

《应用密码学》实验报告

课程: 应用密码学 实验名称: RSA 签名算法的实现

姓名: 杨佳妮 实验日期: 2024.5.6

学号: 2022132006 实验报告日期: 2024.5.6

班级： 信安实验 221

教师评语:	成绩:
签名:	
日期:	

一、实验名称

RSA 签名算法的实现

二、实验环境（详细说明运行的系统、平台及代码等）

- 1 平台: VC
2.操作系统: Windows

三、实验目的

- (1) 加深对数字签名算法的理解;
- (2) 加深消息摘要函数 SHA-1 的掌握;
- (3) 提高编程实践能力。

四、实验内容、步骤及结果

1. 实验内容

- (1) 输入字符作为原始消息，采用 SHA-1 算法，输出固定长度 160 比特的消息摘要；

(2) 将输出的消息摘要作为输入，采用 RSA 算法的私钥进行签名，得到签名消息；

(3) 将原始消息和签名消息作为输入，采用 RSA 算法的公钥进行签名验证。

2. 实验结果

(1)、第 1 组测试要求: 1 组固定的数值设置, RSA 算法测试, $e=5$, $p=23$, $q=29$; 输入为 ASCII 码, 程序的默认输入为 FIPS-180-2 中示例的 “abc”

```
17 //迭代数初始化
18 void IterationValueInit_06(SHAValues* shaValues)
19 {
20     shaValues->IterationValue[0] = 0x67452301;
21     shaValues->IterationValue[1] = 0xEFCDA8B9;
22     shaValues->IterationValue[2] = 0x98BADCFE;
23     shaValues->IterationValue[3] = 0x;
24     shaValues->IterationValue[4] = 0x;
25 }
26
27 //获取无符号字节数组的长度, ustr是无符号
28 size_t strlen_06(uint8_t* ustr)
29 {
30     size_t size = 0;
31     while (ustr[size++] != '\0');
32     size--;
33     return size;
34 }
35
36 //逻辑函数
37 int ft_06(uint32_t B, uint32_t C, uint32_t A)
38 {
39     switch (num)
40     {
41     case 0:
42         //...
43     }
44 }
```

```
C:\WINDOWS\system32\cmd. x + v
输入待Hash的消息M: abc
请顺序输入公钥, 两素数: e p q: 5 23 29
输出模数n: 667
输出模数n的欧拉函数: 616
输出私钥d: 493
输出消息摘要值: a993e36 4706816a ba3e2571 7850c26c 9cd0d89d
开始计算签名值计算
第1字节的摘要值: a9=169
第1个字节的签名s值: 78
第2字节的摘要值: 99=153
第2个字节的签名s值: 474
第3字节的摘要值: 3e=62
第3个字节的签名s值: 238
第4字节的摘要值: 36=54
第4个字节的签名s值: 400
第5字节的摘要值: 47=71
第5个字节的签名s值: 167
第6字节的摘要值: 06=6
第6个字节的签名s值: 361
第7字节的摘要值: 81=129
第7个字节的签名s值: 573
第8字节的摘要值: 6a=106
```

```
17 //迭代数初始化
18 void IterationValueInit_06(SHAValues* shaValues)
19 {
20     shaValues->IterationValue[0] = 0x67452301;
21     shaValues->IterationValue[1] = 0xEFCDA8B9;
22     shaValues->IterationValue[2] = 0x98BADCFE;
23     shaValues->IterationValue[3] = 0x;
24     shaValues->IterationValue[4] = 0x;
25 }
26
27 //获取无符号字节数组的长度, ustr是无符号
28 size_t strlen_06(uint8_t* ustr)
29 {
30     size_t size = 0;
31     while (ustr[size++] != '\0');
32     size--;
33     return size;
34 }
35
36 //逻辑函数
37 int ft_06(uint32_t B, uint32_t C, uint32_t A)
38 {
39     switch (num)
40     {
41     case 0:
42         //...
43     }
44 }
```

```
C:\WINDOWS\system32\cmd. x + v
请输入签名验证消息M: abc
输入签名消息s (十六进制): 4e 1da ee 198 a7 169 23d 9f 28a ee 26b c9 24a 127 70 4d 17c 1
待验证消息Hash值计算结果:
输出摘要值(十六进制): a993e36 4706816a ba3e2571 7850c26c 9cd0d89d
Hash的值第1字节: a9
验证计算签名的哈希值h': a9
第1字节签名验证通过
Hash的值第2字节: 99
验证计算签名的哈希值h': 99
第2字节签名验证通过
Hash的值第3字节: 3e
验证计算签名的哈希值h': 3e
第3字节签名验证通过
Hash的值第4字节: 36
验证计算签名的哈希值h': 36
第4字节签名验证通过
Hash的值第5字节: 47
验证计算签名的哈希值h': 47
第5字节签名验证通过
Hash的值第6字节: 06
验证计算签名的哈希值h': 06
第6字节签名验证通过
Hash的值第7字节: 81
```

```
17 //迭代数初始化
18 void IterationValueInit_06(SHA1Values* sha1Values)
19 {
20     sha1Values->IterationValue[0] = 0x67E;
21     sha1Values->IterationValue[1] = 0xEF;
22     sha1Values->IterationValue[2] = 0x98;
23     sha1Values->IterationValue[3] = 0x10;
24     sha1Values->IterationValue[4] = 0xC3;
25 }
26
27 //获取无符号字节数组的长度, ustr是无符号字
28 size_t strlen_06(uint8_t* ustr)
29 {
30     size_t size = 0;
31     while (ustr[size++] != '\0');
32     size--;
33     return size;
34 }
35
36 //逻辑函数
37 int ft_06(uint32_t B, uint32_t C, uint32_t A)
38 {
39     switch (num)
40     {
41     case 0:
42         //...
43     }
44 }
```

Hash的值第14字节: 50
验证计算签名的哈希值h': 50
第14字节签名验证通过

Hash的值第15字节: c2
验证计算签名的哈希值h': c2
第15字节签名验证通过

Hash的值第16字节: 6c
验证计算签名的哈希值h': 6c
第16字节签名验证通过

Hash的值第17字节: 9c
验证计算签名的哈希值h': 9c
第17字节签名验证通过

Hash的值第18字节: d0
验证计算签名的哈希值h': d0
第18字节签名验证通过

Hash的值第19字节: d8
验证计算签名的哈希值h': d8
第19字节签名验证通过

Hash的值第20字节: 9d
验证计算签名的哈希值h': 9d
第20字节签名验证通过

编译成功!

(2)、

```
51 }
52
53 //64字节数据块处理
54 void blocksProcess_06(uint8_t* bytes, SHA1Values* sha1Values)
55 {
56     static int iteration_times = 1;
57     int i = 0;
58     uint32_t A = sha1Values->Iter;
59     uint32_t B = sha1Values->Iter;
60     uint32_t C = sha1Values->Iter;
61     uint32_t D = sha1Values->Iter;
62     uint32_t E = sha1Values->Iter;
63
64     //对于每个512位的明文分块, sha1
65     int bytes_divided_512bits[16];
66     //int bytes_divided_512bits[16];
67     for (i = 0; i < 16; i++)
68     {
69         for (int j = 0; j < 4; j++)
70         {
71             bytes_divided_512bits[i]
72             bytes_divided_512bits[i]
73         }
74     }
75 }
```

输入待Hash的消息M: 2022132006
请顺序输入公钥, 两素数: e p q:5 23 29
输出模数n: 667
输出模数n的欧拉函数: 616
输出私钥d: 493
输出消息摘要值: 0a9aa3ef 01243caf ba80487b ba424f2f e8196200

开始计算签名值计算

第1字节的摘要值: 0a=10
第1个字节签名s值: 595

第2字节的摘要值: 9a=154
第2个字节签名s值: 468

第3字节的摘要值: a3=163
第3个字节签名s值: 374

第4字节的摘要值: ef=239
第4个字节签名s值: 140

第5字节的摘要值: 01=1
第5个字节签名s值: 1

第6字节的摘要值: 24=36
第6个字节签名s值: 256

第7字节的摘要值: 3c=60
第7个字节签名s值: 21

第8字节的摘要值: af=175

```
51 }
52
53 //64字节数据块处理
54 void blocksProcess_06(uint8_t* bytes, SHAValues* shaValues)
55 {
56     static int iteration_times = 1;
57     int i = 0;
58     uint32_t A = shaValues->iterationValue[0];
59     uint32_t B = shaValues->iterationValue[1];
60     uint32_t C = shaValues->iterationValue[2];
61     uint32_t D = shaValues->iterationValue[3];
62     uint32_t E = shaValues->iterationValue[4];
63
64     //对于每个512位的明文分组，SHA1将其分 (int)0
65     int bytes_divided_512bits[16] = { 0 };
66     //int bytes_divided_512bits_size = sizeof
67     for (i = 0; i < 16; i++)
68     {
69         for (int j = 0; j < 4; j++)
70         {
71             bytes_divided_512bits[i] <<= 8;
72             bytes_divided_512bits[i] += bytes
73         }
74     }
75 }
```

请输入签名验证消息M: 2022132086
输入签名消息s (十六进制) : 253 1d4 176 8c 01 100 15 140 28a 12e 9c 11d 28a 61 87 1cd
待验证消息Hash值计算结果:
输出摘要值(十六进制): 0a9aa3ef 01243caf ba80487b ba424f2f e8196200
Hash的值第1字节: 0a
验证计算签名的哈希值h': 0a
第1字节签名验证通过
Hash的值第2字节: 9a
验证计算签名的哈希值h': 9a
第2字节签名验证通过
Hash的值第3字节: a3
验证计算签名的哈希值h': a3
第3字节签名验证通过
Hash的值第4字节: ef
验证计算签名的哈希值h': ef
第4字节签名验证通过
Hash的值第5字节: 01
验证计算签名的哈希值h': 01
第5字节签名验证通过
Hash的值第6字节: 24
验证计算签名的哈希值h': 24
第6字节签名验证通过
Hash的值第7字节: 3c

```
51 }
52
53 //64字节数据块处理
54 void blocksProcess_06(uint8_t* bytes, SHAValues* shaValues)
55 {
56     static int iteration_times = 1;
57     int i = 0;
58     uint32_t A = shaValues->iterationValue[0];
59     uint32_t B = shaValues->iterationValue[1];
60     uint32_t C = shaValues->iterationValue[2];
61     uint32_t D = shaValues->iterationValue[3];
62     uint32_t E = shaValues->iterationValue[4];
63
64     //对于每个512位的明文分组，SHA1将其再分成16
65     int bytes_divided_512bits[16] = { 0 };
66     //int bytes_divided_512bits_size = sizeof
67     for (i = 0; i < 16; i++)
68     {
69         for (int j = 0; j < 4; j++)
70         {
71             bytes_divided_512bits[i] <<= 8;
72             bytes_divided_512bits[i] += bytes
73         }
74     }
75 }
```

Hash的值第14字节: 42
验证计算签名的哈希值h': 42
第14字节签名验证通过
Hash的值第15字节: 4f
验证计算签名的哈希值h': 4f
第15字节签名验证通过
Hash的值第16字节: 2f
验证计算签名的哈希值h': 2f
第16字节签名验证通过
Hash的值第17字节: e8
验证计算签名的哈希值h': e8
第17字节签名验证通过
Hash的值第18字节: 19
验证计算签名的哈希值h': 19
第18字节签名验证通过
Hash的值第19字节: 62
验证计算签名的哈希值h': 62
第19字节签名验证通过
Hash的值第20字节: 00
验证计算签名的哈希值h': 00
第20字节签名验证通过
请按任意键继续. . .

五、实验中的问题及心得

对于 RSA 签名算法的实现，在编写程序过程中，核心部分的程序是 RSA 签名算法的逻辑实现（综合了 sha-1 和 rsa 加解密算法），课后自己实现了一遍代码，对 RSA 签名算法的实现有了更深的了解和掌握，也掌握了 RSA 签名算法的实现的规则。

附件：程序代码

```
#define _CRT_SECURE_NO_WARNINGS
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <stdint.h>
```

```
#include <math.h>
```

```
#define CHUNK 1024
```

```
const uint32_t K[] = { 0x5A827999,0x6ED9EBA1,0x8F1BBCDC,0xCA62C1D6 };
```

```
typedef struct
```

```
{
```

```
    uint32_t IterationValue[5]; //存放 5 个迭代数
```

```
}SHA1Values;
```

```
//迭代数初始化
```

```
void IterationValueInit_06(SHA1Values* sha1Values)
```

```
{
```

```
    sha1Values->IterationValue[0] = 0x67452301;
```

```
    sha1Values->IterationValue[1] = 0xEFCDAB89;
```

```
    sha1Values->IterationValue[2] = 0x98BADCFE;
```

```
    sha1Values->IterationValue[3] = 0x10325476;
```

```
    sha1Values->IterationValue[4] = 0xC3D2E1F0;
```

```
}
```

```
//获取无符号字节数组的长度，ustr 是无符号字节数组的首地址指针
```

```
size_t strlen_06(uint8_t* ustr)
```

```
{
```

```
    size_t size = 0;
```

```
    while (ustr[size++] != '\0');
```

```
    size--;
```

```
    return size;
```

```
}
```

```
//逻辑函数
```

```
int ft_06(uint32_t B, uint32_t C, uint32_t D, uint32_t num)
```

```
{
```

```
    switch (num)
```

```
    {
```

```
        case 0:
```

```

        return (B & C) | (~B & D);
    case 1:
        return B ^ C ^ D;
    case 2:
        return (B & C) | (B & D) | (C & D);
    case 3:
        return B ^ C ^ D;
    }
    return 0;
}

```

```

//64 字节数据块处理
void blocksProcess_06(uint8_t* bytes, SHA1Values* sha1Values)
{
    static int iteration_times = 1;
    int i = 0;
    uint32_t A = sha1Values->IterationValue[0];
    uint32_t B = sha1Values->IterationValue[1];
    uint32_t C = sha1Values->IterationValue[2];
    uint32_t D = sha1Values->IterationValue[3];
    uint32_t E = sha1Values->IterationValue[4];
}

```

```

//对于每个 512 位的明文分组，SHA1 将其再分成 16 份更小的明文分组,M[t](t= 0, 1,...,15)
int bytes_divided_512bits[16] = { 0 };
//int bytes_divided_512bits_size =
sizeof(bytes_divided_512bits)/sizeof(bytes_divided_512bits[0]);
for (i = 0; i < 16; i++)
{
    for (int j = 0; j < 4; j++)
    {
        bytes_divided_512bits[i] <= 8;
        bytes_divided_512bits[i] += bytes[j + 4 * i];
    }
}
}

```

```

//将这 16 个子明文分组扩充到 80 个子明文分组，记为 W[t](t= 0, 1,...,79)
int bytes_divided2_512bits[80] = { 0 };
//当 0<t<15 时，Wt = Mt
for (i = 0; i < 16; i++)
{
    bytes_divided2_512bits[i] = bytes_divided_512bits[i];
}
//当 16<t<79 时，Wt = ( Wt-3 ⊕ Wt-8⊕ Wt-14⊕ Wt-16) <<< 1
for (i = 16; i < 80; i++)

```

```

{
    bytes_divided2_512bits[i] = _rotr(
        bytes_divided2_512bits[i - 3] ^ bytes_divided2_512bits[i - 8] ^
        bytes_divided2_512bits[i - 14] ^ bytes_divided2_512bits[i - 16], 1);
}

```

```

//80 次迭代，分 4 组进行，每组迭代常数和逻辑函数组合不同
for (i = 0; i < 80; i++)
{
    uint32_t temp = _rotr(A, 5) + E + bytes_divided2_512bits[i] + K[i / 20] + ft_06(B, C, D,
i / 20);
    E = D;
    D = C;
    C = _rotr(B, 30);
    B = A;
    A = temp;
}
sha1Values->IterationValue[0] += A;
sha1Values->IterationValue[1] += B;
sha1Values->IterationValue[2] += C;
sha1Values->IterationValue[3] += D;
sha1Values->IterationValue[4] += E;
}

```

```

//对 str 进行 SHA1 摘要计算，并把结果赋给 data
SHA1Values SHA1Encrypt_06(uint8_t* str, SHA1Values sha1Values)
{
    //SHA1Values sha1Values = { 0 };
    IterationValueInit_06(&sha1Values); //对 5 个迭代常数进行初始化
    long str_size = strlen_06(str); //获取无符号字符数组的长度
    uint8_t bytes_512bits[64] = { 0 };
    int packets = (int)ceil(str_size * 8.0 / 512); //以 512 比特为单位获取数据的分块个数
    int left_packets = packets; //剩余未处理的分块个数
    long str_index = 0; //数据索引位置

```

```

while (left_packets > 0)
{
    //当剩余未处理的分块个数为 1 是执行以下代码
    if (left_packets == 1)
    {
        int j = 0;

```

```

//按 512 位的长度进行分组
//要注意处理的特例是 str_size 等于 64，它为 64 时不进入下面的 for 循环

```

```

if (str_size % 64 != 0)
    for (; j < str_size % 64; j++)
    {
        bytes_512bits[j] = str[str_index];
        str_index++;
    }
else
{
    for (; j < 64; j++)
    {
        bytes_512bits[j] = str[str_index];
        str_index++;
    }
    blocksProcess_06(bytes_512bits, &sha1Values);
    j = 0;
}

```

//将最后一组数据添入数组以后，判断这组数据的长度，如果小于 56，那么直接处理这些数据以后就结
束了

//如果大于等于 56，则说明还需要再多处理一组数据，因为根据算法流程，需要在最后一组数据后先添
1，

//然后补零直至数据长度对 512 求余后为 448，如果最后一组数据长度大于等于 56，添 1 以后补 0 会超
过当前

```

//64 字节的数据块，所以要多处理一组数据。
if (j >= 56)
{
    bytes_512bits[j++] = 0x80;
    for (; j < 64; j++)
        bytes_512bits[j] = 0x00;
    blocksProcess_06(bytes_512bits, &sha1Values);
    for (j = 0; j < 56; j++)
        bytes_512bits[j] = 0x00;
}
else
{
    bytes_512bits[j++] = 0x80;
    for (; j < 56; j++)
        bytes_512bits[j] = 0x00;
}
//将有效数据长度用 8 字节表示填充到数据末尾
for (j = 63; j >= 56; j--)
{
    long str_size1 = str_size * 8;
    for (int k = 0; k < 63 - j; k++)
        str_size1 >>= 8;
}

```



```

        bytes_512bits[j] = str_size1 % 256;
    }
    blocksProcess_06(bytes_512bits, &sha1Values);
}
//当数据分块个数超过 1 个是执行以下代码
else
{
    //以 512 比特（64 字节）为单位对数组进行赋值
    for (int j = 0; j < 64; j++)
    {
        bytes_512bits[j] = str[str_index];
        str_index++;
    }
    //对 64 字节空间的数组进行处理，将迭代值赋给 sha1Values
    blocksProcess_06(bytes_512bits, &sha1Values);
}
left_packets--;
}
/*sprintf_s(data,45, "%08x %08x %08x %08x %08x",
    sha1Values.IterationValue[0],
    sha1Values.IterationValue[1],
    sha1Values.IterationValue[2],
    sha1Values.IterationValue[3],
    sha1Values.IterationValue[4]);
printf("消息摘要为: %s\n", data);*/
return sha1Values;
}

```

```

//接收用户输入，为防止溢出，给指针动态分配内存
char* readinput_06()
{
    char* old_input = NULL;
    char* new_input = NULL;
    char tempbuf[CHUNK];
    size_t inputlen = 0, templen = 0;
    rewind(stdin);
    do {
        fgets(tempbuf, CHUNK, stdin);
        templen = strlen(tempbuf);
        new_input = (char*)realloc(old_input, inputlen + templen + 1);
        if (new_input == NULL)
        {
            return NULL;
        }
    }
}

```

```

        old_input = new_input;
        memcpy(new_input + inputlen, tempbuf, templen + 1);
        inputlen += templen;
    } while (templen == CHUNK - 1 && tempbuf[CHUNK - 2] != '\n');
    new_input[strlen(new_input) - 1] = '\0';
    return new_input;
}

```

```

// 模幂运算
uint32_t mod_exp_06(uint32_t base, uint32_t exponent, uint32_t mod) {
    uint32_t result = 1;
    base = base % mod;
    while (exponent > 0) {
        if (exponent % 2 == 1)
            result = (result * base) % mod;

```

```

        base = (base * base) % mod;
        exponent >>= 1;
    }
    return (uint32_t)result;
}

```

```

// RSA 数字签名
void rsa_sign_06(unsigned int* c, uint32_t d, uint32_t n, uint32_t* signature, unsigned int* byte)
{
    for (int i = 0; i < 5; i++) {
        // 提取 c[i] 中的四个字节
        byte[0 + i * 4] = (c[i] >> 24) & 0xFF; // 获取高 8 位
        byte[1 + i * 4] = (c[i] >> 16) & 0xFF; // 获取次高 8 位
        byte[2 + i * 4] = (c[i] >> 8) & 0xFF; // 获取次低 8 位
        byte[3 + i * 4] = c[i] & 0xFF; // 获取低 8 位

```

```

    }
    for (int j = 0; j < 4; j++) {
        signature[j + i * 4] = mod_exp_06(byte[j + i * 4], d, n);
    }
}

```

```

void rsa_verify_06(uint32_t* signature, uint32_t e, uint32_t n, unsigned int* verify) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 4; j++) {
            verify[j + i * 4] = mod_exp_06(signature[j + i * 4], e, n);
        }
    }
}

```

```
}
```

```
int main() {  
    int input_num, i = 0;  
    uint8_t* str_1 = NULL;  
    uint8_t* str_2 = NULL;  
    SHA1Values sha1Values_1 = { 0 };  
    SHA1Values sha1Values_2 = { 0 };  
    printf("输入待 Hash 的消息 M: ");  
    //读取输入的数据，并将最终结果的字符首地址赋给 str  
    str_1 = (uint8_t*)readinput_06();
```

```
    // 输入公钥，两素数: e, p, q  
    uint32_t e, p, q;  
    printf("请顺序输入公钥，两素数: e p q:");  
    scanf("%u %u %u", &e, &p, &q);
```

```
    // 计算模数 n 和私钥 d  
    uint32_t n = p * q;  
    uint32_t phi = (p - 1) * (q - 1);  
    uint32_t d = 0;  
    while ((d * e) % phi != 1) {  
        d++;  
    }
```

```
    // 输出模数 n，欧拉函数，私钥 d  
    printf("输出模数 n: %u\n", n);  
    printf("输出模数 n 的欧拉函数: %u\n", phi);  
    printf("输出私钥 d: %u\n", d);
```

```
    // 计算消息摘要  
    sha1Values_1 = SHA1Encrypt_06(str_1, sha1Values_1);  
    printf("输出消息摘要值: %08x %08x %08x %08x %08x\n", sha1Values_1.IterationValue[0],  
    sha1Values_1.IterationValue[1], sha1Values_1.IterationValue[2], sha1Values_1.IterationValue[3],  
    sha1Values_1.IterationValue[4]);  
    printf("\n 开始计算签名值计算\n");
```

```
    // RSA 数字签名  
    uint32_t signature[20];  
    unsigned int byte[20] = { 0 };  
    rsa_sign_06(sha1Values_1.IterationValue, d, n, signature, byte);
```

```
    // 输出签名值  
    for (int i = 0; i <= 4; i++) {
```

```

        for (int j = 0; j <= 3; j++) {
            printf("第%d 字节的摘要值: %02x=%d\n", j + 1 + i * 4, byte[j + i * 4], byte[j + i * 4]);
            printf("第%d 个字节签名 s 值: %u\n", j + 1 + i * 4, signature[j + i * 4]);
            printf("\n");
        }
    }
}

```

```

// 验证签名

printf("请输入签名验证消息 M: ");

str_2 = (uint8_t*)readinput_06();

```

```

printf("输入签名消息 s (十六进制): ");

for (int i = 0; i < 5; i++) {
    for (int j = 0; j <= 3; j++) {
        printf("%02x ", signature[j + i * 4]);
    }
}

printf("\n");

```

```

// 哈希验证

unsigned int verify[20] = { 0 };

rsa_verify_06(signature, e, n, verify);

```

```

printf("待验证消息 Hash 值计算结果: \n");

// 计算消息摘要

sha1Values_2 = SHA1Encrypt_06(str_2, sha1Values_2);

printf("输出摘要值(十六进制): %08x %08x %08x %08x %08x\n", sha1Values_2.IterationValue[0],
sha1Values_2.IterationValue[1], sha1Values_2.IterationValue[2], sha1Values_2.IterationValue[3],
sha1Values_2.IterationValue[4]);

```

```

for (int i = 0; i <= 4; i++) {
    // 提取 sha1Values_2.IterationValue[i] 中的四个字节
    byte[0 + i * 4] = (sha1Values_2.IterationValue[i] >> 24) & 0xFF; // 获取高 8 位
    byte[1 + i * 4] = (sha1Values_2.IterationValue[i] >> 16) & 0xFF; // 获取次高 8 位
    byte[2 + i * 4] = (sha1Values_2.IterationValue[i] >> 8) & 0xFF; // 获取次低 8 位
    byte[3 + i * 4] = sha1Values_2.IterationValue[i] & 0xFF; // 获取低 8 位
    for (int j = 0; j <= 3; j++) {
        printf("Hash 的值第%d 字节: %02x\n", j + 1 + i * 4, byte[j + i * 4]);
        printf("验证计算签名的哈希值 h': %02x\n", verify[j + i * 4]);
        if (verify[j + i * 4] != byte[j + i * 4]) {
            printf("签名验证失败\n");
            return 0;
        }
    }
    else {

```

```
        printf("第%d 字节签名验证通过\n", j + 1 + i * 4);  
        printf("\n");  
    }  
}  
}
```

```
free(str_1);  
free(str_2);
```

```
return 0;  
}
```