武汉大学国家网络安全学院

密 码 学 实 验 报 告

学　　号　2021302181156

姓　　名　赵伯侯

实验名称　分组密码 AES

指导教师　何琨

一、 实验名称: 分组密码 DES

二、 实验目的及要求:

2.1 实验目的

(1) 掌握分组密码的基本概念

(2) 掌握 AES 密码算法

(3) 了解 AES 密码的安全性

(4) 掌握分组密码常用工作模式及其特点

(5) 熟悉分组密码的应用

2.2 实验要求

(1) 熟悉 AES 算法的基本结构

(2) 掌握 AES 算法的基本运算

(3) 掌握 AES 算法的实现与优化方法

(4) 熟悉 AES 算法的安全性

三、 实验设备环境及要求:

Windows 操作系统，python 高级语言开发环境

四、 实验内容与步骤:

4.1 AES 算法基本结构

4.1.1 算法整体结构

AES 的整体算法结构如下图所示。

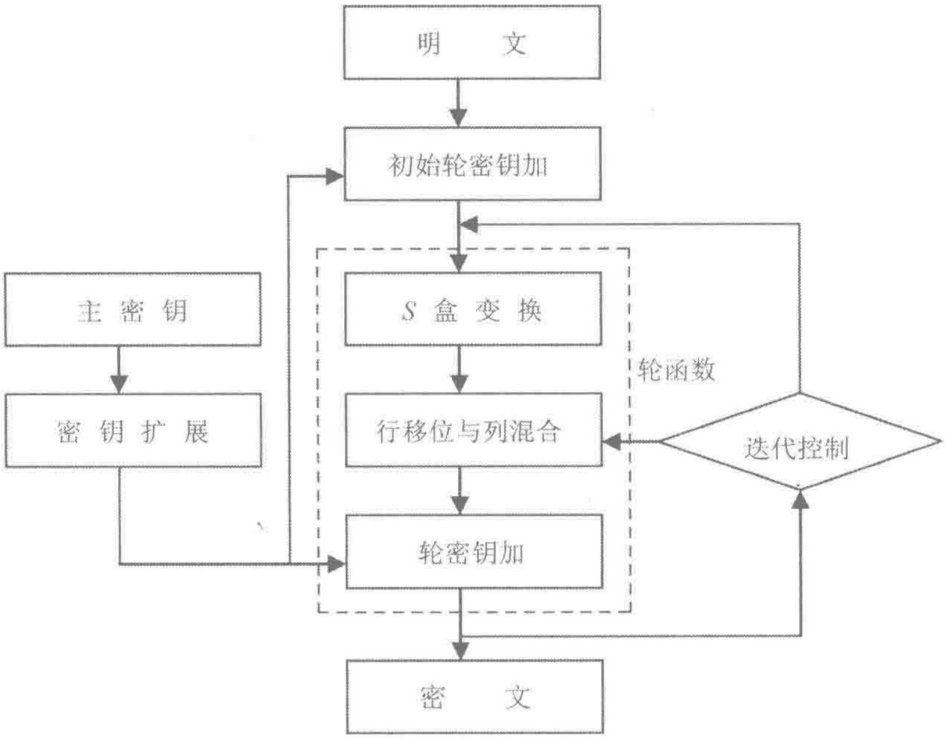

图 1: AES 算法结构

明文在经过初始轮密钥加之后进入到轮密钥中迭代一定的次数之后得到密文，在轮密钥加步骤中需要的轮密钥需要由主密钥通过密钥拓展得到。

AES 加密过程的代码如下所示

```
1  def encode(key, message):
2      """AES加密"""
3
4      # 计算相关参数
5      Nrs = [[10, 12, 14], [12, 12, 14], [14, 14, 14]]
6      Nb = int(len(message) * 4 / 32)  # 4
7      Nk = int(len(key) * 4 / 32)  # 4
8      Nr = Nrs[int(Nk / 2) - 2][int(Nb / 2) - 2]  # 10
```

```
9
10      # 计算加密拓展密钥得到轮密钥
11      round_keys = get_round_keys(Nb, Nr, Nk, key)
12
13      # 论函数得到密文
14      secret = Round(Nb, Nr, round_keys, message)
15
16      # 输出密文
17      return secret
```

代码 1: AES 整体加密代码

4.1.2 轮函数

轮函数由三层结构组成：

（1）非线性层：进行非线性 S 盒变换 ByteSub，由 16 个 S 盒并置而成，起到混淆的作用

（2）线性混合层：进行行移位变换 ShiftRow 和列混合变换 MixColumn 以确保多轮之上的高度扩散

（3）密钥加层：进行轮密钥加变换 AddRoundKey，将轮密钥与中间状态进行异或运算，实现密钥的加密控制作用

轮密钥函数算法的结构如下图所示

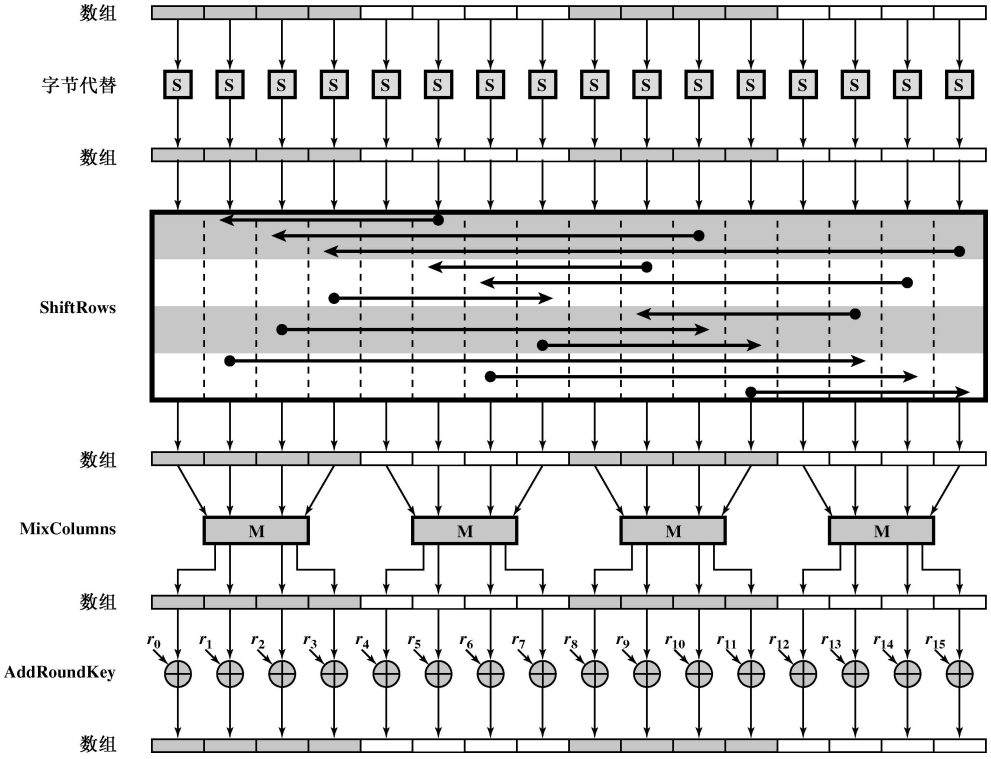

图 2: 轮函数算法结构

轮函数的程序代码如下所示

```
def Round(Nb, Nr, round_keys, message):
    """轮函数"""
    state = message
    # 初始密钥加
    state = AddRoundKey(state, round_keys[0])

    # 迭代控制
    for i in range(1, Nr):
        Round_key = round_keys[i]
        state = SubByte(state)
        state = ShiftRow(Nb, state)
        state = MixColumn(state)
        state = AddRoundKey(state, Round_key)
```

```
14
15      # 最后一轮迭代
16      Round_key = round_keys[Nr]
17      state = SubByte(state)
18      state = ShiftRow(Nb, state)
19      state = AddRoundKey(state, Round_key)
20
21      # 返回密文
22      return state
```

代码 2: 轮函数代码

4.1.3 S 盒变换

S 盒变换是以字节为单位进行代替变换，首先将字节的值用它的乘法逆来代替，其中‘00’的乘法逆为其本身，然后再将处理后的字节按照如下图所示的变换公式进行仿射变换

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

图 3: S 盒仿射变换

4.1.4 行移位变换 ShiftRow

行移位变换是将状态的行进行循环左移变换，对于状态的第 0 行不进行移位操作，对状态的第 i 行分别循环左移 $C[i]$ 个字节得到行移位变换后的结果其中每

一行的移位值与 Nb 有关，其关系如下表所示

表 1: 移位值表

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

行移位操作的程序代码如下所示

```
def ShiftRow(Nb, State):
    """行移位变换"""

    # 行移位矩阵
    shift_table = [[1, 2, 3], [1, 2, 3], [1, 3, 4]]
    c = shift_table[int(Nb / 2) - 2]

    # 整个状态拆分成4块
    states = ['', '', '', '']
    for j in range(4):
        states[j] = [State[i + j * 2:i + j * 2 + 2] for i in range(0, len(State
            ), 8)]
        states[j] = ''.join(states[j])

    # 对后三块按照移位矩阵移位后保存到result中
    result = ['', '', '', '']
    result[0] = states[0]
    result[1] = shift(states[1], c[0])
    result[2] = shift(states[2], c[1])
    result[3] = shift(states[3], c[2])

```

```
    # 结果转换成一整个字符串
    output = ''
    for i in range(4):
        output += ''.join(item[i * 2:2 + i * 2] for item in result)
    return output
```

代码 3: 行移位代码

4.1.5 列混合变换 MixColumn

列混合变换时对状态的列进行混合变换，把状态中的每一列看作 $GF(2^8)$ 上的一个多项式，并与一个固定的多项式 $c(x) =' 03' x^3 +' 01' x^2 +' 01' x^1 +' 02'$ 相乘然后模多项式 $x^4 + 1$ 得到列混合变换的结果。在本次实验中通过将每一个字节进行如下图所示的变换规则进行多项式乘法得到变换后的多项式。

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

图 4: 列混合变换

列混合变换的实现代码如下所示

```
def MixColumn(State):
    """列混合变换"""
    # 状态转为字节形式存储
    states = [State[i * 2:i * 2 + 2] for i in range(16)]

    # 变换矩阵
    MIX_C = [['2', '3', '1', '1'], ['1', '2', '3', '1'], ['1', '1', '2', '3'],
```

```
        ['3', '1', '1', '2']]
8
9       # 矩阵乘法
10      result = ['00'] * 16
11      for row in range(4):
12          for col in range(4):
13              for Round in range(4):
14                  result[row + col * 4] = xor(mul_x(MIX_C[row][Round], states[
                        Round + col * 4]), result[row + col * 4])
15              result[row + col * 4] = mod_x(result[row + col * 4])  # 模不可约多
                    项式
16
17      # 结果拼接成整个字符串
18      state = ''
19      for i in range(16):
20          while len(result[i]) < 2:
21              result[i] = '0' + result[i]
22          state += result[i]
23      return state
```

代码 4: 列混合代码

4.1.6 轮密钥加变换 AddRoundKey

轮密钥加函数主要功能是将当前的轮密钥当作二进制多项式与当前状态进行加和即进行异或操作，其代码如下所示

```
1   def AddRoundKey(State, round_key):
2       """轮密钥加"""
3       # 直接异或
4       return xor(State, round_key)
```

代码 5: 轮密钥加代码

4.2 AES 算法的基本运算

4.2.1 $GF(2^8)$ 上的加法与多项式加法

在本次实验中由于将状态看作二进制多项式形式并用 16 进制进行表示，那么两个多项式之间的加法可以看作是将其各位进行异或操作，其代码如下所示

```
def xor(byte_a, byte_b):
    """2进制多项式加法"""
    # 获取两个加数最长长度
    size = max(len(byte_a), len(byte_b))

    # 将被加数转换成二进制列表形式并补齐最高位
    a_bits = bin(int(byte_a, 16))[2:]
    a = [int(bit) for bit in a_bits]
    b_bits = bin(int(byte_b, 16))[2:]
    b = [int(bit) for bit in b_bits]
    while len(a) < size * 4:
        a.insert(0, 0)
    while len(b) < size * 4:
        b.insert(0, 0)

    # 按位异或
    result = ''
    for index in range(size * 4):
        result += str(a[index] ^ b[index])

    # 运算结果转为16进制
    result = int(result, 2)
    result = hex(result)[2:]

```

```
25      # 若结果为0则返回'0'
26      while len(result) < size:
27          result = '0' + result
28      return result
```

代码 6: 多项式加法代码

首先将得到的两个二进制多项式的字节表示转换成为列表的形式，然后补齐高位后对于每一位进行异或操作，最后得到的结果即为两个多项式相加的结果

4.2.2 在不同 CPU 架构下，不同表示方法的执行速度快慢关系

（1）数据表示和内存访问模式：

字节级操作：在一些处理器上，如 8 位或 16 位微控制器，按字节操作可能更高效，因为这些处理器的数据路径宽度较小。

32 位字操作：在 32 位处理器上，一次处理一个 32 位的字通常更加高效，因为它可以在单个操作中处理更多的数据。

128 位状态操作：在支持 SIMD（单指令多数据）指令集的现代处理器（如 x86 的 SSE 或 ARM 的 NEON）上，可以一次性以 128 位块的形式处理整个状态，这通常是最快的方法。

（2）CPU 指令集：

一些现代处理器拥有专门的指令集和硬件优化，可以一次性处理大量数据。例如，支持 SIMD（单指令多数据）指令集的处理器可以同时处理多个 32 位或更大的数据单元。（3）CPU 位宽：

对于 8 位或 16 位微控制器，按字节（8 位）操作可能更有效率，因为这些处理器的数据路径和寄存器宽度通常适合较小的数据单元。

对于 32 位处理器，按字（32 位）操作可能更加高效，因为它可以一次处理更多数据，与处理器的自然字大小相匹配。

对于 64 位处理器，尽管可以有效地执行 32 位操作，但它们通常能够一次处理

更大的数据块，因此可能在某些情况下，按 64 位操作会更加高效。

（4）内存访问和数据对齐：

在执行异或操作时，内存访问模式也很重要。如果数据没有正确对齐，即数据的内存地址不符合其自然边界，可能会导致性能下降。

一些处理器在处理未对齐的数据时效率较低，这可能会影响按字节或字操作的性能。

4.2.3 $GF(2^8)$ 上的乘法

（1）借助 xtime 运算快速实现

借助 xtime 运算快速实现乘法的代码如下所示

```
def xtime(a):
    """实现多项式在GF(2^8)上乘以{x}的运算"""
    # 左移一位
    result = a << 1
    # 如果最高位是1，执行模约简
    if a & 0x80:
        result ^= 0x1b
    # 确保结果仍然在一个字节范围内
    return result & 0xFF

def poly_mul(a, b):
    """使用xtime实现两个多项式的乘法"""
    result = 0
    for i in range(8):
        # 如果b的当前位是1，则将a的当前值加到结果中
        if b & 0x01:
            result ^= a
        # 检查a的最高位是否为1
```

```
        high_bit_set = a & 0x80
        # a左移一位
        a = xtime(a)
        # 如果之前a的最高位是1，将结果与约简多项式异或
        if high_bit_set:
            a ^= 0x1b
        # b右移一位
        b >>= 1
    return result
```

代码 7: 借助 xtime 实现多项式加法代码

该算法检查 b 的每一位，如果该位为 1，则将 a 的当前值累加到结果中。同时，a 通过应用 xtime 函数被逐步乘以多项式 x，而 b 则右移一位。这个过程持续进行，直到 b 被完全处理完毕。

该算法的效率分析:

在最好情况下，发生在乘数（即 b）的二进制表示中只有一个位为 1，而其他位都为 0。这种情况下，poly_mul 函数只需要进行一次非零位的处理。具体来说：如果 b 是形如 00000001 的二进制数，那么在第一次迭代中，结果就会直接等于 a，后续的迭代中不会再有任何累加，因为 b 的其他位都是 0。在这种情况下，函数几乎不需要进行额外的计算，仅需执行一次异或操作和一些基本的位操作，如位移。最好情况的时间复杂度是 $O(1)$，即常数时间复杂度，因为它几乎不需要进行计算。

在最坏情况下出现在乘数 b 的所有位都是 1，即 b 是形如 11111111 的二进制数。在这种情况下，每一次迭代中 b 的当前位都为 1，因此函数需要执行最多的异或操作。对于 a，每次迭代都会执行 xtime 函数，可能伴随着额外的模约简操作（即异或 0x1b）。在 8 次迭代中，每次都需要进行一次异或操作，以及其他的位操作和可能的模约简操作。最坏情况的时间复杂度是 $O(n)$，其中 n 是乘数的位数（在这个情况下，n 为 8），因为它需要对每一位进行处理。

（2）借助生成元实现

借助生成元运算快速实现乘法的代码如下所示

```
def gf_multiply(a, b, irreducible_poly=0x11b, field_size=256):
    """GF(2^8)域上的乘法"""
    result = 0
    for i in range(8):
        if b & 1:
            result ^= a
        carry = a & 0x80
        a <<= 1
        if carry:
            a ^= irreducible_poly
        b >>= 1
    return result % field_size


def polynomial_multiply(poly1, poly2):
    """多项式乘法"""
    result_size = len(poly1) + len(poly2) - 1
    result = [0] * result_size
    for i, coeff1 in enumerate(poly1):
        for j, coeff2 in enumerate(poly2):
            result[i + j] ^= gf_multiply(coeff1, coeff2)
    return result
```

代码 8: 借助生成元多项式乘法代码

首先造表生成有限域 $GF(2^8)$ 的所有元素。这通常是通过从 0 到 255 的整数表示，并将它们视为多项式的系数。构建乘法和加法表，用于快速查找结果。然后查表进行计算：对于多项式乘法中的每一对系数，使用乘法表来获取结果。加法相当于二进制下的异或操作（XOR）。最后进行结果合并：将所有的乘法和加法（异或）结果合并，得到最终的多项式乘法结果。

时间复杂度分析：

首先分析 gf_multiply 函数，这个函数在最坏情况下执行 8 次循环（对应于二进制下 8 位的乘法）。在每次循环中，它执行固定数量的操作（赋值、位移、异或等），这些操作的时间复杂度是常数级别的 $O(1)$。因此，gf_multiply 函数的时间复杂度是 $O(8) = O(1)$，即常数时间复杂度。

然后分析 polynomial_multiply 函数，这个函数包含两个嵌套循环，分别对应于两个输入多项式的长度，设为 n 和 m。对于每对系数组合，它调用一次 gf_multiply。因此，总共有 n * m 次 gf_multiply 调用。

由于 gf_multiply 是常数时间复杂度，所以 polynomial_multiply 的总时间复杂度是 $O(n * m)$。对于两个长度分别为 n 和 m 的多项式，该算法的时间复杂度是 $O(n * m)$。

空间复杂度分析：

首先分析 gf_multiply 函数，这个函数仅使用了固定数量的局部变量，不依赖于输入大小，因此它的空间复杂度是 $O(1)$。

然后分析 polynomial_multiply 函数，除了输入多项式外，这个函数创建了一个新的数组来存储结果，其长度为 n + m - 1。

因此，polynomial_multiply 函数的空间复杂度是 $O(n+m)$。该算法的空间复杂度是 $O(n+m)$，这是由于它需要存储两个输入多项式和一个长度为 n + m - 1 的结果多项式。

4.2.4 $GF(2^8)$ 上的多项式乘法实现

在本次实验中采用的多项式乘法的代码如下所示

```
    while len(a) < size * 4:
        a.insert(0, 0)
    while len(b) < size * 4:
        b.insert(0, 0)

    # 按位异或
```

```
    result = ''
    for index in range(size * 4):
        result += str(a[index] ^ b[index])

    # 运算结果转为16进制
    result = int(result, 2)
    result = hex(result)[2:]

    # 若结果为0则返回'0'
    while len(result) < size:
        result = '0' + result
```

代码 9: 多项式乘法代码

在该代码中，函数接收两个十六进制字符串作为输入。然后从十六进制转换为二进制数列表。然后初始化一个乘积多项式长度为两个输入列表长度之和减 1，然后进行多项式乘法运算，找到多项式 a 中最高的 1 位，然后遍历多项式 b 中的每一项，与找到的位进行与操作后异或乘积多项式的对应位置的值即可

4.2.5 列混合运算实现

（1）列混合运算优化 1 的实现

在列混合运算优化 1 的算法中，预先计算所有 256*6 个乘法（01 不用计算），这样需要 1.5K 字节空间，但可省去大量乘法运算，其对应的代码如下所示

```
def create_precomputed_tables():
    """创建预计算表"""
    coefficients = [0x02, 0x03, 0x0e, 0x0b, 0x0d, 0x09]
    tables = {}
    for coef in coefficients:
        tables[coef] = [gf_multiply(coef, x) for x in range(256)]
    return tables
```

```
8
9  def mix_columns(state, table):
10     """列混合运算"""
11     # 此处的实现取决于state的结构，假设它是一个4x4的矩阵
12     new_state = [[0] * 4 for _ in range(4)]
13     for i in range(4):
14         new_state[0][i] = table[0x02][state[0][i]] ^ table[0x03][state[1][i]] ^
               state[2][i] ^ state[3][i]
15         new_state[1][i] = state[0][i] ^ table[0x02][state[1][i]] ^ table[0x03][
               state[2][i]] ^ state[3][i]
16         new_state[2][i] = state[0][i] ^ state[1][i] ^ table[0x02][state[2][i]]
               ^ table[0x03][state[3][i]]
17         new_state[3][i] = table[0x03][state[0][i]] ^ state[1][i] ^ state[2][i]
               ^ table[0x02][state[3][i]]
18     return new_state
```

代码 10: 列混合优化 1 代码

（2）列混合优化算法 2 的实现

在该优化算法中，加密算法圈变换中的每一列变换，可通过作 4 次查表和 4 次异或运算得到。节省了大量的乘法运算

```
1  def create_t_tables():
2      """创建预计算表 T0 到 T3"""
3      # 请根据具体的线性变换来填充这些表
4      T0 = [0] * 256
5      T1 = [0] * 256
6      T2 = [0] * 256
7      T3 = [0] * 256
8      # 预计算填充表的值...
9      return T0, T1, T2, T3
```

```
10
11 def optimized_mix_columns(state, T0, T1, T2, T3):
12     """优化后的列混合运算"""
13     new_state = [[0] * 4 for _ in range(4)]
14     for j in range(4):
15         new_state[0][j] = T0[state[0][j]] ^ T1[state[1][(j + 1) % 4]] ^ T2[
               state[2][(j + 2) % 4]] ^ T3[state[3][(j + 3) % 4]]
16         new_state[1][j] = T0[state[1][j]] ^ T1[state[2][(j + 1) % 4]] ^ T2[
               state[3][(j + 2) % 4]] ^ T3[state[0][(j + 3) % 4]]
17         new_state[2][j] = T0[state[2][j]] ^ T1[state[3][(j + 1) % 4]] ^ T2[
               state[0][(j + 2) % 4]] ^ T3[state[1][(j + 3) % 4]]
18         new_state[3][j] = T0[state[3][j]] ^ T1[state[0][(j + 1) % 4]] ^ T2[
               state[1][(j + 2) % 4]] ^ T3[state[2][(j + 3) % 4]]
19     return new_state
20
21 # 创建预计算表
22 T0, T1, T2, T3 = create_t_tables()
23
24 # 示例状态矩阵
25 state = [[0x32, 0x88, 0x31, 0xe0],
26          [0x43, 0x5a, 0x31, 0x37],
27          [0xf6, 0x30, 0x98, 0x07],
28          [0xa8, 0x8d, 0xa2, 0x34]]
29
30 # 执行优化后的列混合运算
31 optimized_state = optimized_mix_columns(state, T0, T1, T2, T3)
```

代码 11: 列混合优化 2 代码

（3）在单片机、手机、PDA 等资源受限环境下的实现

在该资源受限的实现算法中，xtime 函数接收一个字节并在有限域 $GF(2^8)$ 中将其乘以 2，如果需要的话会与 0x1B 进行异或运算以保持结果在域内。mix_column 函数接收一个四字节的列作为输入，并执行列混合操作。这种实现方式不需要预计算表，相比前述的方法，它在存储上更为高效，但在每次操作时需要更多的计算。该算法的实现代码如下所示。

```
def mix_column(column):
    """执行列混合操作"""
    t = column[0] ^ column[1] ^ column[2] ^ column[3]
    u = column[0]

    v = column[0] ^ column[1]
    v = xtime(v)
    column[0] ^= v ^ t

    v = column[1] ^ column[2]
    v = xtime(v)
    column[1] ^= v ^ t

    v = column[2] ^ column[3]
    v = xtime(v)
    column[2] ^= v ^ t

    v = column[3] ^ u
    v = xtime(v)
    column[3] ^= v ^ t

    return column
```

代码 12: 列混合优化 2 代码

时间复杂度分析：

首先分析 xtime 函数，该函数执行的是基本的位操作，这些操作的时间复杂度是 $O(1)$，即常数时间。然后分析 mix_column 函数，在该函数中，主要操作是几个异或运算和对 xtime 函数的调用。对于每个元素，都执行了一次 xtime 调用和几次异或运算。由于这些操作都是常数时间的，所以每个元素的处理时间是常数时间。因此，对于单个四字节的列，mix_column 函数的时间复杂度是 $O(1)$。

空间复杂度分析：

这两个函数都只使用了固定数量的局部变量。xtime 函数只接受一个字节作为输入，而 mix_column 函数处理的是一个固定长度为 4 的字节数组。这意味着这两个函数的空间复杂度是 $O(1)$，即它们占用的空间不随输入大小的变化而变化。

因此, 该算法的空间复杂度也是 $O(1)$，因为它不需要额外的存储空间，只使用固定数量的局部变量。

正确性证明：

在证明中通过计算 $a[0]$ 的值对该算法的正确性进行证明。在标准运算的列混合中 $a'[0] = 2 * a[0] + 3 * a[1] + a[2] + a[3]$

而在优化算法中，$a'[0] = a[0] \oplus xtime(a[0] \oplus a[1]) \oplus t$

因为 $t = a[0] \oplus a[1] \oplus a[2] \oplus a[3]$

所以 $a'[0] = a[0] \oplus xtime(a[0] \oplus a[1]) \oplus (a[0] \oplus a[1] \oplus a[2] \oplus a[3])$

将 xtime 展开结果为 $a'[0] = a[0] \oplus (2 * a[0] \oplus 2 * a[1]) \oplus a[0] \oplus a[1] \oplus a[2] \oplus a[3]$

简化表达式后得到的结果为 $a'[0] = 2 * a[0] + 3 * a[1] + a[2] + a[3]$

优化算法与标准算法得到的结果相同，可以证明该优化算法的正确性

五、 AES 的安全性

5.1 AES 的 S 盒的实现

5.1.1 算法实现

在本次实验中为了省去计算 S 盒的运算时间将 S 盒变换的结果直接保存到表中，在使用时直接查表得到替换后的字节值 S 盒变换 ByteSub 部分的代码如下所示

```
def SubByte(byte):
    """S盒变换"""
    S_box = [['63', '7C', '77', '7B', 'F2', '6B', '6F', 'C5', '30', '01', '67',
         '2B', 'FE', 'D7', 'AB', '76'],
             ['CA', '82', 'C9', '7D', 'FA', '59', '47', 'F0', 'AD', 'D4', 'A2',
                  'AF', '9C', 'A4', '72', 'C0'],
             ['B7', 'FD', '93', '26', '36', '3F', 'F7', 'CC', '34', 'A5', 'E5',
                  'F1', '71', 'D8', '31', '15'],
             ['04', 'C7', '23', 'C3', '18', '96', '05', '9A', '07', '12', '80',
                  'E2', 'EB', '27', 'B2', '75'],
             ['09', '83', '2C', '1A', '1B', '6E', '5A', 'A0', '52', '3B', 'D6',
                  'B3', '29', 'E3', '2F', '84'],
             ['53', 'D1', '00', 'ED', '20', 'FC', 'B1', '5B', '6A', 'CB', 'BE',
                  '39', '4A', '4C', '58', 'CF'],
             ['D0', 'EF', 'AA', 'FB', '43', '4D', '33', '85', '45', 'F9', '02',
                  '7F', '50', '3C', '9F', 'A8'],
             ['51', 'A3', '40', '8F', '92', '9D', '38', 'F5', 'BC', 'B6', 'DA',
                  '21', '10', 'FF', 'F3', 'D2'],
             ['CD', '0C', '13', 'EC', '5F', '97', '44', '17', 'C4', 'A7', '7E',
                  '3D', '64', '5D', '19', '73'],
             ['60', '81', '4F', 'DC', '22', '2A', '90', '88', '46', 'EE', 'B8',
                  '14', 'DE', '5E', '0B', 'DB'],
```

```
            ['E0', '32', '3A', '0A', '49', '06', '24', '5C', 'C2', 'D3', 'AC',
                '62', '91', '95', 'E4', '79'],
            ['E7', 'C8', '37', '6D', '8D', 'D5', '4E', 'A9', '6C', '56', 'F4',
                'EA', '65', '7A', 'AE', '08'],
            ['BA', '78', '25', '2E', '1C', 'A6', 'B4', 'C6', 'E8', 'DD', '74',
                '1F', '4B', 'BD', '8B', '8A'],
            ['70', '3E', 'B5', '66', '48', '03', 'F6', '0E', '61', '35', '57',
                'B9', '86', 'C1', '1D', '9E'],
            ['E1', 'F8', '98', '11', '69', 'D9', '8E', '94', '9B', '1E', '87',
                'E9', 'CE', '55', '28', 'DF'],
            ['8C', 'A1', '89', '0D', 'BF', 'E6', '42', '68', '41', '99', '2D',
                '0F', 'B0', '54', 'BB', '16']]
    result = ''
    # 对每一个字节分别操作，高四位为行，第四位为列
    for i in range(0, len(byte), 2):
        x = int(byte[i], 16)
        y = int(byte[i + 1], 16)
        result += S_box[x][y]
    return result
```

代码 13: S 盒变换代码

5.1.2 时间复杂度分析

在 S 盒初始化过程中，S 盒是一个固定大小的二维数组（16x16），其初始化时间是常数记位 $O(1)$。在对每个字节的处理中函数通过循环遍历输入字节字符串。循环的次数与输入字节的长度成正比。对于每个字节，函数执行一系列固定时间的操作（计算行和列索引，然后从 S 盒中检索相应的值）。因此，这部分的时间复杂度与输入字节的长度成正比，即 $O(n)$，其中 n 是输入字节字符串的长度。

由此可见整个 S 盒变换函数的的总体时间复杂度是 $O(n)$。

5.1.3 空间复杂度分析

S 盒是一个固定大小的二维数组，不论输入数据的大小如何，其占用的空间都是常数。所以，这部分的空间复杂度是 $O(1)$。对于结果字符串来说其长度与输入字符串 byte 的长度成正比。因此，这部分的空间复杂度与输入的长度成正比，即 $O(n)$，其中 n 是输入字节字符串的长度。

所以，整个 S 盒变换函数的总体空间复杂度也是 $O(n)$。

5.2 编程研究 AES 的 S 盒的以下特性

5.2.1 明文输入改变 1 位，密文输出平均改变多少位？

将明文输入使用原本的明文，密钥保持不变，每一次改变明文中的一位然后统计密文修改的位数，将得到的密文的改变统计平均值编写函数完成这一过程的代码如下所示

```
from AES import *


"""
明文输入改变1位，密文输出平均改变多少位？
"""
def count_differences(str1, str2):
    """ 计算两个字符串之间不同字符的数量。 """
    return sum(c1 != c2 for c1, c2 in zip(str1, str2))



def test_input_changes_m(initial_string):
    """统计密文修改"""
    total_differences = 0
    total_tests = 0
    key = '00012001710198aeda79171460153594'

```

```
    for i in range(len(initial_string)):
        for hex_digit in '0123456789abcdef':
            # 一次只改变一个字符
            modified_string = initial_string[:i] + hex_digit + initial_string[i
                + 1:]

            # 对两个字符串应用'encode'函数
            encoded_initial = encode(key, initial_string)
            encoded_modified = encode(key, modified_string)

            # 计算输出中的差异
            total_differences += count_differences(encoded_initial,
                encoded_modified)
            total_tests += 1

    return total_differences / total_tests


# 生成初始的128位16进制字符串
initial_hex_string = '0001000101a198afda78173486153566'

# 计算输出变化的平均位数
average_changes = test_input_changes_m(initial_hex_string)
print("在输入修改一位的情况下，输出的平均位数变化是:", average_changes)
```

代码 14: 统计密文修改代码

统计得到的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\密码学\lab_2\test.py
在输入修改一位的情况下，输出的平均位数变化是: 28.22265625

Process finished with exit code 0
```

图 5: 统计密文修改结果

5.2.2 S 盒输入改变 1 位，S 盒输出平均改变多少位？

将 S 盒的初始值设置为'a0' 每一次改变其中的一位然后统计得到的结果修改的位数，将得到的修改结果统计平均值编写函数完成这一过程的代码如下所示

```python
from AES import *


"""
S盒输入改变1位，S盒输出平均改变多少位？
"""



def count_differences(str1, str2):
    """ 计算两个字符串之间不同字符的数量。 """
    return sum(c1 != c2 for c1, c2 in zip(str1, str2))



def test_input_changes_s(initial_string):
    total_differences = 0
    total_tests = 0
    key = '00012001710198aeda79171460153594'

    for i in range(len(initial_string)):
        for hex_digit in '0123456789abcdef':
            # 一次只改变一个字符
```

```
            modified_string = initial_string[:i] + hex_digit + initial_string[i
                + 1:]

            encoded_initial = SubByte(initial_string)
            encoded_modified = SubByte(modified_string)

            # 计算输出中的差异
            total_differences += count_differences(encoded_initial,
                encoded_modified)
            total_tests += 1

    return total_differences / total_tests


S1 = '0a'
average_changes = test_input_changes_s(S1)
print("在输入修改一位的情况下，输出的平均位数变化是:", average_changes)
```

代码 15: 统计 S 盒修改代码

统计得到的结果如下图所示

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\密码学\lab_2\test_change.py
在输入修改一位的情况下，输出的平均位数变化是: 1.6875

Process finished with exit code 0
```

图 6: 统计密文修改结果

5.2.3 对于一个输入，连续施加 S 盒变换，变换多少次时出现输出等于输入？

选取 S 盒中的每一项作为输入，统计该项经过数次 S 盒变换得到原值的次数，最后计算总体的平均值实现这一步骤的代码如下所示

```
"""
对于一个输入，连续施加S盒变换，变换多少次时出现输出等于输入？
"""

from AES import *


def s(s1):
    s2 = SubByte(s1)
    i = 1
    while (s2 != s1):
        s2 = SubByte(s2)
        i += 1
    return i


S_box = [['63', '7C', '77', '7B', 'F2', '6B', '6F', 'C5', '30', '01', '67', '2B
    ', 'FE', 'D7', 'AB', '76'],
         ['CA', '82', 'C9', '7D', 'FA', '59', '47', 'F0', 'AD', 'D4', 'A2', 'AF
             ', '9C', 'A4', '72', 'C0'],
         ['B7', 'FD', '93', '26', '36', '3F', 'F7', 'CC', '34', 'A5', 'E5', 'F1
             ', '71', 'D8', '31', '15'],
         ['04', 'C7', '23', 'C3', '18', '96', '05', '9A', '07', '12', '80', 'E2
             ', 'EB', '27', 'B2', '75'],
         ['09', '83', '2C', '1A', '1B', '6E', '5A', 'A0', '52', '3B', 'D6', 'B3
             ', '29', 'E3', '2F', '84'],
```

```
22          ['53', 'D1', '00', 'ED', '20', 'FC', 'B1', '5B', '6A', 'CB', 'BE', '39
                ', '4A', '4C', '58', 'CF'],
23          ['D0', 'EF', 'AA', 'FB', '43', '4D', '33', '85', '45', 'F9', '02', '7F
                ', '50', '3C', '9F', 'A8'],
24          ['51', 'A3', '40', '8F', '92', '9D', '38', 'F5', 'BC', 'B6', 'DA', '21
                ', '10', 'FF', 'F3', 'D2'],
25          ['CD', '0C', '13', 'EC', '5F', '97', '44', '17', 'C4', 'A7', '7E', '3D
                ', '64', '5D', '19', '73'],
26          ['60', '81', '4F', 'DC', '22', '2A', '90', '88', '46', 'EE', 'B8', '14
                ', 'DE', '5E', '0B', 'DB'],
27          ['E0', '32', '3A', '0A', '49', '06', '24', '5C', 'C2', 'D3', 'AC', '62
                ', '91', '95', 'E4', '79'],
28          ['E7', 'C8', '37', '6D', '8D', 'D5', '4E', 'A9', '6C', '56', 'F4', 'EA
                ', '65', '7A', 'AE', '08'],
29          ['BA', '78', '25', '2E', '1C', 'A6', 'B4', 'C6', 'E8', 'DD', '74', '1F
                ', '4B', 'BD', '8B', '8A'],
30          ['70', '3E', 'B5', '66', '48', '03', 'F6', '0E', '61', '35', '57', 'B9
                ', '86', 'C1', '1D', '9E'],
31          ['E1', 'F8', '98', '11', '69', 'D9', '8E', '94', '9B', '1E', '87', 'E9
                ', 'CE', '55', '28', 'DF'],
32          ['8C', 'A1', '89', '0D', 'BF', 'E6', '42', '68', '41', '99', '2D', '0F
                ', 'B0', '54', 'BB', '16']]
33  result = 0
34  for i in range(16):
35      for j in range(16):
36          input = S_box[i][j]
37          result += s(input)
38  result = result / 256
39  print("将S盒的输入连续施加S盒变换得到原输出的改变位数平均值：")
```

```
40 print(result)
```

代码 16: 统计 s 盒变换回原值平均值代码

```
E:\Python_code\venvs\Scripts\python.exe E:\Python_code\codes\密码学\lab_2\test_change.py
将S盒的输入连续施加S盒变换得到原输出的改变位数平均值：
71.65625
```

图 7: 连续施加 s 盒结果

六、 拓展思考

6.1 比较 AES 和 DES，说明它们各有什么特点？

6.1.1 AES

（1）密钥长度：AES 支持 128 位、192 位和 256 位密钥长度。

（2）安全性：由于较长的密钥长度和更复杂的加密机制，AES 比 DES 提供更高的安全性。它被认为是抵抗所有已知攻击的。

（3）算法结构：AES 使用了一个称为 Rijndael 的加密算法，基于一个称为“状态”的字节矩阵，其加密过程包括多轮的替换、置换和混合操作。

（4）效率：在现代硬件上，AES 通常比 DES 更快，尤其是在处理大量数据时。

（5）应用广泛：AES 是目前最流行和最广泛使用的加密标准，用于保护政府、金融机构和其他许多行业的敏感数据。

6.1.2 DES

（1）密钥长度：DES 仅使用 56 位密钥长度（加上 8 位奇偶校验，实际长度为 64 位）。

（2）安全性：由于密钥长度较短，DES 容易受到暴力破解攻击。1990 年代后期，DES 的安全性已经不被普遍认可。

（3）算法结构：DES 基于 Feistel 网络，它通过 16 轮相同的操作进行加密，每轮使用不同的密钥。

（4）效率：在较老的或资源受限的硬件上，DES 可能仍然有效，但它在现代系统中通常比 AES 慢。

（5）应用范围：虽然现在已经不推荐使用 DES，但它在历史上是最广泛使用的加密算法之一，对加密技术的发展有重大影响。

6.2 AES 的解密算法与加密算法有什么不同？

（1）操作顺序：解密过程中的操作是加密过程中操作的逆序。例如，加密中的字节替换在解密时变为逆字节替换。

（2）轮密钥使用：虽然使用的是相同的轮密钥集，但在解密过程中，这些轮密钥的使用顺序与加密过程相反。

6.3 在 $GF(2^8)$ 中，01 的逆元素是什么？

由于’0x01’ 本身就是单位元，它与任何多项式相乘的结果都是那个多项式本身。因此，在 $GF(2^8)$ 中，’0x01’ 的逆就是它自己，即’0x01’。

6.4 在 AES 中，对于字节’00’ 和’01’ 计算 S 盒的输出。

6.4.1 ’00’

（1）求逆元：在 $GF(2^8)$ 中，’00’ 没有逆元。对于 AES 的 S 盒，当输入是’00’ 时，我们直接跳过求逆元的步骤。

（2）应用仿射变换：对于’00’，仿射变换是简单的，因为不需要与任何值异或。应用 AES S 盒的固定仿射变换后，结果为’63’。这是因为仿射变换在所有输入为 0 时输出的固定值。

因此，对于’00’，S 盒的输出是’63’。

6.4.2 '01'

（1）求逆元：$GF(2^8)$ 中，'01' 的逆元是它自己，因为任何元素与其逆元相乘的结果为 1

（2）应用仿射变换：在找到逆元之后，我们需要对它应用 AES S 盒的固定仿射变换。仿射变换是一个固定的线性变换，通常通过矩阵乘法和向量加法来实现。仿射变换的公式是：

$b(x) = (x \oplus (x \ll 1) \oplus (x \ll 2) \oplus (x \ll 3) \oplus (x \ll 4) \oplus' 63')$

其中 $\ll$ 表示循环左移，'63' 是十六进制数，对应于二进制的'01100011'。对于'01'（二进制为'00000001'），

应用上述变换得到：$00000001 \oplus 00000010 \oplus 00000100 \oplus 00001000 \oplus 00010000 \oplus 01100011 = 01111110$

这个结果是二进制形式，转换为十六进制是'7E'。

6.5 证明：模 x4+1，$c(x)$ 与 $d(x)$ 互逆。

因为

$c(x) = \{03\}\, x^3 + \{01\}\, x^2 + \{01\}\, x + \{02\}$

$d(x) = \{0B\}\, x^3 + \{0D\}\, x^2 + \{09\}\, x + \{0E\}$

两个多项式在二进制下表示为

$c(x) = (11)_2\, x^3 + (01)_2\, x^2 + (01)_2\, x + (10)_2$

$d(x) = (1011)_2\, x^3 + (1101)_2\, x^2 + (1001)_2\, x + (1110)_2$

将两个二进制多项式直接相乘得到的结果应该为 $x^6 + x^4 + x^2$

与 $x^4 + 1$ 取模之后得到 1，得证 $c(x), d(x)$ 互逆

6.6 证明：xi mod $(x4+1)$=xi mod 4 。

对于任意的 i, 将 i 分解为 i=4k+r 的形式，其中 r=i mod 4。

将 x^i 进行拆分得到 $x^i = x^{4k+1} = (x^4)^k * x^r$

由于 $x^4 \equiv -1 mod(x^4+1)$

所以 $(x^4)^k * x^r$ 在模 x^4+1 的情况下等价于 $(-1)^k \equiv 1 mod 2$

所以可以得到 $x^i \equiv x^r mod(x^4+1)$, 又因为 r=i mod 4

可以得到结论 $x^i mod(x^4+1) = x^{imod4}$

6.7 利用 AES 的对数表或反对数表计算 ByteSub(25)。

首先通过查找 AES 的对数表和反对数表得到 0x25 的逆元为 0x4d, 再将得到的 0x4d=01001101 进行仿射变换

$$y0 = (x0 + x4 + x5 + x6 + x7) mod 2 \oplus 1 = 0$$

$$y1 = (x0 + x1 + x5 + x6 + x7) mod 2 \oplus 1 = 0$$

$$y2 = (x0 + x1 + x2 + x6 + x7) mod 2 \oplus 0 = 1$$

$$y3 = (x0 + x1 + x2 + x3 + x7) mod 2 \oplus 0 = 1$$

$$y4 = (x0 + x1 + x2 + x3 + x4) mod 2 \oplus 0 = 1$$

$$y5 = (x1 + x2 + x3 + x4 + x5) mod 2 \oplus 1 = 1$$

$$y6 = (x2 + x3 + x4 + x5 + x6) mod 2 \oplus 1 = 1$$

$$y7 = (x3 + x4 + x5 + x6 + x7) mod 2 \oplus 0 = 1$$

仿射变换后得到 S 盒的输出为 0x3f, 所以 ByteSub（25）=3f

6.8 求出 AES 的 S 盒的逆矩阵

若 S 盒的输入为 0xXY, 则需要在 S 盒矩阵中查找第 X 行和第 Y 列的值，假设得到的 S 盒的出为 0xGH, 则根据该 S 盒构造的逆 S 盒应当将 G 行 H 列位置的元素

设置为 XY 以实现求出 S 盒的逆。计算 S 盒的逆矩阵的算法如下所示

```
"""
求出AES的S盒的逆矩阵
"""

def InverseSbox(sbox):
    inv_sbox = [[0 for _ in range(16)] for _ in range(16)]

    # 遍历S盒的每个元素
    for x in range(16):
        for y in range(16):
            # S盒的输入是0xXY
            value = sbox[x][y]

            # 将value分解为G和H
            g = value // 16
            h = value % 16

            # 将逆S盒的G行H列设置为0xXY
            inv_sbox[g][h] = x * 16 + y

    return inv_sbox


s_box = [[0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0
    x2B, 0xFE, 0xD7, 0xAB, 0x76],
         [0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0
             xAF, 0x9C, 0xA4, 0x72, 0xC0],
         [0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0
```

```
            xF1, 0x71, 0xD8, 0x31, 0x15],
        [0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0
            xE2, 0xEB, 0x27, 0xB2, 0x75],
        [0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0
            xB3, 0x29, 0xE3, 0x2F, 0x84],
        [0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0
            x39, 0x4A, 0x4C, 0x58, 0xCF],
        [0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0
            x7F, 0x50, 0x3C, 0x9F, 0xA8],
        [0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0
            x21, 0x10, 0xFF, 0xF3, 0xD2],
        [0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0
            x3D, 0x64, 0x5D, 0x19, 0x73],
        [0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0
            x14, 0xDE, 0x5E, 0x0B, 0xDB],
        [0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0
            x62, 0x91, 0x95, 0xE4, 0x79],
        [0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0
            xEA, 0x65, 0x7A, 0xAE, 0x08],
        [0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0
            x1F, 0x4B, 0xBD, 0x8B, 0x8A],
        [0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0
            xB9, 0x86, 0xC1, 0x1D, 0x9E],
        [0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0
            xE9, 0xCE, 0x55, 0x28, 0xDF],
        [0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0
            x0F, 0xB0, 0x54, 0xBB, 0x16]]

# 计算逆S盒
```

```
inverse_sbox = InverseSbox(s_box)


# 打印逆S盒
print("得到的逆S盒为：")
for row in inverse_sbox:
    print(' '.join(format(x, '02x') for x in row))
```

代码 17: 计算 S 盒逆矩阵代码

算法运行后的结果如下图所示，能够求出给出的 S 盒的逆矩阵

图 8: 使用的 S 盒矩阵

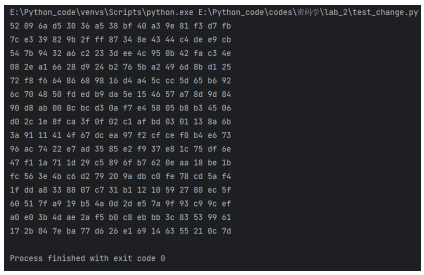

图 9: 求解出的 S 盒逆矩阵

6.9 证明

6.9.1 $InvShiftRow(InvByteSub(S)) = InvByteSub(InvShiftRow(S))$

证明过程如下图所示，两个逆过程的先后并不会影响最终的结果，因为逆行位移会影响到每个字节的位置，而逆 S 变换会影响到每个字节的值，两者并不存在操作顺序的先后关系

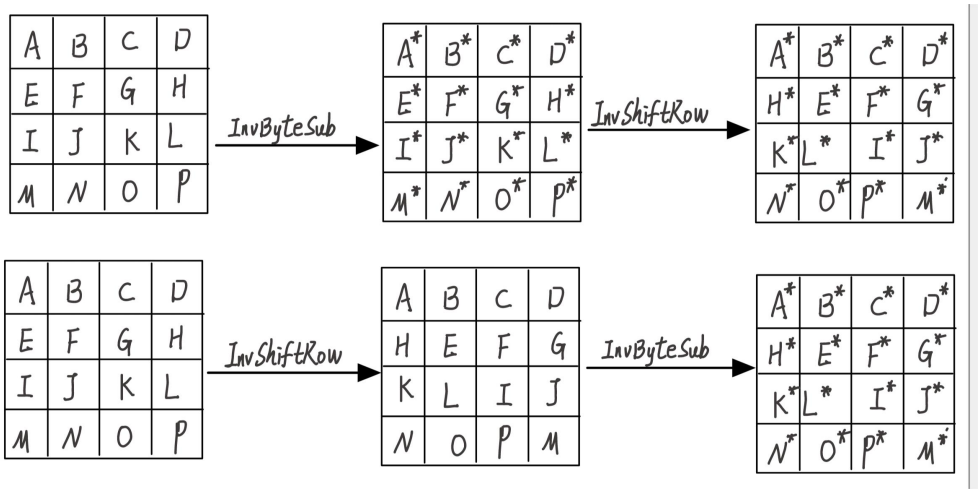

图 10: 证明 1

6.9.2 $InvMixColunm(S \oplus W) = InvMixColunm(S) \oplus InvMixColunm(W)$

（1）首先，AES 算法中的 InvMixColumn 操作是矩阵乘法，它对输入矩阵的每一列应用一个特定的线性变换。

（2）将 $S \oplus W$ 表示为一个新的矩阵 X，即 $X = S \oplus W$

（3）由于 InvMixColumn 操作是线性的，我们可以将等式分解为每一列的等式。对于每一列，我们可以证明

$InvMixColumn(X_i) = InvMixColumn(S_i) \oplus InvMixColumn(W_i)$

其中 X_i, S_i, W_i 分别表示矩阵 X、S 和 W 的第 i 列

（4）因此，我们得出结论，对于每一列

$InvMixColumn(X_i) = InvMixColumn(S_i) \oplus InvMixColumn(W_i)$ 成立

（5）最终，由于这个等式对于矩阵的每一列都成立，我们可以得出整个矩阵的

等式：$InvMixColumn(X) = InvMixColumn(S) \oplus InvMixColumn(W)$

6.9.3 说明上述结论对 AES 解密算法的设计有何作用

（1）算法优化和灵活性：通过这些等式，我们了解到特定的操作步骤可以在不改变最终结果的情况下交换顺序。这种灵活性允许算法设计者根据特定的硬件或软件环境调整步骤的执行顺序，以优化性能。例如，在某些硬件上，执行顺序的调整可以利用并行处理能力或减少必要的内存访问，从而提高效率。

（2）简化实现：这些数学关系简化了 AES 解密算法的实现。理解这些等式可以帮助设计者减少算法中的冗余步骤，从而减少实现复杂性。这对于需要在资源受限的环境（如嵌入式系统或物联网设备）中实施 AES 的情况尤为重要。

（3）提高安全性：这些等式还有助于确保算法实现的正确性，这对于保持加密算法的安全性至关重要。错误的实现可能会引入安全漏洞。通过确保这些等式在实现中得到正确应用，可以减少由于实现错误导致的安全风险。

（4）可扩展性和可维护性：在软件工程的角度来看，这种数学上的灵活性使得代码更易于维护和扩展。例如，如果未来需要对算法进行修改或优化，理解这些基本的数学关系可以帮助设计者在不影响整体安全性和性能的情况下，更容易地进行调整。

6.10 了解 AES 采用的 SP（代替-置换）结构的特点。

（1）层次结构：

SP 网络由多个轮（rounds）组成，每一轮都包含一个或多个 S-boxes（代替盒子）和 P-boxes（置换盒子）。在 AES 中，每个轮包括字节代替、行位移、列混淆和轮密钥加操作。

（2）代替（S-boxes）：

S-box 是一种非线性替换操作，它将输入的每个字节替换为另一个字节，这种替换基于一个固定的替换表。在 AES 中，S-box 用于提供混淆（confusion），即隐藏明文和密文之间的关系。

（3）置换（P-boxes）：

P-box 是一种线性变换操作，它重新排列或置换输入数据的位。在 AES 中，行位移和列混淆操作共同起到置换的作用，提供扩散（diffusion），即将明文的一个部分分散影响到密文的多个部分。

（4）安全性：

通过将非线性 S-boxes 和线性 P-boxes 组合起来，SP 网络能够提供很强的抵抗密码分析攻击的能力。这种结构有效地破坏了数据的统计结构，使得密码分析变得更加困难。

（5）效率和可实现性：

SP 网络特别适合于实现高效的软件和硬件加密算法，因为它们的操作可以并行执行，并且容易转换为电路设计。AES 特别考虑了在不同平台上（从小型嵌入式设备到大型服务器）的执行效率和实现复杂性。

七、 拓展练习

7.1 S 盒的安全性测试

对于 AES S 盒，计算其差分分布表和非线性度；

注: 差分分布表的定义是对于一对任意输入 x1 和 x2, 满足 $\triangle x = x1 + x2$, 输出等于 $\triangle y = y1 + y2$ 的统计计数中的最大次数例如对于 $\triangle x = 0$, 则 $\triangle y = 0$ 出现 256 次, 其余 $\triangle y$=1,2,3⋯,255 出现 0 次, 则 $\triangle x = 0$ 的差分次数是 256;

计算差分分布表的代码如下所示

```
def S_ddt():
    """S盒的差分分布表"""
    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]

    size = 256  # AES S-盒的大小
```

```
    ddt = [[0] * size for _ in range(size)]

    for x1 in range(size):
        for x2 in range(size):
            delta_x = x1 ^ x2
            y1 = s_box[x1]
            y2 = s_box[x2]
            delta_y = y1 ^ y2
            ddt[delta_x][delta_y] += 1

    print(ddt)
```

代码 18: 计算 S 盒逆矩阵代码

由于计算得到的差分分布表过长，在这里只展示计算结果的一部分如下图所示

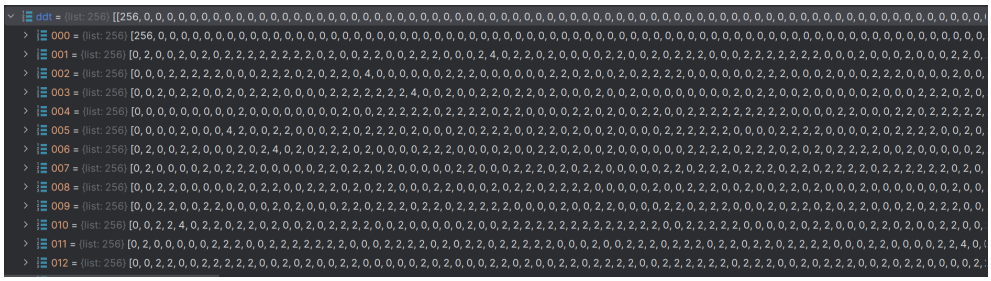

图 11: 差分分布表

计算非线性度的代码如下所示

```
def aes_sbox_nonlinearity():
    from itertools import product

    # AES 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
```

```
    ]

    max_nonlinearity = 0

    for i in range(256):
        for j in range(256):
            if i != j:
                a = i
                b = j
                output_diff = sbox[a] ^ sbox[b]
                count = bin(output_diff).count('1')
                nonlinearity = 64 - count
                if nonlinearity > max_nonlinearity:
                    max_nonlinearity = nonlinearity

    return max_nonlinearity


S_ddt()
result = aes_sbox_nonlinearity()
print("AES S-Box Nonlinearity:", result)
```

代码 19: 计算非线性度代码

计算非线性度得到的结果如下图所示

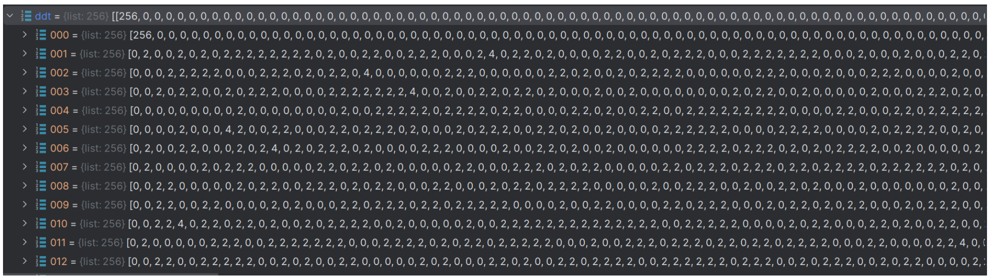

图 12: 差分分布表

7.2 S 盒的设计

产生新的 S 盒使其达到题 1 中的性质最优；例：AES 的 S 盒是计算输入 X 的逆，然后做仿射变换得出输出 Y=AX-1+B= AX254+B。尝试 Y=AXC+B 的形式,

（1）C 要求汉明重量为 7（例如 AES 中 254=11111110）,

（2）新盒可以改变仿射变换使用的（满秩）矩阵 A 或向量 B

给出结果, 并计算其差分分布表和非线性度。

仅实现了题目一，题目二无法实现

八、 实验结果与数据处理

在本次实验中选用的明文为’0001000101a198afda78173486153566’,

密钥为’00012001710198aeda79171460153594’，程序会将明文使用密钥进行加密之后将得到的密文输出，并使用密文运行解密程序，最后将解密得到的明文进行输出，程序运行的结果如下图所示

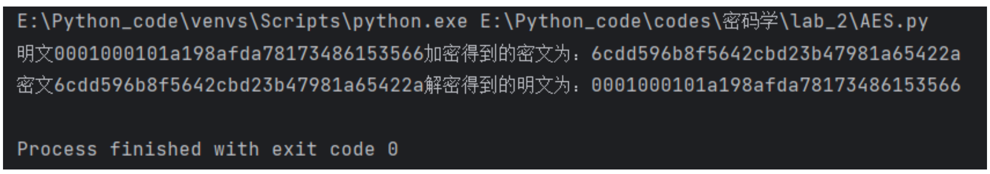

图 13: AES 加密解密结果

九、 分析与讨论

（1）深入了解了分组密码的基本概念。分组密码是一种加密算法，它将明文数据分割成固定大小的数据块，并通过密钥对每个数据块进行加密和解密。理解了分组密码的工作原理和它在数据加密中的重要性。

（2）全面学习了高级加密标准（AES）密码算法。了解了 AES 的基本结构，包括轮数、密钥扩展和四个主要操作：字节替代、行位移、列混淆和轮密钥加。还学习了 AES 算法的加密和解密过程，以及如何生成和管理密钥。

（3）讨论了 AES 密码的安全性。了解了 AES 被广泛认可为安全可靠的密码算法，其安全性经过了严格的评估和测试。学习了 AES 密码抵抗不同类型的攻击，并了解了为什么它是许多应用中的首选加密算法。

（4）最后熟悉了分组密码的广泛应用。分组密码在网络通信、数据存储、安全通信和加密文件等领域都有重要应用。了解了如何在实际应用中使用 AES 算法来保护数据的机密性和完整性。

十、 教师评语