



山东大学
SHANDONG UNIVERSITY

Project4 SM3 实现和优化

学院 网络空间安全

专业 网络空间安全

学号 202200460149

班级姓名 网安 22.1 班张弛

2025 年 8 月 6 日

目录

1	实验任务	1
2	SM3 实现	2
2.1	SM3 概述	2
2.2	消息填充	2
2.3	消息扩展	3
2.4	迭代压缩	4
2.5	输出结果	6
3	SM3 优化	7
3.1	优化 1: 宏定义展开	7
3.2	优化 2: 减少函数调用	10
3.3	优化 3: 宏定义 plus 版	11
3.4	优化 4: SIMD 优化消息扩展	12
3.5	优化总对比	14
4	参考链接	14

1 实验任务

Project 4: SM3 的软件实现与优化

a): 与 Project 1 类似, 从 SM3 的基本软件实现出发, 参考付勇老师的 PPT, 不断对 SM3 的软件执行效率进行改进

b): 基于 sm3 的实现, 验证 length-extension attack

c): 基于 sm3 的实现, 根据 RFC6962 构建 Merkle 树 (10w 叶子节点), 并构建叶子的存在性证明和不存在性证明

2 SM3 实现

2.1 SM3 概述

SM3 密码杂凑算法是中国国家密码管理局 2010 年公布的中国商用密码杂凑算法标准。该算法于 2012 年发布为密码行业标准 (GM/T 0004-2012)，2016 年发布为国家密码杂凑算法标准 (GB/T 32905-2016)。

SM3 采用 Merkle-Damgard 结构。消息分组长度为 512 位，摘要值长度为 256 位。

整个算法的执行过程可以概括成四个步骤：消息填充、消息扩展、迭代压缩、输出结果。

2.2 消息填充

SM3 的消息扩展步骤是以 512 位的数据分组作为输入的。因此，我们需要在一开始就把数据长度填充至 512 位的倍数。数据填充规则和 MD5 一样，具体步骤如下：

1、先填充一个“1”，后面加上 k 个“0”。其中 k 是满足 $(n+1+k) \bmod 512 = 448$ 的最小正整数。

2、追加 64 位的数据长度 (bit 为单位，大端序存放 1。)

一个例子如下所示：

例如：对消息 01100001 01100010 01100011，其长度 $l=24$ ，经填充得到比特串：

01100001 01100010 01100011 1 $\overbrace{00 \dots 00}^{423 \text{ 比特}}$ $\overbrace{00 \dots 011000}^{64 \text{ 比特}}$
l 的二进制表示

图 1 消息填充

代码实现如下所示：

```
1 // 消息填充
2 vector<u8> message_padding(const vector<u8>& msg) {
3     size_t len = msg.size() * 8;
4     vector<u8> padded = msg;
5
6     // 追加 '1' bit
7     padded.push_back(0x80);
8 }
```

```

9      // 追加 '0' bits
10     while ((padded.size() * 8) % 512 != 448)
11         padded.push_back(0x00);
12
13     // 添加64位长度
14     for (int i = 7; i >= 0; --i)
15         padded.push_back((u8)((len >> (i * 8)) & 0xFF));
16
17     return padded;
18 }

```

2.3 消息扩展

SM3 这一步骤产生 132 个消息字。（一个消息字的长度为 32 位，4 个字节，8 个 16 进制数字）概括来说，先将一个 512 位数据分组划分为 16 个消息字，并且作为生成的 132 个消息字的前 16 个。再用这 16 个消息字递推生成剩余的 116 个消息字。

在最终得到的 132 个消息字中，前 68 个消息字构成数列 $\{W_j\}$ ，后 64 个消息字构成数列 $\{W'_j\}$ ，其中下标 j 从 0 开始计数。

5.3.2 消息扩展

将消息分组 $B^{(i)}$ 按以下方法扩展生成 132 个字 $W_0, W_1, \dots, W_{67}, W'_0, W'_1, \dots, W'_{63}$ ，用于压缩函数 CF ：

a) 将消息分组 $B^{(i)}$ 划分为 16 个字 W_0, W_1, \dots, W_{15} 。

b) FOR $j=16$ TO 67

$$W_j \leftarrow P_1(W_{j-16} \oplus W_{j-9} \oplus (W_{j-3} \ll 15)) \oplus (W_{j-13} \ll 7) \oplus W_{j-6}$$

ENDFOR

c) FOR $j=0$ TO 63

$$W'_j = W_j \oplus W_{j+4}$$

ENDFOR

图 2 消息扩展

代码实现如下所示：

```

1  // 消息扩展
2  void message_expansion(const u8* block, u32 W[68], u32 W1[64]) {
3      for (int i = 0; i < 16; ++i) {
4          W[i] = (block[4 * i] << 24) |
5              (block[4 * i + 1] << 16) |
6              (block[4 * i + 2] << 8) |
7              (block[4 * i + 3]);
8      }
9  }

```

```

10     for (int i = 16; i < 68; ++i) {
11         u32 tmp = P1(W[i - 16] ^ W[i - 9] ^ left_rotate(W[i - 3], 15));
12         W[i] = tmp ^ left_rotate(W[i - 13], 7) ^ W[i - 6];
13     }
14
15     for (int i = 0; i < 64; ++i) {
16         W1[i] = W[i] ^ W[i + 4];
17     }
18 }
    
```

2.4 迭代压缩

SM3 使用消息扩展得到的消息字进行迭代压缩运算：

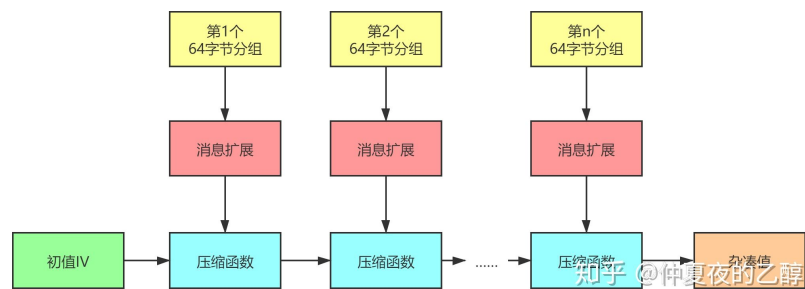


图 3 迭代压缩

初值 IV 被放在 A、B、C、D、E、F、G、H 八个 32 位变量中。整个算法中最核心、也最复杂的地方就在于压缩函数。压缩函数将这八个变量进行 64 轮相同的计算，64 轮的计算过程如下图所示：

令 A, B, C, D, E, F, G, H 为寄存器, $SS1, SS2, TT1, TT2$ 为中间变量, 压缩函数 $V^{i+1} = CF(V^{(i)}, B^{(i)})$, $0 \leq i \leq n-1$ 。计算过程描述如下:

```

 $ABCDEFGH \leftarrow V^{(i)}$ 
FOR j=0 TO 63
     $SS1 \leftarrow ((A \lll 12) + E + (T_j \lll j)) \lll 7$ 
     $SS2 \leftarrow SS1 \oplus (A \lll 12)$ 
     $TT1 \leftarrow FF_j(A, B, C) + D + SS2 + W'_j$ 
     $TT2 \leftarrow GG_j(E, F, G) + H + SS1 + W_j$ 
     $D \leftarrow C$ 
     $C \leftarrow B \lll 9$ 
     $B \leftarrow A$ 
     $A \leftarrow TT1$ 
     $H \leftarrow G$ 
     $G \leftarrow F \lll 19$ 
     $F \leftarrow E$ 
     $E \leftarrow P_0(TT2)$ 
ENDFOR
 $V^{(i+1)} \leftarrow ABCDEFGH \oplus V^{(i)}$ 
其中, 字的存储为大端(big-endian)格式。
    
```

图 4 轮操作

其中, 在更新 SS1 时式子中的常量如图所示:

$$T_j = \begin{cases} 79cc4519 & 0 \leq j \leq 15 \\ 7a879d8a & 16 \leq j \leq 63 \end{cases}$$

图 5 T 表常量

代码实现如下所示:

```

1 // 压缩函数
2 void compression(u32 V[8], const u8* block) {
3     u32 W[68], W1[64];
4     message_expansion(block, W, W1);
5
6     u32 A = V[0], B = V[1], C = V[2], D = V[3];
7     u32 E = V[4], F = V[5], G = V[6], H = V[7];
8
9     for (int j = 0; j < 64; ++j) {
10         u32 SS1 = left_rotate((left_rotate(A, 12) + E + left_rotate(T[j], j
            % 32)) & 0xFFFFFFFF, 7);
11         u32 SS2 = SS1 ^ left_rotate(A, 12);
12         u32 TT1 = (FF(A, B, C, j) + D + SS2 + W1[j]) & 0xFFFFFFFF;
13         u32 TT2 = (GG(E, F, G, j) + H + SS1 + W[j]) & 0xFFFFFFFF;
14
15         D = C;
16         C = left_rotate(B, 9);
17         B = A;
18         A = TT1;
    
```

```

19     H = G;
20     G = left_rotate(F, 19);
21     F = E;
22     E = PO(TT2);
23 }
24
25 V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
26 V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
27 }

```

2.5 输出结果

将得到的 A、B、C、D、E、F、G、H 八个变量拼接输出，就是 SM3 算法的输出，一共 256bit。

最后综合上述所有内容，进行完整的算法过程的代码实现如下所示：

```

1 // SM3 主函数
2 vector<u8> sm3_hash(const string& input) {
3     init_T();
4
5     // 初始 IV
6     u32 V[8] = {
7         0x7380166F, 0x4914B2B9, 0x172442D7, 0xDA8A0600,
8         0xA96F30BC, 0x163138AA, 0xE38DEE4D, 0xB0FB0E4E
9     };
10
11     vector<u8> msg_bytes = string_to_bytes(input);
12     vector<u8> padded = message_padding(msg_bytes);
13
14     size_t block_count = padded.size() / 64;
15
16     for (size_t i = 0; i < block_count; ++i) {
17         compression(V, &padded[i * 64]);
18     }
19
20     vector<u8> hash_result;
21     for (int i = 0; i < 8; ++i) {
22         hash_result.push_back((V[i] >> 24) & 0xFF);
23         hash_result.push_back((V[i] >> 16) & 0xFF);
24         hash_result.push_back((V[i] >> 8) & 0xFF);
25         hash_result.push_back(V[i] & 0xFF);
26     }

```

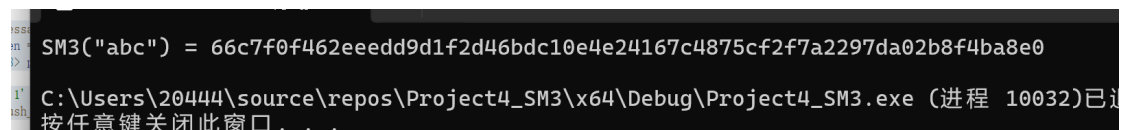


```

27
28     return hash_result;
29 }

```

然后我们测试字符串”abc”在 SM3 算法下的输出结果，最终运行结果如下所示：



```

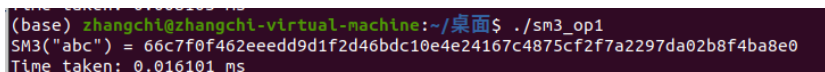
SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
C:\Users\20444\source\repos\Project4_SM3\x64\Debug\Project4_SM3.exe (进程 10032)已
按任意键关闭此窗口...

```

图 6 运行结果

这和官网上给出的结果一致，说明基本实现正确。

为了和后续的优化进行对比，我们查看 abc 在 SM3 算法下的运行时间：



```

(base) zhangchi@zhangchi-virtual-machine:~/桌面$ ./sm3_op1
SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
Time taken: 0.016101 ms

```

图 7 基础实现运行结果

3 SM3 优化

3.1 优化 1：宏定义展开

根据 ppt，我们知道在底层优化中，特别是嵌入式实现，推荐宏定义，核心代码，代码规模较小，几十行/一两百行的代码，推荐用宏定义；SM3 的设计正符合此，因此想把轮函数变成宏定义来展开，修改如下：

```

1 // 压缩轮次宏展开
2 #define COMPRESS_ROUND(j, A, B, C, D, E, F, G, H, W, W1) { \
3     u32 SS1 = left_rotate((left_rotate(A, 12) + E + left_rotate(T[j], j %
4         32)) & 0xFFFFFFFF, 7); \
5     u32 SS2 = SS1 ^ left_rotate(A, 12); \
6     u32 TT1 = (((j < 16) ? (A ^ B ^ C) : ((A & B) | (A & C) | (B & C))) + D
7         + SS2 + W1[j]) & 0xFFFFFFFF; \
8     u32 TT2 = (((j < 16) ? (E ^ F ^ G) : ((E & F) | (~E & G))) + H + SS1 + W
9         [j]) & 0xFFFFFFFF; \
10    D = C; C = left_rotate(B, 9); B = A; A = TT1; \
11    H = G; G = left_rotate(F, 19); F = E; E = P0(TT2); \
12 }

```

```

11 // 压缩函数
12 void compression(u32 V[8], const u8* block) {
13     u32 W[68], W1[64];
14     message_expansion(block, W, W1);
15
16     u32 A = V[0], B = V[1], C = V[2], D = V[3];
17     u32 E = V[4], F = V[5], G = V[6], H = V[7];
18
19     // 展开所有64轮
20     COMPRESS_ROUND(0, A, B, C, D, E, F, G, H, W, W1);
21     COMPRESS_ROUND(1, A, B, C, D, E, F, G, H, W, W1);
22     COMPRESS_ROUND(2, A, B, C, D, E, F, G, H, W, W1);
23     COMPRESS_ROUND(3, A, B, C, D, E, F, G, H, W, W1);
24     COMPRESS_ROUND(4, A, B, C, D, E, F, G, H, W, W1);
25     COMPRESS_ROUND(5, A, B, C, D, E, F, G, H, W, W1);
26     COMPRESS_ROUND(6, A, B, C, D, E, F, G, H, W, W1);
27     COMPRESS_ROUND(7, A, B, C, D, E, F, G, H, W, W1);
28     COMPRESS_ROUND(8, A, B, C, D, E, F, G, H, W, W1);
29     COMPRESS_ROUND(9, A, B, C, D, E, F, G, H, W, W1);
30     COMPRESS_ROUND(10, A, B, C, D, E, F, G, H, W, W1);
31     COMPRESS_ROUND(11, A, B, C, D, E, F, G, H, W, W1);
32     COMPRESS_ROUND(12, A, B, C, D, E, F, G, H, W, W1);
33     COMPRESS_ROUND(13, A, B, C, D, E, F, G, H, W, W1);
34     COMPRESS_ROUND(14, A, B, C, D, E, F, G, H, W, W1);
35     COMPRESS_ROUND(15, A, B, C, D, E, F, G, H, W, W1);
36     COMPRESS_ROUND(16, A, B, C, D, E, F, G, H, W, W1);
37     COMPRESS_ROUND(17, A, B, C, D, E, F, G, H, W, W1);
38     COMPRESS_ROUND(18, A, B, C, D, E, F, G, H, W, W1);
39     COMPRESS_ROUND(19, A, B, C, D, E, F, G, H, W, W1);
40     COMPRESS_ROUND(20, A, B, C, D, E, F, G, H, W, W1);
41     COMPRESS_ROUND(21, A, B, C, D, E, F, G, H, W, W1);
42     COMPRESS_ROUND(22, A, B, C, D, E, F, G, H, W, W1);
43     COMPRESS_ROUND(23, A, B, C, D, E, F, G, H, W, W1);
44     COMPRESS_ROUND(24, A, B, C, D, E, F, G, H, W, W1);
45     COMPRESS_ROUND(25, A, B, C, D, E, F, G, H, W, W1);
46     COMPRESS_ROUND(26, A, B, C, D, E, F, G, H, W, W1);
47     COMPRESS_ROUND(27, A, B, C, D, E, F, G, H, W, W1);
48     COMPRESS_ROUND(28, A, B, C, D, E, F, G, H, W, W1);
49     COMPRESS_ROUND(29, A, B, C, D, E, F, G, H, W, W1);
50     COMPRESS_ROUND(30, A, B, C, D, E, F, G, H, W, W1);
51     COMPRESS_ROUND(31, A, B, C, D, E, F, G, H, W, W1);
52     COMPRESS_ROUND(32, A, B, C, D, E, F, G, H, W, W1);
53     COMPRESS_ROUND(33, A, B, C, D, E, F, G, H, W, W1);

```

```

54 COMPRESS_ROUND(34, A, B, C, D, E, F, G, H, W, W1);
55 COMPRESS_ROUND(35, A, B, C, D, E, F, G, H, W, W1);
56 COMPRESS_ROUND(36, A, B, C, D, E, F, G, H, W, W1);
57 COMPRESS_ROUND(37, A, B, C, D, E, F, G, H, W, W1);
58 COMPRESS_ROUND(38, A, B, C, D, E, F, G, H, W, W1);
59 COMPRESS_ROUND(39, A, B, C, D, E, F, G, H, W, W1);
60 COMPRESS_ROUND(40, A, B, C, D, E, F, G, H, W, W1);
61 COMPRESS_ROUND(41, A, B, C, D, E, F, G, H, W, W1);
62 COMPRESS_ROUND(42, A, B, C, D, E, F, G, H, W, W1);
63 COMPRESS_ROUND(43, A, B, C, D, E, F, G, H, W, W1);
64 COMPRESS_ROUND(44, A, B, C, D, E, F, G, H, W, W1);
65 COMPRESS_ROUND(45, A, B, C, D, E, F, G, H, W, W1);
66 COMPRESS_ROUND(46, A, B, C, D, E, F, G, H, W, W1);
67 COMPRESS_ROUND(47, A, B, C, D, E, F, G, H, W, W1);
68 COMPRESS_ROUND(48, A, B, C, D, E, F, G, H, W, W1);
69 COMPRESS_ROUND(49, A, B, C, D, E, F, G, H, W, W1);
70 COMPRESS_ROUND(50, A, B, C, D, E, F, G, H, W, W1);
71 COMPRESS_ROUND(51, A, B, C, D, E, F, G, H, W, W1);
72 COMPRESS_ROUND(52, A, B, C, D, E, F, G, H, W, W1);
73 COMPRESS_ROUND(53, A, B, C, D, E, F, G, H, W, W1);
74 COMPRESS_ROUND(54, A, B, C, D, E, F, G, H, W, W1);
75 COMPRESS_ROUND(55, A, B, C, D, E, F, G, H, W, W1);
76 COMPRESS_ROUND(56, A, B, C, D, E, F, G, H, W, W1);
77 COMPRESS_ROUND(57, A, B, C, D, E, F, G, H, W, W1);
78 COMPRESS_ROUND(58, A, B, C, D, E, F, G, H, W, W1);
79 COMPRESS_ROUND(59, A, B, C, D, E, F, G, H, W, W1);
80 COMPRESS_ROUND(60, A, B, C, D, E, F, G, H, W, W1);
81 COMPRESS_ROUND(61, A, B, C, D, E, F, G, H, W, W1);
82 COMPRESS_ROUND(62, A, B, C, D, E, F, G, H, W, W1);
83 COMPRESS_ROUND(63, A, B, C, D, E, F, G, H, W, W1);
84
85 // 更新向量
86 V[0] ^= A; V[1] ^= B; V[2] ^= C; V[3] ^= D;
87 V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
88 }
89
90 \\将T表展开同时，减少初始化时间
91 static const u32 T[64] = {
92     0x79CC4519,0x79CC4519,0x79CC4519,0x79CC4519,
93     0x79CC4519,0x79CC4519,0x79CC4519,0x79CC4519,
94     0x79CC4519,0x79CC4519,0x79CC4519,0x79CC4519,
95     0x79CC4519,0x79CC4519,0x79CC4519,0x79CC4519,
96     0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,

```

```

97     0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
98     0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
99     0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
100    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
101    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
102    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
103    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
104    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
105    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
106    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A,
107    0x7A879D8A,0x7A879D8A,0x7A879D8A,0x7A879D8A
108 };

```

优化 1 运行结果如下所示：

```

(base) zhangchi@zhangchi-virtual-machine:~/桌面$ ./sm3_op1
SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
Time taken: 0.010292 ms

```

图 8 优化 1

我们可以看出来，优化结果的时间会短于基础实现的时间要短，但是提升不是很多，没有提升一倍。

3.2 优化 2：减少函数调用

由于在基础版的实现中有多次调用消息拓展函数等等，现在将其集成到一个函数中，并且尝试设置宏 `#define FF` 和 `#define GG` 实现布尔函数，减少函数调用开销，有助于编译器进行内联和指令级优化。修改部分的代码如下：

```

1  #define FF(j, x, y, z) ((j < 16) ? (x ^ y ^ z) : ((x & y) | (x & z) | (y & z)
    ))
2  #define GG(j, x, y, z) ((j < 16) ? (x ^ y ^ z) : ((x & y) | ((~x) & z)))
3
4  void compression(u32* V, const u8* block) {
5      u32 W[68], W1[64];
6
7      // 消息扩展
8      for (int i = 0; i < 16; ++i)
9          W[i] = (block[4*i] << 24) | (block[4*i+1] << 16) | (block[4*i+2] <<
            8) | block[4*i+3];
10
11     for (int i = 16; i < 68; ++i) {
12         u32 tmp = W[i-16] ^ W[i-9] ^ left_rotate(W[i-3], 15);

```

```

13     W[i] = P1(tmp) ^ left_rotate(W[i-13], 7) ^ W[i-6];
14 }
15
16 for (int i = 0; i < 64; ++i)
17     W1[i] = W[i] ^ W[i+4];
18
19 u32 A = V[0], Bv = V[1], C = V[2], D = V[3];
20 u32 E = V[4], F = V[5], G = V[6], H = V[7];
21
22 for (int j = 0; j < 64; ++j) {
23     u32 SS1 = left_rotate((left_rotate(A, 12) + E + left_rotate(T[j], j
        % 32)) & 0xFFFFFFFF, 7);
24     u32 SS2 = SS1 ^ left_rotate(A, 12);
25     u32 TT1 = (FF(j, A, Bv, C) + D + SS2 + W1[j]) & 0xFFFFFFFF;
26     u32 TT2 = (GG(j, E, F, G) + H + SS1 + W[j]) & 0xFFFFFFFF;
27
28     D = C;
29     C = left_rotate(Bv, 9);
30     Bv = A;
31     A = TT1;
32     H = G;
33     G = left_rotate(F, 19);
34     F = E;
35     E = P0(TT2);
36 }
37
38 V[0] ^= A; V[1] ^= Bv; V[2] ^= C; V[3] ^= D;
39 V[4] ^= E; V[5] ^= F; V[6] ^= G; V[7] ^= H;
40 }

```

运行时间如下所示，可以看到这个优化依旧并没有太多的优化：

```

(base) zhangchi@zhangchi-virtual-machine:~/桌面$ ./sm3_op2
SM3("abc") = 66c7f0f462eedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
Time: 0.011153 ms

```

图 9 优化 2

3.3 优化 3: 宏定义 plus 版

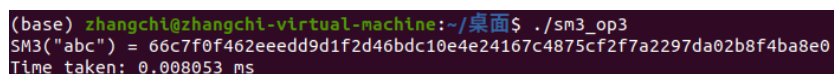
我们前面只是将压缩函数进行了展开，现在我们考虑将前面几个小函数都进行展开，因为 SM3 等对性能敏感的密码算法中，布尔函数与置换函数被频繁调用，使用宏定义替代常规函数可以避免函数调用开销：

```

1 #define rol(x,j) (((x) << (j)) | ((x) >> (32 - (j))))
2 #define P0(x) ((x) ^ rol((x), 9) ^ rol((x), 17))
3 #define P1(x) ((x) ^ rol((x), 15) ^ rol((x), 23))
4 #define FF0(x,y,z) ((x) ^ (y) ^ (z))
5 #define FF1(x,y,z) (((x)&(y)) | ((x)&(z)) | ((y)&(z)))
6 #define GG0(x,y,z) ((x) ^ (y) ^ (z))
7 #define GG1(x,y,z) (((x)&(y)) | ((~(x)) & (z)))

```

优化结果如图所示：



```

(base) zhangchl@zhangchl-virtual-machine:~/桌面$ ./sm3_op3
SM3("abc") = 66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
Time taken: 0.008053 ms

```

图 10 宏定义优化 2

3.4 优化 4: SIMD 优化消息扩展

核心思路如下所示：

1. SIMD 并行处理——用 SSE 128-bit 指令

- `__m128i` 是 Intel SSE SIMD 指令集的 128-bit 向量类型。
- SIMD：一条指令同时操作多个数据。
- 在 SM3 的消息扩展中，很多对多个 32-bit 字的位运算（异或、轮转等）可以用 SIMD 并行处理 4 个 `uint32_t`。

修改相应代码如下：

```

1 static inline __m128i _m128i_left(__m128i a, int k) {
2     k &= 31;
3     return _mm_or_si128(_mm_slli_epi32(a, k), _mm_srli_epi32(a, 32 - k));
4 }
5
6 static inline __m128i _m128i_P1_simd(__m128i X) {
7     return _mm_xor_si128(_mm_xor_si128(X, _m128i_left(X, 15)), _m128i_left(X, 23));
8 }

```

在消息扩展阶段：

- 使用 `_mm_setr_epi32` 载入连续 4 个 `uint32_t`，做成 128 位向量。

- 利用 `_mm_xor_si128`（按位异或）、自定义的 SIMD 左移和 P1 函数，实现消息扩展的复杂变换。

2. 结合上述优化中的宏定义展开 FF0、FF1、GG0、GG1、P0、P1 等都宏实现，且表达式直接展开，避免函数调用开销。

3. 将消息扩展和填充部分进行循环展开

利用宏定义展开循环：

```
1  #define LOOP_BODY1(i) \
2  temp = rol(A, 12) + E + rol(0x79cc4519, i);\
3  SS1 = rol(temp, 7); SS2 = SS1 ^ rol(A, 12);\
4  TT1 = FF0(A, B, C) + D + SS2 + W_t[i]; TT2 = GG0(E, F, G) + H + SS1 + W[i];\
5  D = C; C = rol(B, 9); B = A; A = TT1; H = G; G = rol(F, 19); F = E; E = P0(\
    TT2);
6
7  #define LOOP_BODY2(i) \
8  temp = rol(A, 12) + E + rol(0x7a879d8a, i % 32);\
9  SS1 = rol(temp, 7); SS2 = SS1 ^ rol(A, 12);\
10 TT1 = FF1(A, B, C) + D + SS2 + W_t[i]; TT2 = GG1(E, F, G) + H + SS1 + W[i];\
11 D = C; C = rol(B, 9); B = A; A = TT1; H = G; G = rol(F, 19); F = E; E = P0(\
    TT2);
```

减少循环开销，提升流水线效率。

4. 使用 `memcpy`

```
1  memcpy(&W[(j << 2)], &re, 16);
```

直接从 128-bit 寄存器写入内存，避免对齐、别名等问题，安全且高效。

注意点：大小端转换

SM3 使用大端格式存储数据，x86/x64 是小端平台，所以用 `byte_swap32` 把中间状态字节序调整为大端，保证输出正确。

注意点：内存对齐

用 `uint32_t*` 类型参数，且用 `_mm_setr_epi32` 对 4 个 32 位数连续加载，避免错位访问和性能损失。

得到运行结果如下所示：

```
Time taken: 0.005715 ms
(base) zhangchi@zhangchi-virtual-machine:~/桌面$ ./sm3_op3
66c7f0f462eeedd9d1f2d46bdc10e4e24167c4875cf2f7a2297da02b8f4ba8e0
Time taken: 0.005521 ms
(base) zhangchi@zhangchi-virtual-machine:~/桌面$
```

图 11 SIMD 优化结果

3.5 优化总对比

我们总结运行单次算法所需要的时间：

项目	时间 (ms)
基础实现	0.016101
优化 1	0.010292
优化 2	0.011153
优化 2	0.008053
优化 3	0.005521

表 1 不同优化方案的执行时间比较

可以看到或多或少相较于原始代码有优化，同时 SIMD 最后这个优化优化最好。

4 参考链接

1、https://blog.csdn.net/qz_40662424/article/details/121637732