

Project 1: 做 SM4 的软件实现和优化

a): 从基本实现出发 优化 SM4 的软件执行效率, 至少应该覆盖 T-table、AESNI 以及最新的指令集 (GFNI、VPROLD 等)

SM4 分组密码算法规范

加密时

一、基本参数

分组长度: 128 位 (16 字节)

密钥长度: 128 位 (16 字节)

结构类型: 非平衡 Feistel 网络

迭代轮数: 32 轮固定轮次

安全强度: 128 位安全级别 (目前无有效攻击方法)

二、加密流程

1. 输入参数:

明文分组 P (128 位)

加密主密钥 MK (128 位)

2. 密钥扩展:

从主密钥 MK 生成 32 个轮密钥 $rk_0 \sim rk_{31}$ (每个 32 位)

3. 初始变换:

分割明文: $P = (X_0, X_1, X_2, X_3)$

系统参数异或操作:

$$X_0 = X_0 \oplus 0xA3B1BAC6$$

$$X_1 = X_1 \oplus 0x56AA3350$$

$$X_2 = X_2 \oplus 0x677D9197$$

$$X_3 = X_3 \oplus 0xB27022DC$$

4. 轮函数迭代 (32 轮):

每轮运算公式:

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

核心变换 $T(\cdot) = L(\tau(\cdot))$ 包含:

非线性变换 τ :

- 32 位输入分割为 4 个字节
- 每个字节通过 8×8 S 盒置换 (基于有限域仿射变换)

线性变换 L :

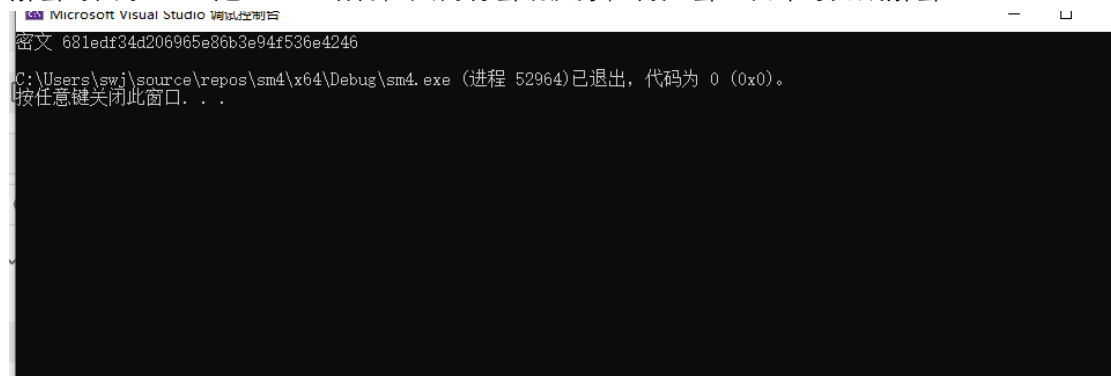
$$*L(B) = B \oplus (B \lll 2) \oplus (B \lll 10) \oplus (B \lll 18) \oplus (B \lll 24)*$$

(\lll 表示循环左移操作)

5. 输出转换:

- 最终状态: $(X_{32}, X_{33}, X_{34}, X_{35})$
- 反序重组: $(Y_0, Y_1, Y_2, Y_3) = (X_{35}, X_{34}, X_{33}, X_{32})$
- 输出密文: $C = (Y_0, Y_1, Y_2, Y_3)$ (128 位)

解密时由于 sm4 是 feistel 结构, 只需将密钥反序, 再加密一次即可完成解密



T-Table 优化是一种**预计算查表法**, 通过将 SM4 轮函数中的非线性变换 (τ) 和线性变换 (L) 合并为预计算的查表操作, 显著减少实时计算量。其核心思想是将耗时的复合变换转换为内存访问操作, 适用于软件和硬件实现场景。

1. 数学原理

SM4 轮函数中的关键变换为合成置换 $T(\cdot) = L \circ \tau(\cdot)$ $\tau(\cdot) = L \circ \tau(\cdot)$, 其中:

非线性变换 τ : 输入 32 位数据 $A = (a_0, a_1, a_2, a_3)$ $A = (a_0, a_1, a_2, a_3)$, 通过 4 个 S 盒并行替换:

$\tau(A) = (\text{Sbox}(a_0), \text{Sbox}(a_1), \text{Sbox}(a_2), \text{Sbox}(a_3))$ $\tau(A) = (\text{Sbox}(a_0), \text{Sbox}(a_1), \text{Sbox}(a_2), \text{Sbox}(a_3))$

线性变换 L : 对 τ 的输出 B 进行扩散:

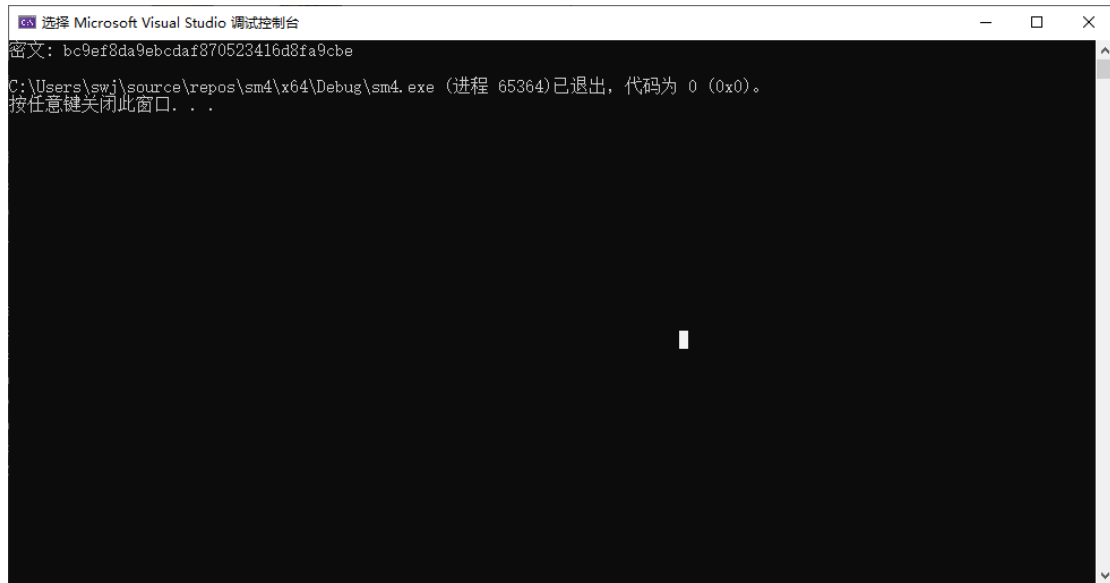
$L(B) = B \oplus (B \ll 2 \oplus (B \ll 10) \oplus (B \ll 18) \oplus (B \ll 24))$ $L(B) = B \oplus (B \ll 2 \oplus (B \ll 10) \oplus (B \ll 18) \oplus (B \ll 24))$

```
void sm4_encrypt_optimized(const uint8_t input_block[16], uint8_t output_block[16], const uint32_t round_keys[32]) {
    uint32_t state_words[36];

    for (int i = 0; i < 4; i++) {
        state_words[i] = ((uint32_t)input_block[4 * i] << 24) |
            ((uint32_t)input_block[4 * i + 1] << 16) |
            ((uint32_t)input_block[4 * i + 2] << 8) |
            input_block[4 * i + 3];
    }

    for (int i = 0; i < 32; i++) {
        state_words[i + 4] = state_words[i] ^
            optimized_t_transform(state_words[i + 1] ^
                state_words[i + 2] ^
                state_words[i + 3] ^
                round_keys[i]);
    }

    for (int i = 0; i < 4; i++) {
        output_block[4 * i] = (state_words[35 - i] >> 24) & 0xFF;
        output_block[4 * i + 1] = (state_words[35 - i] >> 16) & 0xFF;
        output_block[4 * i + 2] = (state_words[35 - i] >> 8) & 0xFF;
        output_block[4 * i + 3] = state_words[35 - i] & 0xFF;
    }
}
```



SIMD 优化

优化通过并行处理提高加密性能，AVX2 指令集并行化，将 4 个独立 128 位数据块重组为 4 个 256 位 SIMD 向量

每个向量包含 4 个块中相同位置的数据（跨块数据对齐），从而实现单条指令处理 4 个块的数据。

```
// 并行加密4个数据块
void sm4_encrypt_four_blocks_parallel(const uint8_t input_blocks[4][16], uint8_t output_blocks[4][16], const uint32_t round
// 状态向量，每个 _mm256i 包含4个不同位置的相同字节
    _mm256i state_vector_0, state_vector_1, state_vector_2, state_vector_3;

// 加载数据
uint32_t temp[16];
for (int i = 0; i < 4; i++) {
    memcpy(&temp[i * 4], input_blocks[i], 16);
}

state_vector_0 = _mm256_set_epi32(temp[12], temp[8], temp[4], temp[0],
    temp[12], temp[8], temp[4], temp[0]);
state_vector_1 = _mm256_set_epi32(temp[13], temp[9], temp[5], temp[1],
    temp[13], temp[9], temp[5], temp[1]);
state_vector_2 = _mm256_set_epi32(temp[14], temp[10], temp[6], temp[2],
    temp[14], temp[10], temp[6], temp[2]);
state_vector_3 = _mm256_set_epi32(temp[15], temp[11], temp[7], temp[3],
    temp[15], temp[11], temp[7], temp[3]);
```

使用并行轮函数计算，单次迭代完成 4 个数据块的轮函数计算

```
// 32轮迭代
for (int round = 0; round < 32; round++) {
    _mm256i round_key_vector = _mm256_set1_epi32(round_keys[round]);
    _mm256i temp_vector = _mm256_xor_si256(_mm256_xor_si256(state_vector_1, state_vector_2), state_vector_3);
    temp_vector = _mm256_xor_si256(temp_vector, round_key_vector);
    temp_vector = _mm256_xor_si256(state_vector_0, avx2_composite_transform_t(temp_vector));

    state_vector_0 = state_vector_1;
    state_vector_1 = state_vector_2;
    state_vector_2 = state_vector_3;
    state_vector_3 = temp_vector;
}
```

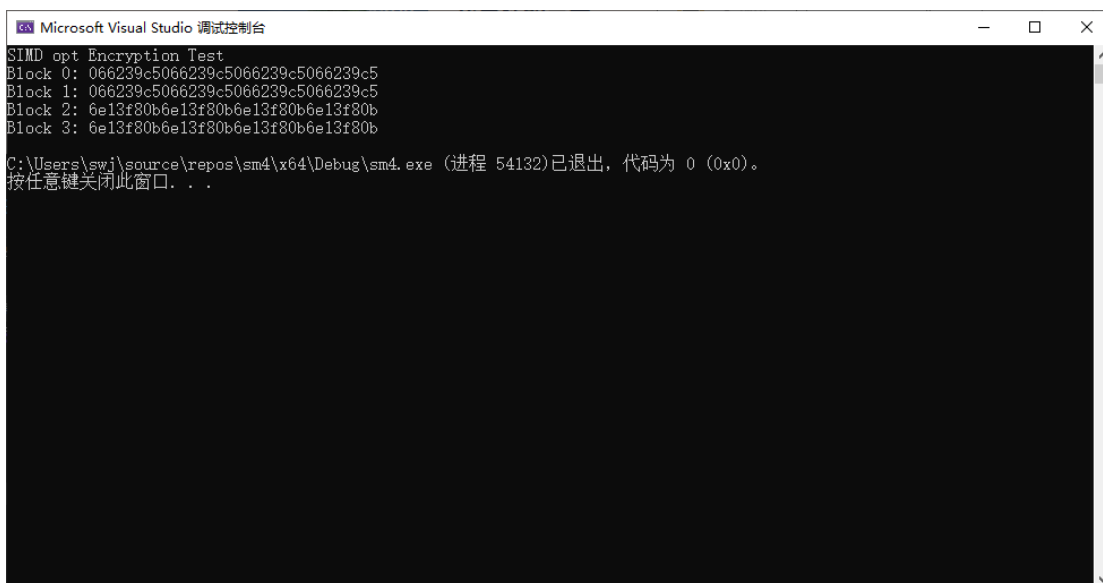
Avx2 指令实现并行循环位移，单指令完成 32 位字的循环移位操作，线性变换 L 向量化

```

// AVX2循环左移操作
static inline __m256i avx2_rotate_left_32(__m256i input_vector, int shift_bits) {
    return _mm256_or_si256(_mm256_slli_epi32(input_vector, shift_bits),
        _mm256_srli_epi32(input_vector, 32 - shift_bits));
}

// AVX2线性变换L
static inline __m256i avx2_linear_transform_l(__m256i input_vector) {
    return _mm256_xor_si256(
        _mm256_xor_si256(
            _mm256_xor_si256(
                _mm256_xor_si256(input_vector, avx2_rotate_left_32(input_vector, 2)),
                avx2_rotate_left_32(input_vector, 10)),
            avx2_rotate_left_32(input_vector, 18)),
        avx2_rotate_left_32(input_vector, 24));
}

```



```

Microsoft Visual Studio 调试控制台
SIMD opt Encryption Test
Block 0: 066239c5066239c5066239c5066239c5
Block 1: 066239c5066239c5066239c5066239c5
Block 2: 6e13f80b6e13f80b6e13f80b6e13f80b
Block 3: 6e13f80b6e13f80b6e13f80b6e13f80b

C:\Users\swj\source\repos\sm4\x64\Debug\sm4.exe (进程 54132)已退出, 代码为 0 (0x0)。
按任意键关闭此窗口。 . . .

```

b): 基于 SM4 的实现，做 SM4-GCM 工作模式的软件优化实现

SM4 算法在 Galois/Counter Mode 下的实现，优化核心在于协同加速计数器模式(CTR)加密与伽罗瓦认证(GHASH)，利用 SIMD 指令(AVX2/AVX-512)并行加密多个计数器值，单指令同时生成 4/8 个密钥流块，与明文块并行异或实现流水线吞吐，采用 PCLMULQDQ 硬件指令加速 $GF(2^{128})$ 有限域乘法，批量处理数据块减少函数调用开销，使共享密钥扩展与预计算资源，内存对齐访问提升缓存利用率。

