山东大学网络空间安全学院

创新创业实践 课程实验报告

学号：202100141038	姓名：沈文慧	班级：21 密码 2 班
实验题目：do your best to optimize SM3 implementation (software)		
实验学时：	实验日期：2023-07	

问题分析：do your best to optimize SM3 implementation (software)

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)** mod 512 = 448 的最小正整数（512-64=448 bit 位）2、追加 64 位的数据长度（bit 为单位，大端序存放 1。

优化思路:优化方向一：消息扩展的快速实现。优化实现：在执行 64 轮压缩函数之前，只计算最初的四个词，其余的在每一轮压缩函数中计算。 Wi+4 在压缩函数的第 I 轮生成，将 w1’替换为 wiˆwi+4。优化方向二：预计算常数：常数是预先计算和存储的。这样可以避免对每个消息包进行不断地移位操作。

硬件环境：Windows 11，X64

软件环境：visual studio code

实验步骤与内容：

1. 实验代码：

```
#include <iostream>
#include <cstring>
#include "SM3_class"
#include <fstream>

#include <vector>
#include <iomanip>
#include <memory>
#include <stdint.h>
#include <ctime>
#include <ratio>

#include <chrono>
#include <stdlib.h>
using namespace std;
```

```
//代码文件说明,使用 SM3 对输入文本进行加密
#define MAX_NUM 1024*1024
#define MAXSIZE 1024*MAX_NUM

//设定加密文件的最大字节数为 4KB
//超过该字节数，该程序会自动进行分块
//分为若干文本片段进行分别加密


unsigned int hash_result = 0;
//总的消息块
unsigned int Rate_of_hash = 0;
//当前已经计算完成 hash 的文本数据比例



static const int endianTest = 1;
#define IsLittleEndian() (*(char *)&endianTest == 1)
//首先程序需要判断运行环境是否为小端
#define LeftRotate(word, bits) ( (word) << (bits) | (word) >> (32 - (bits)) )
//遵循 sm3 的标准加密方案，向左循环移位，同时反转四字节整型字节序

unsigned int* Reverse_word(unsigned int* word)
{
    unsigned char* byte, temp;

    byte = (unsigned char*)word;
    temp = byte[0];
    byte[0] = byte[3];
    byte[3] = temp;

    temp = byte[1];
    byte[1] = byte[2];
    byte[2] = temp;
    return word;

}
```

```
//接下来分别实现 sm3 的各个部件
//T
unsigned int T(int i)
{
    if (i >= 0 && i <= 15)
        return 0x79CC4519;
    else if (i >= 16 && i <= 63)
        return 0x7A879D8A;
    else
        return 0;
}

//FF
unsigned int FF(unsigned int X, unsigned int Y, unsigned int Z, int i)
{
    if (i >= 0 && i <= 15)
        return X ^ Y ^ Z;
    else if (i >= 16 && i <= 63)
        return (X & Y) | (X & Z) | (Y & Z);
    else
        return 0;
}

//GG
unsigned int GG(unsigned int X, unsigned int Y, unsigned int Z, int i)
{
    if (i >= 0 && i <= 15)
        return X ^ Y ^ Z;
    else if (i >= 16 && i <= 63)
        return (X & Y) | (~X & Z);
    else
        return 0;
}

//P0
unsigned int P0(unsigned int X)
{
    return X ^ LeftRotate(X, 9) ^ LeftRotate(X, 17);
```

```
}

//P1
unsigned int P1(unsigned int X)
{
    return X ^ LeftRotate(X, 15) ^ LeftRotate(X, 23);
}

//对 sm3 进行初始化函数
void SM3_Init(SM3::SM3Context* context) {
    context->intermediateHash[0] = 0x7380166F;
    context->intermediateHash[1] = 0x4914B2B9;
    context->intermediateHash[2] = 0x172442D7;
    context->intermediateHash[3] = 0xDA8A0600;
    context->intermediateHash[4] = 0xA96F30BC;
    context->intermediateHash[5] = 0x163138AA;
    context->intermediateHash[6] = 0xE38DEE4D;
    context->intermediateHash[7] = 0xB0FB0E4E;
}

// 对 input 的文本进行分块处理
void SM3_dealwith_MessageBlock(SM3::SM3Context* context)
{
    int i;
    unsigned int W[68];
    unsigned int W_[64];
    unsigned int A, B, C, D, E, F, G, H, SS1, SS2, TT1, TT2;

    //message extence
    for (i = 0; i < 16; i++)
    {
        W[i] = *(unsigned int*)(context->messageBlock + i * 4);
        if (IsLittleEndian())
            ReverseWord(W + i);
    }
    for (i = 16; i < 68; i++)
    {
        //P1
```

```
        W[i] = (W[i - 16] ^ W[i - 9] ^ LeftRotate(W[i - 3], 15)) ^ LeftRotate((W[i - 16] ^ W[i - 9] ^ LeftRotate(W[i - 3], 15)), 15) ^ LeftRotate((W[i - 16] ^ W[i - 9] ^ LeftRotate(W[i - 3], 15)), 23)
                ^ LeftRotate(W[i - 13], 7)
                ^ W[i - 6];
    }
    for (i = 0; i < 64; i++)
    {
        W_[i] = W[i] ^ W[i + 4];
    }
    if (i < 12) {
        W[i + 4] = *(unsigned int*)(context->messageBlock + (i + 4) * 4);
        if (IsLittleEndian())   ReverseWord(W + i + 4);
    }
    else {
        W[i + 4] = ((W[i - 12] ^ W[i - 5] ^ LeftRotate(W[i + 1], 15)) ^ LeftRotate((W[i - 12] ^ W[i - 5] ^ LeftRotate(W[i + 1], 15)), 15) ^ LeftRotate((W[i - 12] ^ W[i - 5] ^ LeftRotate(W[i + 1], 15)), 23)) ^ LeftRotate(W[i - 9], 7) ^ W[i - 2];
    }

    //message compression
    A = context->intermediateHash[0];
    B = context->intermediateHash[1];
    C = context->intermediateHash[2];
    D = context->intermediateHash[3];
    E = context->intermediateHash[4];
    F = context->intermediateHash[5];
    G = context->intermediateHash[6];
    H = context->intermediateHash[7];

    for (i = 0; i < 64; i++)
    {
        unsigned int SS3;

        SS1 = LeftRotate((LeftRotate(A, 12) + E + LeftRotate(T(i), i)), 7);
        SS2 = SS1 ^ LeftRotate(A, 12);
        TT1 = FF(A, B, C, i) + D + SS2 + W_[i];
```

```
            TT2 = GG(E, F, G, i) + H + SS1 + W[i];



            D = C;
            C = LeftRotate(B, 9);
            B = A;
            A = TT1;
            H = G;
            G = LeftRotate(F, 19);
            F = E;
            E = TT2 ^ LeftRotate(TT2, 9) ^ LeftRotate(TT2, 17);
        }
        context->intermediateHash[0] ^= A;
        context->intermediateHash[1] ^= B;
        context->intermediateHash[2] ^= C;
        context->intermediateHash[3] ^= D;
        context->intermediateHash[4] ^= E;
        context->intermediateHash[5] ^= F;
        context->intermediateHash[6] ^= G;
        context->intermediateHash[7] ^= H;
    }

    /*
    * SM3 运算的主体过程:
        message 代表需要加密的消息字节串;
        messagelen 是消息的字节数;
        digset 表示返回的哈希值
    */
    unsigned char* SM3::SM3Calc(const unsigned char* message,
        unsigned int messageLen, unsigned char digest[SM3_HASH_SIZE])
    {
        SM3::SM3Context context;
        unsigned int i, remainder, bitLen;

        SM3_Init(&context);
        hash_result = messageLen / 64 + 1;
        //计算总块数
        remainder = messageLen % 64;
```

```
    if (remainder > 111) {
        hash_result += 1;
        //mod64 之后如果大于 111，说明超出了 4KB，我们需要额外一块进行消息填充
        //总块数还要+1
    }
    //对前面的消息分组进行处理
    for (i = 0; i < messageLen / 64; i++)
    {
        memcpy(context.messageBlock, message + i * 64, 64);
        Rate_of_hash = i + 1;
        //每处理一个 512bit 的消息块，进度就+1
        SM3_dealwith_MessageBlock(&context);
    }

    //填充消息分组，并处理
    bitLen = messageLen * 8;
    if (IsLittleEndian())
        ReverseWord(&bitLen);
    memcpy(context.messageBlock, message + i * 64, remainder);
    context.messageBlock[remainder] = 0x80;
    //添加 bit‘0x1000 0000’到末尾
    if (remainder <= 111)
    {
        //长度按照大端法占 8 个字节，只考虑长度在 2^32 - 1(bit)以内的情况，
        //故将高 4 个字节赋为 0
        memset(context.messageBlock + remainder + 1, 0, 64 - remainder - 1 - 8 + 4);
        memcpy(context.messageBlock + 64 - 4, &bitLen, 4);
        Rate_of_hash += 1;//计算最后一个短块
        SM3_dealwith_MessageBlock(&context);
    }
    else
    {
        memset(context.messageBlock + remainder + 1, 0, 64 - remainder - 1);
        hash_rate += 1;
        //计算我额外添加的短块
        SM3_dealwith_MessageBlock(&context);
```

```
        //长度按照大端法占 8 个字节，只考虑长度在 2^32 - 1(bit)以内的情况，
        //故将高 4 个字节赋为 0
        memset(context.messageBlock, 0, 64 - 4);
        memcpy(context.messageBlock + 64 - 4, &bitLen, 4);
        Rate_of_hash += 1;
        //计算最后一个短块
        SM3_dealwith_MessageBlock(&context);
    }
    if (IsLittleEndian())
        for (i = 0; i < 8; i++)
            ReverseWord(context.intermediateHash + i);
    memcpy(digest, context.intermediateHash, SM3_HASH_SIZE);
    return digest;
}

/*
* call_hash_sm3 函数
    输入参数：文件地址字符串
    输出：向量 vector<unit32_t> hash_result(32)
*/
std::vector<uint32_t> SM3::call_hash_sm3(char* filepath)
{
    std::vector<uint32_t> hash_result(32, 0);
    std::ifstream infile;
    uint32_t FILESIZE = 0;
    //进行文件操作，将该文件作为文本数据待加密
    unsigned char* buffer = new unsigned char[MAXSIZE];
    unsigned char hash_output[32];
    struct _stat info;
    _stat(filepath, &info);
    FILESIZE = info.st_size;
    infile.open(filepath, std::ifstream::binary);
    infile >> buffer;

    auto start = std::chrono::high_resolution_clock::now();
    SM3::SM3Calc(buffer, FILESIZE, hash_output);
    auto end = std::chrono::high_resolution_clock::now();
    // 以毫秒为单位，返回所用时间
```

```
    std::cout << "in millisecond time:";
    std::chrono::duration<double, std::ratio<1, 1000>> diff = end - start;
    std::cout << "Time is " << diff.count() << " ms\n";
    hash_result.assign(&hash_output[0], &hash_output[32]);
    delete[]buffer;
    return hash_result;
}

//对当前的哈希进度进行计算与反馈
double progress() {
    return (double(Rate_of_hash) / hash_result);
}

//创建固定大小的文件
void CreatTxt(char* pathName, int length)//创建 txt 文件
{
    ofstream fout(pathName);
    char char_list[] = "abcdefghijklmnopqrstuvwxyz";
    int n = 26;
    if (fout) {
        for (int i = 0; i < length; i++)
        {
            fout << char_list[rand() % n];
            // 使用和输出流同样的方式进行写入
        }

        fout.close();
        // 执行完操作后关闭文件句柄，
        //一定要写这一句，否则下次运行的时候会出现问题
    }
}
int main() {
    char filepath[] = "test.txt";
    CreatTxt(filepath, MAX_NUM);
    std::vector<uint32_t> hash_result;
    hash_result = SM3::call_hash_sm3(filepath);
    for (int i = 0; i < 32; i++) {
        std::cout << std::hex << std::setw(2) << std::setfill('0') << hash_result[i];
```

```
            if (((i + 1) % 4) == 0) std::cout << " ";
        }
        std::cout << std::endl;
        double rate = progress();
        printf("\n 当前进度: %f", rate);
        return 0;
}
```

结论分析与体会：

在软件上尽最大努力实现 SM3，使得 SM3 的加密速度尽可能的快；确保你理解所写的每一行（不要复制粘贴）通过更快的运行获得更高的分数。整个 SM3 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。下面分块进行详细说明：

国密算法 SM3 方案概述

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。具体算法标准原始文本参见参考文献【1】。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 适用于商用密码应用中的数字签名和验证，是在[SHA-256]基础上改进实现的一种算法，其安全性可以认为与 SHA-256 类似。在进行杂凑过程中 SM3 和 MD5 的迭代过程类似，也采用 Merkle-Damgard 结构。消息分组长度为 512 位，hash 值计算出的长度为 256 位。【4】

首先 SM3 是一种 hash 函数，那么首先其具有如下特性：

1.对于任何一个给定的消息，它都很容易就能运算出散列数值。（多项式时间可计算）

2.难以由一个已知的散列数值，去推算出原始的消息。（单向性，求逆困难，一般认为在多项式时间内只能以可忽略的概率计算出其逆）

3.在不更动散列数值的前提下，修改消息内容是不可行的。（认证性）

4.对于两个不同的消息，它不能给与相同的散列数值。（抗碰撞性）

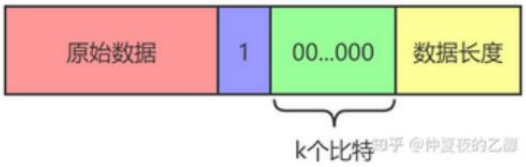
一.消息填充

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) mod 512 = 448 的最小正整数（512-64=448 bit 位）

2、追加 64 位的数据长度（bit 为单位，大端序存放 1。观察国家密码管理局关于发布《SM3 密码杂凑算法》公告的原文附录 A 运算示例可以推知。）

消息填充示例：



二. 消息扩展

SM3 的迭代压缩步骤没有直接使用数据分组进行运算，而是使用这个步骤产生的 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。在最终得到的 132 个消息字中，前 68 个消息字构成数列 {Wj}，后 64 个消息字构成数列 {Wj'}，其中下标 j 从 0 开始计数。

具体消息扩展运算过程如下图所示：

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

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

b)FOR j=16 TO 67

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

ENDFOR

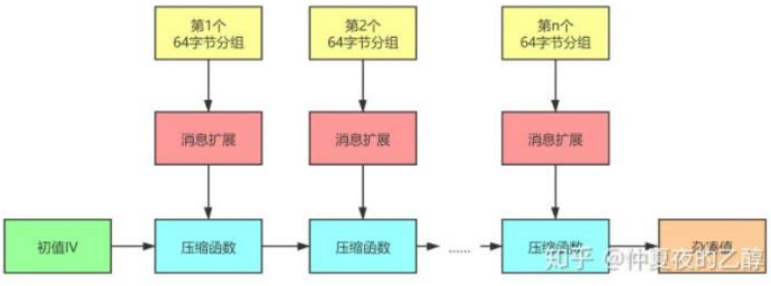
c)FOR j=0 TO 63

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

ENDFOR

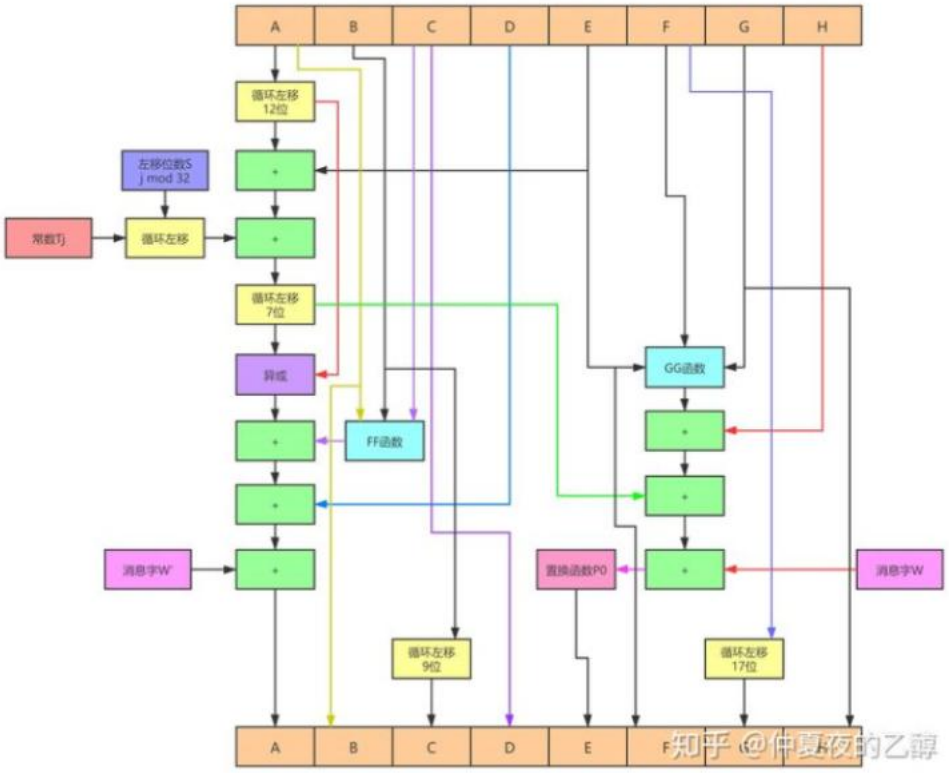
三. 迭代压缩

在上文已经提过，SM3 的迭代过程和 MD5 类似，也是 Merkle-Damgard 结构。但和 MD5 不同的是，SM3 使用消息扩展得到的消息字进行运算。这个迭代过程可以用下图表示：



初值 IV 被放在 A、B、C、D、E、F、G、H 八个 32 位变量中，其具体数值参见 SM3 公告【1】。整个算法中最核心、也最复杂的地方就在于压缩函数（compression）。压缩函数将这八个变量进行 64 轮相同的计算。

其中一轮的计算过程如下图所示：



图中不同的数据流向用不同颜色的箭头表示。最后，再将计算完成的 A、B、C、D、E、F、G、H 和原来的 A、B、C、D、E、F、G、H 分别进行异或，就是压缩函数的输出。这个输出再作为下一次调用压缩函数时的初值。进入递归函数，递归的终止条件是直到用完最后一组 132 个消息字为止。

输出结果：

将上面第三步中计算出的 A、B、C、D、E、F、G、H 八个变量拼接输出，就是 SM3 算法的最终输出的杂凑值。

四.代码实现思路

SM3 的每个部件的相关实现

标准 FF 与 GG 部件相关的布尔运算的实现：

```
//FF
unsigned int FF(unsigned int X, unsigned int Y, unsigned int Z, int i)
{
    if (i >= 0 && i <= 15)
        return X ^ Y ^ Z;
    else if (i >= 16 && i <= 63)
        return (X & Y) | (X & Z) | (Y & Z);
    else
        return 0;
}
//GG
unsigned int GG(unsigned int X, unsigned int Y, unsigned int Z, int i)
{
    if (i >= 0 && i <= 15)
        return X ^ Y ^ Z;
    else if (i >= 16 && i <= 63)
        return (X & Y) | (~X & Z);
    else
        return 0;
}
```

P0/P1 置换函数：

```
//P0
unsigned int P0(unsigned int X)
{
    return X ^ LeftRotate(X, 9) ^ LeftRotate(X, 17);
}
//P1
unsigned int P1(unsigned int X)
{
    return X ^ LeftRotate(X, 15) ^ LeftRotate(X, 23);
}
```

字节翻转操作：

在 SM3 算法中，string 以大端格式存储所以在标准运算中，针对每个待加密的字符串应该进行翻转后再进行加密运算

```
unsigned int* Reverse_word(unsigned int* word)
{
    unsigned char* byte, temp;

    byte = (unsigned char*)word;
    temp = byte[0];
    byte[0] = byte[3];
    byte[3] = temp;
    temp = byte[1];
    byte[1] = byte[2];
    byte[2] = temp;
    return word;
}
```

消息填充：

将数据长度填充为 512bit 的倍数。长度按照大端法占用 8 个字节，只考虑 2^32 - 1（单位：位）以内的长度，*所以分配高 4 个字节为 0。

```
//填充消息分组，并处理
bitLen = messageLen * 8;
if (IsLittleEndian())
    ReverseWord(&bitLen);
memcpy(context.messageBlock, message + i * 64, remainder);
context.messageBlock[remainder] = 0x80;
//添加bit '0x1000 0000' 到末尾
if (remainder <= 111)
{
    //长度按照大端法占8个字节，只考虑长度在 2^32 - 1(bit)以内的情况，
    //故将高 4 个字节赋为 0
    memset(context.messageBlock + remainder + 1, 0, 64 - remainder - 1 - 8 + 4);
    memcpy(context.messageBlock + 64 - 4, &bitLen, 4);
    Rate_of_hash += 1;//计算最后一个短块
    SM3_dealwith_MessageBlock(&context);
}
```

剩余部件如消息扩展，消息压缩等的详细代码实现操作，请参看 SM3_Enc.cpp 文件中相应部分，均已写明注释。

五.SM3 算法部分优化方向

优化思路相关参考上传的 SM3_Reference.pdf 文件与参考资料【5】中的优化思路。

优化方向一：消息扩展的快速实现

优化原理：

优化前：计算前 16 个 wi 时，每个需要执行一次 load 和一个 store，计算后 52 个 wi 时，每个需要执行 5 次 load，1 次 store，6 次 XOR 和 1 次 rot；

计算 64 个 wi，每个 wi 需要执行 2 次 load，1 次 store 和 1 次 rot。

优化后：

在计算前 12 个 wi+4 时，每个都需要执行一次加载和一次保存。 在计算最后 52 个 wi+4 时，每个需要进行 5 次 load，1 次 store，6 次 XOR，4 次 rot；主要是减少计算和存储 W' 时的存取操作。 在测试中，优化也提高了算法的执行速度

优化实现： 在执行 64 轮压缩函数之前，只计算最初的四个词，其余的在每一轮压缩函数中计算。 Wi+4 在压缩函数的第 I 轮生成，将 w1' 替换为 wi^wi+4。

```
if (i < 12) {
    W[i + 4] = *(unsigned int*)(context->messageBlock + (i + 4) * 4);
    if (IsLittleEndian())  ReverseWord(W + i + 4);
}
else {
    W[i + 4] = ((W[i - 12] ^ W[i - 5] ^ LeftRotate(W[i + 1], 15)) ^ LeftRotate((W[i - 12] ^ W[i - 5] ^ LeftRotate(W[i + 1], 15)), 15) ^ LeftRotate((W[i - 12] ^ W[i - 5]
}
```

优化方向二：预计算常数

常数是预先计算和存储的。 这样可以避免对每个消息包进行不断的移位操作，优化后占用的存储空间也很小，只有 256 字节。

```
unsigned int T(int i) {
    if (i >= 0 && i <= 15)      return 0x79CC4519;
    else if (i >= 16 && i <= 63)    return 0x7A879D8A;
    else                return 0;
}

void cacuIT() {
    for (int i = 0; i < 64; i++)    t[i] = LeftRotate(T(i), i);
    return;
}
```

优化方向三：优化压缩函数中间变量的生成过程

去除了不必要的赋值，减少了中间变量的数量。 方法 3、4 优化后，只更新了 D、h、B、F，减少了赋值操作。压缩函数的结构进行了调整，大大减少了 load 和 store 的数量，而中间变量 TT1 和 TT2 的优化进一步减少了 rot 的数量。

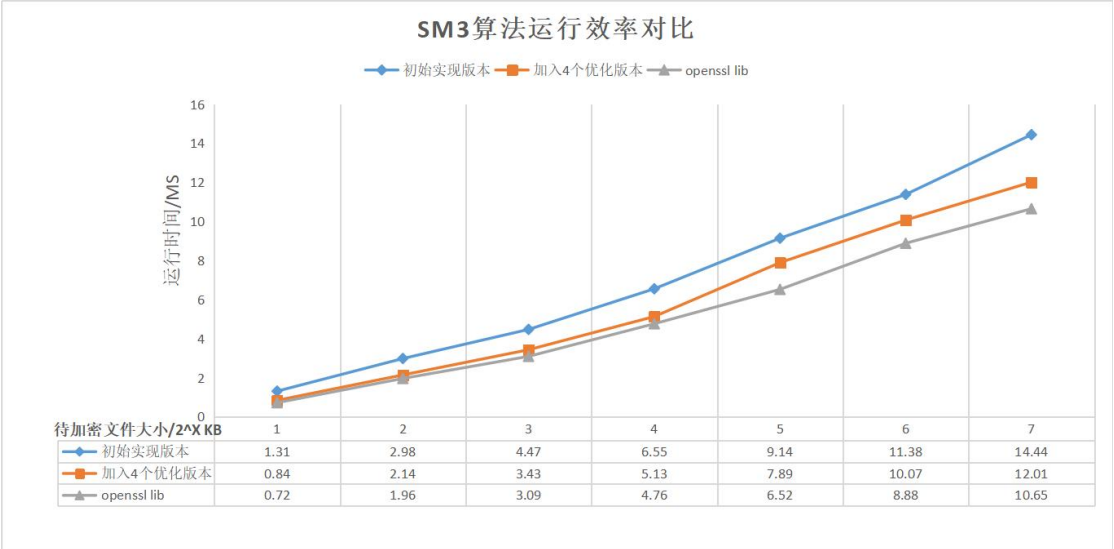
```
TT2 = LeftRotate(A, 12);
TT1 = TT2 + E + t[i];
TT1 = LeftRotate(TT1, 7);
TT2 ^= TT1;

D = D + FF(A, B, C, i) + TT2 + (W[i] ^ W[i + 4]);
H = H + GG(E, F, G, i) + TT1 + W[i];
B = LeftRotate(B, 9);
F = LeftRotate(F, 19);
H = H ^ LeftRotate(H, 9) ^ LeftRotate(H, 17);
```

优化方向四：对压缩函数进行结构性调整

在每一轮压缩函数结束时，将执行一次循环右移。 将 string 循环向右移动可以改变每轮输入 string 的顺序，这个顺序变化会在 4 轮后恢复，代码实现如下：

```
for (i = 0; i <= 60; i += 4) {
    one_round(i, A, B, C, D, E, F, G, H, W, context);
    one_round(i + 1, D, A, B, C, H, E, F, G, W, context);
    one_round(i + 2, C, D, A, B, G, H, E, F, W, context);
    one_round(i + 3, B, C, D, A, F, G, H, E, W, context);
}
```

六. 运行结果与效率对比截图：



待加密文件大小/2^X KB	1	2	3	4	5	6	7
初始实现版本	1.31	2.98	4.47	6.55	9.14	11.38	14.44
加入4个优化版本	0.84	2.14	3.43	5.13	7.89	10.07	12.01
openssl lib	0.72	1.96	3.09	4.76	6.52	8.88	10.65

如下图所示，在加入上面的优化方法后，对比初始实现方案与 openssl 中的 sm3 库的运行效率，算法的运行效率得到了提高，但仍然与标准库函数的运行效率有一定的差距。

参考资料：

【1】国家密码管理局关于发布《SM3 密码杂凑算法》公告：
http://www.oscca.gov.cn/sca/xxgk/2010-12/17/content_1002389.shtml

【2】SM3 密码杂凑算法原理简述： https://zhuanlan.zhihu.com/p/129692191

【3】SM3 加密算法详解（2021-12-8）： https://blog.csdn.net/qq_40662424/article/details/121637732

【4】SM3 百度百科： https://baike.baidu.com/item/SM3/4421797?fr=aladdin

【5】SM3 密码杂凑算法-->大文件做摘要优化： https://blog.csdn.net/oprim/article/details/124179928