# 实验一

(密码学与网络安全课程报告)

姓　　名：肖文韬

学　　号：2020214245

专 业 方 向：电子信息（计算机技术）

邮　　箱：xwt20@mails.tsinghua.edu.cn

二〇二一年三月十五日

# 第 1 章　实验介绍

AES 加密算法是迭代型分组密码算法，涉及 4 种操作：字节替代、行移位、列混合和轮密钥加。本次实验中密钥长度和分组长度都为 128 比特，加密轮数为 10 轮。实验使用 Rust 语言实现 AES128 的加密解密操作。

实验目的：

1. 熟练掌握 AES 加密算法的理论
2. 学习 rust 语言的基本用法

实验平台：

1. rust 语言
2. Arch Linux (不依赖具体操作系统，rust 亦可在 windows/macOS 上使用)

# 第 2 章　实验内容

## 2.1　字节替代变换和逆字节替代变换

根据 16×16 的字节替代矩阵和逆替代矩阵，对于每个字节，将其前 4bit 作为横坐标，后 4bit 作为纵坐标，使用下方替代矩阵对应位置的字节进行替代。完成对一个长文本的字节替代变换和逆字节替代变换，对于不足 16 字符的输入需要补位，为了方便之后的加解密，16 字节的文本应转换为 4×4 的字节矩阵。

**解：**

因为 AES128 是作用于块大小为 128 bits 的块加密算法。所以本题首先需要实现将明文消息分块，并且对于最后一块不足 16 字节时进行补位。再对每一块进行字节替代，再进行逆字节替代。最后进行逆补位得到原文。

### 2.1.1　补位

```rust
fn pad(states: &mut Vec<u8>) -> () {
  let mut pad_size = states.len() % BLOCK_SIZE;
  if pad_size != 0 {
      pad_size = BLOCK_SIZE - pad_size;
  }
  states.extend(vec![PAD_BYTE; pad_size]);
}
```

其中 `PAD_BYTE` 为 0，即用 0 来补位，`BLOCK_SIZE` 为 16 字节，即 128 bits。

### 2.1.2　字节替代

```rust
fn sub_bytes(states: &mut Vec<u8>) -> () {
  for idx in 0..states.len() {
      let b = states[idx];
      states[idx] = SBOX[(b >> 4) as usize][(b & 0x0f) as usize];
  }
}
```

其中 SBOX 是提前给定的。

### 2.1.3　逆字节替代

```rust
fn inv_sub_bytes(states: &mut Vec<u8>) -> () {
    for idx in 0..states.len() {
        let b = states[idx];
        states[idx] = INV_SBOX[(b >> 4) as usize][(b & 0x0f) as usize];
    }
}
```

INV_SBOX 也是根据 SBOX 对应的逆替换表。

### 2.1.4　逆补位

```rust
fn unpad(states: &mut Vec<u8>) -> () {
    let blocks = states.chunks_mut(BLOCK_SIZE);
    let last_block: &mut [u8] = blocks.last().unwrap();
    let padding_len = BLOCK_SIZE - match last_block.iter().position(
        |&x| x == PAD_BYTE) {
        Some(pos) => pos,
        None => BLOCK_SIZE,
    };
    for _ in 0..padding_len {
        states.remove(states.len() - 1);
    }
}
```

逆补位稍微复杂一点，首先需要在逆变换后的补位长度，然后将补位部分截断即可。

### 2.1.5　运行结果

本题使用字符串’abcdefghijklmn’ 作为输入，完成字符串的补位、转码、字节替代、逆字节替代、转码、去补位，输出每一步的结果。运行结果如图 2.1所示。

## 2.2　行移位变换和逆行移位变换，列混合和逆列混合

编写对 4×4 字节矩阵的行移位变换和逆行移位变换代码。编写对 4×4 字节矩阵的列混合和逆列混合代码，列混合需要使用 x 乘法。

```
plaintext: abcdefghijklmn
Pad:
Block 0:
61 65 69 6D
62 66 6A 6E
63 67 6B 00
64 68 6C 00

Sub Bytes:
Block 0:
EF 4D F9 3C
AA 33 02 9F
FB 85 7F 63
43 45 50 63

Inv Sub Bytes:
Block 0:
61 65 69 6D
62 66 6A 6E
63 67 6B 00
64 68 6C 00

Unpad:
Result text: abcdefghijklmn
```

图 2.1　字节替换运行结果

### 2.2.1　行移位变换

行移位就是将每一块（block）用状态矩阵（state）表示，然后对每一行做一些简单的循环左移运算。具体的，第一行不移位，第二行循环左移一位，第三行循环左移两位，第四行循环左移三位。

```rust
fn shift_rows(states: &mut Vec<u8>) -> () {
    let blocks = states.chunks_mut(BLOCK_SIZE);
    for state in blocks {
        let mut temp: u8;
        // row 1
        temp = state[1];
        state[1] = state[5];
        state[5] = state[9];
        state[9] = state[13];
        state[13] = temp;
        // row 2
        temp = state[2];
        state[2] = state[10];
        state[10] = temp;
        temp = state[6];
        state[6] = state[14];
        state[14] = temp;
        // row 3
        temp = state[15];
        state[15] = state[11];
        state[11] = state[7];
        state[7] = state[3];
        state[3] = temp;
    }
}
```

### 2.2.2　列混合

在列混合中，状态矩阵中的每一个字节都可以看作是 $GF(2^8)$ 上的多项式，且最高项次数不超过 7。在 $GF(2^8)$ 上乘以另外一个模多项式 $c(x)$ 再取模可以写作一个矩阵运算（由 $GF(2^8)$ 上定义的乘法实现的）。对于输入多项式 $a(x)$，乘以模多项式 $c(x) =' 03' x^3 +' 01' x^2 +' 01' x +' 02'$ 可以表示为：

$$
\begin{aligned}
b(x) &= c(x) \otimes a(x) \\
&= \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}
\end{aligned} \tag{2-1}
$$

注意 $GF(2^8)$ 上的加法就是异或运算。同时，$GF(2^8)$ 上一个多项式乘以任意多项式（任意常数）都可以归约为乘以常数'01' (也就是多项式 $x$)。例如，上面矩阵的结果中，有以下推导：

$$b_0 =' 02' \cdot a_0 +' 03' \cdot a_1 +' 01' \cdot a_2 +' 01' \cdot a_3$$

$$'02' \cdot a_0 + a_1 \cdot ('01' +' 02') + a_2 + a_3$$

$$= (a_0 + a_1) \cdot' 02' + a_1 + a_2 + a_3$$

$$= (a_0 + a_1) \cdot' 02' + a_0 + a_0 + a_1 + a_2 + a_3$$

```rust
fn mix_columns(states: &mut Vec<u8>) {
    let blocks = states.chunks_mut(BLOCK_SIZE);
    for state in blocks {
        let columns = state.chunks_mut(ROW_COUNT);
        for column in columns {
            let tmp = column[0] ^ column[1] ^ column[2] ^ column[3];
            let bak_c0 = column[0];
            column[0] = xtime(column[0] ^ column[1]) ^ column[0] ^ tmp;
            column[1] = xtime(column[1] ^ column[2]) ^ column[1] ^ tmp;
            column[2] = xtime(column[2] ^ column[3]) ^ column[2] ^ tmp;
            column[3] = xtime(column[3] ^ bak_c0) ^ column[3] ^ tmp;
        }
    }
}
```

### 2.2.3　逆列混合

```rust
fn inv_mix_columns(states: &mut Vec<u8>) {
    let blocks = states.chunks_mut(BLOCK_SIZE);
    for state in blocks {
        let columns = state.chunks_mut(ROW_COUNT);
        for column in columns {
            let mut t = column[0] ^ column[1] ^ column[2] ^ column[3];
            let u = xtime(xtime(column[0] ^ column[2]));
            let v = xtime(xtime(column[1] ^ column[3]));
            let bak_c0 = column[0];
            column[0] = t ^ column[0] ^ xtime(column[0] ^ column[1]);
            column[1] = t ^ column[1] ^ xtime(column[1] ^ column[2]);
            column[2] = t ^ column[2] ^ xtime(column[2] ^ column[3]);
            column[3] = t ^ column[3] ^ xtime(column[3] ^ bak_c0);
            t = xtime(u ^ v);
            column[0] ^= t ^ u;
            column[1] ^= t ^ v;
            column[2] ^= t ^ u;
            column[3] ^= t ^ v;
        }
    }
}
```

### 2.2.4　逆行移位变换

逆行移位变换就是行移位运算的逆过程，就是将之前的循环左移变成循右移。

```rust
fn inv_shift_rows(states: &mut Vec<u8>) -> () {
    let blocks = states.chunks_mut(BLOCK_SIZE);
    for state in blocks {
        let mut temp: u8;
        temp = state[13];
        state[13] = state[9];
        state[9] = state[5];
        state[5] = state[1];
        state[1] = temp;
        temp = state[14];
        state[14] = state[6];
        state[6] = temp;
        temp = state[10];
        state[10] = state[2];
        state[2] = temp;
        temp = state[3];
        state[3] = state[7];
        state[7] = state[11];
        state[11] = state[15];
        state[15] = temp;
    }
}
```

### 2.2.5　运行结果

本题使用字符串'abcdefghijklmnop' 作为验证输入，将其转换为字节矩阵后对四个变换进行验证，记录其输出。

## 2.3　轮密钥生成

使用原始密钥'abcdefghijklmnop' 生成总计 11 组扩展密钥，将用于之后的加解密。初始密钥 K 转换为字节矩阵后的 4 列为 $W_0 \sim W_3$，后续的 $W_4 \sim W_{43}$ 使用递归计算得到。