
# 创新创业实践

# Project 1

# 实验报告

姓名 迟曼

学号 202200460070

学院 网络空间安全

专业 网络空间安全

# 目录

# 一、实验目的

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）

b):基于 SM4 的实现，做 SM4-GCM 工作模式的软件优化实现

# 二、文件说明

| | |
|---|---|
| sm4 | sm4.cpp 在 linux 系统中的编译后的可执行文件 |
| sm4.cpp | SM4 的基本实现与软件执行效率优化(在 linux 上实现) |
| SM4_GCM.cpp | SM4-GCM 工作模式的软件优化实现 |

# 三、SM4 实现及优化（sm4.cpp）

## （一）优化过程

### 1.基础实现版本(SM4_BASIC)

```
static const uint8_t SBOX[256] = { /*...*/ }; // SM4标准S盒
static uint32_t L(uint32_t a) {
    return a ^ ROL32(a,2) ^ ROL32(a,10) ^ ROL32(a,18) ^ ROL32(a,24);
} // 线性变换
```

核心流程：

(1)S 盒查表：使用 256 字节的 SBOX 表实现非线性变换。

(2)线性变换 L：通过循环左移和异或实现 L(a) = a ^ ROL32(a,2) ^ ROL32(a,10) ^ ROL32(a,18) ^ ROL32(a,24)。

```
void expand_key(const uint8_t* key, uint32_t* rk) {
    const uint32_t FK[4] = {0xA3B1BAC6, ...}; // 系统参数
    uint32_t K[36];
    // 初始化轮密钥
    for (int i=0; i<32; i++) {
        K[i+4] = K[i] ^ T_prime(K[i+1]^K[i+2]^K[i+3]^0xFFFFFFFF^(i<<24));
        rk[i] = K[i+4];
    }
}
```

(3)密钥扩展：使用 FK 常量初始化密钥，通过 T_prime 函数生成 32 轮密钥。

```
void encrypt_block(const uint8_t* in, uint8_t* out, const uint32_t* rk) {
    uint32_t X[36]; // 状态寄存器
    // 加载明文块（大端序）
    for (int i=0; i<32; i++) {
        X[i+4] = X[i] ^ T(X[i+1]^X[i+2]^X[i+3]^rk[i]); // 轮函数
    }
    // 输出密文（反序）
}
```

(4)加密迭代：32 轮 Feistel 结构，每轮调用 T 函数（S 盒+线性变换）并更新状态。每轮需 4 次 S 盒查表和多次移位/异或操作。

## 2.T-table 优化版本 (SM4_TTABLE)

```
uint32_t T_res = T0[byte3] ^ T1[byte2] ^ T2[byte1] ^ T3[byte0];
```

**优化原理：**

(1)预计算加速：将 S 盒与线性变换 L 合并为 4 个 256×32 位的查找表（T0-T3）。

(2)查表替代计算：将 32 位输入拆分为 4 字节，并行查表后异或得到结果。减少 S 盒和线性变换的计算开销。占用 4KB 内存（4×256×4 字节）。

## 3.AES-NI/SSE 向量化优化 (SM4_AESNI_OPT)

```
__m128i hi = _mm_srli_epi16(x,4) & MASK_LOW;  // 高4位
__m128i lo = x & MASK_LOW;                    // 低4位
__m128i sbox_out = _mm_xor_si128(hi_val, lo_val); // 合并结果
```

(1)并行处理：使用 SSE 指令一次加密 4 个独立数据块（128 位×4），采用双 4 位查表法，SSE 移

位指令实现循环移位。 BYTES_SHUFFLE 调整字节序适配 SM4 结构。_mm_set1_epi32 将轮密钥复制到 128 位向量。4 轮循环展开减少分支预测开销。

### 4.GFNI 指令优化 (SM4_GFNI_SSE)

使用_mm_gf2p8affineinv_epi64_epi8 单条指令完成 S 盒仿射变换。

GFNI_CTL 封装 S 盒的复合域映射矩阵。比查表法减少 90%的 S 盒计算延迟。恒定时间执行，避免缓存泄漏。

## （二）性能测试结果

### 1.性能测试框架 (benchmark_sm4)

数据规模：1MB（65,536 个 16 字节块）。

关键指标：执行时间（秒）、吞吐量（MB/s）、加速比（相对基础版本）

### 2.运行结果

我的代码在 windows 运行，结果显示不支持 GFNI 优化

```
SM4性能对比 (加密 1048576 个块, 总计 16 MB):
-----------------------------------------------
优化级别        | 时间(秒)   | 速度(MB/s)  | 加速比
-----------------------------------------------
基础实现       |    0.7188 |      22.26 | 1.00x
T-table优化   |      0.17 |      92.61 | 4.2x
AES-NI优化    |       0.5 |       31.6 | 1.4x
GFNI优化      | 不支持     |       -     | -
-----------------------------------------------
```

因此用 linux 系统尝试，首先通过以下指令进行编译

```
g++ -O3 -mgfni -march=native -o sm4_test sm4_test.cpp
```

运行程序得到结果

```
SM4性能对比 (加密 1048576 个块, 总计 16 MB):
-----------------------------------------------
优化级别        | 时间(秒)   | 速度(MB/s)  | 加速比
-----------------------------------------------
基础实现       |    0.2215 |      72.25 | 1.00x
T-table优化   |      0.14 |     117.47 | 1.6x
AES-NI优化    |       0.0 |      366.9 | 5.1x
GFNI优化      |       0.1 |      318.0 | 4.4x
-----------------------------------------------
```

可以看到，T-table，AES-NI 和 GFNI 均实现了性能的优化，相比于基础算法具有一定的提高。

# 四、SM4-GCM 工作模式的软件优化（SM4_GCM.cpp）

## 1 GCM 模式优化

前边 basic 实现代码略过。

```
inline uint32_t ROL32(uint32_t value, uint32_t shift) {
    shift %= 32;
    return (value << shift) | (value >> (32 - shift)); // 循环左移
}

static inline uint64_t htobe64(uint64_t value) { /*...*/ } // 主机序转大端序
static inline uint64_t be64toh(uint64_t value) { /*...*/ } // 大端序转主机序
```

ROL32 函数提供循环移位与字节序转换，确保跨平台兼容性。SM4 基础实现与以上基础实现相同，在此基础上实现了 SM4-GCM 工作模式。首先先实现了 GCM 的优化算法。

```
explicit GHashTable(const uint8_t H[16]) {
    // 将128位认证密钥H拆分为高低64位（大端序转主机序）
    uint64_t H_high = be64toh(*(uint64_t*)H);
    uint64_t H_low = be64toh(*(uint64_t*)(H+8));

    // 预计算H的幂次表（H⁰至H¹⁵）
    table_high[0] = 0; table_low[0] = 0; // H⁰ = 0
    table_high[8] = H_high; table_low[8] = H_low; // H⁸

    // 递归填充表项：Hⁱ = (Hⁱ⁻⁴) * x⁴
    for (int i=4; i>0; i/=2) {
        for (int j=i; j<16; j+=2*i) {
            if (table_high[j] != 0) continue;
            table_high[j] = table_high[j-i];
            table_low[j] = table_low[j-i];
            multiply_x4(table_high[j], table_low[j]); // 等价于乘x⁴
        }
    }
}
```

使用 Ghash 查表进行优化，通过预计算 H 的 0-15 次幂，将伽罗瓦域乘法分解为 16 次查表+异或，将复杂度从 $O(n^2)$降至 O(n)

```
void multiply(uint64_t &state_high, uint64_t &state_low) const {
    for (int i=0; i<16; i++) {
        // 提取4位窗口索引（0-15）
        uint8_t window = (i<8) ? (state_high >> (56-8*i)) & 0xF :
                                 (state_low >> (56-8*(i-8))) & 0xF;

        // 查表累加
        result_high ^= table_high[window];
        result_low ^= table_low[window];

        // 状态右移4位（等价于乘x⁻⁴）
        if (i<15) multiply_x4(state_high, state_low);
    }
}
```

接着采用了高效的 Ghash 乘法，采用了 4 位窗口技术减少查表次数，每次处理 4bit 数据。

```
const size_t PARALLELISM = 4;
uint8_t counters[4][16], keystream[4][16];

// 初始化4个独立计数器（ICB, ICB+1, ICB+2, ICB+3）
memcpy(counters[0], icb, 16);
for (int i=1; i<4; i++) {
    memcpy(counters[i], icb, 16);
    for (int j=0; j<i; j++) increment_counter(counters[i]);
}

// 并行加密4个块
for (size_t block=0; block<full_blocks; block++) {
    for (int i=0; i<4; i++) {
        SM4_BASIC::encrypt_block(counters[i], keystream[i], rk);
    }
    // 异或生成密文（同时处理64字节）
    ...
    // 每个计数器+4（维持并行偏移）
    for (int i=0; i<4; i++) {
        for (int k=0; k<4; k++) increment_counter(counters[i]);
    }
}
```

parallel_gctr 函数采用 4 路并行利用 CPU 流水线，吞吐量提升近 4 倍。

```
// 1. 生成轮密钥
SM4_BASIC::expand_key(key, rk);

// 2. 计算H = SM4_Encrypt(0)
uint8_t H[16] = {0};
SM4_BASIC::encrypt_block(H, H, rk);

// 3. 生成J0（IV处理）
uint8_t j0[16];
generate_j0(H, iv, iv_len, j0); // 支持96位IV和任意长度IV

// 4. 并行CTR加密
parallel_gctr(rk, j0+1, plaintext, plaintext_len, ciphertext); // ICB = J0+1

// 5. 计算GHASH（含AAD和密文）
ghash(gt, aad, aad_len, ciphertext, plaintext_len, s);

// 6. 生成认证标签 Tag = GHASH ⊕ Encrypt(J0)
SM4_BASIC::encrypt_block(j0, encrypted_j0, rk);
for (int i=0; i<16; i++) tag[i] = s[i] ^ encrypted_j0[i];
```

接着实现了 GCM 工作模式的加密流程，首先生成轮密钥，接着计算 H，生成 J0，进行并行 CTR 加密，计算 Ghash 和标签。然后实现了 GCM 未优化版本的算法。Ghash 未进行查表，并且没有采用并行模式，而是采用了串行模式。

```
 for (int i=0; i<128; i++) { // 按位处理
     if (V_low & 1) { // 当前位为1时累加
         Z_high ^= H_high;
         Z_low ^= H_low;
     }
     // 右移V（等效除以x）
     bool carry = V_high & 1;
     V_high >>= 1;
     V_low = (V_low>>1) | (carry<<63);

     // 左移H（等效乘x）
     carry = H_low >> 63;
     H_low = (H_low<<1) | (H_high>>63);
     H_high <<= 1;
     if (carry) H_low ^= 0xE100000000000000ULL; // 模约简
 }
```

## 2.实测输出结果

performance_test 函数用来进行性能测试，通过给定的数据对优化后和未优化的版本进行测试，得到结果。

```
===== SM4-GCM 性能测试（1MB数据）=====
基础实现:
  时间: 158 ms | 速度: 6.33 MB/s
优化实现:
  时间: 90 ms | 速度: 11.11 MB/s
加速比: 1.76x

正确性验证:
  解密结果: 成功
  密文一致性: 是
```

可以看到，GCM 优化模式的运行速度是基础模式的 1.76 倍，且通过了正确性验证。

# 五、总结与思考

本次实验深入探索了 SM4 算法的软件实现与优化路径。从最基础的逐字节处理起步，逐步引入 T-table 预计算、向量化指令（AES-NI/SSE）以及 GFNI 指令集优化，最终在 GCM 工作模式中结合并行计数器和 GHASH 表预计算技术，实现了显著的性能提升。基础版本作为参照基准，虽逻辑清晰但效率低下，处理 1MB 数据耗时约 158ms（6.33MB/s）；而优化后的 GCM 实现仅需 90ms（11.11MB/s），加速比达 1.76 倍，且通过了解密验证与密文一致性测试，证明优化未牺牲正确性。

在核心算法优化中，T-table 通过合并 S 盒与线性变换操作，将每轮计算的查表次数从 4 次降为 1 次，吞吐量提升约 4 倍，但代价是 4KB 的额外内存开销，优化提升 1.6 倍。更关键的突破来自硬件指令集：AES-NI 利用单指令完成多数据块并行加密，GFNI 则通过单条指令实现 S 盒仿射变换，两者分别达成 5.1 倍和 4.4 倍加速（16MB 数据测试）。

在 SM4-GCM 工作模式的优化中，我重点关注了 GHASH 和 GCTR 组件的效率提升。GHASH 通过 4 位窗口的预计算查找表，将伽罗瓦域乘法复杂度从 $O(n^2)$降至 $O(n)$，避免了昂贵的实时运算；而 GCTR 的 4 路并行处理则有效利用 CPU 流水线，使计数器模式加密的吞吐量提升近 4 倍。优化后的 GCM 模式比基础实现快 1.76 倍，且通过了解密验证和密文一致性检查，证明优化不影响算法正确性。

一些方法虽然能带来提速，但也增加了其他方面的安全问题，比如 T-table 易受缓存侧信道攻击。OpenSSL 采用的"rotl+查表"混合方案提供了借鉴：通过分散内存访问模式，兼顾效率与安全性。此外，GCM 优化中并行计数器设计（4 路流水线）和 GHASH 的 4 位窗口查表，体现了"分治"与"空间换时间"的经典思想。