北京电子科技学院（BESTI）

**实 验 报 告**

课程：信息安全系统设计 班级：2214

姓名：徐鹿鸣 学号：20221414

成绩： 指导教师：娄嘉鹏 实验日期：2024/11/03

实验密级：无 预习程度：全预习

实验时间：3，4节 仪器组次：无 必修/选修：必修

实验序号：实验二

实验名称： 密码算法实现

实验目的与要求：掌握常见商用密码算法的原理与实现。基于Arm等平台和国产化操作系统使用C语言编程实现SM2、SM3、SM4算法；对比分析算法实现的正确性和效率。

实验仪器：

|  |  |  |
| --- | --- | --- |
| 名称 | 型号 | 数量 |
| 计算机 |  | 1 |
|  |  |  |

实验内容、步骤与体会（附纸）：

**1. 实验内容**

基于国产化平台使用C语言编程实现SM2、SM3、SM4等算法；

**2. 实验步骤**

**2.1步骤1** 在 Ubuntu中调试运行商用密码检测中心供的源代码，至少运行SM2，SM3，SM4代码。使用GmSSL命令验证你代码的正确性。

**2.1.1 操作内容**

进入[商用密码检测中心](https://www.scctc.org.cn/xzzx/sfydm/ydmxz/)下载源代码PDF文件。将其转为word文件。由于转换不是很准确，所以先在VS2022中运行测试，修正错误后再拷贝到Ubuntu中运行。

在测试中发现源代码均无main函数，需要自己编写。同时注意到自检函数。所以首要的目标就是编写一个简单的调用自检函数的mian函数进行测试，测试成功后就可以模仿自检函数自己构建数据、调用函数、进行测试了。

按照先易后难的顺序，依次完成SM3、SM4和SM2的内容。其中SM3较为简单，这里不多论述。

首先要判断这是ECB模式下的SM4实现。在测试中发现SM4程序对输入与输出有严格的要求，这需要对数据进行填充与解填充。这里采用PKCS#7填充方法，在SM4代码内部也加入了相关填充功能，同时使用AI辅助生成了python脚本实现填充相关功能。

实际测试时，使用命令行传入要测试的数据”xlm”，输出十六进制密钥、密文和解密的明文。使用测试数据生成明文文件，使用pad.py将明文文件填充生成填充后的明文文件，用gmssl命令和十六进制密钥加密填充后的明文文件，查看密文是否与程序输出的密文一致。如果一致，则程序功能验证成功。

对于SM2，情况复杂很多。

首先要在Ubuntu中下载编译Miracl库。我的摸索过程是通过GitHub上的官方文档和其他教程文章，这里不再赘言。我们需要编译32位的Miracl库，所以需要预先准备32位环境，可以参考[这个对话中的命令](https://kimi.moonshot.cn/share/cser2rgjdo7r0h8fqahg)准备环境。从GitHub上下载Miracl库的zip包到一个新的文件夹，使用unzip -j -aa -L MIRACL-master.zip命令无视文件结构解压。执行bash linux命令即可编译32位的Miracl库。使用./pk-demo命令测试安装是否成功。在miracl.h的第96行取消注释，避免找不到compare函数。

为了在不同地方都能链接这个库，我们将其拷贝到用户文件夹的指定位置。可以参考以下命令：

cp ./miracl.a /usr/lib/libmiracl\_32.a

mkdir /usr/include/miracl

cp \*.h /usr/include/miracl

采用这个方法编译Miracl库后，之前源代码的相关头文件需要修改，一般是加上”miracl/”。编译时采取32位编译即加上-m32，链接时使用-lmiracl\_32。

在实践中发现，sm2密钥交换代码在vs2022中可以正常运行，但在Linux环境下会报错:SM2\_KEY\_EX.c:95:23: error: lvalue required as unary ‘&’ operand。将源代码中将错误代码

memcpy(IDlen, &(unsigned char)ELAN + 1, 1);

memcpy(IDlen + 1, &(unsigned char)ELAN, 1);

改为：

memcpy(IDlen, (unsigned char\*)&ELAN + 1, 1);

memcpy(IDlen + 1, (unsigned char\*)&ELAN, 1);

由此可以完成调试运行商用密码检测中心供的SM2，SM3，SM4源代码。其中使用GmSSL命令验证的部分需要根据自己的main函数实现与测试数据进行调整，具体的可以参考下面操作结果截图。

**2.1.2操作结果**

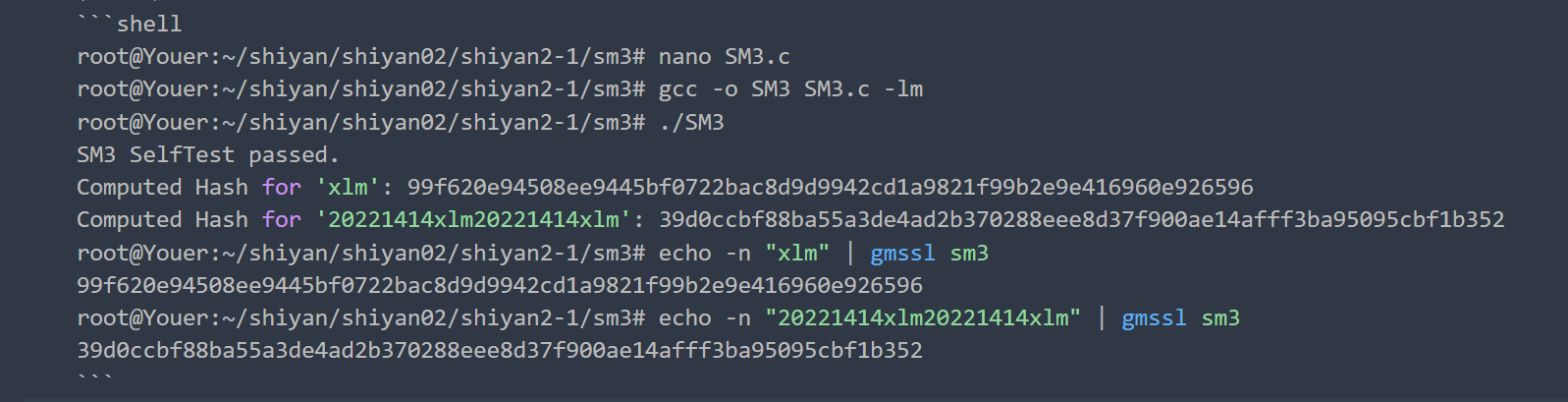


图 1 SM3程序运行与验证过程

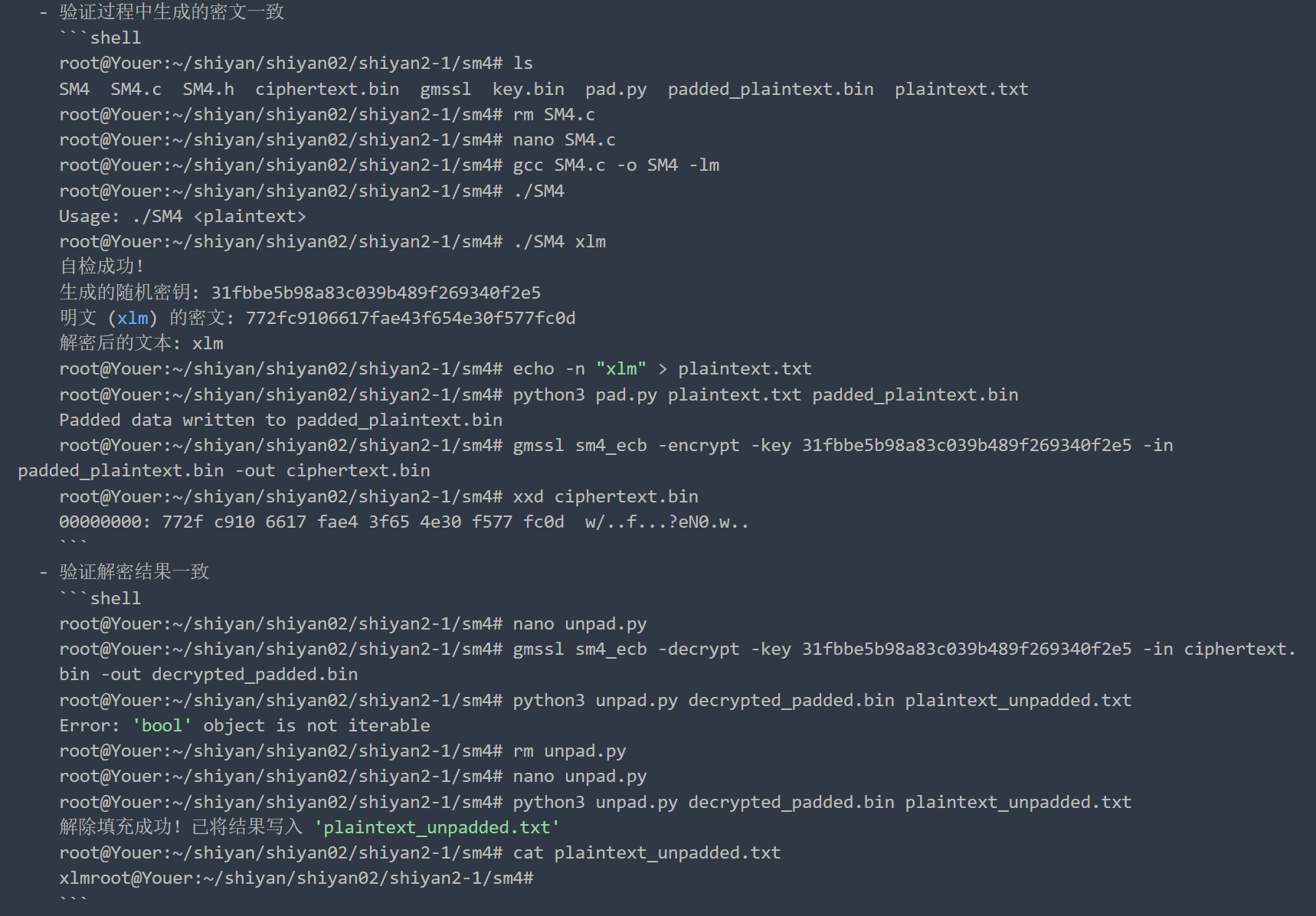


图 2 SM4的程序运行与GmSSL命令验证过程



图 3 SM2程序运行过程

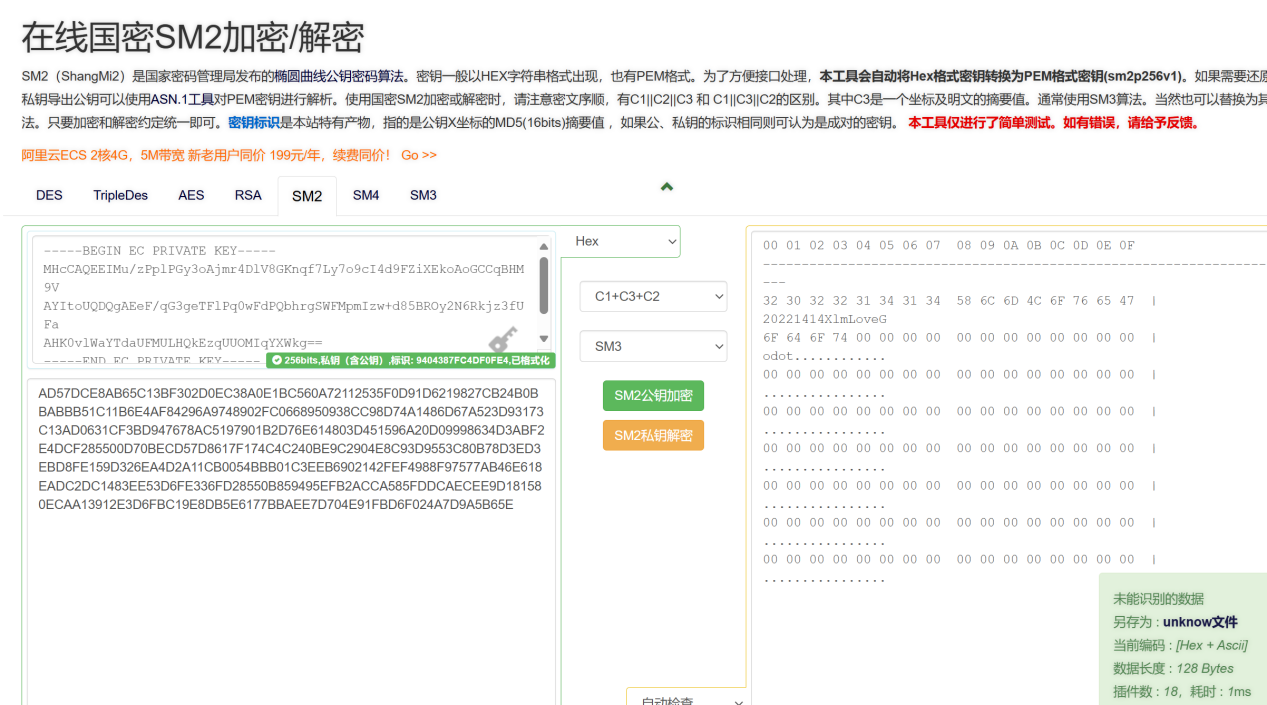


图 4 SM2程序验证（通过[在线网站](https://the-x.cn/cryptography/Sm2.aspx)）

**2.2步骤2** 在[密标委网站](http://www.gmbz.org.cn/main/bzlb.html)查找SM2，SM3，SM4相关标准，分析步骤一中代码实现与标准的对应关系

**2.2.1 操作内容**

在[密标委网站](http://www.gmbz.org.cn/main/bzlb.html)通过标准中文名称（例如“SM2”）和标准状态(“现行”)快速查找SM2，SM3，SM4相关标准。

通过目录和AI的总结把握标准的大体内容，再与代码进行对比。从参数命名和数值的一致、算法实现的一致、测试方法与数据的一致、输入输出的格式要求的一致等方面寻找对应关系。可以使用AI来辅助理解代码逻辑。

**2.2.2操作结果**



图 5 查询标准的方法

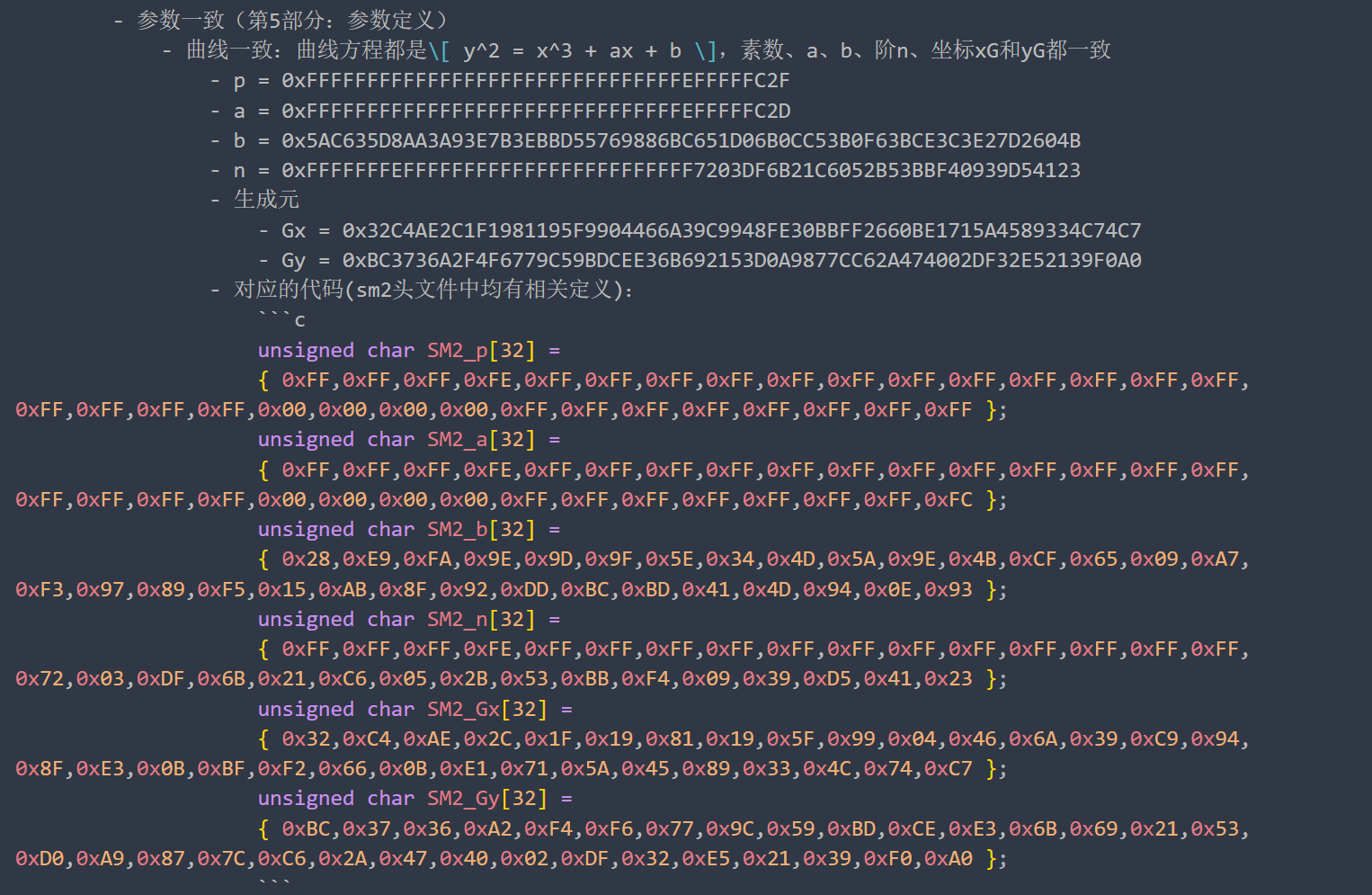


图 6 SM2曲线参数的一致



图 7 AI辅助理解代码([对话链接](https://kimi.moonshot.cn/share/csf21hcrmee54nn7ne30))

**2.3步骤3** 在Ubuntu中调试运行教材提供SM2，SM3，SM4源代码，使用GmSSL命令验证你代码的正确性。

**2.3.1 操作内容**

下载并解压云班课/资源/课程参考资料/Windows.C.C++.加密解密实战.rocsrc.zip并结合教材确定代码所属内容。（见[我的GitHub仓库](https://github.com/youer0219/Information-Security-System-Design-Assignment)中“实验/实验二/代码/教材源代码”）

根据步骤一的经验按SM3、SM4、SM2的顺序进行实验。

在SM3的实践中，4-1的代码会报错。修复后的代码见实验报告第三部分。同时注意，SM3需要在32位环境中运行，所以编译时要加上-m32。同时，包括之后的代码，预编译头文件pch.h与其cpp文件都会被删除，因为这些并没有被使用。

4-1到4-3的测试内容基本一致，所以用gmssl sm3 命令输出哈希值再对比即可验证。4-4的密钥为string类型，转换后的长度也不符合gmssl命令要求，所以在[在线网站](https://www.lddgo.net/encrypt/hmac)中验证。

这次的SM4代码调试与运行并没有难度，正常编译执行即可。

在SM2的实践中，由于步骤一已经安装了Miracl库且可以在任何地方链接，这里就不再按照教材的方法进行（教材的方法也不是适用于Linux的方法）。所以需要对源代码的头文件进行修改。同时修复重复定义报错（具体修复见实验报告第三部分），对照教材修复一些乱码。

由于SM4和SM2代码均有自检函数，不需要再通过GmSSL命令进行验证。由此便可完成步骤三。

**2.3.2操作结果**

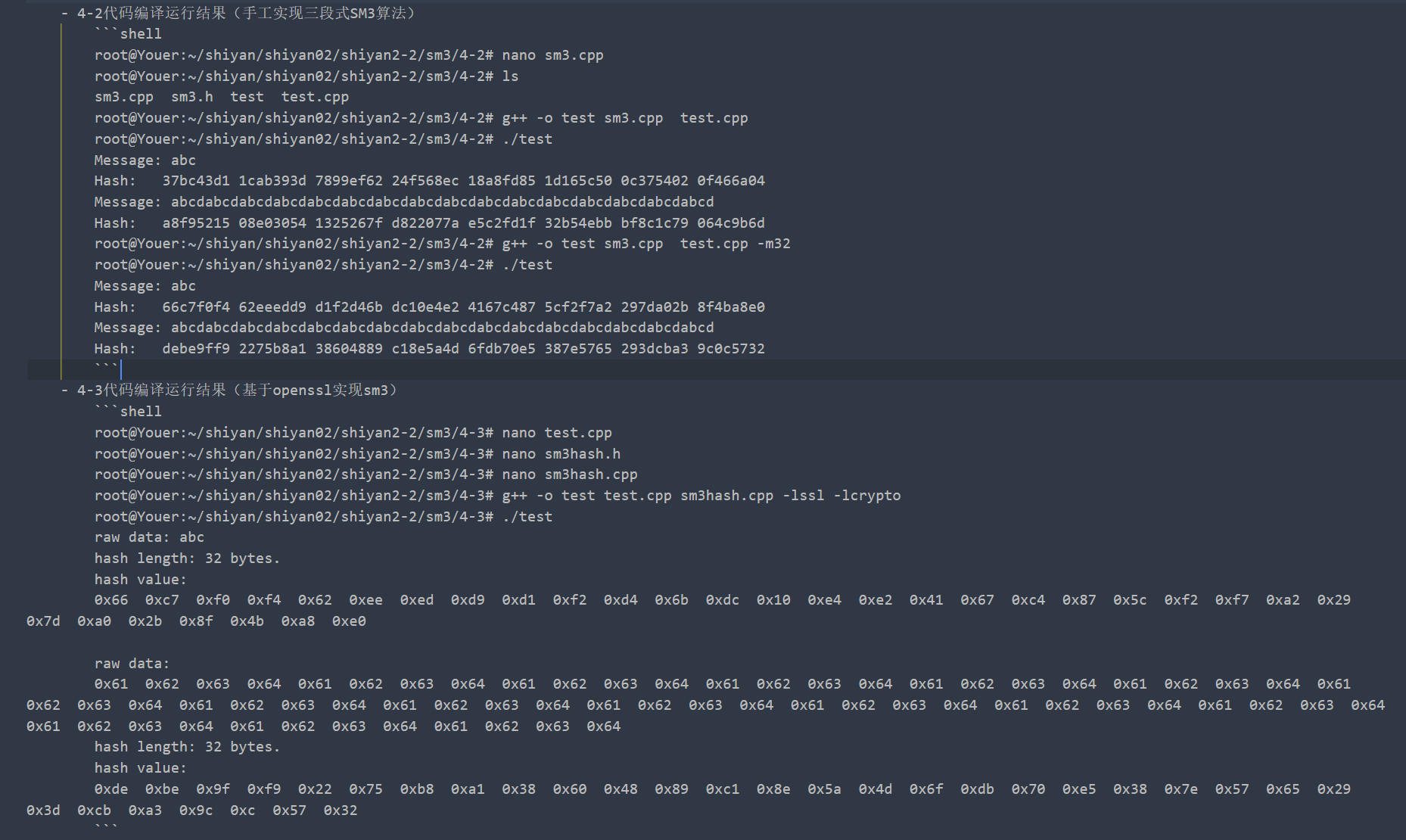


图 8 SM3程序运行结果

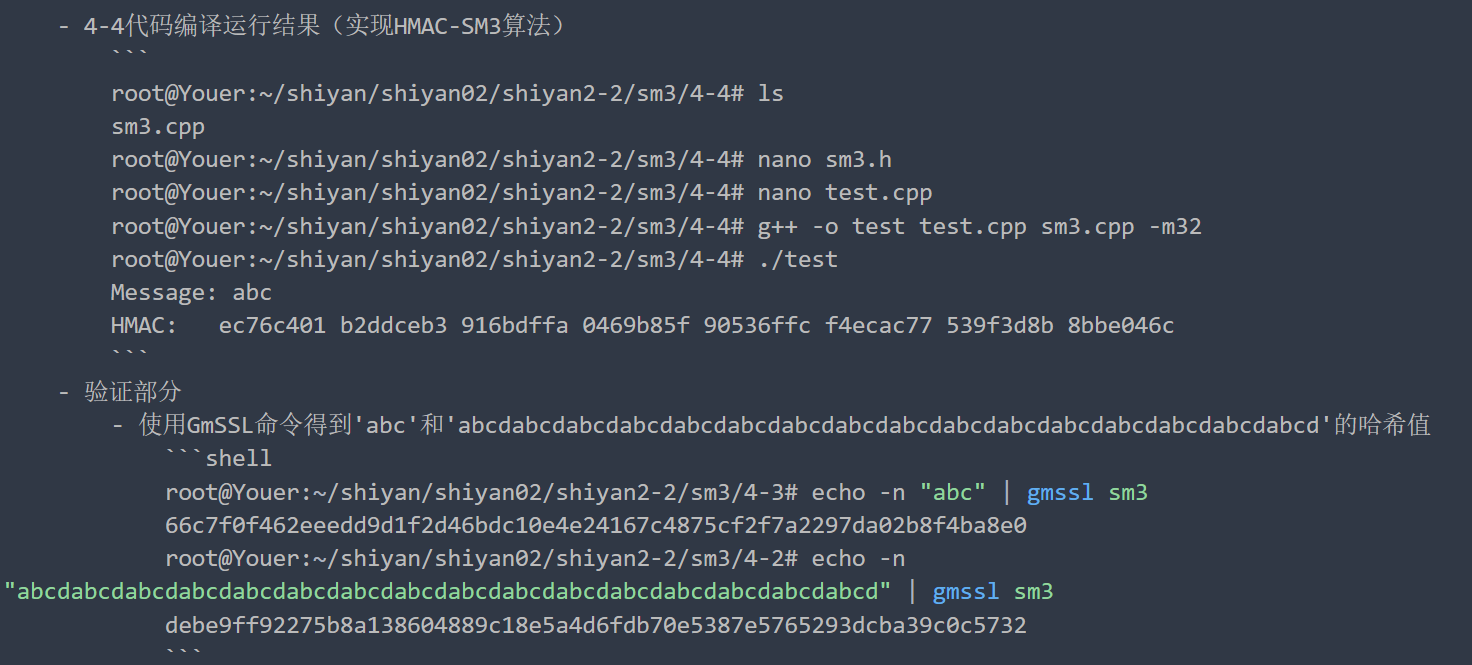


图 9 SM3 4-4运行结果 和gmssl命令验证结果

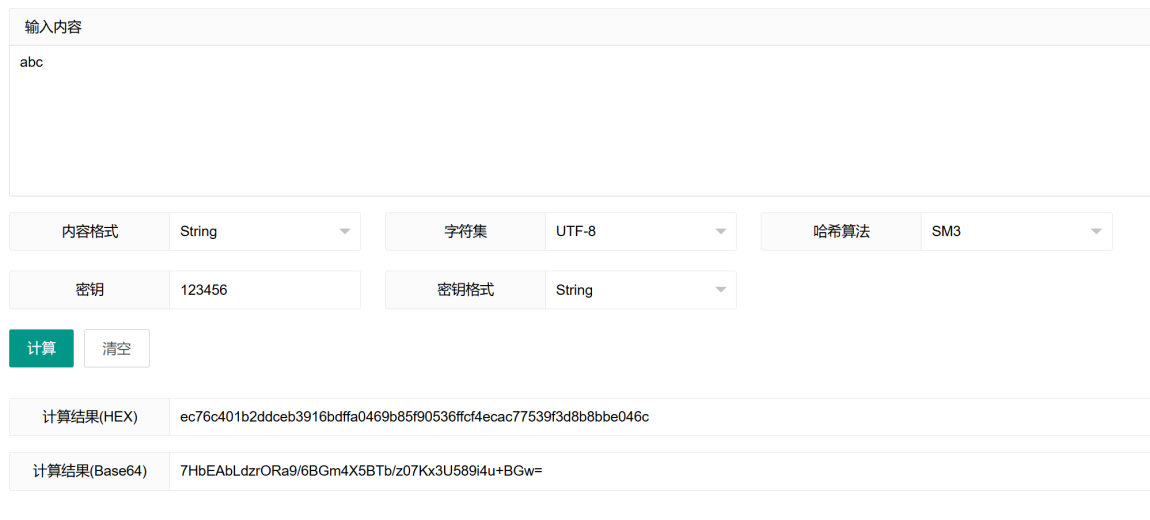


图 10 对SM3 4-4的验证（[在线网站](https://www.lddgo.net/encrypt/hmac)）

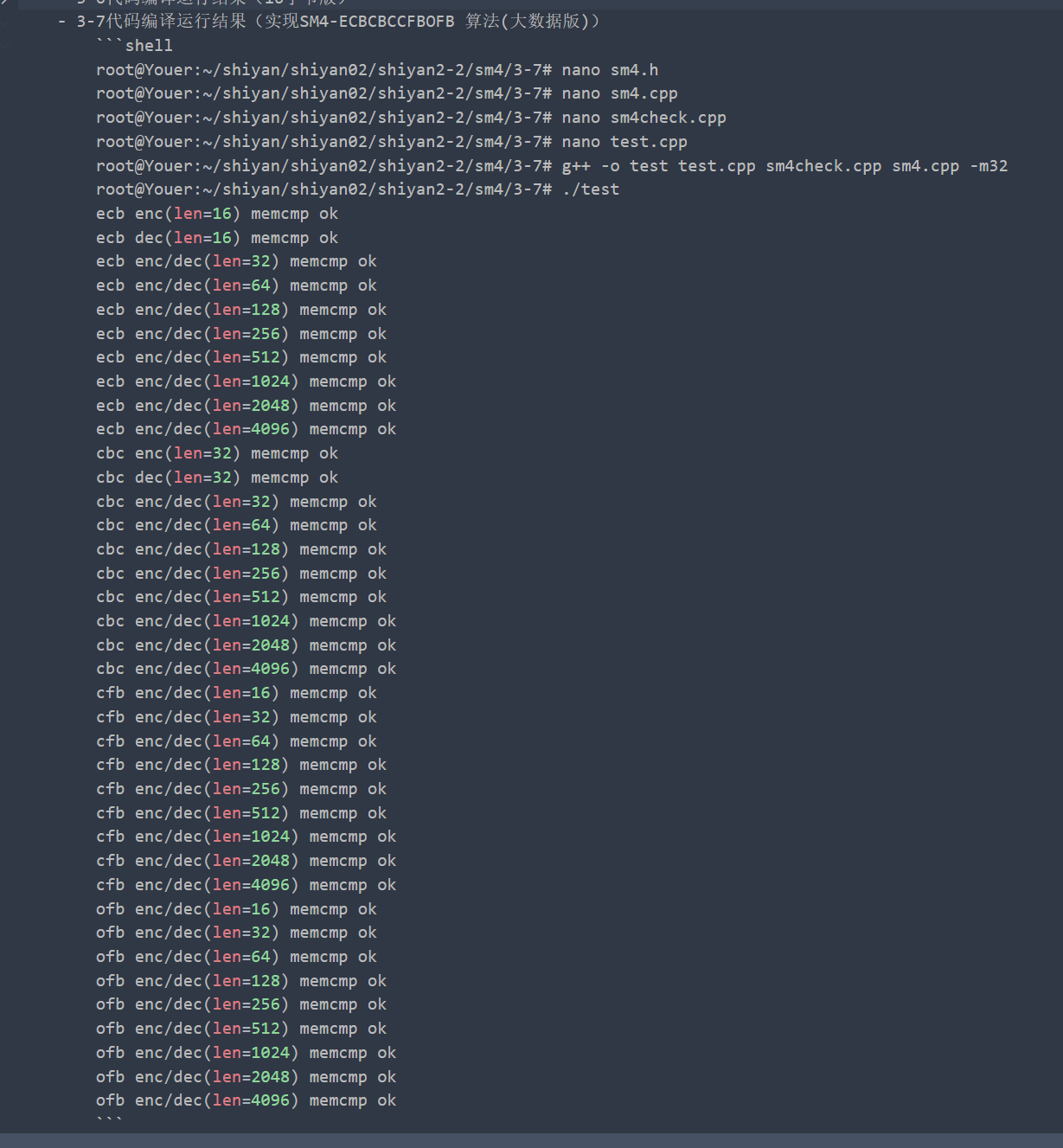


图 11 SM4运行结果

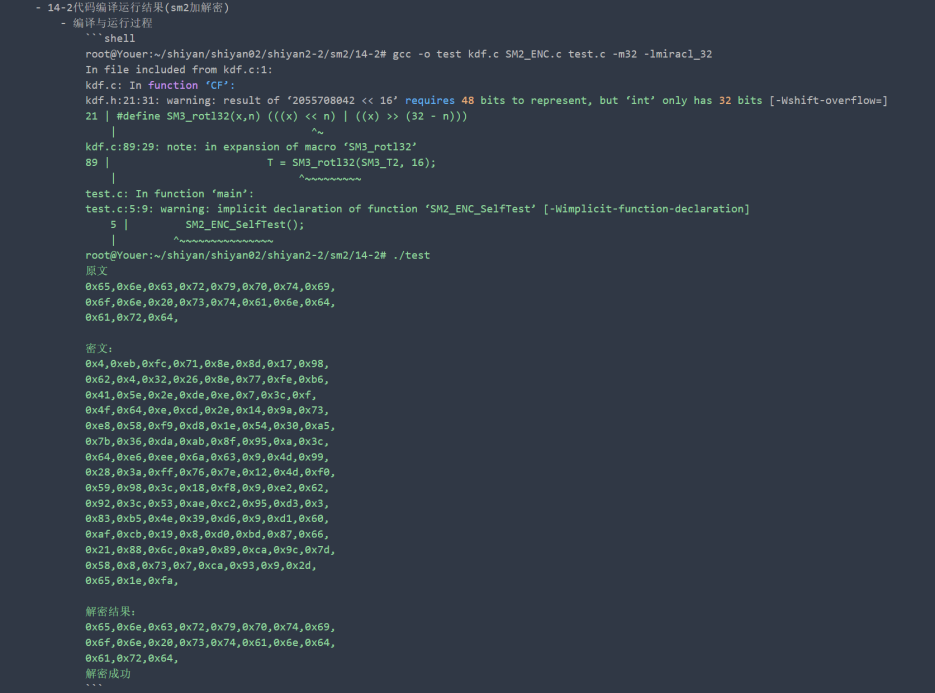


图 12 SM2加解密的运行结果

**2.4步骤4** 在[密标委网站](http://www.gmbz.org.cn/main/bzlb.html)查找SM2，SM3，SM4相关标准，分析步骤三中代码实现与标准的对应关系。

**2.4.1 操作内容**

参考步骤二获取标准文件并分析代码实现与标准的对应关系。

这次由于程序很多，主要采用AI辅助的方法进行分析。将代码发送给kimi，基于步骤二总结出来的各种方面来询问其代码与标准的对应关系。

**2.4.2操作结果**

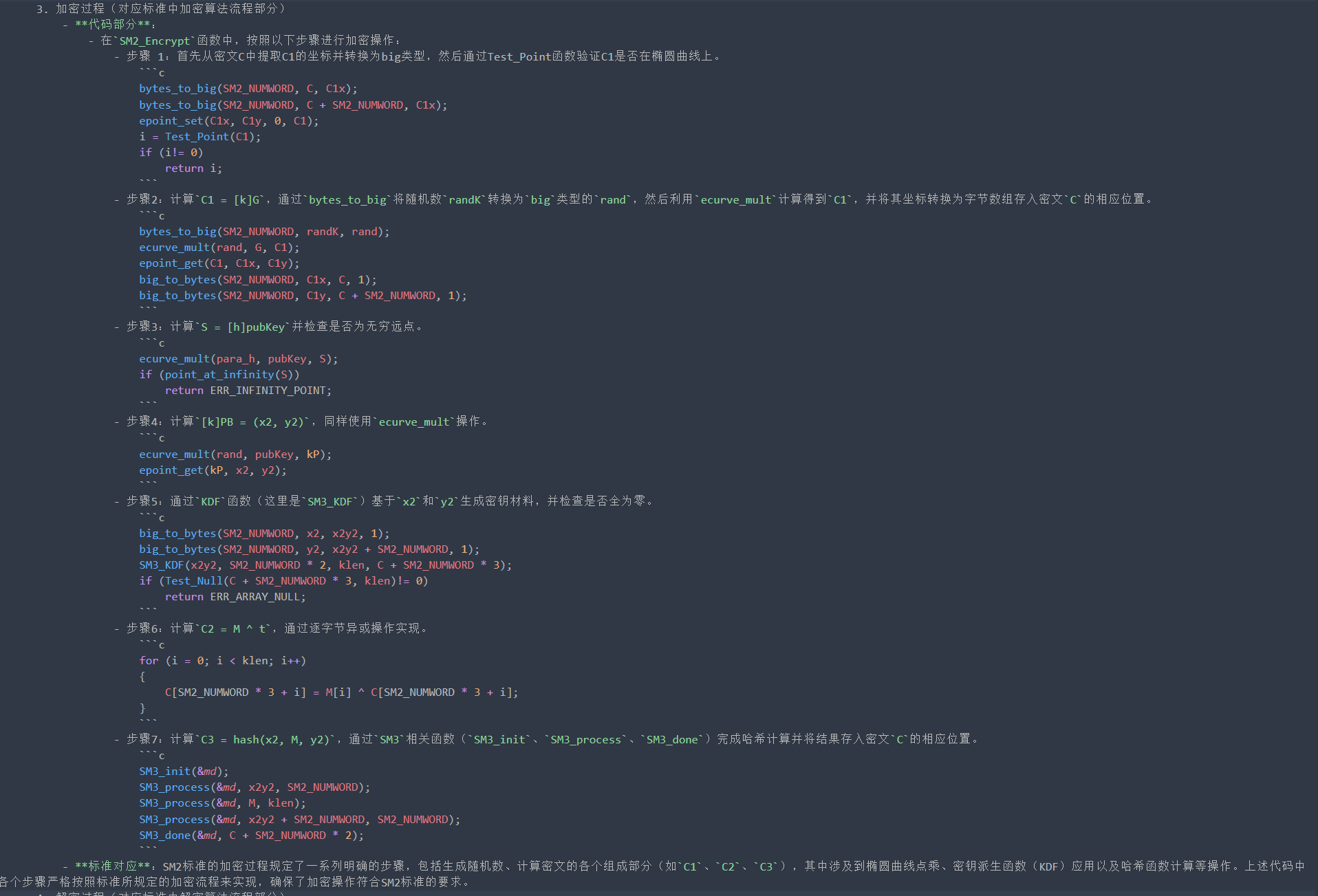


图 13 对SM2加密过程的代码分析

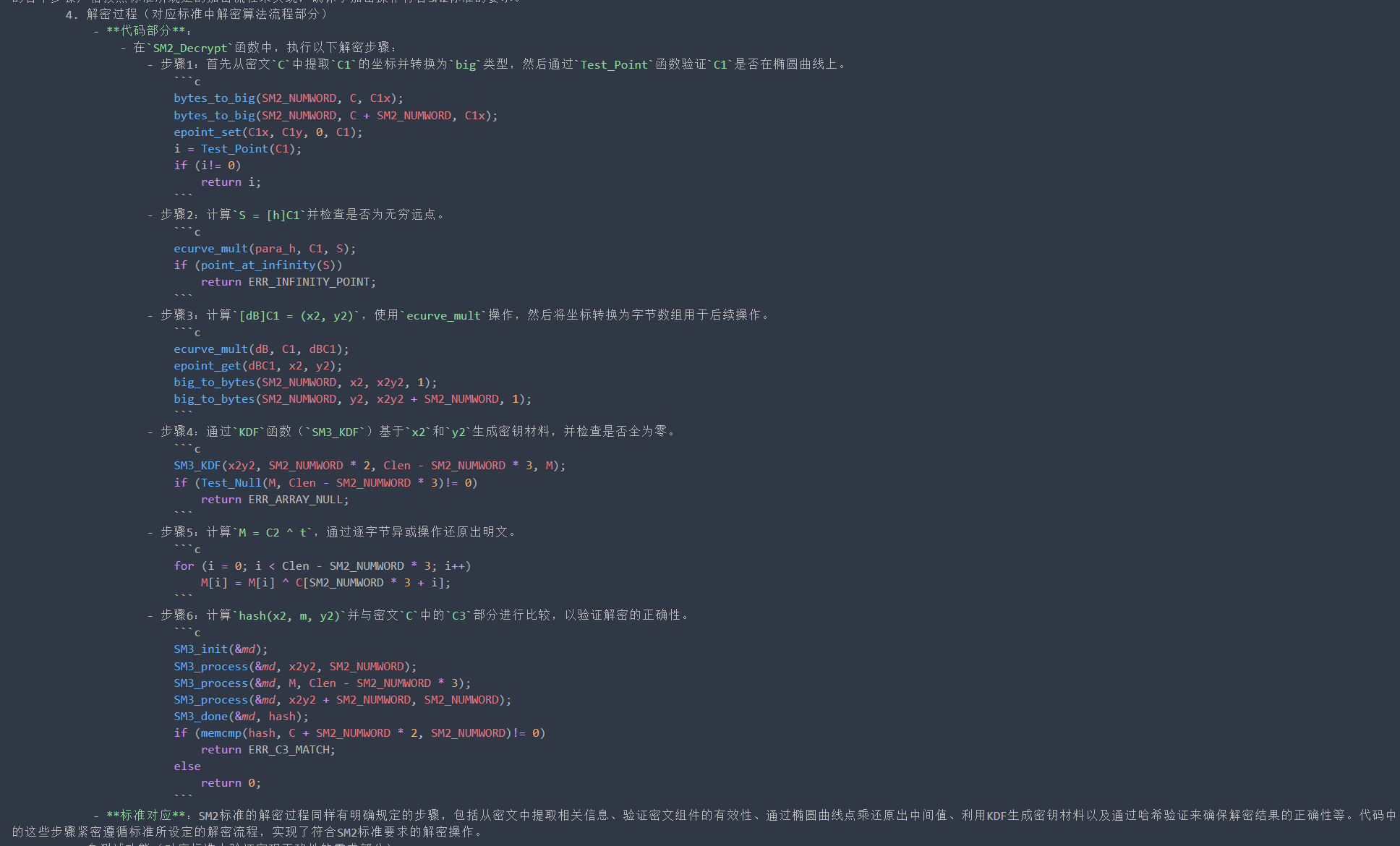


图 14 对SM2解密部分的代码的分析

**3．实验体会**

**3.1** 调试中出现的问题及解决过程

**3.1.1** 步骤一中SM2代码运行环境问题

表现：尽管在vs2022中运行良好，但只要我把代码移植到Ubuntu中，就会解密失败（报错代码：6）。哪怕是自检函数都无法运行成功（报错代码：9）。之前一直以为源代码与系统位数无关，并且编译的库的位数需要和系统位数一样，不然会报错，直到这时才意识到这可能都需要在32位中运行才可。

解决问题的过程：

1.首先是解决如何在64位系统的编译32位代码

2.按照AI的回复安装各种兼容的程序与C语言库

   3.改编译命令为：gcc -m32 -o SM2\_ENC SM2\_ENC.c -lmiracl

   4.此时仍会报错，因为miracl是64位的，我们需要编译一个32位的miracl库——主要是生成一个miracl\_32.a文件，因为头文件两者是一致的。

   5.按照当初64位的编译方式编译，但改变“bash linux64”为“bash linux”

.将新生成的库复制到用户文件夹：cp ./miracl.a /usr/lib/libmiracl\_32.a，不需要复制新的头文件到用户文件夹中，同样，原先代码中的"miracl/miracl.h"等不需要改变，因为头文件不变。

6.最后，修改编译命令为：gcc -m32 -o SM2\_ENC SM2\_ENC.c -lmiracl\_32

经验教训：

1.像这种处理字节的代码对系统位数比较敏感，要额外注意其适用于什么操作系统。

   2.重视自检函数的作用，优先判断其自检函数能否运行，再进行自己的main函数实现。同时学习将自检函数一步步转换成自己的测试函数。

3.1.2 步骤一中SM2密钥交换报错

在运行SM2密钥交换的代码时，发现代码在VS2022中可以正常运行，但在Ubuntu中会报错：

SM2\_KEY\_EX.c: In function ‘SM3\_Z’:

SM2\_KEY\_EX.c:95:23: error: lvalue required as unary ‘&’ operand

95 | memcpy(IDlen, &(unsigned char)ELAN + 1, 1);

| ^

SM2\_KEY\_EX.c:96:27: error: lvalue required as unary ‘&’ operand

96 | memcpy(IDlen + 1, &(unsigned char)ELAN, 1);

| ^

解决过程：

1. 首先不要直接就去调试。要先尝试更加简便的方法。
2. 我们可以尝试抓住问题的本质。先搞明白原本的代码的作用是什么，再尝试他有没有其他的合法的表达。
3. 通过AI（[对话链接](https://share.aichatos58.com/" \l "/share/5cca84d9c38549229319c95bbf1e0f33)）完成第二步，即可得到正确的代码：

memcpy(IDlen, (unsigned char\*)&ELAN + 1, 1);

memcpy(IDlen + 1, (unsigned char\*)&ELAN, 1);

原理解释：原本代码的意思是：将 ELAN 的下一个字节（即 ELAN 的内存地址加 1 处的字节）复制到 IDlen 指向的内存位置中。如果不希望对 ELAN 进行类型转换，但仍想取下一个字节，可以直接使用指针算术。

至于为什么在不同环境下编译结果不同，这是因为不同环境下C语言实现有差异，vs2022更加宽松一些。可以参考[这个对话](https://www.doubao.com/thread/w8f20d21b4f59d489)的解释（但他提供的解决方法都是错的）。

3.1.3 4-1代码（一段式SM3算法）编译报错与运行结果不正确

报错信息：

test.cpp: In function ‘long unsigned int SL(long unsigned int, int)’:

test.cpp:17:26: error: expected initializer before ‘x’

17 | unsigned \_\_int64 x = X;

| ^

test.cpp:18:9: error: ‘x’ was not declared in this scope

18 | x = x << (n % 32);

| ^

test.cpp: In function ‘void SM3Hash(unsigned char\*, int, unsigned char\*)’:

test.cpp:154:9: error: ‘memset’ was not declared in this scope

154 | memset(m1, 0, m1l);

| ^~~~~~

test.cpp:9:1: note: ‘memset’ is defined in header ‘<cstring>’; did you forget to ‘#include <cstring>’?

8 | #include <memory>

+++ |+#include <cstring>

9 |

test.cpp:155:9: error: ‘memcpy’ was not declared in this scope

155 | memcpy(m1, m, l / 8);

| ^~~~~~

test.cpp:155:9: note: ‘memcpy’ is defined in header ‘<cstring>’; did you forget to ‘#include <cstring>’?

test.cpp: At global scope:

test.cpp:223:1: error: ‘::main’ must return ‘int’

223 | void main()

| ^~~~

这次报错可能是因为源代码有些古老了，不符合新的规范。将报错信息交给AI解决即可。

正确代码如下：

#include <stdio.h>

#include <memory>

#include <cstring>

#include <cstdint> // 包含标准整数类型

unsigned char IV[256 / 8] = { 0x73,0x80,0x16,0x6f,0x49,0x14,0xb2,0xb9,0x17,0x24,0x42,0xd7,0xda,0x8a,0x06,0x00,0xa9,0x6f,0x30,0xbc,0x16,0x31,0x38,0xaa,0xe3,0x8d,0xee,0x4d,0xb0,0xfb,0x0e,0x4e };

// 循环左移

unsigned long SL(unsigned long X, int n)

{

uint64\_t x = X; // 修改为标准类型

x = x << (n % 32);

unsigned long l = (unsigned long)(x >> 32);

return x | l;

}

unsigned long Tj(int j)

{

if (j <= 15)

{

return 0x79cc4519;

}

else

{

return 0x7a879d8a;

}

}

unsigned long FFj(int j, unsigned long X, unsigned long Y, unsigned long Z)

{

if (j <= 15)

{

return X ^ Y ^ Z;

}

else

{

return (X & Y) | (X & Z) | (Y & Z);

}

}

unsigned long GGj(int j, unsigned long X, unsigned long Y, unsigned long Z)

{

if (j <= 15)

{

return X ^ Y ^ Z;

}

else

{

return (X & Y) | (~X & Z);

}

}

unsigned long P0(unsigned long X)

{

return X ^ SL(X, 9) ^ SL(X, 17);

}

unsigned long P1(unsigned long X)

{

return X ^ SL(X, 15) ^ SL(X, 23);

}

// 扩展

void EB(unsigned char Bi[512 / 8], unsigned long W[68], unsigned long W1[64])

{

// Bi 分为W0~W15

for (int i = 0; i < 16; ++i)

{

W[i] = Bi[i \* 4] << 24 | Bi[i \* 4 + 1] << 16 | Bi[i \* 4 + 2] << 8 | Bi[i \* 4 + 3];

}

for (int j = 16; j <= 67; ++j)

{

W[j] = P1(W[j - 16] ^ W[j - 9] ^ SL(W[j - 3], 15)) ^ SL(W[j - 13], 7) ^ W[j - 6];

}

for (int j = 0; j <= 63; ++j)

{

W1[j] = W[j] ^ W[j + 4];

}

}

// 压缩函数

void CF(unsigned char Vi[256 / 8], unsigned char Bi[512 / 8], unsigned char Vi1[256 / 8])

{

// Bi 扩展为132个字

unsigned long W[68] = { 0 };

unsigned long W1[64] = { 0 };

EB(Bi, W, W1);

// 串联 ABCDEFGH = Vi

unsigned long R[8] = { 0 };

for (int i = 0; i < 8; ++i)

{

R[i] = ((unsigned long)Vi[i \* 4]) << 24 | ((unsigned long)Vi[i \* 4 + 1]) << 16 | ((unsigned long)Vi[i \* 4 + 2]) << 8 | ((unsigned long)Vi[i \* 4 + 3]);

}

unsigned long A = R[0], B = R[1], C = R[2], D = R[3], E = R[4], F = R[5], G = R[6], H = R[7];

unsigned long SS1, SS2, TT1, TT2;

for (int j = 0; j <= 63; ++j)

{

SS1 = SL(SL(A, 12) + E + SL(Tj(j), j), 7);

SS2 = SS1 ^ SL(A, 12);

TT1 = FFj(j, A, B, C) + D + SS2 + W1[j];

TT2 = GGj(j, E, F, G) + H + SS1 + W[j];

D = C;

C = SL(B, 9);

B = A;

A = TT1;

H = G;

G = SL(F, 19);

F = E;

E = P0(TT2);

}

// Vi1 = ABCDEFGH 串联

R[0] = A, R[1] = B, R[2] = C, R[3] = D, R[4] = E, R[5] = F, R[6] = G, R[7] = H;

for (int i = 0; i < 8; ++i)

{

Vi1[i \* 4] = (R[i] >> 24) & 0xFF;

Vi1[i \* 4 + 1] = (R[i] >> 16) & 0xFF;

Vi1[i \* 4 + 2] = (R[i] >> 8) & 0xFF;

Vi1[i \* 4 + 3] = (R[i]) & 0xFF;

}

// Vi1 = ABCDEFGH ^ Vi

for (int i = 0; i < 256 / 8; ++i)

{

Vi1[i] ^= Vi[i];

}

}

//参数 m 是原始数据，ml 是数据长度，r 是输出参数,存放hash结果

void SM3Hash(unsigned char\* m, int ml, unsigned char r[32])

{

int l = ml \* 8;

int k = 448 - 1 - l % 512; // 添加k个0，k 是满足 l + 1 + k ≡ 448mod512 的最小的非负整数

if (k <= 0)

{

k += 512;

}

int n = (l + k + 65) / 512;

int m1l = n \* 512 / 8; // 填充后的长度，512位的倍数

unsigned char\* m1 = new unsigned char[m1l];

memset(m1, 0, m1l);

memcpy(m1, m, l / 8);

m1[l / 8] = 0x80; // 消息后补1

// 再添加一个64位比特串，该比特串是长度l的二进制表示

unsigned long l1 = l;

for (int i = 0; i < 64 / 8 && l1 > 0; ++i)

{

m1[m1l - 1 - i] = l1 & 0xFF;

l1 = l1 >> 8;

}

//将填充后的消息m′按512比特进行分组：m′ = B(0)B(1)· · · B(n−1),其中n=(l+k+65)/512。

unsigned char\*\* B = new unsigned char\*[n];

for (int i = 0; i < n; ++i)

{

B[i] = new unsigned char[512 / 8];

memcpy(B[i], m1 + (512 / 8)\*i, 512 / 8);

}

delete[] m1;

unsigned char\*\* V = new unsigned char\*[n + 1];

for (int i = 0; i <= n; ++i)

{

V[i] = new unsigned char[256 / 8];

memset(V[i], 0, 256 / 8);

}

// 初始化 V0 = VI

memcpy(V[0], IV, 256 / 8);

// 压缩函数，V 与扩展的B

for (int i = 0; i < n; ++i)

{

CF(V[i], B[i], V[i + 1]);

}

for (int i = 0; i < n; ++i)

{

delete[] B[i];

}

delete[] B;

// V[n]是结果

memcpy(r, V[n], 32);

for (int i = 0; i <= n; ++i)

{

delete[] V[i];

}

delete[] V;

}

// 打印缓冲区内容

void dumpbuf(unsigned char\* buf, int len)

{

for (int i = 0; i < len; ++i) {

printf("%02x", buf[i]);

if ((i + 1) % 16 == 0)

printf("\n");

else

printf(" ");

}

printf("\n");

}

int main()

{

// 输入数据

unsigned char message[] = "abc";

int message\_len = strlen((char\*)message);

// 输出缓冲区

unsigned char hash\_result[32] = { 0 };

// 调用 SM3 哈希函数

SM3Hash(message, message\_len, hash\_result);

// 输出哈希结果

printf("SM3 Hash:\n");

dumpbuf(hash\_result, 32);

return 0;

}

但编译后运行发现’abc’的哈希结果是37bc43d1 1cab393d 7899ef62 24f568ec 18a8fd85 1d165c50 0c375402 0f466a04。这与标准值不符合。

当时的第一反应是程序大小端，因为SM3需要保持大端存储但在这个程序的实现中似乎没有相关功能的函数，同时，Ubuntu又是一个小端存储的系统。

第二反应则是运行环境位数的问题，之前按64位编译，该程序也可能应该在32位环境下运行。

第二个猜测更容易验证。我直接在编译命令后面加上了-m32，其结果就与标准哈希值一致了。

但理论上讲，这个程序确实没有处理字节序的功能且理应在大端系统中运行才好，为什么又可以出现正确结果呢？这可能是数据处理过程中的字节序补偿机制。在这个 SM3 哈希算法的实现代码中，虽然代码中存在一些按照大端序组合和拆分字节的操作，但实际上在整个哈希计算过程中，这些操作是相对 “对称” 的，从而在一定程度上抵消了字节序的影响。

对于一段式SM3算法，不必考虑大小端问题；对于三段式SM3算法来说，就会有一个处理机制：宏GET\_ULONG\_BE。其会根据不同端正确处理字节序。

#ifndef GET\_ULONG\_BE

#define GET\_ULONG\_BE(n,b,i)                             \

    {                                                       \

        (n) = ( (unsigned long) (b)[(i)    ] << 24 )        \

            | ( (unsigned long) (b)[(i) + 1] << 16 )        \

            | ( (unsigned long) (b)[(i) + 2] <<  8 )        \

            | ( (unsigned long) (b)[(i) + 3]       );       \

    }

#endif

#ifndef PUT\_ULONG\_BE

#define PUT\_ULONG\_BE(n,b,i)                             \

    {                                                       \

        (b)[(i)    ] = (unsigned char) ( (n) >> 24 );       \

        (b)[(i) + 1] = (unsigned char) ( (n) >> 16 );       \

        (b)[(i) + 2] = (unsigned char) ( (n) >>  8 );       \

        (b)[(i) + 3] = (unsigned char) ( (n)       );       \

    }

#endif

为什么会有这个差异，我们不妨看看宏GET\_ULONG\_BE都会用在哪里。

宏GET\_ULONG\_BE只用于sm3\_process函数。函数一开始使用GET\_ULONG\_BE宏从data数组中获取 16 个 32 位无符号长整数W[0] - W[15]，宏保证了无论是大端程序还是小端程序都可以得到一样的值。所以它是字节序问题可能导致哈希计算错误的主要环节，因为它涉及数据的读取、大量的中间计算和状态更新，这些操作都依赖于正确的字节序来保证计算结果的准确性。

3.1.4 步骤三SM2代码编译报错：重复定义与找不到定义

现象：按照源码编译，就会显示para\_p, para\_a等变量重复定义；如果在这些变量前面加上extern，又会显示这些变量找不到。

通过与AI对话，我学习了extern关键字的用法：extern关键字用于变量声明时，主要是告诉编译器变量的定义在其他地方，这样编译器在编译当前文件时，就知道该变量是一个外部变量，不会因为找不到变量定义而报错。

如果一个变量在头文件中被声明为 extern，那么它应该在某个 .c 文件中有且仅有一次定义。如果没有定义，链接器会报错说找不到该变量的定义；如果多个 .c 文件中都定义了同一个 extern 变量，链接器则会报错说有多重定义。

所以解决方法就是：

在SM2\_ENC.c中加入变量的定义

big para\_p, para\_a, para\_b, para\_n, para\_Gx, para\_Gy, para\_h;

epoint \*G;

miracl \*mip;

在SM2\_ENC.h中给原先变量加上extern守卫

extern big para\_p, para\_a, para\_b, para\_n, para\_Gx, para\_Gy, para\_h;

extern epoint \*G;

extern miracl \*mip;

对于签名的相关代码也做如此修改。

**3.2 心得体会**

这次实践了“先易后难”和“抓住本质”的原则，先做SM3再做SM4，最后完成SM2，同时解决问题中尝试思考问题的本质是什么，我的解决思路是否正确等问题，这些都帮助我更快、更深入地解决了许多问题。

我对 SM2、SM3、SM4 算法的标准与实现有了更深入的理解，特别是它们在不同环境下的运行特点和可能出现的问题。例如，SM3 算法在处理字节序方面有其独特之处，一段式 SM3 算法虽然没有明确处理大小端的函数，但在数据处理过程中存在字节序补偿机制；而三段式 SM3 算法则通过宏 GET\_ULONG\_BE 来正确处理字节序。

**3.3 实验改进建议**

在实验开始前，可以更加详细地介绍实验环境的配置要求和方法。例如，如何在 Ubuntu 中安装和编译 Miracl 库，以及如何处理 32 位和 64 位环境的问题。这样可以避免学生在环境配置上花费过多的时间和精力，提高实验效率。