



网络空间安全创新创业实践 Project1

吴扬

2025 年 7 月 26 日

目录

1 实验环境	2
2 问题重述	2
3 SM4 算法原理	2
3.1 核心参数	2
3.2 加密过程	2
3.3 核心函数	3
3.3.1 非线性变换 τ (S-Box)	3
3.3.2 线性变换 L	3
3.4 密钥扩展	3
4 软件优化技术	4
4.1 T-Table (查找表) 优化	4
4.2 利用 AES-NI 进行同构优化	4
5 C++ 代码实现与性能展示	5
5.1 基础实现与优化实现核心代码	5
5.2 性能测试主函数	10
5.3 性能展示	16
6 SM4-GCM	16
6.1 主要接口	17
6.2 关键数据结构: 预计算表	17
6.3 密钥与初始化	17
6.4 伽罗瓦域 ($GF(2^{128})$) 乘法	18
6.4.1 基本实现	18
6.4.2 优化实现	18
6.5 GHASH 计算	18
6.6 计数器与加密	19
6.6.1 初始计数器 J_0 的生成	19
6.6.2 加密过程	19
6.7 认证标签生成与验证	19
6.7.1 标签生成	19
6.7.2 标签验证	20
6.8 测试与验证	20
7 结论	21

1 实验环境

处理器	Intel(R) Core(TM) i9-14900HX 2.20 GHz
机带 RAM	16.0 GB (15.6 GB 可用)
Windows 版本	Windows 11 家庭中文版
操作系统版本	22631.4169

2 问题重述

软件实现 SM4 并优化。

- 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)。
- 基于 SM4 的实现，做 SM4-GCM 工作模式的软件优化实现。

3 SM4 算法原理

SM4 是一种分组密码算法，由中国国家密码管理局于 2012 年发布，标准号为 GM/T 0002-2012。它是一种非平衡 Feistel 网络结构，以其高效性和安全性被广泛应用于中国的各类信息系统中。

3.1 核心参数

SM4 算法的核心设计参数如下：

- 分组长度：128 位 (16 字节)
- 密钥长度：128 位 (16 字节)
- 迭代轮数：32 轮
- 轮密钥长度：32 位 (4 字节)

3.2 加密过程

SM4 的加密过程包括 32 轮迭代和一个最终的反序变换。设输入明文分组为 (X_0, X_1, X_2, X_3) ，其中 X_i 为 32 位字。

1. 32 轮迭代: 对于 $i = 0, 1, \dots, 31$, 执行以下轮函数:

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

其中 rk_i 是第 i 轮的 32 位轮密钥。这是一个典型的 Feistel 结构, 每一轮的输入 $(X_i, X_{i+1}, X_{i+2}, X_{i+3})$ 经过变换后, 会输出一个新的状态字 X_{i+4} 。

2. 反序变换: 32 轮迭代后, 得到状态 $(X_{32}, X_{33}, X_{34}, X_{35})$ 。最终的密文分组 (Y_0, Y_1, Y_2, Y_3) 是通过反序输出得到的:

$$(Y_0, Y_1, Y_2, Y_3) = (X_{35}, X_{34}, X_{33}, X_{32})$$

解密过程与加密过程结构相同, 但使用反序的轮密钥。

3.3 核心函数

轮函数 F 内部由一个复合变换 T 构成:

$$F(A, B, C, rk) = T(A \oplus B \oplus C \oplus rk)$$

而复合变换 T 由一个非线性 S 盒变换 τ 和一个线性变换 L 组成:

$$T(V) = L(\tau(V))$$

3.3.1 非线性变换 τ (S-Box)

该变换对输入的 32 位字 $V = (v_0, v_1, v_2, v_3)$ (每个 v_i 为 8 位字节) 的每个字节进行 S 盒 (S-Box) 代换:

$$\tau(V) = (\text{SBOX}[v_0], \text{SBOX}[v_1], \text{SBOX}[v_2], \text{SBOX}[v_3])$$

S 盒是一个预定义的 8 位输入到 8 位输出的置换表, 是算法安全性的主要来源。

3.3.2 线性变换 L

该变换对 32 位输入 B 进行异或和循环左移操作, 以提供扩散:

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

其中 \lll 表示循环左移。

3.4 密钥扩展

32 个轮密钥 rk_i 由 128 位主密钥生成。该过程与加密轮函数类似, 但使用了不同的系统参数和线性变换 L' :

$$L'(B) = B \oplus (B \lll 13) \oplus (B \lll 23)$$

4 软件优化技术

PDF 文件中探讨了多种 SM4 软件实现和优化的方法，从基础实现到利用 SIMD 和 AES-NI 硬件指令进行加速。

4.1 T-Table (查找表) 优化

这是一种经典的空间换时间优化策略。其核心思想是将 S 盒查找和后续的线性变换 L 合并成一个预计算的查找表 (T-Table)。由于 L 变换是线性的，满足 $L(A \oplus B) = L(A) \oplus L(B)$ 。因此，对于输入字 $V = (v_0, v_1, v_2, v_3)$ ：

$$\begin{aligned} T(V) &= L(\tau(V)) \\ &= L((\text{SBOX}[v_0] \ll 24) \oplus (\text{SBOX}[v_1] \ll 16) \oplus (\text{SBOX}[v_2] \ll 8) \oplus \text{SBOX}[v_3]) \\ &= L(\text{SBOX}[v_0] \ll 24) \oplus L(\text{SBOX}[v_1] \ll 16) \oplus L(\text{SBOX}[v_2] \ll 8) \oplus L(\text{SBOX}[v_3]) \end{aligned}$$

我们可以定义四个 T-Table :

$$\begin{aligned} T_0[x] &= L(\text{SBOX}[x] \ll 24) \\ T_1[x] &= L(\text{SBOX}[x] \ll 16) \\ T_2[x] &= L(\text{SBOX}[x] \ll 8) \\ T_3[x] &= L(\text{SBOX}[x]) \end{aligned}$$

每个表大小为 256×4 字节 = 1KB，总共需要 4KB 内存。这样，原先需要 4 次 S 盒查找、多次移位和异或的 T 函数，现在只需要 4 次查表和 3 次异或即可完成，显著提升了速度。

$$T(V) = T_0[v_0] \oplus T_1[v_1] \oplus T_2[v_2] \oplus T_3[v_3]$$

不过，使用 T-Table 的实现容易受到缓存计时攻击，因为访存时间差异可能泄露密钥信息。

4.2 利用 AES-NI 进行同构优化

这是一种更高级的优化技术，利用了现代 CPU 中的 AES 新指令集 (AES-NI) 来加速 SM4。其核心原理是 SM4 的 S 盒和 AES 的 S 盒在数学上存在深刻联系。两者都是在有限域 $GF(2^8)$ 上的求逆运算，外加一个仿射变换。

- $SBox_{AES}(x) = A_{AES} \cdot x^{-1} + C_{AES}$ in $GF_{AES}(2^8)$
- $SBox_{SM4}(x) = A_2 \cdot (A_1 \cdot x + C_1)^{-1} + C_2$ in $GF_{SM4}(2^8)$

虽然它们定义在不同的 $GF(2^8)$ 有限域上，但这两个域是同构的。这意味着存在一个可逆的线性映射 M ，可以将一个域中的元素转换到另一个域中。利用此特性，可以通过 AES-NI 硬件指令高效地完成求逆运算，从而极大地加速 SM4 的 S 盒计算。

5 C++ 代码实现与性能展示

以下为完整的 C++ 代码，用于演示基础实现与 T-Table 优化实现，并对比其性能。

5.1 基础实现与优化实现核心代码

以下代码包含了包含 SM4 的基础实现和基于 T-Table 的优化实现。

Listing 1: sm4_impl.cpp - 密钥扩展与加密函数

```
1 #include "sm4_shared.h"

2

3 // 循环左移宏
4 #define ROTL(x, n) (((x) << (n)) | ((x) >> (32 - (n))))
5
6 // --- 密钥扩展相关函数 ---
7
8 // 密钥扩展中的非线性变换
9 uint32_t tau_key(uint32_t A) {
10     uint8_t b[4];
11     from_uint32(A, b);
12     b[0] = SM4_SBOX[b[0]];
13     b[1] = SM4_SBOX[b[1]];
14     b[2] = SM4_SBOX[b[2]];
15     b[3] = SM4_SBOX[b[3]];
16     return to_uint32(b);
17 }

18
19 // 密钥扩展中的线性变换 L'
20 uint32_t L_prime(uint32_t B) {
21     return B ^ ROTL(B, 13) ^ ROTL(B, 23);
22 }

23
24 // SM4 密钥扩展函数
25 void sm4_set_key(const uint8_t* key, uint32_t* rk) {
26     uint32_t K[4];
27     K[0] = to_uint32(key);
28     K[1] = to_uint32(key + 4);
29     K[2] = to_uint32(key + 8);
30     K[3] = to_uint32(key + 12);
31 }
```

```
32     K[0] ^= FK[0];
33     K[1] ^= FK[1];
34     K[2] ^= FK[2];
35     K[3] ^= FK[3];
36
37     for (int i = 0; i < SM4_NUM_ROUNDS; ++i) {
38         rk[i] = K[0] ^ L_prime(tau_key(K[1] ^ K[2] ^ K[3] ^ CK[i]));
39         // 更新K数组作为滑动窗口
40         K[0] = K[1];
41         K[1] = K[2];
42         K[2] = K[3];
43         K[3] = rk[i];
44     }
45 }
46
47
48 // --- 基础实现 ---
49
50 // 加密/解密中的线性变换 L
51 uint32_t L(uint32_t B) {
52     return B ^ ROTL(B, 2) ^ ROTL(B, 10) ^ ROTL(B, 18) ^ ROTL(B, 24);
53 }
54
55 // 加密/解密中的非线性变换
56 uint32_t tau(uint32_t A) {
57     uint8_t b[4];
58     from_uint32(A, b);
59     b[0] = SM4_SBOX[b[0]];
60     b[1] = SM4_SBOX[b[1]];
61     b[2] = SM4_SBOX[b[2]];
62     b[3] = SM4_SBOX[b[3]];
63     return to_uint32(b);
64 }
65
66 // 复合变换 T
67 uint32_t T(uint32_t V) {
68     return L(tau(V));
69 }
70
```

```
71 // SM4 基础加密函数
72 void sm4_encrypt_basic(const uint8_t* in, uint8_t* out, const
73     uint32_t* rk) {
74     uint32_t X[4];
75     X[0] = to_uint32(in);
76     X[1] = to_uint32(in + 4);
77     X[2] = to_uint32(in + 8);
78     X[3] = to_uint32(in + 12);
79
80     for (int i = 0; i < SM4_NUM_ROUNDS; ++i) {
81         uint32_t temp = X[1] ^ X[2] ^ X[3] ^ rk[i];
82         uint32_t X_new = X[0] ^ T(temp);
83         X[0] = X[1];
84         X[1] = X[2];
85         X[2] = X[3];
86         X[3] = X_new;
87     }
88
89     // 反序变换
90     from_uint32(X[3], out);
91     from_uint32(X[2], out + 4);
92     from_uint32(X[1], out + 8);
93     from_uint32(X[0], out + 12);
94 }
95
96 // SM4 基础解密函数
97 void sm4_decrypt_basic(const uint8_t* in, uint8_t* out, const
98     uint32_t* rk) {
99     uint32_t X[4];
100    X[0] = to_uint32(in);
101    X[1] = to_uint32(in + 4);
102    X[2] = to_uint32(in + 8);
103    X[3] = to_uint32(in + 12);
104
105    // 解密使用反序的轮密钥
106    for (int i = 0; i < SM4_NUM_ROUNDS; ++i) {
107        uint32_t temp = X[1] ^ X[2] ^ X[3] ^ rk[31 - i];
108        uint32_t X_new = X[0] ^ T(temp);
109        X[0] = X[1];
110    }
111 }
```

```
108     X[1] = X[2];
109     X[2] = X[3];
110     X[3] = X_new;
111 }
112
113 // 反序变换
114 from_uint32(X[3], out);
115 from_uint32(X[2], out + 4);
116 from_uint32(X[1], out + 8);
117 from_uint32(X[0], out + 12);
118 }
119
120 // --- T-Table 优化实现 ---
121
122 static uint32_t T_TABLE[4][256];
123 static bool t_tables_generated = false;
124
125 void generate_ttables() {
126     if (t_tables_generated) return;
127     for (int i = 0; i < 256; ++i) {
128         uint32_t s_val = SM4_SBOX[i];
129         T_TABLE[0][i] = L(s_val << 24);
130         T_TABLE[1][i] = L(s_val << 16);
131         T_TABLE[2][i] = L(s_val << 8);
132         T_TABLE[3][i] = L(s_val);
133     }
134     t_tables_generated = true;
135 }
136
137 // 使用T-Table的复合变换
138 uint32_t T_ttable(uint32_t V) {
139     return T_TABLE[0][(V >> 24) & 0xFF] ^
140             T_TABLE[1][(V >> 16) & 0xFF] ^
141             T_TABLE[2][(V >> 8) & 0xFF] ^
142             T_TABLE[3][(V) & 0xFF];
143 }
144
145 // SM4 T-Table 加密函数
146 void sm4_encrypt_ttable(const uint8_t* in, uint8_t* out, const
```

```
147     uint32_t* rk) {
148
149     if (!t_tables_generated) generate_ttables();
150
151     uint32_t X[4];
152     X[0] = to_uint32(in);
153     X[1] = to_uint32(in + 4);
154     X[2] = to_uint32(in + 8);
155     X[3] = to_uint32(in + 12);
156
157     for (int i = 0; i < 32; i++) {
158         uint32_t temp = X[1] ^ X[2] ^ X[3] ^ rk[i];
159         uint32_t X_new = X[0] ^ T_ttable(temp);
160         X[0] = X[1];
161         X[1] = X[2];
162         X[2] = X[3];
163         X[3] = X_new;
164     }
165
166     from_uint32(X[3], out);
167     from_uint32(X[2], out + 4);
168     from_uint32(X[1], out + 8);
169     from_uint32(X[0], out + 12);
170 }
171 // SM4 T-Table 解密函数
172 void sm4_decrypt_ttable(const uint8_t* in, uint8_t* out, const
173     uint32_t* rk) {
174
175     if (!t_tables_generated) generate_ttables();
176
177     uint32_t X[4];
178     X[0] = to_uint32(in);
179     X[1] = to_uint32(in + 4);
180     X[2] = to_uint32(in + 8);
181     X[3] = to_uint32(in + 12);
182
183     for (int i = 0; i < 32; i++) {
184         uint32_t temp = X[1] ^ X[2] ^ X[3] ^ rk[31 - i];
185         uint32_t X_new = X[0] ^ T_ttable(temp);
186         X[0] = X[1];
```

```
184     X[1] = X[2];
185     X[2] = X[3];
186     X[3] = X_new;
187 }
188
189 from_uint32(X[3], out);
190 from_uint32(X[2], out + 4);
191 from_uint32(X[1], out + 8);
192 from_uint32(X[0], out + 12);
193 }
```

5.2 性能测试主函数

以下代码用于正确性验证和性能测试，并展示优化效果。

Listing 2: main.cpp - 性能测试与结果展示

```
194 #include "sm4_shared.h"
195 #include <chrono>
196 #include <vector>
197 #include <memory.h> // For memcmp
198
199 // 性能和正确性测试函数
200 void benchmark_and_verify() {
201     // 测试数据来自 GB/T 32907-2016 标准附录A
202     const uint8_t key[16] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0xab, 0
203         xcd, 0xef, 0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0x10 };
204     const uint8_t plaintext[16] = { 0x01, 0x23, 0x45, 0x67, 0x89, 0
205         xab, 0xcd, 0xef, 0xfe, 0xdc, 0xba, 0x98, 0x76, 0x54, 0x32, 0
206         x10 };
207     const uint8_t expected_ciphertext[16] = { 0x68, 0x1e, 0xdf, 0x34
208         , 0xd2, 0x06, 0x96, 0x5e, 0x86, 0xb3, 0xe9, 0x4f, 0x53, 0x6e,
209         0x42, 0x46 };
210
211     uint32_t round_keys[SM4_NUM_ROUNDS];
212     uint8_t basic_ct[SM4_BLOCK_SIZE], basic_pt[SM4_BLOCK_SIZE];
213     uint8_t ttable_ct[SM4_BLOCK_SIZE], ttable_pt[SM4_BLOCK_SIZE];
214
215     // 生成轮密钥
216     sm4_set_key(key, round_keys);
```

```
213 // --- 正确性验证 ---
214 std::cout << "--- Correctness Verification ---" << std::endl;
215 std::cout << "Plaintext: " ; print_hex(plaintext,
216 16);
217 std::cout << "Expected Ciphertext: " ; print_hex(
218 expected_ciphertext, 16);

219 // 基础版加解密
220 sm4_encrypt_basic(plaintext, basic_ct, round_keys);
221 std::cout << "Basic Encrypted: " ; print_hex(basic_ct,
222 16);
223 sm4_decrypt_basic(basic_ct, basic_pt, round_keys);
224 std::cout << "Basic Decrypted: " ; print_hex(basic_pt,
225 16);

226 // T-Table版加解密
227 sm4_encrypt_ttable(plaintext, ttable_ct, round_keys);
228 std::cout << "T-Table Encrypted: " ; print_hex(ttable_ct,
229 16);
230 sm4_decrypt_ttable(ttable_ct, ttable_pt, round_keys);
231 std::cout << "T-Table Decrypted: " ; print_hex(ttable_pt,
232 16);

233 // 比较结果
234 bool ok = true;
235 if (memcmp(basic_ct, expected_ciphertext, 16) != 0) {
236     std::cout << "[FAIL] Basic encryption output does not match
237     expected value." << std::endl;
238     ok = false;
239 }
240 if (memcmp(ttable_ct, expected_ciphertext, 16) != 0) {
241     std::cout << "[FAIL] T-Table encryption output does not
242     match expected value." << std::endl;
243     ok = false;
244 }
245 if (memcmp(basic_pt, plaintext, 16) != 0) {
246     std::cout << "[FAIL] Basic decryption failed." << std::endl;
247     ok = false;
248 }
```

```
244     if (memcmp(ttable_pt, plaintext, 16) != 0) {
245         std::cout << "[FAIL] T-Table decryption failed." << std::
246             endl;
247         ok = false;
248     }
249     if (ok) {
250         std::cout << "[PASS] All correctness checks passed!" << std
251             ::endl;
252     }
253     std::cout << std::endl;
254
255
256 // --- 性能测试 ---
257 std::cout << "--- Performance Benchmark ---" << std::endl;
258 const int num_iterations = 2000000; // 增加迭代次数以获得更稳定的
259 // 结果
260 uint8_t temp_buffer[SM4_BLOCK_SIZE]; // 避免编译器优化掉循环
261
262 // 测试基础版
263 auto start_basic = std::chrono::high_resolution_clock::now();
264 for (int i = 0; i < num_iterations; ++i) {
265     sm4_encrypt_basic(plaintext, temp_buffer, round_keys);
266 }
267 auto end_basic = std::chrono::high_resolution_clock::now();
268 std::chrono::duration<double, std::milli> duration_basic =
269     end_basic - start_basic;
270 double gb_per_sec_basic = (double)num_iterations *
271     SM4_BLOCK_SIZE / (duration_basic.count() / 1000.0) / (1024 *
272     1024 * 1024);
273 std::cout << "Basic Implementation (" << num_iterations << "
274     blocks): "
275     << duration_basic.count() << " ms (" << gb_per_sec_basic <<
276     " GB/s)" << std::endl;
277
278 // 测试T-Table版
279 auto start_ttable = std::chrono::high_resolution_clock::now();
280 for (int i = 0; i < num_iterations; ++i) {
281     sm4_encrypt_ttable(plaintext, temp_buffer, round_keys);
282 }
```

```

275     auto end_ttable = std::chrono::high_resolution_clock::now();
276     std::chrono::duration<double, std::milli> duration_ttable =
277         end_ttable - start_ttable;
278     double gb_per_sec_ttable = (double)num_iterations *
279         SM4_BLOCK_SIZE / (duration_ttable.count() / 1000.0) / (1024 *
280         1024 * 1024);
281     std::cout << "T-Table Optimized (" << num_iterations << " blocks
282         ): "
283         << duration_ttable.count() << " ms (" << gb_per_sec_ttable
284         << " GB/s)" << std::endl;
285
286     double improvement = (duration_basic.count() - duration_ttable.
287         count()) / duration_basic.count() * 100.0;
288     std::cout << "\nOptimization Effect (T-Table vs Basic):" << std
289         ::endl;
290     std::cout << " - Speedup: " << std::fixed << std::setprecision
291         (2) << duration_basic.count() / duration_ttable.count() << "x
292         " << std::endl;
293     std::cout << " - Time Reduction: " << std::fixed << std::
294         setprecision(2) << improvement << "%" << std::endl;
295 }
296
297 int main() {
298     benchmark_and_verify();
299     return 0;
300 }
```

在头文件中，定义通用常量、S 盒、密钥扩展参数以及辅助函数。

Listing 3: sm4_shared.h

```

291 #pragma once
292 #include <cstdint>
293 #include <vector>
294 #include <string>
295 #include <iostream>
296 #include <iomanip>
297
298 // 定义SM4算法的核心参数
299 constexpr int SM4_NUM_ROUNDS = 32;
300 constexpr int SM4_KEY_SIZE = 16;    // 128 bits
```

```
301 constexpr int SM4_BLOCK_SIZE = 16; // 128 bits
302
303 // SM4 S-Box
304 const uint8_t SM4_SBOX[256] = {
305     0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14
306         , 0xc2, 0x28, 0xfb, 0x2c, 0x05,
307     0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13
308         , 0x26, 0x49, 0x86, 0x06, 0x99,
309     0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b
310         , 0x43, 0xed, 0xcf, 0xac, 0x62,
311     0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94
312         , 0xfa, 0x75, 0x8f, 0x3f, 0xa6,
313     0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c
314         , 0x19, 0xe6, 0x85, 0x4f, 0xa8,
315     0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f
316         , 0x4b, 0x70, 0x56, 0x9d, 0x35,
317     0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c
318         , 0x3b, 0x01, 0x21, 0x78, 0x87,
319     0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02
320         , 0xe7, 0xa0, 0xc4, 0xc8, 0x9e,
321     0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2
322         , 0xce, 0xf9, 0x61, 0x15, 0xa1,
323     0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32
324         , 0x30, 0xf5, 0x8c, 0xb1, 0xe3,
325     0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23
326         , 0xab, 0xd, 0x53, 0x4e, 0x6f,
327     0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a
328         , 0x72, 0x6d, 0x6c, 0x5b, 0x51,
329     0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c
330         , 0x41, 0x1f, 0x10, 0x5a, 0xd8,
331     0xa, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0
332         , 0x12, 0xb8, 0xe5, 0xb4, 0xb0,
333     0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1
334         , 0x09, 0xc5, 0x6e, 0xc6, 0x84,
335     0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f
336         , 0x3e, 0xd7, 0xcb, 0x39, 0x48 };
337
338 // 密钥扩展中的系统参数 FK 和固定参数 CK
339 const uint32_t FK[4] = { 0xa3b1bac6, 0x56aa3350, 0x677d9197, 0
340 }
```

```
    xb27022dc };
```

```
324 const uint32_t CK[32] = {  
325     0x00070e15, 0x1c232a31, 0x383f464d, 0x545b6269,  
326     0x70777e85, 0x8c939aa1, 0xa8afb6bd, 0xc4cbd2d9,  
327     0xe0e7eef5, 0xfc030a11, 0x181f262d, 0x343b4249,  
328     0x50575e65, 0x6c737a81, 0x888f969d, 0xa4abb2b9,  
329     0xc0c7ced5, 0xdce3eaf1, 0xf8ff060d, 0x141b2229,  
330     0x30373e45, 0x4c535a61, 0x686f767d, 0x848b9299,  
331     0xa0a7aeb5, 0xbcc3cad1, 0xd8dfe6ed, 0xf4fb0209,  
332     0x10171e25, 0x2c333a41, 0x484f565d, 0x646b7279 };
```

```
333  
334 // 辅助函数：字节序转换（大端）  
335 inline uint32_t to_uint32(const uint8_t* p) {  
336     return ((uint32_t)p[0] << 24) | ((uint32_t)p[1] << 16) | ((  
337         uint32_t)p[2] << 8) | p[3];  
338 }  
339 inline void from_uint32(uint32_t v, uint8_t* p) {  
340     p[0] = (uint8_t)(v >> 24);  
341     p[1] = (uint8_t)(v >> 16);  
342     p[2] = (uint8_t)(v >> 8);  
343     p[3] = (uint8_t)v;  
344 }  
345  
346 // 辅助函数：打印16进制数据  
347 inline void print_hex(const uint8_t* data, size_t len) {  
348     for (size_t i = 0; i < len; ++i) {  
349         std::cout << std::hex << std::setw(2) << std::setfill('0')  
            << (int)data[i];  
350     }  
351     std::cout << std::dec << std::endl;  
352 }  
353  
354 // 声明所有实现中的函数，以便在 main.cpp 中调用  
355 void sm4_set_key(const uint8_t* key, uint32_t* rk);  
356 void sm4_encrypt_basic(const uint8_t* in, uint8_t* out, const  
    uint32_t* rk);  
357 void sm4_decrypt_basic(const uint8_t* in, uint8_t* out, const  
    uint32_t* rk);
```

```

358 void sm4_encrypt_ttable(const uint8_t* in, uint8_t* out, const
   uint32_t* rk);
359 void sm4_decrypt_ttable(const uint8_t* in, uint8_t* out, const
   uint32_t* rk);

```

5.3 性能展示

```

--- Correctness Verification ---
Plaintext:          0123456789abcdefedcba9876543210
Expected Ciphertext: 681edf34d206965e86b3e94f536e4246
Basic Encrypted:    681edf34d206965e86b3e94f536e4246
Basic Decrypted:    0123456789abcdefedcba9876543210
T-Table Encrypted: 681edf34d206965e86b3e94f536e4246
T-Table Decrypted: 0123456789abcdefedcba9876543210
[PASS] All correctness checks passed!

--- Performance Benchmark ---
Basic Implementation (2000000 blocks): 1290.05 ms (0.0231017 GB/s)
T-Table Optimized (2000000 blocks): 353.472 ms (0.0843132 GB/s)

Optimization Effect (T-Table vs Basic):
- Speedup: 3.65x
- Time Reduction: 72.60%

```

D:\Study\study\2024-2025 Semester2\practice\project\lab1\Project1\x64\Debug\Project1.exe [进程 6152]已退出，代码为 0 (0x0)。按任意键关闭此窗口 . . .

Correctness Verification 部分的结果清晰地表明，无论是基础实现还是 T-Table 优化实现，加密结果与解密结果都正确，证明了代码的正确性。

Performance Benchmark 部分展示了优化的实际效果，基础实现耗时约 1290.05ms，T-Table 优化实现仅耗时 353.472ms，优化后的吞吐率从 0.023GB/s 提升到了 0.084GB/s，处理数据的效率大大提高。

加速比达到了 3.65 倍，这意味着优化后的版本比基础版本的速度快了近 4 倍。执行时间减少了 72.6%，这是一个巨大的性能提升，说明优化策略非常有效。

6 SM4-GCM

SM4-GCM 为对称块密码 SM4 提供了一种高效且安全的运行模式。它不仅能保证数据的机密性，还能通过生成认证标签来确保数据的完整性和真实性。此实现包含两个关键部分：

- GCTR: 使用 SM4 加密一个递增的计数器来生成密钥流，然后与明文进行异或操作以产生密文。
- GHASH: 一个在伽罗瓦域 $GF(2^{128})$ 上的哈希函数，用于处理附加认证数据 (AAD) 和密文，以生成最终的认证标签。

该实现的显著特点是采用了预算算表来优化 $GF(2^{128})$ 乘法，从而大幅提升了 GHASH 的性能。

6.1 主要接口

实现的功能通过 sm4_gcm.h 中定义的几个关键函数暴露给用户：

- sm4_gcm_init(key, H_out): 初始化 GCM 模式。此函数接收一个 128 位密钥，生成哈希子密钥 H ，并构建用于加速 GF 乘法的预算算表。
- sm4_gcm_encrypt(...): 执行加密和认证操作。它输入密钥、初始化向量 (IV)、附加数据 (AAD) 和明文，输出密文和认证标签。
- sm4_gcm_decrypt(...): 执行解密和验证操作。它输入密钥、IV、AAD、密文和待验证的标签。只有在标签验证通过时，才会返回解密后的明文。

6.2 关键数据结构：预算算表

为了优化 GHASH 中的 $GF(2^{128})$ 乘法，实现使用了一个全局的预算算表：

```
1 // 预算算的 GF(2^128) 乘法表
2 uint64_t H_TABLE[16][256] = { {0} };
3 bool h_tables_initialized = false;
```

这个 H_TABLE 是一个二维数组，用于存储哈希子密钥 H 与不同字节值相乘的结果，从而将耗时的逐位乘法转换成高效的查表和异或操作。

6.3 密钥与初始化

GCM 模式的初始化过程在 sm4_gcm_init 函数中完成。

1. 生成哈希子密钥 (H): 哈希子密钥 H 是通过使用 SM4 算法加密一个 128 位的零块得到的，即 $H = E_K(0^{128})$ 。

Listing 4: H 的生成过程

```
1 // 生成哈希子密钥 H = E_K(0^128)
2 uint8_t H[16] = { 0 };
3 uint32_t round_keys[SM4_NUM_ROUNDS];
4
5 sm4_set_key(key, round_keys);
6 sm4_encrypt_ttable(H, H, round_keys);
```

2. 初始化预算算表：随后，代码调用 init_gf_tables(H) 函数，使用生成的 H 来填充 H_TABLE 。这一步是性能优化的核心，仅在首次操作时执行一次。

6.4 伽罗瓦域 ($GF(2^{128})$) 乘法

GHASH 的核心是 $GF(2^{128})$ 乘法，其不可约多项式为 $x^{128} + x^7 + x^2 + x + 1$ 。代码中提供了两种实现方式。

6.4.1 基本实现

函数 `gf_multiply_basic` 提供了一个基础的、基于移位和异或的乘法实现。它严格遵循“无进位乘法”的定义，但性能较低，主要用于初始化预算表。

6.4.2 优化实现

函数 `gf_multiply` 是一个高度优化的实现，它利用预算表 `H_TABLE` 来完成乘法。其原理是将一个 128 位的乘数 X 分解为 16 个字节 X_0, X_1, \dots, X_{15} ，然后通过查表获得每个字节对应的预算结果，最后将所有结果异或起来得到最终乘积。

Listing 5: 使用预算表的 GF 乘法优化实现

```
1 void gf_multiply(const uint8_t* X, const uint8_t* Y, uint8_t* out) {
2     if (!h_tables_initialized) {
3         // 如果表格未初始化，则使用基本实现
4         gf_multiply_basic(X, Y, out);
5         return;
6     }
7
8     // 使用表格优化的乘法实现
9     uint64_t Z_high = 0;
10    uint64_t Z_low = 0;
11
12    for (int i = 0; i < 16; i++) {
13        Z_high ^= H_TABLE[i][X[i]];
14        Z_low ^= H_TABLE[i + 8][X[i]];
15    }
16
17    words_to_block(Z_high, Z_low, out);
18 }
```

注意：此处的实现使用 Y 来构建表格，然后用 X 的字节进行查表。

6.5 GHASH 计算

ghash 函数负责计算认证哈希值。它按顺序处理 AAD 和密文：

1. 初始化一个 128 位的累加器 X 为零。
2. 将 AAD 数据分块（128 位），与累加器 X 异或后，再与哈希子密钥 H 进行 GF 乘法。不足一块的数据会先填充零。
3. 以同样的方式处理密文数据。
4. 最后，将 AAD 和密文的长度（以位为单位）编码成一个 128 位块，与累加器 X 异或后，再进行一次 GF 乘法。
5. 最终累加器 X 的值就是 GHASH 的输出，记为 S 。

6.6 计数器与加密

6.6.1 初始计数器 J_0 的生成

初始计数器 J_0 由 generate_J0 函数生成。

- 如果 IV 长度为 96 位（12 字节）， J_0 直接由 IV 拼接一个 32 位的块 0x00000001 构成。
- 如果 IV 长度不是 96 位，则需要通过对 IV 进行 GHASH 计算来生成 J_0 。

6.6.2 加密过程

加密过程在 sm4_gcm_encrypt 中实现，遵循 GCTR 模式：

1. 从 J_0 开始，派生出加密用的第一个计数器 counter（通常是 J_0+1 ），由 increment_counter 函数完成递增。
2. 在一个循环中，对当前的 counter 值使用 SM4 进行加密： $E_K(\text{counter})$ 。
3. 将加密结果与明文块进行异或，得到密文块。
4. 递增 counter，为下一个明文块做准备。

6.7 认证标签生成与验证

6.7.1 标签生成

认证标签 T 由 generate_tag 函数生成，其计算公式为：

$$T = S \oplus E_K(J_0)$$

其中 S 是 ghash 函数的输出。代码实现如下：

Listing 6: 认证标签生成

```
1 void generate_tag(const uint8_t* J0, const uint8_t* S, const
2     uint32_t* round_keys, uint8_t* tag) {
3     uint8_t mask[16];
4     sm4_encrypt_ttable(J0, mask, round_keys); // 计算 E_K(J0)
5
6     // 与 GHASH 输出异或得到标签
7     for (int i = 0; i < 16; i++) {
8         tag[i] = mask[i] ^ S[i];
9     }
}
```

6.7.2 标签验证

在解密函数 sm4_gcm_decrypt 中，程序首先使用收到的 AAD 和密文重新计算一个期望的认证标签 calculated_tag。然后，将这个计算出的标签与用户提供的标签 tag 进行逐字节比较。

- 如果两个标签完全匹配，则认证通过，程序继续执行解密步骤。
- 如果不匹配，则认证失败。函数会立即返回 false，并且不会输出任何解密后的数据，以防止对非法数据的处理。

6.8 测试与验证

main.cpp 文件中提供了完整的正确性和性能测试用例。

- test_sm4_gcm_correctness(): 使用一组固定的密钥、IV、AAD 和明文来测试加密和解密流程是否正确。它还通过故意篡改标签 ($tag[0] \hat{=} 1;$) 来验证认证失败的逻辑是否能被正确触发。
- benchmark_sm4_gcm(): 在不同数据大小下对加密和解密进行性能基准测试，并报告吞吐量 (MB/s)。

```
--- SM4-GCM 正确性测试 ---
加密结果: 成功
明文: 0123456789abcdeffedcba98765432100123456789abcdeffedcba9876543210
密文: 54025df6381a64f09a5c3933ad08544589da6ccac9fa7350b86813b7e6b9216b
认证标签: 83720e2558fc50def97202cf35e50258
解密结果: 成功
解密后的数据: 0123456789abcdeffedcba98765432100123456789abcdeffedcba9876543210
解密数据与原始明文匹配
使用无效标签解密: 正确拒绝

--- SM4-GCM 性能基准测试 ---
数据大小 | 加密速度 | 解密速度 | 综合速度
-----|-----|-----|-----
00000016B | 9.71 MB/s | 9.85 MB/s | 9.78 MB/s
00000064B | 24.84 MB/s | 24.82 MB/s | 24.83 MB/s
00000256B | 41.72 MB/s | 41.42 MB/s | 41.57 MB/s
00001024B | 50.36 MB/s | 49.33 MB/s | 49.84 MB/s
00004096B | 52.67 MB/s | 52.39 MB/s | 52.53 MB/s
```

可以看到，成功实现了加解密以及标签解密的验证。

7 结论

SM4 算法的设计使其可以通过多种方式进行软件优化。本文成功实现了 SM4 的基础版本和 T-Table 优化版本，并通过性能测试验证了优化的有效性。T-Table 优化是一种有效的折衷方案，用适度的内存开销换取了显著的速度提升。而报告中探讨的更高级优化方法，如利用 AES-NI 同构特性，则展示了如何结合深刻的数学理论与底层硬件特性，以实现极致的密码学算法性能。