Source Code

sbox.py

```
"""
sbox.py
Contains S-box and inverse S-box tables, and SubBytes/InvSubBytes functions
for AES.
"""
d
# AES S-box (substitution box)
sbox = [
    # 0     1     2     3     4     5     6     7     8     9
A     B     C     D     E     F
    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
]

# AES inverse S-box
inv_sbox = [
```

```
    0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e,
0x81, 0xf3, 0xd7, 0xfb,
    0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44,
0xc4, 0xde, 0xe9, 0xcb,
    0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b,
0x42, 0xfa, 0xc3, 0x4e,
    0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49,
0x6d, 0x8b, 0xd1, 0x25,
    0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc,
0x5d, 0x65, 0xb6, 0x92,
    0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57,
0xa7, 0x8d, 0x9d, 0x84,
    0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05,
0xb8, 0xb3, 0x45, 0x06,
    0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03,
0x01, 0x13, 0x8a, 0x6b,
    0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce,
0xf0, 0xb4, 0xe6, 0x73,
    0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
0x1c, 0x75, 0xdf, 0x6e,
    0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e,
0xaa, 0x18, 0xbe, 0x1b,
    0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe,
0x78, 0xcd, 0x5a, 0xf4,
    0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59,
0x27, 0x80, 0xec, 0x5f,
    0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f,
0x93, 0xc9, 0x9c, 0xef,
    0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c,
0x83, 0x53, 0x99, 0x61,
    0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
0x55, 0x21, 0x0c, 0x7d
]

def sub_bytes(state):
    """
    Applies S-box substitution to each byte in the state.
    state: 4x4 matrix of bytes
    Returns new state matrix.
    """
    return [[sbox[b] for b in row] for row in state]

def inv_sub_bytes(state):
    """
    Applies inverse S-box substitution to each byte in the state.
    state: 4x4 matrix of bytes
    Returns new state matrix.
    """
    return [[inv_sbox[b] for b in row] for row in state]

# Helper for SubWord (used in key expansion)
def sub_word(word):
    """
    Applies S-box to a 4-byte word.
```

```
    word: list of 4 bytes
    Returns new word.
    """
    return [sbox[b] for b in word]

# Helper for InvSubWord (not needed for key expansion)
def inv_sub_word(word):
    """
    Applies inverse S-box to a 4-byte word.
    word: list of 4 bytes
    Returns new word.
    """
    return [inv_sbox[b] for b in word]
```

key_expansion.py

```
"""
key_expansion.py
Implements AES key expansion for 128-bit keys, with RotWord, SubWord, Rcon,
and key schedule logic.
"""
from sbox import sub_word

# AES parameters for 128-bit key
Nk = 4  # Number of 32-bit words in key
Nb = 4  # Number of columns (block size in words)
Nr = 10 # Number of rounds

# Round constants (Rcon)
def rcon_gen():
    """
    Generates Rcon array for AES key expansion.
    Each Rcon[i] is [RCi, 0x00, 0x00, 0x00], RCi = 2^(i-1) in GF(2^8)
    """
    rcon = [[0x00, 0x00, 0x00, 0x00]]
    rc = 1
    for i in range(1, 11):
        rcon.append([rc, 0x00, 0x00, 0x00])
        # Multiply rc by 2 in GF(2^8)
        rc = rc << 1
        if rc & 0x100:
            rc ^= 0x11b
    return rcon

Rcon = rcon_gen()

def rot_word(word):
    """
    RotWord: Circular left shift of a word by 1 byte.
    [a0, a1, a2, a3] -> [a1, a2, a3, a0]
    """
```

```
    return word[1:] + word[:1]

# Key expansion function
def key_expansion(key_bytes):
    """
    Expands 128-bit key into 44 words (4 bytes each) for AES-128.
    key_bytes: list of 16 bytes
    Returns: list of 44 words (each word is 4 bytes)
    """
    w = []
    # Step 1: Copy original key
    for i in range(Nk):
        w.append(key_bytes[4*i:4*i+4])
    # Step 2: Expand key
    for i in range(Nk, Nb*(Nr+1)):
        temp = w[i-1].copy()
        if i % Nk == 0:
            temp = rot_word(temp)
            temp = sub_word(temp)
            # XOR with round constant
            temp = [t ^ r for t, r in zip(temp, Rcon[i//Nk])]
        w.append([wi ^ ti for wi, ti in zip(w[i-Nk], temp)])
    return w

# Example usage (for testing):
if __name__ == "__main__":
    # Example 128-bit key (hex string)
    key_hex = "2b7e151628aed2a6abf7158809cf4f3c"
    key_bytes = [int(key_hex[i:i+2], 16) for i in range(0, 32, 2)]
    expanded = key_expansion(key_bytes)
    print("Expanded Key Schedule:")
    for i, word in enumerate(expanded):
        print(f"w[{i}]:", [hex(b) for b in word])
```

aes_core.py

```
"""
aes_core.py
Implements AES block operations: AddRoundKey, ShiftRows, MixColumns, and
their inverses.
"""
from sbox import sub_bytes, inv_sub_bytes

# AddRoundKey: XOR state with round key
def add_round_key(state, round_key):
    """
    XORs each byte of the state with the round key.
    state: 4x4 matrix
    round_key: 4x4 matrix
    Returns new state matrix.
```

```
    """
    return [[state[r][c] ^ round_key[r][c] for c in range(4)] for r in
range(4)]

# ShiftRows operation
def shift_rows(state):
    """
    Cyclically shifts each row by its index.
    Row 0: no shift, Row 1: shift left by 1, etc.
    """
    return [
        state[0],
        state[1][1:] + state[1][:1],
        state[2][2:] + state[2][:2],
        state[3][3:] + state[3][:3]
    ]

def inv_shift_rows(state):
    """
    Inverse of ShiftRows: shift right by row index.
    """
    return [
        state[0],
        state[1][-1:] + state[1][:-1],
        state[2][-2:] + state[2][:-2],
        state[3][-3:] + state[3][:-3]
    ]

# MixColumns helpers (GF(2^8) multiplication)
def xtime(a):
    """
    Multiplies by x (i.e., 2) in GF(2^8).
    """
    return ((a << 1) ^ 0x1b) & 0xff if (a & 0x80) else (a << 1)

def mix_single_column(col):
    """
    Mixes one column for MixColumns.
    """
    t = col[0] ^ col[1] ^ col[2] ^ col[3]
    u = col[0]
    col[0] ^= t ^ xtime(col[0] ^ col[1])
    col[1] ^= t ^ xtime(col[1] ^ col[2])
    col[2] ^= t ^ xtime(col[2] ^ col[3])
    col[3] ^= t ^ xtime(col[3] ^ u)
    return col

def mix_columns(state):
    """
    MixColumns transformation: mixes each column using GF(2^8) arithmetic.
    """
    for c in range(4):
        col = [state[r][c] for r in range(4)]
        mixed = mix_single_column(col)
```

```
        for r in range(4):
            state[r][c] = mixed[r]
    return state

# Inverse MixColumns
def inv_mix_single_column(col):
    """
    Inverse MixColumns for one column.
    """
    # Multiply by fixed matrix in GF(2^8)
    def mul(a, b):
        p = 0
        for i in range(8):
            if b & 1:
                p ^= a
            hi_bit_set = a & 0x80
            a = (a << 1) & 0xff
            if hi_bit_set:
                a ^= 0x1b
            b >>= 1
        return p
    return [
        mul(col[0], 0x0e) ^ mul(col[1], 0x0b) ^ mul(col[2], 0x0d) ^
mul(col[3], 0x09),
        mul(col[0], 0x09) ^ mul(col[1], 0x0e) ^ mul(col[2], 0x0b) ^
mul(col[3], 0x0d),
        mul(col[0], 0x0d) ^ mul(col[1], 0x09) ^ mul(col[2], 0x0e) ^
mul(col[3], 0x0b),
        mul(col[0], 0x0b) ^ mul(col[1], 0x0d) ^ mul(col[2], 0x09) ^
mul(col[3], 0x0e)
    ]

def inv_mix_columns(state):
    """
    Inverse MixColumns transformation.
    """
    for c in range(4):
        col = [state[r][c] for r in range(4)]
        mixed = inv_mix_single_column(col)
        for r in range(4):
            state[r][c] = mixed[r]
    return state
```

cfb_mode.py

```
"""
cfb_mode.py
Implements AES-128 CFB mode encryption and decryption using AES block
functions.
"""
```

```
from aes_core import add_round_key, shift_rows, mix_columns, sub_bytes
from key_expansion import key_expansion, Nb, Nr

# Helper: Convert 16-byte array to 4x4 matrix
def bytes_to_matrix(b):
    """
    Converts a 16-byte array to a 4x4 matrix (column-major).
    """
    return [[b[4*c + r] for c in range(4)] for r in range(4)]

# Helper: Convert 4x4 matrix to 16-byte array
def matrix_to_bytes(m):
    """
    Converts a 4x4 matrix to a 16-byte array (column-major).
    """
    return [m[r][c] for c in range(4) for r in range(4)]

# AES block encryption (single block, no padding)
def aes_encrypt_block(plaintext, round_keys):
    """
    Encrypts a single 16-byte block using AES-128.
    plaintext: 16-byte array
    round_keys: list of 44 words (from key_expansion)
    Returns: 16-byte array (ciphertext)
    """
    state = bytes_to_matrix(plaintext)
    # Initial round
    state = add_round_key(state, bytes_to_matrix(sum(round_keys[0:4], [])))
    # 9 main rounds
    for rnd in range(1, Nr):
        state = sub_bytes(state)
        state = shift_rows(state)
        state = mix_columns(state)
        state = add_round_key(state,
bytes_to_matrix(sum(round_keys[4*rnd:4*(rnd+1)], [])))
    # Final round (no MixColumns)
    state = sub_bytes(state)
    state = shift_rows(state)
    state = add_round_key(state, bytes_to_matrix(sum(round_keys[4*Nr:4*
(Nr+1)], [])))
    return matrix_to_bytes(state)

# CFB mode encryption
def     aes_cfb_encrypt(plaintext, key_bytes, iv):
    """
    Encrypts plaintext using AES-128 in CFB mode.
    plaintext: byte array
    key_bytes: 16-byte array
    iv: 16-byte array (initialization vector)
    Returns: ciphertext byte array
    """
    round_keys = key_expansion(key_bytes)
    block_size = 16
    ciphertext = []
```

```
    prev = iv.copy()
    for i in range(0, len(plaintext), block_size):
        block = plaintext[i:i+block_size]
        # Encrypt previous ciphertext (or IV for first block)
        enc = aes_encrypt_block(prev, round_keys)
        # XOR with plaintext block
        cipher_block = [b ^ e for b, e in zip(block, enc[:len(block)])]
        ciphertext.extend(cipher_block)
        prev = cipher_block.copy()
    return ciphertext

# CFB mode decryption
def aes_cfb_decrypt(ciphertext, key_bytes, iv):
    """
    Decrypts ciphertext using AES-128 in CFB mode.
    ciphertext: byte array
    key_bytes: 16-byte array
    iv: 16-byte array
    Returns: plaintext byte array
    """
    round_keys = key_expansion(key_bytes)
    block_size = 16
    plaintext = []
    prev = iv.copy()
    for i in range(0, len(ciphertext), block_size):
        block = ciphertext[i:i+block_size]
        enc = aes_encrypt_block(prev, round_keys)
        plain_block = [b ^ e for b, e in zip(block, enc[:len(block)])]
        plaintext.extend(plain_block)
        prev = block.copy()
    return plaintext
```

main.py

```
"""
main.py
Main program for AES-128 CFB mode encryption and decryption from scratch.
Accepts plaintext, key, and IV from user, runs encryption/decryption, and
verifies output.
"""
from cfb_mode import aes_cfb_encrypt, aes_cfb_decrypt

# Helper: Convert hex string to byte array
def hex_to_bytes(hex_str):
    """
    Converts hex string to byte array.
    """
    return [int(hex_str[i:i+2], 16) for i in range(0, len(hex_str), 2)]

# Helper: Convert string to byte array
```

```
def str_to_bytes(s):
    """
    Converts ASCII string to byte array.
    """
    return [ord(c) for c in s]

# Helper: Convert byte array to string
def bytes_to_str(b):
    """
    Converts byte array to ASCII string.
    """
    return ''.join(chr(x) for x in b)

if __name__ == "__main__":
    print("AES-128 CFB Mode Encryption/Decryption (from scratch)")
    # Get plaintext
    plaintext = input("Enter plaintext: ")
    pt_bytes = str_to_bytes(plaintext)
    # Get key (hex string, 32 chars for 128 bits)
    key_hex = input("Enter 128-bit key (hex, 32 chars): ")
    key_bytes = hex_to_bytes(key_hex)
    if len(key_bytes) != 16:
        print("Key must be 16 bytes (32 hex chars)")
        exit(1)
    # Get IV (hex string, 32 chars for 128 bits). If blank, generate from
timestamp.
    import time
    iv_hex = input("Enter IV (hex, 32 chars) [leave blank to auto-
generate]: ")
    if iv_hex.strip() == "":
        # Generate IV using current timestamp
        ts = int(time.time() * 1000000)  # microseconds for more entropy
        iv_bytes = [(ts >> (8 * i)) & 0xff for i in range(16)]
        print("Generated IV (hex):", ''.join(f'{b:02x}' for b in iv_bytes))
    else:
        iv_bytes = hex_to_bytes(iv_hex)
        if len(iv_bytes) != 16:
            print("IV must be 16 bytes (32 hex chars)")
            exit(1)
    # Pad plaintext to multiple of 16 bytes (CFB can work with partial
blocks, but we keep it simple)
    if len(pt_bytes) % 16 != 0:
        pt_bytes += [0] * (16 - len(pt_bytes) % 16)
    # Encrypt
    ciphertext = aes_cfb_encrypt(pt_bytes, key_bytes, iv_bytes)
    print("Ciphertext (hex):", ''.join(f'{b:02x}' for b in ciphertext))
    # Decrypt
    recovered = aes_cfb_decrypt(ciphertext, key_bytes, iv_bytes)
    # Remove padding
    recovered_str = bytes_to_str(recovered[:len(plaintext)])
    print("Recovered Plaintext:", recovered_str)
    # Verify
    if recovered_str == plaintext:
        print("Success: Plaintext recovered correctly!")
```

```
    else:
        print("Error: Plaintext does not match!")
```

performance_analysis.py

```
"""
performance_analysis.py
Measures AES-128 CFB encryption/decryption performance on a 1MB text file.
Records encryption time, decryption time, and ciphertext size.
"""
import time
from cfb_mode import aes_cfb_encrypt, aes_cfb_decrypt
from main import hex_to_bytes

# Settings
KEY_HEX = "2b7e151628aed2a6abf7158809cf4f3c"  # Example 128-bit key
IV_HEX = "000102030405060708090a0b0c0d0e0f"    # Example IV
INPUT_FILE = "input_1mb.txt"  # Path to 1MB text file

# Read file as bytes
def read_file_bytes(filename):
    with open(filename, "rb") as f:
        return list(f.read())

# Write bytes to file
def write_file_bytes(filename, data):
    with open(filename, "wb") as f:
        f.write(bytes(data))

if __name__ == "__main__":
    print("AES-128 CFB Performance Analysis")
    # Prepare key and IV
    key_bytes = hex_to_bytes(KEY_HEX)
    iv_bytes = hex_to_bytes(IV_HEX)
    # Read input file
    plaintext = read_file_bytes(INPUT_FILE)
    print(f"Plaintext size: {len(plaintext)} bytes")
    # Encrypt
    start_enc = time.time()
    ciphertext = aes_cfb_encrypt(plaintext, key_bytes, iv_bytes)
    end_enc = time.time()
    enc_time = end_enc - start_enc
    write_file_bytes("ciphertext.bin", ciphertext)
    # Decrypt
    start_dec = time.time()
    recovered = aes_cfb_decrypt(ciphertext, key_bytes, iv_bytes)
    end_dec = time.time()
    dec_time = end_dec - start_dec
    write_file_bytes("recovered.txt", recovered)
    # Results table
```

```
    print("\nResults:")
    print("+-------------------+-------------------+")
    print("| Operation         | Time (seconds)    |")
    print("+-------------------+-------------------+")
    print(f"| Encryption        | {enc_time:.6f}        |")
    print(f"| Decryption        | {dec_time:.6f}        |")
    print("+-------------------+-------------------+")
    print(f"Ciphertext size: {len(ciphertext)} bytes")
    print("\nTable for report:")
    print("| Operation   | Time (s)   | Size (bytes) |")
    print(f"| Encrypt     | {enc_time:.6f} | {len(ciphertext)}        |")
    print(f"| Decrypt     | {dec_time:.6f} | {len(recovered)}        |")
    print("\nCheck 'ciphertext.bin' and 'recovered.txt' for output files.")
```

Github Link - https://github.com/sponge-24/TPM-Assesment