



山东大学
SHANDONG UNIVERSITY

AES 密码的软件实现及优化

密码工程

2023 年 9 月 9 日

李 昕 202100460065

21 级密码 2 班

摘 要

AES, 高级加密标准, 在密码学中又称 Rijndael 加密法, 是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的 DES, 目前已经被全世界广泛使用, 同时 AES 已经成为对称密钥加密中最流行的算法之一。AES 支持三种长度的密钥: 128 位, 192 位, 256 位。AES 的代码实现较为简单, 但在 AES 的软件实现中, 为了平衡加密算法的时间和空间复杂度, 利于计算机软件实现, 常常采用构造查找表的实现方法。本篇实验报告即利用查找表法构造 AES-128 和 AES-256 加密算法, 并对其进行优化。

基于查找表 AES-128/AES-256 算法, 构造 2^8 比特的 S 盒查找表, 同时, 对于列混淆, 建立 2^8 个表项的查找表 L, 实现 $x \rightarrow (2x, x, x, 3x)$ 的查找替换, 列混淆表达式即可转化为: $(y_0, y_1, y_2, y_3) = L(x_0) \oplus L(x_1) \gg 8 \oplus L(x_2) \gg 16 \oplus L(x_3) \gg 24$, 那么, AES 每一轮的 MixColumns 操作只需要 $4 \times 4 = 16$ 次内存访问、若干位移和异或操作即可。同时需要 $256 \times 4 = 1\text{KBytes}$ 的内存用于存放以上的查找表。

同时, 基于字节代换和 ShiftRows 可以交换顺序, 将每一轮中的前三层操作 (字节代换、ShiftRows 和 MixColumn) 合并为查找表。如果将它们交换, 那么 ShiftRows 层实际上就是按照特定的顺序去读取这一轮操作的输入数据。字节代换和 MixColumn 融合为查找表, 令输入状态为 $s_{i,j}$, 构造查找表 $T = L(S(\cdot))$, 则输出为 $T(s_{0,i}) \oplus T(s_{0, \text{mod}(i+1,4)}) \gg 8 \oplus T(s_{0, \text{mod}(i+2,4)}) \gg 16 \oplus T(s_{0, \text{mod}(i+3,4)}) \gg 24$, 大大提升 AES 的软件实现效率。

关键词: AES 加密算法软件实现 性能优化

目录

| | | |
|-----|------------------------------|----|
| 1 | 实验要求 | 1 |
| 2 | 符号说明 | 1 |
| 3 | 实验准备 | 1 |
| 3.1 | AES 加密算法的数学原理 ^[1] | 1 |
| 3.2 | 本实验需要的 AES 数据 | 2 |
| 4 | 实验过程 | 2 |
| 4.1 | 利用查找表，实现 AES 加密 | 2 |
| 4.2 | 利用查找表，实现 AES 解密 | 8 |
| 5 | 结果分析 | 12 |
| 5.1 | 加密程序运行结果 | 12 |
| 5.2 | 对查找表优化的性能分析 | 12 |
| A | 附录 | 14 |
| A.1 | aes.cpp 源代码 | 14 |

1 实验要求

问题一 分析 AES 的 S 盒结构和其他实现细节，利用基于查找表的优化方法，完成 AES 算法的软件实现。

2 符号说明

| 符号 | 解释 |
|---------------|------------|
| \oplus | 异或 |
| L,T | 查找表 |
| \rightarrow | 基于查找表的映射关系 |

3 实验准备

3.1 AES 加密算法的数学原理^[1]

AES-128 是一种常用的对称加密算法，它的加密原理主要包括以下步骤：

密钥扩展：通过对输入的 128 位密钥进行扩展，生成一系列的轮密钥，用于后续的轮函数。

初始轮：将明文与第 0 轮密钥进行异或运算，得到加密结果的初始值。

轮函数：该步骤是 AES-128 加密的核心，它包括四个步骤：字节代换、行移位、列混淆和轮密钥加。

字节代换：将初始值中的每个字节都用一个固定的 S 盒进行替换。这个 S 盒是由一系列数学运算构造出来的，可以将输入的 8 位字节映射为另一个 8 位字节。

行移位：将每一行中的字节向左循环移位，移位的数量与行号相关。比如，第一行不移位，第二行向左移动 1 个字节，第三行向左移动 2 个字节，第四行向左移动 3 个字节。

列混淆：将每一列中的 4 个字节进行一定的数学运算，将其转换为新的 4 个字节。这个运算可以增强 AES 的安全性，使其能够抵御更多的攻击。

轮密钥加：将当前轮的密钥与上一轮的加密结果进行异或运算，得到本轮的加密结果。

最终轮：与初始轮类似，最后一轮不进行列混淆操作。

CBC 工作模式：将前一个密文分组与当前明文分组的内容异或起来进行加密，以避免 ECB 模式密文重复的弱点，即明文分组在加密之前会与“前一个密文分组”进行 XOR 运算（第一组明文与初始化向量异或），因此即便明文分组 A 和 B 的值是相等的，密文分组 和 的值也不一定是相等的。

3.2 本实验需要的 AES 数据

本实验代码（包括 S 盒和 T 表具体数据）参考了 Openssl 1.1.0 版本中的 AES 快速实现，以及 GITHUB 中开源的轻量级 AES 实现（CycloneCRYPTO 库）

4 实验过程

4.1 利用查找表，实现 AES 加密

AES 算法实现主要包含以下五步：密钥扩展、字节代换、行移位、列混淆和轮密钥运算，其中，字节替换和列混淆使用查找表的方式实现，由于查找表实现较为复杂且重要，下面首先简述查找表的实现。

对于 AES 的 S 盒，8 比特输入 8 比特输出的置换，一种实现方法即将其表示成 $2^8 = 256$ 个元素查找表，其中第 i 个表项是输入值为 i 的 S 盒输入（ i 的范围是 0 到 255）。在密码算法运行的时候，可以根据 8 比特的输入值来选择对应的输出。对于 AES 每一轮，以上的方法只需要 16 次内存访问操作即可完成 S 盒的运算。另外，还需要 0.25 KBytes 的内存用于存放以上的查找表。

在实践中，我将 T 表提前编写有关程序算出，然后作为一个常量数组硬编码在 AES 的软件实现代码中。这样，就将繁琐的伽罗瓦域上的运算和矩阵运算变成了对于计算机而言非常简单高效的查表和按位运算。

但对于 AES 的 MixColumns 运算，由于考虑查找表大小限制，不能直接使用列出查找表实现。但进一步分析 MixColumns 的结构可知，输入输出的数据可以分别表示为 4 个有限域 F_{2^8} 上的元素，这里以两个列向量表示，分别是 (x, x_1, x_2, x_3) 和 (y_0, y_1, y_2, y_3) 。MixColumns 对输入的列向量左乘一个固定的矩阵，得到输出列向量。其中矩阵中的元素也是 F_{2^8} 上的元素。

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

MixColumns 操作是由 $2 \cdot$ 、 $3 \cdot$ 以及一些有限域上的加法（即异或运算）运算所组成。对于 $2 \cdot$ 操作，建立一个有 256 个表项的表，其中第 i 个表项的值存放 $2 \cdot i$ 。这样，我们可以利用 $8 \times 4 = 32$ 次内存访问操作和若干次异或操作就可以实现 MixColumns。同时需要 $2 \times 2^8 = 0.5\text{KBytes}$ 的内存空间来存放查找表。由于 $2 \cdot x + x = 3 \cdot x$ ，可以进一步把 MixColumns 写成：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 2(x_0 \oplus x_1) \oplus x_1 \oplus x_2 \oplus 3x_3 \\ x_0 \oplus 2(x_1 \oplus x_2) \oplus x_2 \oplus 3x_3 \\ x_0 \oplus x_1 \oplus 2(x_2 \oplus x_3) \oplus x_3 \\ 2(x_0 \oplus x_3) \oplus x_0 \oplus x_1 \oplus x_2 \end{bmatrix}$$

由此，只需要对 $x \rightarrow 2x$ 建立查找表即可。这样，我们可以利用 $4 \times 4 = 16$ 次内存访问操作和若干次异或操作就可以实现 MixColumns。同时需要 $2^8 = 0.25\text{KBytes}$ 的内存空间来存放查找表。对于 32 位和以上位宽处理器，将 MixColumns 操作写为：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 02 \\ 01 \\ 03 \\ 04 \end{bmatrix} x_0 + \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} x_1 + \begin{bmatrix} 01 \\ 02 \\ 02 \\ 01 \end{bmatrix} x_2 + \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} x_3$$

根据以上表示的形式，我们可以建立 $x \rightarrow (2x, x, x, 3x)$ 的查找表，即查找表中有 $2^8 = 256$ 个表项，其中第 x 个表项为 4 个 Bytes 的值 $(2x, x, x, 3x)$ 。同时，我们注意到，要计算 $(3x, 2x, x, x)$ ，只需要对 $(2x, x, x, 3x)$ 右移 1 个 Byte（即 8 比特）即可。而计算 $(x, 3x, 2x, x)$ 和 $(x, x, 3x, 2x)$ 也类似，只需要对 $(2x, x, x, 3x)$ 分别右移 2 个和 3 个 Bytes 即可。综上所述，令 $x \rightarrow (2x, x, x, 3x)$ 的查找表为 $L(\cdot)$ ，Mixcolumns 的计算可以表达为： $(y_0, y_1, y_2, y_3) = L(x_0) \oplus L(x_1) \gg 8 \oplus L(x_2) \gg 16 \oplus L(x_3) \gg 24$ ，那么，AES 每一轮的 MixColumns 操作只需要 $4 \times 4 = 16$ 次内存访问、若干位移和异或操作即可。同时需要 $256 \times 4 = 1\text{KBytes}$ 的内存用于存放以上的查找表。

进一步的，观察每一轮的加密函数结果，发现可以将每一轮中的前三层操作（字节代换层、ShiftRows 层和 MixColumn 层）合并为查找表，因为字节代换层和 ShiftRows

层可以交换顺序，则 ShiftRows 层实际上就是按照特定的顺序去读取这一轮操作的输入数据，故字节代换层和 MixColumn 层融合为查找表，提高代码运算效率。

具体优化方法为：令输入状态为 $s_{i,j}$, $i, j \in 0, \dots, 3$ ，列混淆的输出为 $L(S(s_{0,i})) \oplus L(S(s_{0,mod(i+1,4)})) \gg 8 \oplus L(S(s_{0,mod(i+2,4)})) \gg 16 \oplus L(S(s_{0,mod(i+3,4)})) \gg 24$ 。

构造查找表 $T = L(S(\cdot))$ ，则输出为 $T(s_{0,i}) \oplus T(s_{0,mod(i+1,4)}) \gg 8 \oplus T(s_{0,mod(i+2,4)}) \gg 16 \oplus T(s_{0,mod(i+3,4)}) \gg 24$ 。

则每一轮加密的前三层操作只需要 16 次内存访问和若干次异或操作即可快速实现，并且查找表 $T()$ 的大小仅需要 $256 * 4 = 1\text{KBytes}$ 即可。

于是利用通过编写程序算法计算出四个 TE 表 (即加密所需的 T-Table)：

$$Te_0(A_x) = \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} S(A_x) \quad (4)$$

$$Te_0(A_x) = \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} S(A_x) \quad (5)$$

$$Te_0(A_x) = \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} S(A_x) \quad (6)$$

$$Te_0(A_x) = \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix} S(A_x) \quad (7)$$

即加密时的轮操作化为：

$$\begin{pmatrix} D_0 \\ D_1 \\ D_2 \\ D_3 \end{pmatrix} = Te_0(A_0) + Te_1(A_5) + Te_2(A_{10}) + Te_3(A_{15}) + W_{K_0} \quad (8)$$

$$\begin{pmatrix} D_4 \\ D_5 \\ D_6 \\ D_7 \end{pmatrix} = Te_0(A_4) + Te_1(A_9) + Te_2(A_{14}) + Te_3(A_3) + W_{K_1} \quad (9)$$

$$\begin{pmatrix} D_8 \\ D_9 \\ D_{10} \\ D_{11} \end{pmatrix} = Te_0(A_8) + Te_1(A_{13}) + Te_2(A_2) + Te_3(A_7) + W_{K_2} \quad (10)$$

$$\begin{pmatrix} D_{12} \\ D_{13} \\ D_{14} \\ D_{15} \end{pmatrix} = Te_0(A_{12}) + Te_1(A_1) + Te_2(A_6) + Te_3(A_{11}) + W_{K_3} \quad (11)$$

通过分析上述原理，利用【预计算的查找表】构造加密部分代码如下（完整代码见附录 A1）：

```

1 for (int i = 1; i < Key.nr; i++) //ShiftRow + SubByte + MixCol
2 {
3     t[0] = TE[(s[0] >> 24) & 0xFF];
4     tmp = TE[(s[1] >> 16) & 0xFF];
5     t[0] ^= rotr32(tmp, 8);
6     tmp = TE[(s[2] >> 8) & 0xFF];
7     t[0] ^= rotr32(tmp, 16);
8     tmp = TE[(s[3] >> 0) & 0xFF];
9     t[0] ^= rotr32(tmp, 24);
10    t[1] = TE[(s[1] >> 24) & 0xFF];
11    tmp = TE[(s[2] >> 16) & 0xFF];
12    t[1] ^= rotr32(tmp, 8);
13    tmp = TE[(s[3] >> 8) & 0xFF];
14    t[1] ^= rotr32(tmp, 16);
15    tmp = TE[(s[0] >> 0) & 0xFF];
16    t[1] ^= rotr32(tmp, 24);
17    t[2] = TE[(s[2] >> 24) & 0xFF];
18    tmp = TE[(s[3] >> 16) & 0xFF];
19    t[2] ^= rotr32(tmp, 8);
20    tmp = TE[(s[0] >> 8) & 0xFF];

```



```

21     t[2] ^= rotr32(tmp, 16);
22     tmp = TE[(s[1] >> 0) & 0xFF];
23     t[2] ^= rotr32(tmp, 24);
24     t[3] = TE[(s[3] >> 24) & 0xFF];
25     tmp = TE[(s[0] >> 16) & 0xFF];
26     t[3] ^= rotr32(tmp, 8);
27     tmp = TE[(s[1] >> 8) & 0xFF];
28     t[3] ^= rotr32(tmp, 16);
29     tmp = TE[(s[2] >> 0) & 0xFF];
30     t[3] ^= rotr32(tmp, 24);
31     s[0] = t[0] ^ Key.ek[4 * i + 0];
32     s[1] = t[1] ^ Key.ek[4 * i + 1];
33     s[2] = t[2] ^ Key.ek[4 * i + 2];
34     s[3] = t[3] ^ Key.ek[4 * i + 3];
35 }

```

其余步骤包括密钥扩展，对于密钥扩展，利用循环模拟 AES-128 的密钥扩展过程：

1. 根据密钥的位数计算出加密轮数 (nr) 和密钥字数 (nk)，然后分配存储加密密钥和解密密钥的空间。

2. 函数使用密钥的字节表示形式填充加密密钥数组 (w)。然后，使用逆 S 盒和循环左移函数 (rotr32) 对加密密钥数组的剩余部分进行填充。

3. 函数将填充后的加密密钥数组 (w) 赋值给 *AES_Key* 结构体中的 ek 字段，并根据加密密钥数组 (w) 计算解密密钥数组 (d) 并赋值给 *AES_Key* 结构体中的 dk 字段。

4. 函数返回 1 表示密钥初始化成功，返回 0 表示内存分配失败。

代码实现：

```

1 int AES_KeyInit(uint8_t* key, AES_Key* Key, size_t bits) {
2     uint32_t Rcon[10] = { 0x01, 0x02, 0x04, 0x08, 0x10,
3         0x20, 0x40, 0x80, 0x1B, 0x36 }; //轮常数
4     uint32_t nr = 10 + (bits - 128) / 32; //加密轮数 Nr
5     uint32_t nk = bits / 32; //密钥字数 Nk
6     uint32_t tmp, tmp1;
7     Key->nr = nr;
8     uint32_t* w = (uint32_t*)malloc(sizeof(uint32_t) * 4 * (nr + 1));
9     if (w == (void*)0) {return 0;}
10    uint32_t* d = (uint32_t*)malloc(sizeof(uint32_t) * 4 * (nr + 1));

```

```

11  if (d == (void*)0) {
12      free(w);
13      return 0;
14  }
15  for (int i = 0; i < nk; i++) {
16      w[i] = (key[4 * i + 0] << 24) | (key[4 * i + 1] << 16) |
17          (key[4 * i + 2] << 8) | (key[4 * i + 3]);
18  }
19  for (int i = nk; i < 4 * (nr + 1); i++) {
20      tmp = w[i - 1];
21      if (i % nk == 0) {
22          tmp1 = tmp;
23          tmp = Sbox[(tmp1 >> 24) & 0xFF];
24          tmp |= Sbox[(tmp1 >> 0) & 0xFF] << 8;
25          tmp |= Sbox[(tmp1 >> 8) & 0xFF] << 16;
26          tmp |= (Sbox[(tmp1 >> 16) & 0xFF] ^ Rcon[i / nk - 1]) << 24;
27      }
28      else if (nk > 6 && i % nk == 4) {
29          tmp1 = tmp;
30          tmp = Sbox[(tmp1 >> 0) & 0xFF];
31          tmp |= Sbox[(tmp1 >> 8) & 0xFF] << 8;
32          tmp |= Sbox[(tmp1 >> 16) & 0xFF] << 16;
33          tmp |= Sbox[(tmp1 >> 24) & 0xFF] << 24;
34      }
35      w[i] = w[i - nk] ^ tmp;
36  }
37  Key->ek = w;
38  for (int i = 0; i < 4; i++) {d[i] = w[i];}
39  for (int i = 4; i < 4 * nr; i++) {
40      d[i] = TD[Sbox[(w[i] >> 24) & 0xFF]];
41      tmp = TD[Sbox[(w[i] >> 16) & 0xFF]];
42      d[i] ^= rotr32(tmp, 8);
43      tmp = TD[Sbox[(w[i] >> 8) & 0xFF]];
44      d[i] ^= rotr32(tmp, 16);
45      tmp = TD[Sbox[(w[i] >> 0) & 0xFF]];
46      d[i] ^= rotr32(tmp, 24);
47  }
48  for (int i = 0; i < 4; i++) {d[4 * nr + i] = w[4 * nr + i];}

```

```

49   Key->dk = d;
50   return 1;
51 }

```

4.2 利用查找表，实现 AES 解密

作为经典的对称加密算法，在 AES 原始实现中，解密也采用和加密过程相对称的流程。

但是观察 AES 的原始实现流程，对于解密过程，无论是否交换逆向字节代换层和逆向 ShiftRows 层，都无法将逆向字节代换层和逆向 MixColumn 层按照先逆向字节代换再逆向 MixColumn 的顺序放在一起，即无法形成 ShiftRows->T-Box->KeyAddition 的操作轮顺序，故按照类似于构造 Te 表的方式构造的 Td 表将无法正确应用这个流程上，且加密与解密函数此时不同构。

此时，本实验实现的 project 中，将原始解密方案中的逆向 MixColumn 层和密钥加法层交换顺序，这样可以构成 ShiftRows->T-Box->KeyAddition 的操作轮顺序，值得注意的是，这一步交换需要正确性证明，因为在 AES 算法中逆向字节代换和逆向 MixColumn 这两个操作不是可交换的。

参考本实验报告参考文献^[3]中的解释，若对逆向 MixColumn 这个操作使用矩阵的乘法分配律：

$$\begin{aligned}
& \begin{pmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} C_0 & C_4 & C_8 & C_{12} \\ C_1 & C_5 & C_9 & C_{13} \\ C_2 & C_6 & C_{10} & C_{14} \\ C_3 & C_7 & C_{11} & C_{15} \end{pmatrix} \\
& = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \left(\begin{pmatrix} D_0 & D_4 & D_8 & D_{12} \\ D_1 & D_5 & D_9 & D_{13} \\ D_2 & D_6 & D_{10} & D_{14} \\ D_3 & D_7 & D_{11} & D_{15} \end{pmatrix} + \begin{pmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{pmatrix} \right) \\
& = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} D_0 & D_4 & D_8 & D_{12} \\ D_1 & D_5 & D_9 & D_{13} \\ D_2 & D_6 & D_{10} & D_{14} \\ D_3 & D_7 & D_{11} & D_{15} \end{pmatrix} + \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} K_0 & K_4 & K_8 & K_{12} \\ K_1 & K_5 & K_9 & K_{13} \\ K_2 & K_6 & K_{10} & K_{14} \\ K_3 & K_7 & K_{11} & K_{15} \end{pmatrix}
\end{aligned}$$

故只要对子密钥也做一次逆向 MixColumn 操作，然后再用于密钥加法层，那么可以保证上述交换逆向 MixColumn 层和密钥加法层的改变不影响解密的正确性。

借用参考文献 [3] 原文中的分析，“虽然这样代码在生成解密用的子密钥的时候要多做一步操作，但是这是值得的——因为在分组密码中，处理一长段内容都采用同一个密钥，即使用同一组子密钥。我们一般只进行一次密钥编排，而将生成的子密钥保存在内存中（这里的安全假设是这台计算机的内存和 CPU 是安全的，没有受到入侵），所以这个额外的过程只需要做一次。而在处理长内容的过程中，AES 核心要对多个块依次进行处理，这里会节省下来非常多的时间。”，虽然解密时多了一次逆向 MixColumn 操作，但查找表带来的优化依然使整体加解密速度获得提升。

由此，仿照加密过程来定义解密过程中需要的 TD 表（即解密时的 T-Table）：

$$Td_0(D_x) = \begin{pmatrix} 0E \\ 09 \\ 0D \\ 0B \end{pmatrix} S^{-1}(D_x) \quad Td_1(D_x) = \begin{pmatrix} 0B \\ 0E \\ 09 \\ 0D \end{pmatrix} S^{-1}(D_x)$$

$$Td_2(D_x) = \begin{pmatrix} 0D \\ 0B \\ 0E \\ 09 \end{pmatrix} S^{-1}(D_x) \quad Td_3(D_x) = \begin{pmatrix} 09 \\ 0D \\ 0B \\ 0E \end{pmatrix} S^{-1}(D_x)$$

同理，给出计算明文的逆轮操作：

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} = Td_0(D_0) + Td_1(D_{13}) + Td_2(D_{10}) + Td_3(D_7) + W_{k0} \quad (12)$$

$$\begin{pmatrix} A_4 \\ A_5 \\ A_6 \\ A_7 \end{pmatrix} = Td_0(D_4) + Td_1(D_1) + Td_2(D_{14}) + Td_3(D_{11}) + W_{k1} \quad (13)$$

$$\begin{pmatrix} A_8 \\ A_9 \\ A_{10} \\ A_{11} \end{pmatrix} = Td_0(D_8) + Td_1(D_5) + Td_2(D_2) + Td_3(D_{15}) + W_{k2} \quad (14)$$

$$\begin{pmatrix} A_{12} \\ A_{13} \\ A_{14} \\ A_{15} \end{pmatrix} = Td_0(D_{12}) + Td_1(D_9) + Td_2(D_6) + Td_3(D_3) + W_{k3} \quad (15)$$

利用上述分析，利用【预计算的查找表】构造解密部分代码如下（完整代码见附录A1）：

```

1 void AES_Decrypt(uint8_t* ciphertext, uint8_t* plaintext, AES_Key Key) {
2     for (int i = 0; i < 4; i++) {
3         s[i] = (ciphertext[4 * i + 0] << 24) | (ciphertext[4 * i + 1] << 16) |
4             (ciphertext[4 * i + 2] << 8) | (ciphertext[4 * i + 3]);
5     }
6     s[0] ^= Key.dk[4 * Key.nr + 0]; //轮密钥加
7     s[1] ^= Key.dk[4 * Key.nr + 1];
8     s[2] ^= Key.dk[4 * Key.nr + 2];
9     s[3] ^= Key.dk[4 * Key.nr + 3];
10    for (int i = Key.nr - 1; i > 0; i--) // ShiftRow IV + SubByte IV + MixCol IV
11    {
12        t[0] = TD[(s[0] >> 24) & 0xFF]; tmp = TD[(s[3] >> 16) & 0xFF];
13        t[0] ^= rotr32(tmp, 8); tmp = TD[(s[2] >> 8) & 0xFF];
14        t[0] ^= rotr32(tmp, 16); tmp = TD[(s[1] >> 0) & 0xFF];
15        t[0] ^= rotr32(tmp, 24);
16        t[1] = TD[(s[1] >> 24) & 0xFF]; tmp = TD[(s[0] >> 16) & 0xFF];
17        t[1] ^= rotr32(tmp, 8); tmp = TD[(s[3] >> 8) & 0xFF];
18        t[1] ^= rotr32(tmp, 16); tmp = TD[(s[2] >> 0) & 0xFF];
19        t[1] ^= rotr32(tmp, 24);
20        t[2] = TD[(s[2] >> 24) & 0xFF]; tmp = TD[(s[1] >> 16) & 0xFF];
21        t[2] ^= rotr32(tmp, 8); tmp = TD[(s[0] >> 8) & 0xFF];
22        t[2] ^= rotr32(tmp, 16); tmp = TD[(s[3] >> 0) & 0xFF];
23        t[2] ^= rotr32(tmp, 24);
24        t[3] = TD[(s[3] >> 24) & 0xFF]; tmp = TD[(s[2] >> 16) & 0xFF];
25        t[3] ^= rotr32(tmp, 8); tmp = TD[(s[1] >> 8) & 0xFF];
26        t[3] ^= rotr32(tmp, 16); tmp = TD[(s[0] >> 0) & 0xFF];
27        t[3] ^= rotr32(tmp, 24);
28        s[0] = t[0] ^ Key.dk[4 * i + 0];
29        s[1] = t[1] ^ Key.dk[4 * i + 1];
30        s[2] = t[2] ^ Key.dk[4 * i + 2];
31        s[3] = t[3] ^ Key.dk[4 * i + 3];

```

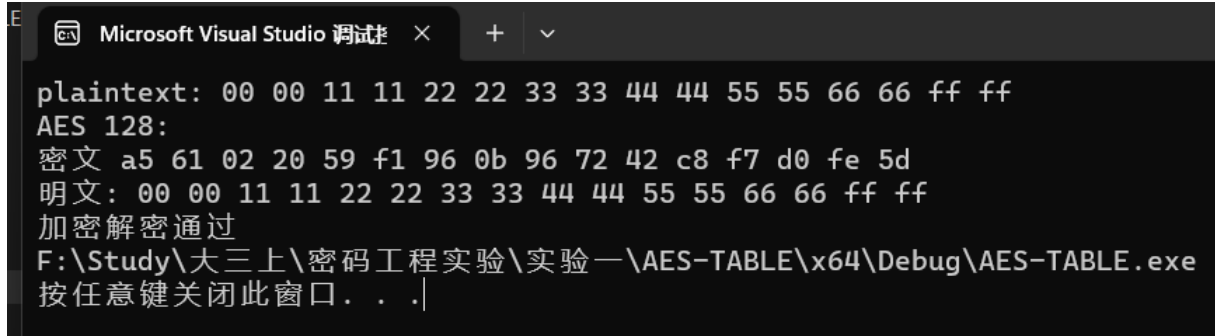
```

32 }
33 t[0] = SboxIV[(s[0] >> 24) & 0xFF] << 24; //行移位/字节变换
34 t[0] |= SboxIV[(s[3] >> 16) & 0xFF] << 16;
35 t[0] |= SboxIV[(s[2] >> 8) & 0xFF] << 8;
36 t[0] |= SboxIV[(s[1] >> 0) & 0xFF] << 0;
37 t[1] = SboxIV[(s[1] >> 24) & 0xFF] << 24;
38 t[1] |= SboxIV[(s[0] >> 16) & 0xFF] << 16;
39 t[1] |= SboxIV[(s[3] >> 8) & 0xFF] << 8;
40 t[1] |= SboxIV[(s[2] >> 0) & 0xFF] << 0;
41 t[2] = SboxIV[(s[2] >> 24) & 0xFF] << 24;
42 t[2] |= SboxIV[(s[1] >> 16) & 0xFF] << 16;
43 t[2] |= SboxIV[(s[0] >> 8) & 0xFF] << 8;
44 t[2] |= SboxIV[(s[3] >> 0) & 0xFF] << 0;
45 t[3] = SboxIV[(s[3] >> 24) & 0xFF] << 24;
46 t[3] |= SboxIV[(s[2] >> 16) & 0xFF] << 16;
47 t[3] |= SboxIV[(s[1] >> 8) & 0xFF] << 8;
48 t[3] |= SboxIV[(s[0] >> 0) & 0xFF] << 0;
49 s[0] = t[0] ^ Key.dk[0];
50 s[1] = t[1] ^ Key.dk[1];
51 s[2] = t[2] ^ Key.dk[2];
52 s[3] = t[3] ^ Key.dk[3];
53 }

```

5 结果分析

5.1 加密程序运行结果

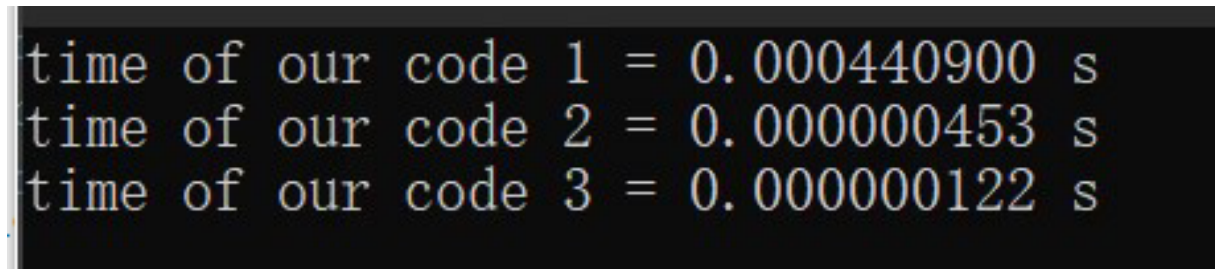


```
Microsoft Visual Studio 调试
plaintext: 00 00 11 11 22 22 33 33 44 44 55 55 66 66 ff ff
AES 128:
密文 a5 61 02 20 59 f1 96 0b 96 72 42 c8 f7 d0 fe 5d
明文: 00 00 11 11 22 22 33 33 44 44 55 55 66 66 ff ff
加密解密通过
F:\Study\大三上\密码工程实验\实验一\AES-TABLE\x64\Debug\AES-TABLE.exe
按任意键关闭此窗口. . .|
```

Figure 1: AES128 算法运行结果验证

5.2 对查找表优化的性能分析

比较 AES 的原始实现, AES 的查找表实现, AES 在 Openssl 中的实现, 三种实现的耗时, 得到一下结果:



```
time of our code 1 = 0.000440900 s
time of our code 2 = 0.000000453 s
time of our code 3 = 0.000000122 s
```

Figure 2: 运行结果, 由上到下依次为原始实现, 查找表实现, Openssl 实现

通过数据比较可以发现, 经过查找表的优化后, AES 计算性能相较原始实现有了很大的提升 (数个数量级), 这是因为将混淆和替换的操作合并到同一个查找表中, 对于状态矩阵中的每个字节, 只需要进行一次查找操作, 可以避免重复访问内存, 从而提高算法的性能。

但可以发现本实验实现的 AES 的耗时仍略高于借助 Openssl 算法库实现的 AES-128 加密算法。通过阅读 Openssl 代码, 发现其不仅采取了算法上的优化, 还针对不同的 CPU 架构进行了优化, 比如优化了 AES 软件实现中除查找表外存在的其他瓶颈, 如编译层面存在其他的内存访问、控制结构、缓存未命中等问题 (通过汇编内联和 SIMD 指令集等), 故其运行效率比本实验所实现的 AES 效率更高。

综上所述，要想获得最优的 AES 软件实现性能，需要综合考虑多个因素，不仅要考虑算法实现细节，还需要考虑 CPU 架构、内存访问模式、缓存优化等等。

参考文献

- [1] [加]Douglas R.Stinson. 密码学原理与实践 [M]. 冯登国. 北京: 电子工业出版社, 2003.
- [2] Openssl 官方文档. <https://www.openssl.org/docs/man1.1.1/man7/>
- [3] AES 对称密码算法介绍.<https://zhuanlan.zhihu.com/p/42264499>

A 附录

A.1 aes.cpp 源代码

```
1 #include "aes.h"
2 #include <stdlib.h>
3 static const uint8_t Sbox[256] = {
4     0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B,
5     0xFE, 0xD7, 0xAB, 0x76, 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
6     0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26,
7     0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
8     0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2,
9     0xEB, 0x27, 0xB2, 0x75, 0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
10    0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84, 0x53, 0xD1, 0x00, 0xED,
11    0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
12    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F,
13    0x50, 0x3C, 0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
14    0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, 0xCD, 0x0C, 0x13, 0xEC,
15    0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
16    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14,
17    0xDE, 0x5E, 0x0B, 0xDB, 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
18    0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D,
19    0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
20    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F,
21    0x4B, 0xBD, 0x8B, 0x8A, 0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
22    0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98, 0x11,
23    0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
24    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F,
25    0xB0, 0x54, 0xBB, 0x16 };
26 static const uint8_t SboxIV[256] = {
27     0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e,
28     0x81, 0xf3, 0xd7, 0xfb, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87,
29     0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb, 0x54, 0x7b, 0x94, 0x32,
30     0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
31     0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49,
32     0x6d, 0x8b, 0xd1, 0x25, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16,
33     0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92, 0x6c, 0x70, 0x48, 0x50,
34     0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
35     0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05,
```

```

36 0xb8, 0xb3, 0x45, 0x06, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02,
37 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b, 0x3a, 0x91, 0x11, 0x41,
38 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
39 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8,
40 0x1c, 0x75, 0xdf, 0x6e, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89,
41 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b, 0xfc, 0x56, 0x3e, 0x4b,
42 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
43 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59,
44 0x27, 0x80, 0xec, 0x5f, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d,
45 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef, 0xa0, 0xe0, 0x3b, 0x4d,
46 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
47 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63,
48 0x55, 0x21, 0x0c, 0x7d };
49 static const uint32_t TE[256] = {
50 0xc66363a5, 0xf87c7c84, 0xee777799, 0xf67b7b8d, 0xfff2f20d, 0xd66b6bbd,
51 0xde6f6fb1, 0x91c5c554, 0x60303050, 0x02010103, 0xce6767a9, 0x562b2b7d,
52 0xe7fefe19, 0xb5d7d762, 0x4dababe6, 0xec76769a, 0x8fcaca45, 0x1f82829d,
53 0x89c9c940, 0xfa7d7d87, 0xeffafa15, 0xb25959eb, 0xe47474c9, 0xfbff0f00b,
54 0x41adadec, 0xb3d4d467, 0x5fa2a2fd, 0x45afafea, 0x239c9cbf, 0x53a4a4f7,
55 0xe4727296, 0x9bc0c05b, 0x75b7b7c2, 0xe1fdfd1c, 0x3d9393ae, 0x4c26266a,
56 0x6c36365a, 0x7e3f3f41, 0xf5f7f702, 0x83cccc4f, 0x6834345c, 0x51a5a5f4,
57 0xd1e5e534, 0xf9f1f108, 0xe2717193, 0xabd8d873, 0x62313153, 0x2a15153f,
58 0x0804040c, 0x95c7c752, 0x46232365, 0x9dc3c35e, 0x30181828, 0x379696a1,
59 0x0a05050f, 0x2f9a9ab5, 0x0e070709, 0x24121236, 0x1b80809b, 0xdfe2e23d,
60 0xcdebeb26, 0x4e272769, 0x7fb2b2cd, 0xea75759f, 0x1209091b, 0xd83838e,
61 0x582c2c74, 0x341a1a2e, 0x361b1b2d, 0xdc6e6eb2, 0xb45a5aee, 0x5ba0a0fb,
62 0xa45252f6, 0x763b3b4d, 0xb7d6d661, 0x7db3b3ce, 0x5229297b, 0xdde3e33e,
63 0x5e2f2f71, 0x13848497, 0xa65353f5, 0xb9d1d168, 0x00000000, 0xc1eded2c,
64 0x40202060, 0xe3fcfc1f, 0x79b1b1c8, 0xb65b5bed, 0xd46a6abe, 0x8dcbcb46,
65 0x67bebed9, 0x7239394b, 0x944a4ade, 0x984c4cd4, 0xb05858e8, 0x85cfcf4a,
66 0xbbd0d06b, 0xc5efef2a, 0x4faaaae5, 0xedfbfb16, 0x864343c5, 0x9a4d4dd7,
67 0x66333355, 0x11858594, 0x8a4545cf, 0xe9f9f910, 0x04020206, 0xfe7f7f81,
68 0xa05050f0, 0x783c3c44, 0x259f9fba, 0x4ba8a8e3, 0xa25151f3, 0x5da3a3fe,
69 0x804040c0, 0x058f8f8a, 0x3f9292ad, 0x219d9dbc, 0x70383848, 0xf1f5f504,
70 0x63bcbcdf, 0x77b6b6c1, 0xafdada75, 0x42212163, 0x20101030, 0xe5ffff1a,
71 0xfdf3f30e, 0xbfd2d26d, 0x81cdcd4c, 0x180c0c14, 0x26131335, 0xc3ecec2f,
72 0xbe5f5fe1, 0x359797a2, 0x884444cc, 0x2e171739, 0x93c4c457, 0x55a7a7f2,
73 0xfc7e7e82, 0x7a3d3d47, 0xc86464ac, 0xba5d5de7, 0x3219192b, 0xe6737395,

```

```

74 0xc06060a0, 0x19818198, 0x9e4f4fd1, 0xa3dcdc7f, 0x44222266, 0x542a2a7e,
75 0x3b9090ab, 0x0b888883, 0x8c4646ca, 0xc7eeee29, 0x6bb8b8d3, 0x2814143c,
76 0xa7dede79, 0xbc5e5ee2, 0x160b0b1d, 0xaddbdb76, 0xdb0e03b, 0x64323256,
77 0x743a3a4e, 0x140a0a1e, 0x924949db, 0xc06060a, 0x4824246c, 0xb85c5ce4,
78 0x9fc2c25d, 0xbdd3d36e, 0x43acacef, 0xc46262a6, 0x399191a8, 0x319595a4,
79 0xd3e4e437, 0xf279798b, 0xd5e7e732, 0x8bc8c843, 0x6e373759, 0xda6d6db7,
80 0x018d8d8c, 0xb1d5d564, 0x9c4e4ed2, 0x49a9a9e0, 0xd86c6cb4, 0xac5656fa,
81 0xf3f4f407, 0xcfeaea25, 0xca6565af, 0xf47a7a8e, 0x47aeae9, 0x10080818,
82 0x6fbabad5, 0xf0787888, 0x4a25256f, 0x5c2e2e72, 0x381c1c24, 0x57a6a6f1,
83 0x73b4b4c7, 0x97c6c651, 0xcbe8e823, 0xa1dddd7c, 0xe874749c, 0x3e1f1f21,
84 0x964b4bdd, 0x61bdbddc, 0xd8b8b86, 0x0f8a8a85, 0xe0707090, 0x7c3e3e42,
85 0x71b5b5c4, 0xcc6666aa, 0x904848d8, 0x06030305, 0xf7f6f601, 0x1c0e0e12,
86 0xc26161a3, 0x6a35355f, 0xae5757f9, 0x69b9b9d0, 0x17868691, 0x99c1c158,
87 0x3a1d1d27, 0x279e9eb9, 0xd9e1e138, 0xebf8f813, 0x2b9898b3, 0x22111133,
88 0xd26969bb, 0xa9d9d970, 0x078e8e89, 0x339494a7, 0x2d9b9bb6, 0x3c1e1e22,
89 0x15878792, 0xc9e9e920, 0x87cece49, 0xaa5555ff, 0x50282878, 0xa5dfdf7a,
90 0x038c8c8f, 0x59a1a1f8, 0x09898980, 0x1a0d0d17, 0x65bfbfda, 0xd7e6e631,
91 0x844242c6, 0xd06868b8, 0x824141c3, 0x299999b0, 0x5a2d2d77, 0x1e0f0f11,
92 0x7bb0b0cb, 0xa85454fc, 0x6dbbbbdb6, 0x2c16163a };

```

```

93 static const uint32_t TD[256] = {
94 0x51f4a750, 0x7e416553, 0x1a17a4c3, 0x3a275e96, 0x3bab6bcb, 0x1f9d45f1,
95 0xacfa58ab, 0x4be30393, 0x2030fa55, 0xad766df6, 0x88cc7691, 0xf5024c25,
96 0x4fe5d7fc, 0xc52acbd7, 0x26354480, 0xb562a38f, 0xdeb15a49, 0x25ba1b67,
97 0x45ea0e98, 0x5dfec0e1, 0xc32f7502, 0x814cf012, 0x8d4697a3, 0x6bd3f9c6,
98 0x038f5fe7, 0x15929c95, 0xbf6d7aeb, 0x955259da, 0xd4be832d, 0x587421d3,
99 0x49e06929, 0x8ec9c844, 0x75c2896a, 0xf48e7978, 0x99583e6b, 0x27b971dd,
100 0xbee14fb6, 0xf088ad17, 0xc920ac66, 0x7dce3ab4, 0x63df4a18, 0xe51a3182,
101 0x97513360, 0x62537f45, 0xb16477e0, 0xbb6bae84, 0xfe81a01c, 0xf9082b94,
102 0x70486858, 0x8f45fd19, 0x94de6c87, 0x527bf8b7, 0xab73d323, 0x724b02e2,
103 0xe31f8f57, 0x6655ab2a, 0xb2eb2807, 0x2fb5c203, 0x86c57b9a, 0xd33708a5,
104 0x302887f2, 0x23bfa5b2, 0x02036aba, 0xed16825c, 0x8acf1c2b, 0xa779b492,
105 0xf307f2f0, 0x4e69e2a1, 0x65daf4cd, 0x0605bed5, 0xd134621f, 0xc4a6fe8a,
106 0x342e539d, 0xa2f355a0, 0x058ae132, 0xa4f6eb75, 0x0b83ec39, 0x4060efaa,
107 0x5e719f06, 0xbd6e1051, 0x3e218af9, 0x96dd063d, 0xdd3e05ae, 0x4de6bd46,
108 0x91548db5, 0x71c45d05, 0x0406d46f, 0x605015ff, 0x1998fb24, 0xd6bde997,
109 0x894043cc, 0x67d99e77, 0xb0e842bd, 0x07898b88, 0xe7195b38, 0x79c8eedb,
110 0xa17c0a47, 0x7c420fe9, 0xf8841ec9, 0x00000000, 0x09808683, 0x322bed48,
111 0x1e1170ac, 0x6c5a724e, 0xfd0efffb, 0xf853856, 0x3daed51e, 0x362d3927,

```

```

112 0x0a0fd964, 0x685ca621, 0x9b5b54d1, 0x24362e3a, 0x0c0a67b1, 0x9357e70f,
113 0xb4ee96d2, 0x1b9b919e, 0x80c0c54f, 0x61dc20a2, 0x5a774b69, 0x1c121a16,
114 0xe293ba0a, 0xc0a02ae5, 0x3c22e043, 0x121b171d, 0x0e09d0b, 0xf28bc7ad,
115 0x2db6a8b9, 0x141ea9c8, 0x57f11985, 0xaf75074c, 0xee99ddbb, 0xa37f60fd,
116 0xf701269f, 0x5c72f5bc, 0x44663bc5, 0x5bfb7e34, 0x8b432976, 0xcb23c6dc,
117 0xb6edfc68, 0xb8e4f163, 0xd731dcca, 0x42638510, 0x13972240, 0x84c61120,
118 0x854a247d, 0xd2bb3df8, 0xae93211, 0xc729a16d, 0x1d9e2f4b, 0xdc230f3,
119 0x0d8652ec, 0x77c1e3d0, 0x2bb3166c, 0xa970b999, 0x119448fa, 0x47e96422,
120 0xa8fc8cc4, 0xa0f03f1a, 0x567d2cd8, 0x223390ef, 0x87494ec7, 0xd938d1c1,
121 0x8ccaa2fe, 0x98d40b36, 0xa6f581cf, 0xa57ade28, 0xdab78e26, 0x3fadbfa4,
122 0x2c3a9de4, 0x5078920d, 0x6a5fcc9b, 0x547e4662, 0xf68d13c2, 0x90d8b8e8,
123 0x2e39f75e, 0x82c3aff5, 0x9f5d80be, 0x69d0937c, 0x6fd52da9, 0xcf2512b3,
124 0xc8ac993b, 0x10187da7, 0xe89c636e, 0xdb3bbb7b, 0xcd267809, 0x6e5918f4,
125 0xec9ab701, 0x834f9aa8, 0xe6956e65, 0xaaffe67e, 0x21bccf08, 0xef15e8e6,
126 0xbae79bd9, 0x4a6f36ce, 0xea9f09d4, 0x29b07cd6, 0x31a4b2af, 0x2a3f2331,
127 0xc6a59430, 0x35a266c0, 0x744ebc37, 0xfc82caa6, 0xe09d0b0, 0x33a7d815,
128 0xf104984a, 0x41ecdaf7, 0x7fcd500e, 0x1791f62f, 0x764dd68d, 0x43efb04d,
129 0xccaa4d54, 0xe49604df, 0x9ed1b5e3, 0x4c6a881b, 0xc12c1fb8, 0x4665517f,
130 0x9d5eea04, 0x018c355d, 0xfa877473, 0xfb0b412e, 0xb3671d5a, 0x92dbd252,
131 0xe9105633, 0x6dd64713, 0x9ad7618c, 0x37a10c7a, 0x59f8148e, 0xeb133c89,
132 0xcea927ee, 0xb761c935, 0xe11ce5ed, 0x7a47b13c, 0x9cd2df59, 0x55f2733f,
133 0x1814ce79, 0x73c737bf, 0x53f7cdea, 0x5ffdaa5b, 0xdf3d6f14, 0x7844db86,
134 0xcaaff381, 0xb968c43e, 0x3824342c, 0xc2a3405f, 0x161dc372, 0xbce2250c,
135 0x283c498b, 0xff0d9541, 0x39a80171, 0x080cb3de, 0xd8b4e49c, 0x6456c190,
136 0x7bcb8461, 0xd532b670, 0x486c5c74, 0xd0b85742 };
137 #define rotr32(value, shift) ((value >> shift) ^ (value << (32 - shift)))
138 int AES_KeyInit(uint8_t* key, AES_Key* Key, size_t bits) {
139     uint32_t Rcon[10] = { 0x01, 0x02, 0x04, 0x08, 0x10,
140                           0x20, 0x40, 0x80, 0x1B, 0x36 };
141     uint32_t nr = 10 + (bits - 128) / 32;
142     uint32_t nk = bits / 32;
143     uint32_t tmp, tmp1;
144     Key->nr = nr;
145     uint32_t* w = (uint32_t*)malloc(sizeof(uint32_t) * 4 * (nr + 1));
146     if (w == (void*)0) {
147         return 0;
148     }
149     uint32_t* d = (uint32_t*)malloc(sizeof(uint32_t) * 4 * (nr + 1));

```

```

150 if (d == (void*)0) {
151     free(w);
152     return 0;
153 }
154 for (int i = 0; i < nk; i++) {
155     w[i] = (key[4 * i + 0] << 24) | (key[4 * i + 1] << 16) |
156         (key[4 * i + 2] << 8) | (key[4 * i + 3]);
157 }
158 for (int i = nk; i < 4 * (nr + 1); i++) {
159     tmp = w[i - 1];
160     if (i % nk == 0) {
161         tmp1 = tmp;
162         tmp = Sbox[(tmp1 >> 24) & 0xFF];
163         tmp |= Sbox[(tmp1 >> 0) & 0xFF] << 8;
164         tmp |= Sbox[(tmp1 >> 8) & 0xFF] << 16;
165         tmp |= (Sbox[(tmp1 >> 16) & 0xFF] ^ Rcon[i / nk - 1]) << 24;
166     }
167     else if (nk > 6 && i % nk == 4) {
168         tmp1 = tmp;
169         tmp = Sbox[(tmp1 >> 0) & 0xFF];
170         tmp |= Sbox[(tmp1 >> 8) & 0xFF] << 8;
171         tmp |= Sbox[(tmp1 >> 16) & 0xFF] << 16;
172         tmp |= Sbox[(tmp1 >> 24) & 0xFF] << 24;
173     }
174     w[i] = w[i - nk] ^ tmp;
175 }
176 Key->ek = w;
177 for (int i = 0; i < 4; i++) d[i] = w[i];
178 for (int i = 4; i < 4 * nr; i++) {
179
180     d[i] = TD[Sbox[(w[i] >> 24) & 0xFF]];
181     tmp = TD[Sbox[(w[i] >> 16) & 0xFF]];
182     d[i] ^= rotr32(tmp, 8);
183     tmp = TD[Sbox[(w[i] >> 8) & 0xFF]];
184     d[i] ^= rotr32(tmp, 16);
185     tmp = TD[Sbox[(w[i] >> 0) & 0xFF]];
186     d[i] ^= rotr32(tmp, 24);
187 }

```

```

188     for (int i = 0; i < 4; i++) {
189         d[4 * nr + i] = w[4 * nr + i];
190     }
191     Key->dk = d;
192
193     return 1;
194 }
195
196 void AES_Encrypt(uint8_t* plaintext, uint8_t* ciphertext, AES_Key Key) {
197     uint32_t s[4];
198     uint32_t t[4];
199     uint32_t tmp;
200     for (int i = 0; i < 4; i++) {
201         s[i] = (plaintext[4 * i + 0] << 24) | (plaintext[4 * i + 1] << 16) |
202             (plaintext[4 * i + 2] << 8) | (plaintext[4 * i + 3]);
203     }
204     s[0] ^= Key.ek[0];
205     s[1] ^= Key.ek[1];
206     s[2] ^= Key.ek[2];
207     s[3] ^= Key.ek[3];
208     for (int i = 1; i < Key.nr; i++) {
209         t[0] = TE[(s[0] >> 24) & 0xFF];
210         tmp = TE[(s[1] >> 16) & 0xFF];
211         t[0] ^= rotr32(tmp, 8);
212         tmp = TE[(s[2] >> 8) & 0xFF];
213         t[0] ^= rotr32(tmp, 16);
214         tmp = TE[(s[3] >> 0) & 0xFF];
215         t[0] ^= rotr32(tmp, 24);
216         t[1] = TE[(s[1] >> 24) & 0xFF];
217         tmp = TE[(s[2] >> 16) & 0xFF];
218         t[1] ^= rotr32(tmp, 8);
219         tmp = TE[(s[3] >> 8) & 0xFF];
220         t[1] ^= rotr32(tmp, 16);
221         tmp = TE[(s[0] >> 0) & 0xFF];
222         t[1] ^= rotr32(tmp, 24);
223         t[2] = TE[(s[2] >> 24) & 0xFF];
224         tmp = TE[(s[3] >> 16) & 0xFF];
225         t[2] ^= rotr32(tmp, 8);

```

```

226     tmp = TE[(s[0] >> 8) & 0xFF];
227     t[2] ^= rotr32(tmp, 16);
228     tmp = TE[(s[1] >> 0) & 0xFF];
229     t[2] ^= rotr32(tmp, 24);
230     t[3] = TE[(s[3] >> 24) & 0xFF];
231     tmp = TE[(s[0] >> 16) & 0xFF];
232     t[3] ^= rotr32(tmp, 8);
233     tmp = TE[(s[1] >> 8) & 0xFF];
234     t[3] ^= rotr32(tmp, 16);
235     tmp = TE[(s[2] >> 0) & 0xFF];
236     t[3] ^= rotr32(tmp, 24);
237     s[0] = t[0] ^ Key.ek[4 * i + 0];
238     s[1] = t[1] ^ Key.ek[4 * i + 1];
239     s[2] = t[2] ^ Key.ek[4 * i + 2];
240     s[3] = t[3] ^ Key.ek[4 * i + 3];
241 }
242 t[0] = Sbox[(s[0] >> 24) & 0xFF] << 24;
243 t[0] |= Sbox[(s[1] >> 16) & 0xFF] << 16;
244 t[0] |= Sbox[(s[2] >> 8) & 0xFF] << 8;
245 t[0] |= Sbox[(s[3] >> 0) & 0xFF] << 0;
246 t[1] = Sbox[(s[1] >> 24) & 0xFF] << 24;
247 t[1] |= Sbox[(s[2] >> 16) & 0xFF] << 16;
248 t[1] |= Sbox[(s[3] >> 8) & 0xFF] << 8;
249 t[1] |= Sbox[(s[0] >> 0) & 0xFF] << 0;
250 t[2] = Sbox[(s[2] >> 24) & 0xFF] << 24;
251 t[2] |= Sbox[(s[3] >> 16) & 0xFF] << 16;
252 t[2] |= Sbox[(s[0] >> 8) & 0xFF] << 8;
253 t[2] |= Sbox[(s[1] >> 0) & 0xFF] << 0;
254 t[3] = Sbox[(s[3] >> 24) & 0xFF] << 24;
255 t[3] |= Sbox[(s[0] >> 16) & 0xFF] << 16;
256 t[3] |= Sbox[(s[1] >> 8) & 0xFF] << 8;
257 t[3] |= Sbox[(s[2] >> 0) & 0xFF] << 0;
258 s[0] = t[0] ^ Key.ek[4 * Key.nr + 0];
259 s[1] = t[1] ^ Key.ek[4 * Key.nr + 1];
260 s[2] = t[2] ^ Key.ek[4 * Key.nr + 2];
261 s[3] = t[3] ^ Key.ek[4 * Key.nr + 3];
262 for (int i = 0; i < 4; i++) {
263     ciphertext[4 * i + 0] = (s[i] >> 24) & 0xFF;

```

```

264     ciphertext[4 * i + 1] = (s[i] >> 16) & 0xFF;
265     ciphertext[4 * i + 2] = (s[i] >> 8) & 0xFF;
266     ciphertext[4 * i + 3] = (s[i] >> 0) & 0xFF;
267 }
268 }
269 void AES_KeyDelete(AES_Key Key) {
270     free(Key.ek);
271     free(Key.dk);
272 }
273 void AES_Decrypt(uint8_t* ciphertext, uint8_t* plaintext, AES_Key Key) {
274     uint32_t s[4];
275     uint32_t t[4];
276     uint32_t tmp;
277     for (int i = 0; i < 4; i++) {
278         s[i] = (ciphertext[4 * i + 0] << 24) | (ciphertext[4 * i + 1] << 16) |
279             (ciphertext[4 * i + 2] << 8) | (ciphertext[4 * i + 3]);
280     }
281     s[0] ^= Key.dk[4 * Key.nr + 0];
282     s[1] ^= Key.dk[4 * Key.nr + 1];
283     s[2] ^= Key.dk[4 * Key.nr + 2];
284     s[3] ^= Key.dk[4 * Key.nr + 3];
285     for (int i = Key.nr - 1; i > 0; i--) {
286         t[0] = TD[(s[0] >> 24) & 0xFF];
287         tmp = TD[(s[3] >> 16) & 0xFF];
288         t[0] ^= rotr32(tmp, 8);
289         tmp = TD[(s[2] >> 8) & 0xFF];
290         t[0] ^= rotr32(tmp, 16);
291         tmp = TD[(s[1] >> 0) & 0xFF];
292         t[0] ^= rotr32(tmp, 24);
293         t[1] = TD[(s[1] >> 24) & 0xFF];
294         tmp = TD[(s[0] >> 16) & 0xFF];
295         t[1] ^= rotr32(tmp, 8);
296         tmp = TD[(s[3] >> 8) & 0xFF];
297         t[1] ^= rotr32(tmp, 16);
298         tmp = TD[(s[2] >> 0) & 0xFF];
299         t[1] ^= rotr32(tmp, 24);
300         t[2] = TD[(s[2] >> 24) & 0xFF];
301         tmp = TD[(s[1] >> 16) & 0xFF];

```

```

302     t[2] ^= rotr32(tmp, 8);
303     tmp = TD[(s[0] >> 8) & 0xFF];
304     t[2] ^= rotr32(tmp, 16);
305     tmp = TD[(s[3] >> 0) & 0xFF];
306     t[2] ^= rotr32(tmp, 24);
307     t[3] = TD[(s[3] >> 24) & 0xFF];
308     tmp = TD[(s[2] >> 16) & 0xFF];
309     t[3] ^= rotr32(tmp, 8);
310     tmp = TD[(s[1] >> 8) & 0xFF];
311     t[3] ^= rotr32(tmp, 16);
312     tmp = TD[(s[0] >> 0) & 0xFF];
313     t[3] ^= rotr32(tmp, 24);
314     s[0] = t[0] ^ Key.dk[4 * i + 0];
315     s[1] = t[1] ^ Key.dk[4 * i + 1];
316     s[2] = t[2] ^ Key.dk[4 * i + 2];
317     s[3] = t[3] ^ Key.dk[4 * i + 3];
318 }
319 t[0] = SboxIV[(s[0] >> 24) & 0xFF] << 24;
320 t[0] |= SboxIV[(s[3] >> 16) & 0xFF] << 16;
321 t[0] |= SboxIV[(s[2] >> 8) & 0xFF] << 8;
322 t[0] |= SboxIV[(s[1] >> 0) & 0xFF] << 0;
323 t[1] = SboxIV[(s[1] >> 24) & 0xFF] << 24;
324 t[1] |= SboxIV[(s[0] >> 16) & 0xFF] << 16;
325 t[1] |= SboxIV[(s[3] >> 8) & 0xFF] << 8;
326 t[1] |= SboxIV[(s[2] >> 0) & 0xFF] << 0;
327 t[2] = SboxIV[(s[2] >> 24) & 0xFF] << 24;
328 t[2] |= SboxIV[(s[1] >> 16) & 0xFF] << 16;
329 t[2] |= SboxIV[(s[0] >> 8) & 0xFF] << 8;
330 t[2] |= SboxIV[(s[3] >> 0) & 0xFF] << 0;
331 t[3] = SboxIV[(s[3] >> 24) & 0xFF] << 24;
332 t[3] |= SboxIV[(s[2] >> 16) & 0xFF] << 16;
333 t[3] |= SboxIV[(s[1] >> 8) & 0xFF] << 8;
334 t[3] |= SboxIV[(s[0] >> 0) & 0xFF] << 0;
335 s[0] = t[0] ^ Key.dk[0];
336 s[1] = t[1] ^ Key.dk[1];
337 s[2] = t[2] ^ Key.dk[2];
338 s[3] = t[3] ^ Key.dk[3];
339 for (int i = 0; i < 4; i++) {

```

```
340     plaintext[4 * i + 0] = (s[i] >> 24) & 0xFF;
341     plaintext[4 * i + 1] = (s[i] >> 16) & 0xFF;
342     plaintext[4 * i + 2] = (s[i] >> 8) & 0xFF;
343     plaintext[4 * i + 3] = (s[i] >> 0) & 0xFF;
344 }
345 }
```