv\_mix\_columns(state)

    state = inv\_shift\_rows(state)

    state = inv\_sub\_bytes(state)

    state = add\_round\_key(state, expanded\_key[0:16])

    return bytes(state)

***# ---------- CBC mode + padding ----------***

def pkcs7\_pad(data):

    pad\_len = 16 - (len(data) % 16)

    return data + bytes([pad\_len]) \* pad\_len

def pkcs7\_unpad(data):

    pad\_len = data[-1]

    return data[:-pad\_len]

def xor\_bytes(a, b):

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

def encrypt\_cbc(plaintext, key16, iv16):

*"""Encrypt plaintext in CBC mode"""*

    expanded = key\_expansion(key16)

    data = pkcs7\_pad(plaintext)

    blocks = [data[i:i+16] for i in range(0, len(data), 16)]

    out, prev = b"", iv16

    for block in blocks:

        x = xor\_bytes(block, prev)

        c = encrypt\_block(x, expanded)

        out += c

        prev = c

    return out

def decrypt\_cbc(ciphertext, key16, iv16):

*"""Decrypt ciphertext in CBC mode"""*

    expanded = key\_expansion(key16)

    blocks = [ciphertext[i:i+16] for i in range(0, len(ciphertext), 16)]

    out, prev = b"", iv16

    for block in blocks:

        d = decrypt\_block(block, expanded)

        p = xor\_bytes(d, prev)

        out += p

        prev = block

    return pkcs7\_unpad(out)

**# ---------- Example usage ----------**

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

*# 1. Take plaintext input*

    plaintext = input("Enter text to encrypt: ").encode()

*# 2. Take key/IV (any length) and adjust to 16 bytes*

    key = input("Enter key (any length, will be adjusted to 16 bytes): ").encode()

    iv  = input("Enter IV  (any length, will be adjusted to 16 bytes): ").encode()

*# Pad/truncate key and IV to 16 bytes*

    key = (key + b'0'\*16)[:16]

    iv  = (iv  + b'0'\*16)[:16]

    print("\n[+] Using key:", key)

    print("[+] Using IV :", iv)

***# 3. Encrypt***

    ct = encrypt\_cbc(plaintext,key,iv)

    print("\n🔒 Ciphertext (hex):", ct.hex())

***# 4. Decrypt***

    pt = decrypt\_cbc(ct,key,iv)

    print("🔓 Decrypted:", pt.decode(errors="ignore"))

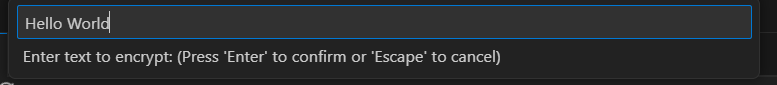
***# 5. Verify***

    if pt==plaintext:

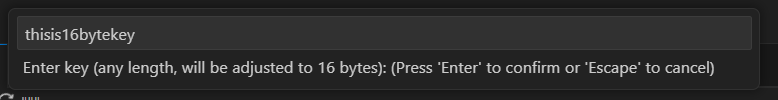
        print("✅ OK - decrypted matches original")

**Output:**

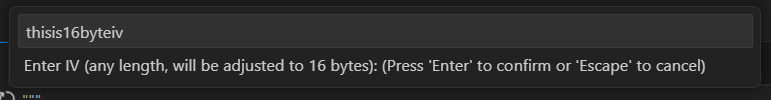
Enter Plain Text:



Enter Key:



Enter IV(Initialization Vector):



After Processing Encryption and Decryption, the Final Output:

