南京邮电大学

《密码学》实验报告（二）

（ 2024 / 2025 学年 第 二 学期）

题　　目：分组密码算法实现

专　　业	信息安全
学 号 姓 名	B230410
	B23041011
	于明宏
指 导 教 师	李琦
指 导 单 位	计算机学院、软件学院、网络空间安全学院
日　　期	2025.4.1

分组密码算法实现

一、课题内容和要求

本实验的目标是实现一个分组密码算法：

1. 编程实现 SM4 或者 AES 算法（二选一），对明文加密并对密文做扩散性、以及随机性测试（0/1、扑克牌、游程测试）。

2. 针对题目 1 里的明文，利用你使用的语言，提供的密码库（选择其他的分组密码算法，或者第一步你使用的），调库实现加密操作，将得到的密文与题目 1 中得到的密文进行比较。

二、实现分析

AES128（CBC 模式）实现方法如下：

明文→填充→分组→密钥扩展→

[轮 1 到轮 9] 每轮操作包括：

1.字节替代

2.行移位

3.列混合

4.轮密钥异或

→[最后一轮] 操作包括：

1.字节替代

2.行移位

3. 轮密钥异或

→密文

三、概要设计

采用 Python 语言编写，完整实现 AES128 算法，并支持 CBC 加密模式。系统核心包括加密/解密流程、密钥扩展、数据填充处理以及随机性测试四大部分，通过模块化设计将各个功能组件清晰分离。算法实现严格遵循标准规范，包含字节替换、行移位、列混淆和轮密钥加等基本操作，同时提供了对应的逆操作用于解密过程。

在加密模式方面，该实现采用 CBC（密码块链接）模式来增强安全性，通过初始化向量实现块间链接，有效隐藏明文模式。系统自动处理数据填充，使用 PKCS#7 标准确保数据长度为 16 字节的倍数，在解密后自动去除填充内容。为验证算法质量，实现中还包含

一套完整的随机性测试工具，包括扩散性测试、0/1 比例测试、扑克牌测试和游程测试等多个统计测试方法，能够全面评估加密结果的随机特性。

四、源程序代码

1. 编程实现 AES128（CBC 模式）算法，对明文加密并对密文做扩散性、以及随机性测试（0/1、扑克牌、游程测试）。

```
import os
from collections import Counter

s_box = [
    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
]

inv_s_box = [
    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 gf_mult(a, b):
    p = 0
    for _ in range(8):
        if b & 1:
            p ^= a
        a <<= 1
        if a & 0x100:
            a ^= 0x11b
        b >>= 1
    return p

def bytes_to_state(data):
    state = [[0]*4 for _ in range(4)]
    for col in range(4):
        for row in range(4):
            state[row][col] = data[col*4 + row]
    return state

def state_to_bytes(state):
    data = bytearray(16)
    for col in range(4):
        for row in range(4):
            data[col*4 + row] = state[row][col]
    return bytes(data)

def sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = s_box[state[i][j]]

def inv_sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = inv_s_box[state[i][j]]

def shift_rows(state):
    state[1] = state[1][1:] + state[1][:1]
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][3:] + state[3][:3]

def inv_shift_rows(state):
    state[1] = state[1][3:] + state[1][:3]
```

```
    state[2] = state[2][2:] + state[2][:2]
    state[3] = state[3][1:] + state[3][:1]

def mix_columns(state):
    for col in range(4):
        s0 = state[0][col]
        s1 = state[1][col]
        s2 = state[2][col]
        s3 = state[3][col]
        state[0][col] = gf_mult(0x02, s0) ^ gf_mult(0x03, s1) ^ s2 ^ s3
        state[1][col] = s0 ^ gf_mult(0x02, s1) ^ gf_mult(0x03, s2) ^ s3
        state[2][col] = s0 ^ s1 ^ gf_mult(0x02, s2) ^ gf_mult(0x03, s3)
        state[3][col] = gf_mult(0x03, s0) ^ s1 ^ s2 ^ gf_mult(0x02, s3)

def inv_mix_columns(state):
    for col in range(4):
        s0 = state[0][col]
        s1 = state[1][col]
        s2 = state[2][col]
        s3 = state[3][col]
        state[0][col] = gf_mult(0x0e, s0) ^ gf_mult(0x0b, s1) ^ gf_mult(0x0d, s2) ^ gf_mult(0x09, s3)
        state[1][col] = gf_mult(0x09, s0) ^ gf_mult(0x0e, s1) ^ gf_mult(0x0b, s2) ^ gf_mult(0x0d, s3)
        state[2][col] = gf_mult(0x0d, s0) ^ gf_mult(0x09, s1) ^ gf_mult(0x0e, s2) ^ gf_mult(0x0b, s3)
        state[3][col] = gf_mult(0x0b, s0) ^ gf_mult(0x0d, s1) ^ gf_mult(0x09, s2) ^ gf_mult(0x0e, s3)

def add_round_key(state, round_key):
    round_key_state = bytes_to_state(round_key)
    for i in range(4):
        for j in range(4):
            state[i][j] ^= round_key_state[i][j]

def key_expansion(key):
    w = [0]*44

    for i in range(4):
        w[i] = int.from_bytes(key[i*4:(i+1)*4], 'big')

    rcon = [0x01000000, 0x02000000, 0x04000000, 0x08000000,
            0x10000000, 0x20000000, 0x40000000, 0x80000000,
            0x1B000000, 0x36000000]

    for i in range(4, 44):
        temp = w[i-1]
        if i % 4 == 0:
            rotated = ((temp << 8) & 0xffffffff) | (temp >> 24)
```

```
            substituted = 0
            for k in range(4):
                b = (rotated >> (24 - 8*k)) & 0xff
                substituted |= s_box[b] << (24 - 8*k)
            temp = substituted ^ rcon[(i//4)-1]
        w[i] = w[i-4] ^ temp

    round_keys = []
    for i in range(11):
        key_bytes = bytearray()
        for j in range(4):
            word = w[i*4 + j]
            for k in range(4):
                byte = (word >> (24 - 8*k)) & 0xff
                key_bytes.append(byte)
        round_keys.append(bytes(key_bytes))

    return round_keys

def pad(data, block_size):
    padding_len = block_size - len(data) % block_size
    return data + bytes([padding_len] * padding_len)

def unpad(data):
    padding_len = data[-1]
    return data[:-padding_len]

def aes_encrypt_block(block, round_keys):
    state = bytes_to_state(block)
    add_round_key(state, round_keys[0])
    for i in range(1, 10):
        sub_bytes(state)
        shift_rows(state)
        mix_columns(state)
        add_round_key(state, round_keys[i])
    sub_bytes(state)
    shift_rows(state)
    add_round_key(state, round_keys[10])
    return state_to_bytes(state)

def aes_decrypt_block(ciphertext_block, round_keys):
    state = bytes_to_state(ciphertext_block)
    add_round_key(state, round_keys[10])
    for i in range(9, 0, -1):
        inv_shift_rows(state)
```

```
            inv_sub_bytes(state)
            add_round_key(state, round_keys[i])
            inv_mix_columns(state)
        inv_shift_rows(state)
        inv_sub_bytes(state)
        add_round_key(state, round_keys[0])
        return state_to_bytes(state)

    def aes_encrypt(plaintext, key, iv):
        round_keys = key_expansion(key)
        padded = pad(plaintext, 16)
        previous = iv
        ciphertext = bytearray()
        for i in range(0, len(padded), 16):
            block = padded[i:i+16]
            xored = bytes([a ^ b for a, b in zip(block, previous)])
            encrypted = aes_encrypt_block(xored, round_keys)
            ciphertext.extend(encrypted)
            previous = encrypted
        return bytes(ciphertext)

    def aes_decrypt(ciphertext, key, iv):
        round_keys = key_expansion(key)
        previous = iv
        plaintext = bytearray()
        for i in range(0, len(ciphertext), 16):
            block = ciphertext[i:i+16]
            decrypted = aes_decrypt_block(block, round_keys)
            xored = bytes([a ^ b for a, b in zip(decrypted, previous)])
            plaintext.extend(xored)
            previous = block
        return unpad(plaintext)

    def bit_flip_diffusion(plaintext, key, iv):
        original = aes_encrypt(plaintext, key, iv)
        modified = bytearray(plaintext)
        modified[0] ^= 0x01
        modified_cipher = aes_encrypt(bytes(modified), key, iv)
        diff = sum(bin(a ^ b).count('1') for a, b in zip(original, modified_cipher))
        return diff / (len(original) * 8)

    def bit_proportion_test(data):
        bits = ''.join(f'{byte:08b}' for byte in data)
        return bits.count('1') / len(bits)
```

```
def poker_test(data):
    bits = ''.join(f'{byte:08b}' for byte in data)
    n = len(bits) // 4
    chunks = [bits[i*4:(i+1)*4] for i in range(n)]
    counts = Counter(chunks)
    x = (16 / n) * sum(v**2 for v in counts.values()) - n
    return x

def runs_test(data):
    bits = ''.join(f'{byte:08b}' for byte in data)
    current = bits[0]
    runs = []
    count = 1
    for bit in bits[1:]:
        if bit == current:
            count +=1
        else:
            runs.append(count)
            current = bit
            count = 1
    runs.append(count)
    return sum(runs)/len(runs)

total_diffusion = 0
total_bit_proportion = 0
total_poker = 0
total_runs = 0

for _ in range(100):
    plaintext = os.urandom(100)
    key = os.urandom(16)
    iv = os.urandom(16)

    print("明文:", plaintext.hex())
    print("密钥:", key.hex())
    print("初始化向量:", iv.hex())

    ciphertext = aes_encrypt(plaintext, key, iv)
    print("密文:", ciphertext.hex())

    decrypted = aes_decrypt(ciphertext, key, iv)
    print("解密所得明文:", decrypted.hex())

    if decrypted == plaintext:
        print("成功！")
```

```
    else:
        print("失败！")

    total_diffusion += bit_flip_diffusion(ciphertext, key, iv)
    total_bit_proportion += bit_proportion_test(ciphertext)
    total_poker += poker_test(ciphertext)
    total_runs += runs_test(ciphertext)

avg_diffusion = total_diffusion / 100
avg_bit_proportion = total_bit_proportion / 100
avg_poker = total_poker / 100
avg_runs = total_runs / 100

print(f"扩散性测试: {avg_diffusion:.4f}")
print(f"0/1 测试: {avg_bit_proportion:.4f}")
print(f"扑克牌测试: {avg_poker:.4f}")
print(f"游程测试: {avg_runs:.4f}")
```

2. 利用提供的密码库，调库实现 AES128（CBC 模式）加密操作，将得到的密文与题目 1 中得到的密文进行比较。这里使用题目 1 中的如下明文、密钥和初始化向量，调库操作。

```
import binascii
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def aes_encrypt(plaintext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.encrypt(pad(plaintext, AES.block_size))

def aes_decrypt(ciphertext, key, iv):
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return unpad(cipher.decrypt(ciphertext), AES.block_size)

plaintext =
binascii.unhexlify("7c7feef136db0d41fdd4c3c7c8a19cea4af5b1ff07b56d2ecc039e6cfc87dc03d496a2e4d851a
27335ab44195e04010d7f1a7c5c06618a95364f1bb7cc873359e0ee8f26111d75ab61e7d3a0ecba21418b3a9c30c
fb56ea0a64ab9aecb24c9afc755c34e")
key = binascii.unhexlify("d078936926b3ee13abaf32fb8895ef80")
iv = binascii.unhexlify("dc5f7b8d416f05fbe3fbf2d3e9215738")

ciphertext = aes_encrypt(plaintext, key, iv)
print("密文:", binascii.hexlify(ciphertext).decode())

decrypted = aes_decrypt(ciphertext, key, iv)
if decrypted == plaintext:
    print("解密验证成功！")
```

```
else:
    print("解密验证失败！")
```

五、测试数据及其结果分析

1. 编程实现 AES128（CBC 模式）算法，对明文加密并对密文做扩散性、以及随机性测试（0/1、扑克牌、游程测试）。

明文:

9f855df5111969e876d71aef7cc0f0cef0d2006a590c72ae2b31286a298d839773770459b4075853847238607a5f

73a768313fca476edf2a2d02e1416a1f00f7a5f8fc85b9f96e9c87ec2d3fb49c4fe5b6236016a8a6ebdf4884194e45

f04e0dc8dceb33

密钥: 431c1941c7a7661ad3b342dfa9b63bfc

初始化向量: f5c7b839f938df27a012549e875d3330

密文:

92cf5a9f4ab6379a345f9755efd0e8517fec1f9f895545d4008cc9e79ac4b18a638b8f2bf1a55a6cff47a69d32652b

0f5a3dffa7c0f631e8fa39ee3e329442d21d6184de85953cad50c8f99a179f4e6f0685a4837fe49ec7e0f25869b5dd

984668351852a9ee084bf41f4d7ee414325c

解密所得明文:

9f855df5111969e876d71aef7cc0f0cef0d2006a590c72ae2b31286a298d839773770459b4075853847238607a5f

73a768313fca476edf2a2d02e1416a1f00f7a5f8fc85b9f96e9c87ec2d3fb49c4fe5b6236016a8a6ebdf4884194e45

f04e0dc8dceb33

成功！

明文:

7c7feef136db0d41fdd4c3c7c8a19cea4af5b1ff07b56d2ecc039e6cfc87dc03d496a2e4d851a27335ab44195e0401

0d7f1a7c5c06618a95364f1bb7cc873359e0ee8f26111d75ab61e7d3a0ecba21418b3a9c30cfb56ea0a64ab9aecb2

4c9afc755c34e

密钥: d078936926b3ee13abaf32fb8895ef80

初始化向量: dc5f7b8d416f05fbe3fbf2d3e9215738

密文:

a9acec265db9815e6d3c46cce41a90498ad4f0f483e58dc7ba2601afa02668f897ec7e5699097d8f601dc76043348

20ec16b0abce4ece2132f7fe7fba1b580169daa5ebd3ae7271a57d0725e99e14d15842b92c2be5f560e9d915b056

034d668be338f3be37af3ca8f12650b6562b586

解密所得明文:

7c7feef136db0d41fdd4c3c7c8a19cea4af5b1ff07b56d2ecc039e6cfc87dc03d496a2e4d851a27335ab44195e0401
0d7f1a7c5c06618a95364f1bb7cc873359e0ee8f26111d75ab61e7d3a0ecba21418b3a9c30cfb56ea0a64ab9aecb2
4c9afc755c34e

成功!

（省略部分输出）

扩散性测试: 0.5025

0/1 测试: 0.4996

扑克牌测试: 15.1786

游程测试: 1.9943

2. 利用提供的密码库，调库实现 AES128（CBC 模式）加密操作，将得到的密文与题目 1 中得到的密文进行比较。这里使用题目 1 中的如下明文、密钥和初始化向量，调库操作。

明文:

7c7feef136db0d41fdd4c3c7c8a19cea4af5b1ff07b56d2ecc039e6cfc87dc03d496a2e4d851a27335ab44195e0401
0d7f1a7c5c06618a95364f1bb7cc873359e0ee8f26111d75ab61e7d3a0ecba21418b3a9c30cfb56ea0a64ab9aecb2
4c9afc755c34e

密钥: d078936926b3ee13abaf32fb8895ef80

初始化向量: dc5f7b8d416f05fbe3fbf2d3e9215738

密文:

a9acec265db9815e6d3c46cce41a90498ad4f0f483e58dc7ba2601afa02668f897ec7e5699097d8f601dc76043348
20ec16b0abce4ece2132f7fe7fba1b580169daa5ebd3ae7271a57d0725e99e14d15842b92c2be5f560e9d915b056
034d668be338f3be37af3ca8f12650b6562b586

解密验证成功!

经比较，二者所得密文相同，因此我们自己编写的 AES128（CBC 模式）加解密代码基本无误。

六、调试过程中的问题

调试过程中未出现问题。

七、课程总结

通过本次实验，深入理解了分组密码的工作原理，掌握了基于 Python 的数据结构和算法的实现方法。实验表明，AES 算法具有良好的扩散性和混淆性，能够有效抵抗统计分析攻击，在 CBC 模式下进一步增强了安全性。