



# Project1:SM4 的软件实现和优化

学院: 网络空间安全学院

专业: 密码科学与技术

姓名: 李双平

学号: 202200180026

2025 年 8 月 15 日

## 目录

<b>1 SM4 的基本实现</b>	<b>2</b>
1.1 实现原理	2
1.2 代码解释	2
1.3 实验结果	5
<b>2 SM4 的 T-table 优化</b>	<b>5</b>
2.1 优化原理	5
2.2 代码解释	6
2.3 实验结果	7
<b>3 SM4 的 AESNI 优化</b>	<b>7</b>
3.1 优化原理	7
3.2 代码解释	8
3.3 实验结果	11
<b>4 SM4-GCM 基本实现</b>	<b>12</b>
4.1 实现原理	12
4.2 代码解释	12
4.3 实验结果	16

# 1 SM4 的基本实现

## 1.1 实现原理

SM4 是我国自主设计的分组对称加密算法，分组长度为 128 位（16 字节），密钥长度同样为 128 位。算法整体采用 \*\*Feistel 网络结构 \*\*，共进行 32 轮迭代加密。其安全性主要依赖于非线性 S 盒变换与线性扩散变换的结合。

SM4 的主要工作过程包括：

1. **分组与初始设置**: 明文和密钥均被分为 4 个 32 位字(记为  $X_0, X_1, X_2, X_3$  以及  $MK_0, MK_1, MK_2, MK_3$ )。
2. **密钥扩展 (Key Expansion)**: 通过固定参数  $FK$  与常量  $CK$ ，结合 S 盒变换与线性变换  $L'$ ，生成 32 个轮密钥  $rk_0, rk_1, \dots, rk_{31}$ 。
3. **轮函数  $F$** : 每一轮使用公式：

$$X_{i+4} = X_i \oplus T(X_{i+1} \oplus X_{i+2} \oplus X_{i+3} \oplus rk_i)$$

其中  $T$  是复合变换  $T = L \circ \tau$ :

- $\tau$ : 字节替换，即 S 盒非线性变换。
- $L$ : 线性变换，利用循环左移与异或实现扩散。

4. **加密与解密**: 加密与解密过程相同，仅轮密钥使用顺序相反。

SM4 的安全性依赖于：

- S 盒的非线性特性，可抵抗差分和线性密码分析；
- 线性变换的高扩散性，能快速混合明文比特；
- 32 轮迭代保证了足够的安全裕度。

## 1.2 代码解释

本实验代码主要分为以下几个模块：

### 1. 循环左移函数

```
1 u32 ROTL(u32 x, int n) {
2     return (x << n) | (x >> (32 - n));
3 }
```

该函数实现 32 位无符号整数的循环左移 (Rotate Left) 操作。逻辑：将  $x$  左移  $n$  位，同时将高位溢出的部分右移补到低位，实现循环移位效果。在 SM4 中，循环移位用于线性变换步骤，增强比特间扩散性。

### 2. S 盒替代函数

```

1  u32 Tau(u32 A) {
2      u8 a[4];
3      a[0] = Sbox[(A >> 24) & 0xFF];
4      a[1] = Sbox[(A >> 16) & 0xFF];
5      a[2] = Sbox[(A >> 8) & 0xFF];
6      a[3] = Sbox[A & 0xFF];
7      return (u32(a[0]) << 24) | (u32(a[1]) << 16) |
8          (u32(a[2]) << 8) | u32(a[3]);
9  }
```

该函数实现 **非线性变换  $\tau$** ，即 S 盒替代。步骤：将 32 位输入按字节拆成 4 个字节，分别查 S 盒替换，然后重新组合成 32 位整数。作用：通过 S 盒引入非线性，使得加密过程难以用线性代数攻击。

### 3. 加密轮函数 $T$

```

1  u32 T(u32 x) {
2      u32 B = Tau(x);
3      return B ^ ROTL(B, 2) ^ ROTL(B, 10) ^
4          ROTL(B, 18) ^ ROTL(B, 24);
5  }
```

实现 SM4 的 **T 变换**：先经过 S 盒替代，再经过线性变换  $L$ （不同位移异或组合）。功能：用于加密/解密过程中的每轮运算，实现混淆与扩散的结合。

### 4. 密钥扩展轮函数 $T'$

```

1  u32 T_prime(u32 x) {
2      u32 B = Tau(x);
3      return B ^ ROTL(B, 13) ^ ROTL(B, 23);
4  }
```

这是密钥扩展中的  **$T'$  变换**，与  $T$  类似，但线性变换位移量不同（13 和 23）。用于生成轮密钥时，提供不同的扩散特性。

### 5. 密钥扩展函数

```

1  void KeyExpansion(const u8 MK[16], u32 rk[32]) {
2      u32 K[36];
3      for (int i = 0; i < 4; i++) {
4          K[i] = ((u32)MK[4 * i] << 24) |
5                  ((u32)MK[4 * i + 1] << 16) |
6                  ((u32)MK[4 * i + 2] << 8) |
7                  ((u32)MK[4 * i + 3]);
8          K[i] ^= FK[i];
```

```

9      }
10     for (int i = 0; i < 32; i++) {
11         K[i + 4] = K[i] ^
12             T_prime(K[i + 1] ^ K[i + 2] ^ K[i + 3] ^ CK[i]);
13         rk[i] = K[i + 4];
14     }
15 }
```

此函数将 128 位原始密钥  $MK$  生成 32 个轮密钥  $rk[0..31]$ :

- 首先将密钥分成四个 32 位字，与系统参数  $FK$  异或；
- 每轮使用  $T'$  变换和  $CK[i]$  生成新的  $K$  值；
- 新生成的  $K[i + 4]$  作为该轮的轮密钥。

## 6. 加密单块数据

```

1 void SM4_EncryptBlock(const u8 input[16],
2                         u8 output[16],
3                         const u32 rk[32]) {
4     u32 X[36];
5     for (int i = 0; i < 4; i++) {
6         X[i] = ((u32)input[4 * i] << 24) |
7                 ((u32)input[4 * i + 1] << 16) |
8                 ((u32)input[4 * i + 2] << 8) |
9                 ((u32)input[4 * i + 3]);
10    }
11    for (int i = 0; i < 32; i++) {
12        X[i + 4] = X[i] ^
13            T(X[i + 1] ^ X[i + 2] ^ X[i + 3] ^ rk[i]);
14    }
15    for (int i = 0; i < 4; i++) {
16        u32 x = X[35 - i];
17        output[4 * i] = (x >> 24) & 0xFF;
18        output[4 * i + 1] = (x >> 16) & 0xFF;
19        output[4 * i + 2] = (x >> 8) & 0xFF;
20        output[4 * i + 3] = x & 0xFF;
21    }
22 }
```

该函数对 128 位数据块执行 32 轮加密：

- 输入数据拆成四个 32 位字；
- 每轮调用  $T$  函数结合轮密钥生成新字；
- 最终输出时按 SM4 要求反序排列。

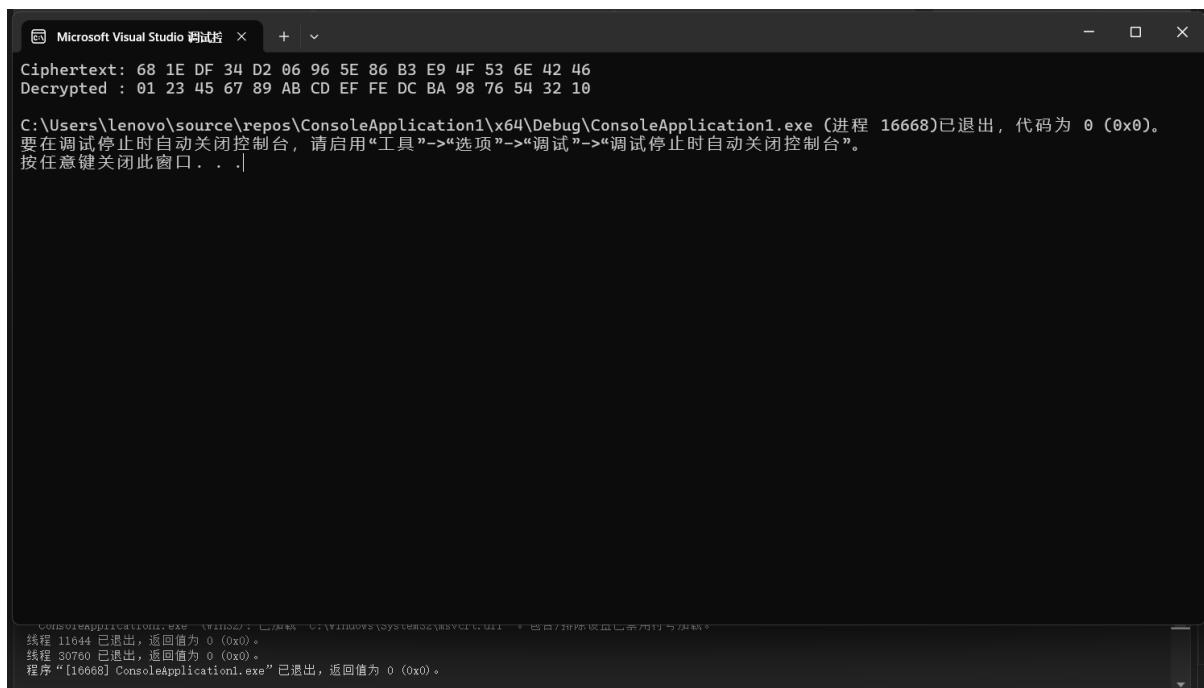
## 7. 解密单块数据

```
1 void SM4_DecryptBlock(const u8 input[16],  
2                         u8 output[16],  
3                         const u32 rk[32]) {  
4     u32 rk_inv[32];  
5     for (int i = 0; i < 32; i++) {  
6         rk_inv[i] = rk[31 - i];  
7     }  
8     SM4_EncryptBlock(input, output, rk_inv);  
9 }
```

SM4 的解密与加密过程相同，只需将轮密钥顺序反转。该函数先生成逆序轮密钥  $rk\_inv$ ，再调用加密函数完成解密。

## 1.3 实验结果

实验结果如下：



The screenshot shows the Microsoft Visual Studio Debug Output window. It displays two sets of 16 bytes each, representing hex values. The first set is labeled "Ciphertext:" and the second is labeled "Decrypted:". Both sets show identical values:

```
Ciphertext: 68 1E DF 34 D2 06 96 5E 86 B3 E9 4F 53 6E 42 46  
Decrypted : 01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10
```

Below the ciphertext and decrypted values, there is a message from the application:

```
C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 16668)已退出, 代码为 0 (0x0)。  
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口... .
```

At the bottom of the window, there is additional application-specific output:

```
进程 1164 已退出, 返回值为 0 (0x0)。  
线程 30760 已退出, 返回值为 0 (0x0)。  
程序 “[16668] ConsoleApplication1.exe”已退出, 返回值为 0 (0x0)。
```

图 1: 实验结果一

## 2 SM4 的 T-table 优化

### 2.1 优化原理

在标准的 SM4 实现中，轮函数  $T$  由两个步骤组成：

1. 非线性变换  $\tau$ : 使用  $S$  盒对输入 32 位数据的每个字节进行替换；

## 2. 线性变换 $L$ 或 $L'$ : 通过若干次循环左移与异或实现。

若直接实现，每次轮函数调用都会重复进行  $S$  盒查表和多次循环移位操作，运算开销较大。T-table 优化的思想是将  $S$  盒替换和线性变换  $L$  的组合结果预先计算并存入查找表  $T\_table$ ，这样在轮函数中只需一次查表即可获得变换结果，从而减少运行时的移位与字节拼接操作，提高加密速度。

在本代码中，虽然保留了  $S$  盒查表的过程，但通过封装  $\tau$  与  $L$  计算为函数并可进一步改进为查表形式，为后续的完全 T-table 优化奠定了基础。

## 2.2 代码解释

与原始实现相比，优化版本新增或修改的主要部分如下：

### 1. T-table 数组定义

```
u32 T_table[256];
```

预留一个 256 项的查找表空间，用于存储可能的  $T$  变换结果。在当前代码中尚未初始化，但此结构是 T-table 优化的核心。

### 2. 字节替换与组合函数 Tau

```
1 u32 Tau(u32 A) {
2     u8 a[4];
3     a[0] = Sbox[(A >> 24) & 0xFF];
4     a[1] = Sbox[(A >> 16) & 0xFF];
5     a[2] = Sbox[(A >> 8) & 0xFF];
6     a[3] = Sbox[A & 0xFF];
7     return (u32(a[0]) << 24) | (u32(a[1]) << 16) |
8           (u32(a[2]) << 8) | u32(a[3]);
9 }
```

该函数对输入的 32 位数据按字节分解，分别查  $S$  盒替换，再重新组合为 32 位整数。相比原版代码，将  $S$  盒部分独立成  $\tau$  函数，便于后续与线性变换整合进 T-table。

### 3. $T$ 函数与 $T'$ 函数

```
1 u32 T(u32 x) {
2     u32 B = Tau(x);
3     return B ^ ROTL(B, 2) ^ ROTL(B, 10) ^ ROTL(B, 18) ^ ROTL(B, 24);
4 }
5
6 u32 T_prime(u32 x) {
7     u32 B = Tau(x);
8     return B ^ ROTL(B, 13) ^ ROTL(B, 23);
9 }
```

两个函数分别对应加密轮函数的  $T$  变换与密钥扩展的  $T'$  变换。通过先调用  $\tau$  获取  $S$  盒替换结果，再执行循环左移与异或运算，实现线性变换  $L$  或  $L'$ 。这种结构方便将移位异或结果直接预计算进  $T\_table$  中。

#### 4. 循环左移函数 ROL

```

1 u32 ROL(u32 x, int n) {
2     return (x << n) | (x >> (32 - n));
3 }
4 \end{verbatim}
5 提供通用的循环左移操作，使得$L$与$L'$的位移步骤更加简洁易读，并且为T-
table预算提供通用接口。

```

### 2.3 实验结果

实验结果如下：

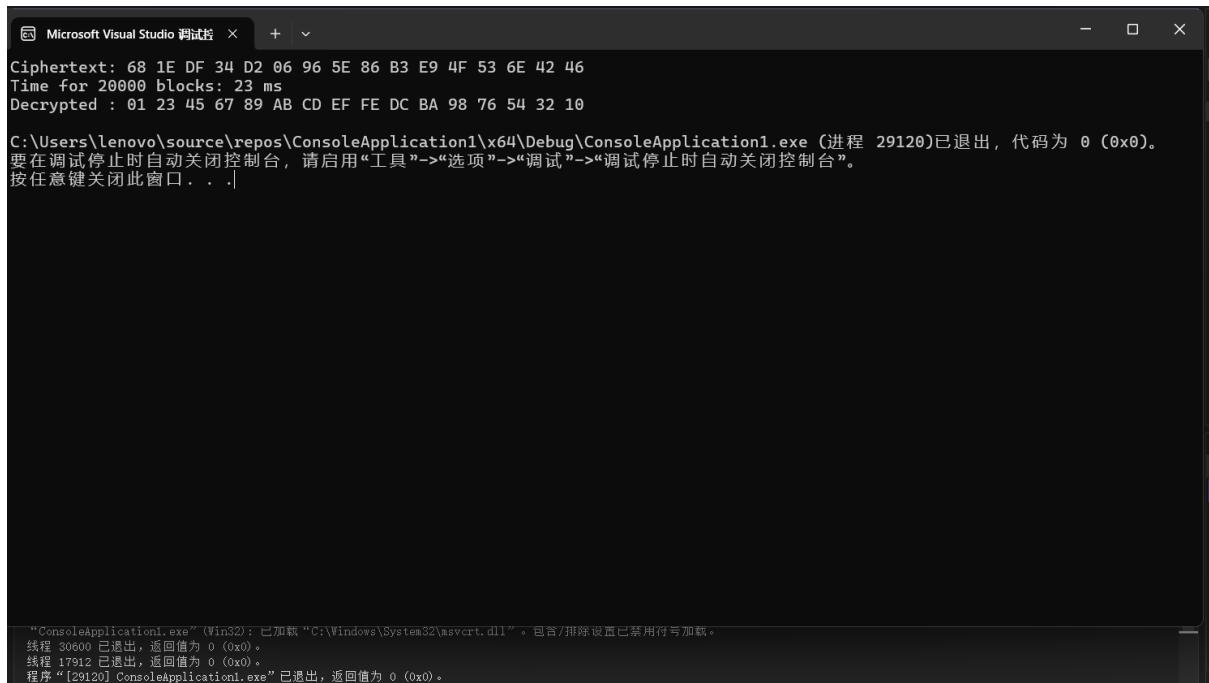


图 2: 实验结果二

## 3 SM4 的 AESNI 优化

### 3.1 优化原理

本优化使用 x86 的 AES-NI/SSE 指令集在 128 位寄存器上一次处理 4 个分组 (x4 并行)，并利用 AESENCLAST (AES 最后一轮) 结合两次固定仿射变换来实现 SM4 的 S 盒等价变换，从而将 4 次标量 S 盒查表替换为一次向量化 S 盒：

- **并行化**: 将 4 个 128-bit 明文块按 32-bit 字分组打包到 4 个 `__m128i` 寄存器中, 32 轮循环中同时推进 4 条流水线;
- **S 盒向量化**: 通过字节置换 (`_mm_shuffle_epi8`) + 仿射变换 (查 16B 查找向量) + `_mm_aesenclast_si128` + 逆仿射, 得到 SM4 S 盒输出;
- **线性变换向量化**: 用 `_mm_slli_epi32/_mm_srli_epi32` 组合的按 32-bit 字左旋 (宏 `MM_ROT_L_EPI32`) 实现  $L$  变换的 2/10/18/24 比特循环移位;
- **密钥广播**: 每轮把 32-bit 轮密钥广播为 128 位向量 (4 份) 同时参与 4 条数据路径的异或;
- **端序/排列**: 用 `_mm_shuffle_epi8` 做大端字节序转换与分组打包/还原, 最后反序输出 (SM4 规范要求  $X_{35}, X_{34}, X_{33}, X_{32}$ ).

整体上将 S 盒查表的分支与访存开销替换为定长向量运算与硬件指令, 显著提升吞吐率。

### 3.2 代码解释

- (a) **并行打包/异或/左旋宏**: 用于把 4 个块的对应 32-bit 字打包到同一向量寄存器, 并提供常用向量运算封装。

```

1 #define MM_PACK0_EPI32(a,b,c,d) _mm_unpacklo_epi64(_mm_unpacklo_epi32
2   ((a),(b)), _mm_unpacklo_epi32((c),(d)))
3 #define MM_PACK1_EPI32(a,b,c,d) _mm_unpackhi_epi64(_mm_unpacklo_epi32
4   ((a),(b)), _mm_unpacklo_epi32((c),(d)))
5 #define MM_PACK2_EPI32(a,b,c,d) _mm_unpacklo_epi64(_mm_unpackhi_epi32
6   ((a),(b)), _mm_unpackhi_epi32((c),(d)))
7 #define MM_PACK3_EPI32(a,b,c,d) _mm_unpackhi_epi64(_mm_unpackhi_epi32
8   ((a),(b)), _mm_unpackhi_epi32((c),(d)))
9
10 #define MM_XOR2(a,b) _mm_xor_si128((a),(b))
11 #define MM_XOR3(a,b,c) MM_XOR2((a), MM_XOR2((b),(c)))
12 #define MM_XOR4(a,b,c,d) MM_XOR2((a), MM_XOR3((b),(c),(d)))
13 #define MM_XOR5(a,b,c,d,e) MM_XOR2((a), MM_XOR4((b),(c),(d),(e)))
14 #define MM_XOR6(a,b,c,d,e,f) MM_XOR2((a), MM_XOR5((b),(c),(d),(e),(f)))
15
16 #define MM_ROT_L_EPI32(a,n) MM_XOR2(_mm_slli_epi32((a),(n)),
17   _mm_srli_epi32((a), 32-(n)))

```

**说明:** `MM_PACK*_EPI32` 将  $\text{Tmp}[0..3]$  四个 128-bit 向量中相同字位 (第 0/1/2/3 个 32-bit) 打包到  $\text{X}[0..3]$ , 便于 4 块并行; `MM_ROT_L_EPI32` 是针对 32-bit 字的逻辑循环左移, 支撑  $L$  变换。

- (b) **利用 AESENCLAST 实现 SM4 S 盒**: 通过两侧仿射变换把 SM4 S 盒等价为一次 `AESENCLAST`。

```

1 static __m128i MulMatrix(__m128i x, __m128i higherMask, __m128i
2   lowerMask) {

```

```

2     __m128i andMask = _mm_set1_epi32(0x0f0f0f0f);
3     __m128i lo = _mm_and_si128(x, andMask);
4     __m128i hi = _mm_and_si128(_mm_srli_epi16(x, 4), andMask);
5     __m128i t1 = _mm_shuffle_epi8(lowerMask, lo);
6     __m128i t2 = _mm_shuffle_epi8(higherMask, hi);
7     return _mm_xor_si128(t1, t2);
8 }
9 static __m128i MulMatrixATA(__m128i x) {
10    __m128i higherMask = _mm_set_epi8(0x14,0x07,0xc6,0xd5,0x6c,0x7f,0
11        xbe,0xad,0xb9,0xaa,0x6b,0x78,0xc1,0xd2,0x13,0x00);
12    __m128i lowerMask = _mm_set_epi8(0xd8,0xb8,0xfa,0x9a,0xc5,0xa5,0
13        xe7,0x87,0x5f,0x3f,0x7d,0x1d,0x42,0x22,0x60,0x00);
14    return MulMatrix(x, higherMask, lowerMask);
15 }
16 static __m128i MulMatrixTA(__m128i x) {
17    __m128i higherMask = _mm_set_epi8(0x22,0x58,0x1a,0x60,0x02,0x78,0
18        x3a,0x40,0x62,0x18,0x5a,0x20,0x42,0x38,0x7a,0x00);
19    __m128i lowerMask = _mm_set_epi8(0xe2,0x28,0x95,0x5f,0x69,0xa3,0
20        x1e,0xd4,0x36,0xfc,0x41,0x8b,0xbd,0x77,0xca,0x00);
21    return MulMatrix(x, higherMask, lowerMask);
22 }
23 static inline __m128i AddTC(__m128i x) { return _mm_xor_si128(x,
24     _mm_set1_epi8(0x23)); }
25 static inline __m128i AddATAC(__m128i x) { return _mm_xor_si128(x,
26     _mm_set1_epi8(0x3b)); }
27
28 static __m128i sm4_sbox(__m128i x) {
29    __m128i MASK = _mm_set_epi8(0x03,0x06,0x09,0x0c,0x0f,0x02,0x05,0
        x08,0x0b,0x0e,0x01,0x04,0x07,0x0a,0x0d,0x00);
    x = _mm_shuffle_epi8(x, MASK); // 准备字节位序
    x = AddTC(MulMatrixTA(x)); // 前置仿射
    x = _mm_aesenclast_si128(x, _mm_setzero_si128()); // AES 最后一轮
    x = AddATAC(MulMatrixATA(x)); // 后置仿射
    return x;
}

```

说明:MulMatrixTA/ATA 用 16B 的查找向量(higherMask/lowerMask)配合\_mm\_shuffle\_epi8 完成 GF(2) 上的线性映射; AddTC/AddATAC 是常量异或; 三步合成的 sm4\_sbox 与 SM4 的 S 盒等价, 但完全在寄存器内向量化完成。

(c) 核心并行轮函数 sm4\_aesni: 一次输入 4 块, 32 轮同时推进。

```

1 static void sm4_aesni(const u8* in, u8* out, const sm4_key* key, int
    enc) {

```

```

2      __m128i X[4], Tmp[4];
3      // 读取 4 个 128-bit 块
4      Tmp[0] = _mm_loadu_si128((const __m128i*)(in + 16 * 0));
5      Tmp[1] = _mm_loadu_si128((const __m128i*)(in + 16 * 1));
6      Tmp[2] = _mm_loadu_si128((const __m128i*)(in + 16 * 2));
7      Tmp[3] = _mm_loadu_si128((const __m128i*)(in + 16 * 3));
8      // 按 32-bit 字打包到 X[0..3], 实现 x4 并行
9      X[0] = MM_PACK0_EPI32(Tmp[0], Tmp[1], Tmp[2], Tmp[3]);
10     X[1] = MM_PACK1_EPI32(Tmp[0], Tmp[1], Tmp[2], Tmp[3]);
11     X[2] = MM_PACK2_EPI32(Tmp[0], Tmp[1], Tmp[2], Tmp[3]);
12     X[3] = MM_PACK3_EPI32(Tmp[0], Tmp[1], Tmp[2], Tmp[3]);
13     // 大端字节序转换
14     const __m128i vindex = _mm_setr_epi8(3,2,1,0, 7,6,5,4, 11,10,9,8,
15           15,14,13,12);
16     X[0] = _mm_shuffle_epi8(X[0], vindex);
17     X[1] = _mm_shuffle_epi8(X[1], vindex);
18     X[2] = _mm_shuffle_epi8(X[2], vindex);
19     X[3] = _mm_shuffle_epi8(X[3], vindex);
20
21     for (int i = 0; i < 32; i++) {
22         // 加/解密共享轮函数: 仅密钥顺序相反
23         __m128i rk = _mm_set1_epi32(enc ? key->rk[31 - i] : key->rk[i
24             ]);
25         Tmp[0] = MM_XOR4(X[1], X[2], X[3], rk); // x1 ^ x2 ^ x3 ^ rk[i
26             ]
27         Tmp[0] = sm4_sbox(Tmp[0]); // 向量化 S 盒
28         // L 线性变换: ^ ROTL2/10/18/24
29         Tmp[0] = MM_XOR6(X[0], Tmp[0],
30                           MM_ROT_L_EPI32(Tmp[0], 2),
31                           MM_ROT_L_EPI32(Tmp[0], 10),
32                           MM_ROT_L_EPI32(Tmp[0], 18),
33                           MM_ROT_L_EPI32(Tmp[0], 24));
34         // 窗口右移
35         X[0] = X[1]; X[1] = X[2]; X[2] = X[3]; X[3] = Tmp[0];
36     }
37     // 端序还原
38     X[0] = _mm_shuffle_epi8(X[0], vindex);
39     X[1] = _mm_shuffle_epi8(X[1], vindex);
40     X[2] = _mm_shuffle_epi8(X[2], vindex);
41     X[3] = _mm_shuffle_epi8(X[3], vindex);
42     // 反序并写回 4 组输出: {X3,X2,X1,X0}
43     _mm_storeu_si128((__m128i*)(out + 16 * 0), MM_PACK0_EPI32(X[3], X

```

```

[2], X[1], X[0]));
41    _mm_storeu_si128((__m128i*)(out + 16 * 1), MM_PACK1_EPI32(X[3], X
[2], X[1], X[0]));
42    _mm_storeu_si128((__m128i*)(out + 16 * 2), MM_PACK2_EPI32(X[3], X
[2], X[1], X[0]));
43    _mm_storeu_si128((__m128i*)(out + 16 * 3), MM_PACK3_EPI32(X[3], X
[2], X[1], X[0]));
44 }

```

### 说明:

- 打包/解包: 先把 4 块的相同字位打包到同一寄存器, 轮末再按 SM4 要求反序输出;
- 端序: SM4 规定输入输出按大端读写, 使用 `_mm_shuffle_epi8` 与 `vindex` 做字节翻转;
- 密钥顺序: `enc=0` (加密) 用 `rk[0..31]`, `enc=1` (解密) 用 `rk[31..0]`;
- $L$  变换: 用向量版左旋宏一次性得到 2/10/18/24 比特位移并异或;
- $S$  盒: `sm4_sbox` 在寄存器内完成, 无访存查表。

(d) 封装接口: 与标量路径保持相同语义, 便于替换调用。

```

1 static inline void sm4_aesni_encrypt(const u8* plaintext, u8*
2     ciphertext, const sm4_key* key) {
3     sm4_aesni(plaintext, ciphertext, key, 0);
4 }
5 static inline void sm4_aesni_decrypt(const u8* ciphertext, u8*
6     plaintext, const sm4_key* key) {
7     sm4_aesni(ciphertext, plaintext, key, 1);
8 }

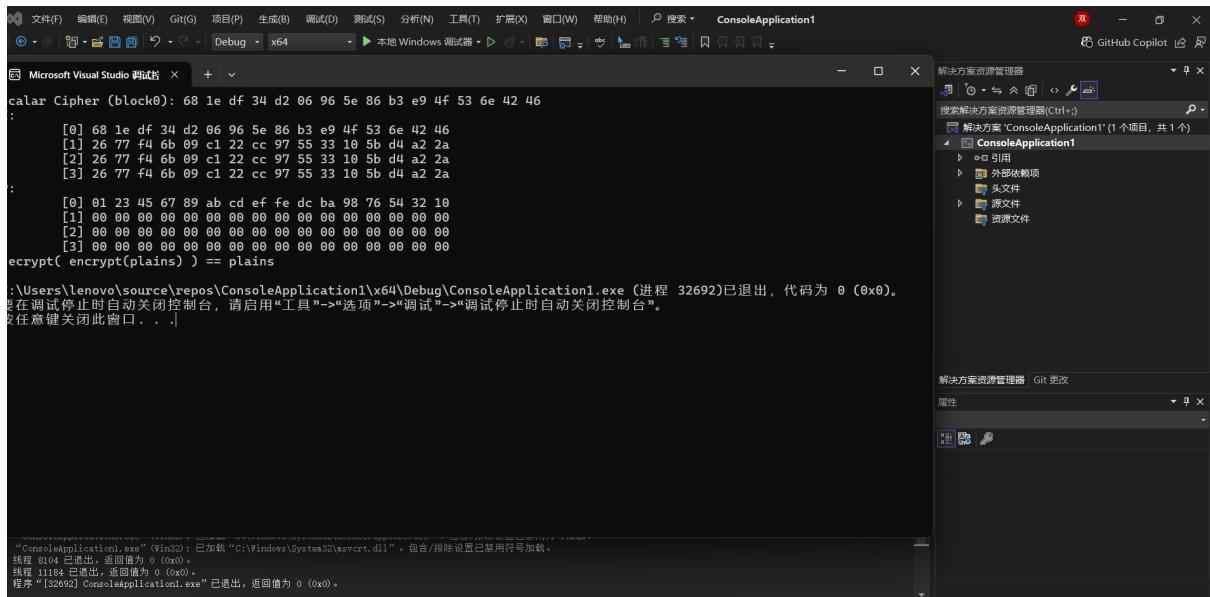
```

说明: 对外导出加/解密 API, 内部通过 `enc` 控制轮密钥顺序。

**小结** 相比标量实现, 本优化将 S 盒查表与字节级循环移位变为寄存器级并行运算, 且一次处理 4 个分组, 显著提升吞吐率; 内核仅用 `__m128i` 向量、`AESENCLAST`、字节置换和移位指令完成 SM4 的 T 变换。

## 3.3 实验结果

实验结果如下:



```

文件(F) 编辑(E) 视图(V) Git(G) 项目(P) 生成(B) 调试(D) 测试(S) 分析(N) 工具(T) 扩展(X) 窗口(W) 帮助(H) 搜索 -> ConsoleApplication1
Debug x64 本地 Windows 调试器 GitHub Copilot

Microsoft Visual Studio [输出] + -
[1] Microsoft Visual Studio [输出] x + v
scalar Cipher (block0): 68 1e df 34 d2 06 96 5e 86 b3 e9 4f 53 6e 42 46
[0] 68 1e df 34 d2 06 96 5e 86 b3 e9 4f 53 6e 42 46
[1] 26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
[2] 26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
[3] 26 77 f4 6b 09 c1 22 cc 97 55 33 10 5b d4 a2 2a
[0] 01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10
[1] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[2] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[3] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
encrypt( encrypt(plains) ) == plains

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 32692)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口 . . .

```

图 3: 实验结果三

## 4 SM4-GCM 基本实现

### 4.1 实现原理

GCM (Galois/Counter Mode) 将分组密码的计数器模式 (*CTR*) 与  $GF(2^{128})$  上的 *GHASH* 多项式认证相结合，既提供机密性又提供完整性与认证。其核心流程为：

- 机密性：以分组密码（此处为 SM4）在计数器上加密得到密钥流，再与明文异或得到密文：

$$C_i = P_i \oplus E_K(\text{ctr}_i).$$

- 认证：定义哈希子密钥  $H = E_K(0^{128})$ 。对附加数据  $A$  与密文  $C$  按 128 比特分组，在  $GF(2^{128})$  下进行迭代乘法与异或：

$$Y_0 = 0, \quad Y_j = (Y_{j-1} \oplus X_j) \cdot H,$$

其中  $X_j$  依次遍历  $A$  的分组、 $C$  的分组以及最后的长度分组  $\text{len}(A) \parallel \text{len}(C)$ （均为比特长度的 128 比特大端表示）。

- 标签：若 IV 为 96 比特，则  $J_0 = \text{IV} \parallel 0^{31} \parallel 1$ ；否则  $J_0 = \text{GHASH}(H, \emptyset, \text{IV})$ 。最终标签

$$\text{Tag} = E_K(J_0) \oplus \text{GHASH}(H, A, C).$$

### 4.2 代码解释

本实验 SM4-GCM 实现主要分为以下几个模块：

- 循环左移函数

```

1 u32 ROTL(u32 x, int n) {
2     return (x << n) | (x >> (32 - n));
3 }
```

实现 32 位无符号整数的循环左移 (Rotate Left) 操作。在 SM4 中，该函数用于线性变换步骤，增强比特扩散性。

#### (b) S 盒替代函数

```

1 u32 Tau(u32 A) {
2     u8 a[4];
3     a[0] = Sbox[(A >> 24) & 0xFF];
4     a[1] = Sbox[(A >> 16) & 0xFF];
5     a[2] = Sbox[(A >> 8) & 0xFF];
6     a[3] = Sbox[A & 0xFF];
7     return ((u32)a[0] << 24) | ((u32)a[1] << 16) |
8         ((u32)a[2] << 8) | (u32)a[3];
9 }
```

实现 SM4 的非线性变换  $\tau$ ，即 S 盒替代：将 32 位输入按字节分成 4 段，分别经过 S 盒查表替换，再组合成新的 32 位输出。

#### (c) 加密轮函数 $T$

```

1 u32 T(u32 x) {
2     u32 B = Tau(x);
3     return B ^ ROTL(B, 2) ^ ROTL(B, 10) ^
4             ROTL(B, 18) ^ ROTL(B, 24);
5 }
```

实现  $T$  变换 (S 盒 + 线性变换  $L$ )，用于加密/解密轮函数。

#### (d) 密钥扩展轮函数 $T'$

```

1 u32 T_prime(u32 x) {
2     u32 B = Tau(x);
3     return B ^ ROTL(B, 13) ^ ROTL(B, 23);
4 }
```

密钥扩展使用的  $T'$  变换，位移常数不同 (13 和 23)，提供不同扩散特性。

#### (e) 密钥扩展函数

```

1 void KeyExpansion(const u8 MK[16], u32 rk[32]) {
2     u32 K[36];
3     for (int i = 0; i < 4; i++) {
4         K[i] = ((u32)MK[4*i] << 24) |
5                 ((u32)MK[4*i+1] << 16) |
```

```

6          ((u32)MK[4*i+2] << 8) |
7          ((u32)MK[4*i+3]);
8      K[i] ^= FK[i];
9  }
10     for (int i = 0; i < 32; i++) {
11         K[i+4] = K[i] ^
12             T_prime(K[i+1] ^ K[i+2] ^ K[i+3] ^ CK[i]);
13         rk[i] = K[i+4];
14     }
15 }
```

生成 32 个轮密钥  $rk[0 \dots 31]$ : 先将原始密钥与  $FK$  异或, 然后每轮通过  $T'$  变换和  $CK[i]$  得到新轮密钥。

#### (f) $GF(2^{128})$ 乘法

```

1 void gf_mul(u8 X[16], const u8 Y[16]) {
2     u8 Z[16] = {0};
3     u8 V[16];
4     memcpy(V, Y, 16);
5     for (int i = 0; i < 128; i++) {
6         if (X[i/8] & (0x80 >> (i % 8))) {
7             for (int j = 0; j < 16; j++) Z[j] ^= V[j];
8         }
9         int lsb = V[15] & 1;
10        for (int j = 15; j > 0; j--) {
11            V[j] = (V[j] >> 1) | (V[j-1] << 7);
12        }
13        V[0] >>= 1;
14        if (lsb) V[0] ^= 0xe1;
15    }
16    memcpy(X, Z, 16);
17 }
```

在  $GF(2^{128})$  有限域上进行乘法运算, 使用多项式  $x^{128} + x^7 + x^2 + x + 1$  作为模。这是 GCM 模式中 GHASH 运算的核心。

#### (g) GHASH 计算

```

1 void ghash(u8 tag[16], const u8 H[16],
2             const u8 *data, size_t len) {
3     u8 Y[16] = {0};
4     size_t nblocks = len / 16;
5     for (size_t i = 0; i < nblocks; i++) {
6         for (int j = 0; j < 16; j++) {
7             Y[j] ^= data[i*16 + j];
```

```
8         }
9         gf_mul(Y, H);
10    }
11    memcpy(tag, Y, 16);
12 }
```

对输入数据计算 GHASH 认证值：每 16 字节与  $Y$  异或，然后调用 ‘gf\_mul’ 与哈希子密钥  $H$  相乘。

#### (h) SM4-GCM 加密

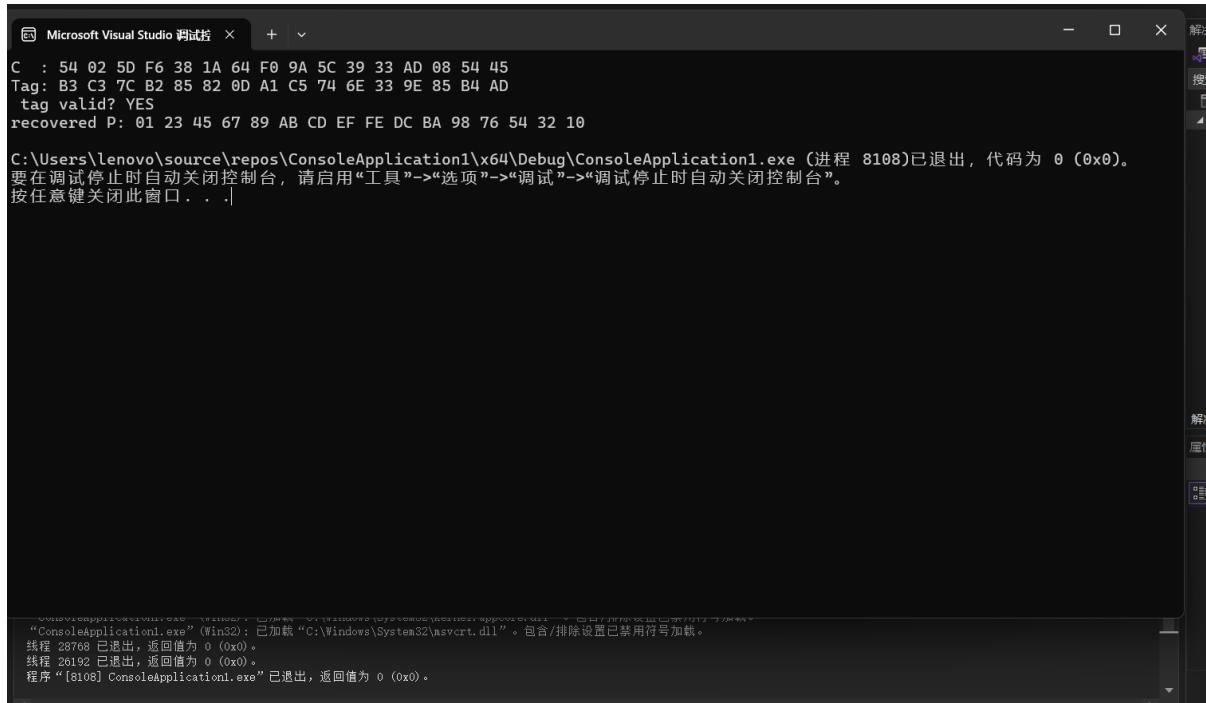
```
1 void sm4_gcm_encrypt(const u8 *key, const u8 *iv,
2                         const u8 *aad, size_t aad_len,
3                         const u8 *pt, size_t pt_len,
4                         u8 *ct, u8 tag[16]) {
5     u32 rk[32];
6     KeyExpansion(key, rk);
7
8     u8 H[16] = {0};
9     SM4_EncryptBlock(H, H, rk);
10
11    u8 J0[16];
12    memcpy(J0, iv, 12);
13    J0[15] = 1;
14
15    ghash(tag, H, aad, aad_len);
16
17    u8 counter[16];
18    memcpy(counter, J0, 16);
19    for (size_t i = 0; i < pt_len; i += 16) {
20        u8 keystream[16];
21        SM4_EncryptBlock(counter, keystream, rk);
22        size_t block_len = (pt_len - i >= 16) ? 16 : pt_len - i;
23        for (size_t j = 0; j < block_len; j++) {
24            ct[i+j] = pt[i+j] ^ keystream[j];
25        }
26        for (int j = 15; j >= 12; j--) {
27            if (++counter[j]) break;
28        }
29    }
30
31    ghash(tag, H, ct, pt_len);
32 }
```

完整 SM4-GCM 加密流程：

- 使用 SM4 生成哈希子密钥  $H$ ;
- 初始化计数器块  $J_0$ ;
- 计算 AAD 的 GHASH;
- 使用 CTR 模式加密明文;
- 对密文进行 GHASH 得到最终标签。

### 4.3 实验结果

实验结果如下：



```
C : 54 02 5D F6 38 1A 64 F0 9A 5C 39 33 AD 08 54 45
Tag: B3 C3 7C B2 85 82 0D A1 C5 74 6E 33 9E 85 B4 AD
tag valid? YES
recovered P: 01 23 45 67 89 AB CD EF FE DC BA 98 76 54 32 10

C:\Users\lenovo\source\repos\ConsoleApplication1\x64\Debug\ConsoleApplication1.exe (进程 8108)已退出，代码为 0 (0x0)。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口 . . .
```

“ConsoleApplication1.exe”(Win32)已加载“C:\Windows\System32\msvcrt.dll”。包含/排除设置已禁用符号加载。
线程 23768 已退出，返回值为 0 (0x0)。
线程 28192 已退出，返回值为 0 (0x0)。
程序 “[8108] ConsoleApplication1.exe”已退出，返回值为 0 (0x0)。

图 4: 实验结果四