《现代密码学》实验报告

实验名称：RSA加密的实现	实验时间：2024年12月18日
学生姓名：黄集瑞	学号：22336090
学生班级：22保密管理	

一、实验目的

> 本实验通过让同学们学习和实现 RSA 加密算法以及填充方案（OAEP），帮助同学们理解公钥密码学中 RSA 加密的基本原理及其安全性问题，加深了同学们对RSA 算法的数学基础的掌握，也增强了对现代加密系统中填充技术（OAEP）重要性的理解。

二、实验内容

> - 用C++实现RSA的加密过程
> - 输入如下：
>
> `uint8_t[256]` 以大端序表示的2048bit的n
>
> `uint8_t[256]` 以大端序表示的2048bit的e
>
> `uint8_t` 加密的消息长度（不超过190bytes）
>
> `uint8_t[]` 加密的消息内容
>
> - 输出如下：
>
> 以大端序输出的256bytes的二进制加密信息
>
> - 需要实现OAEP填充（最优非对称加密填充）

三、实验原理

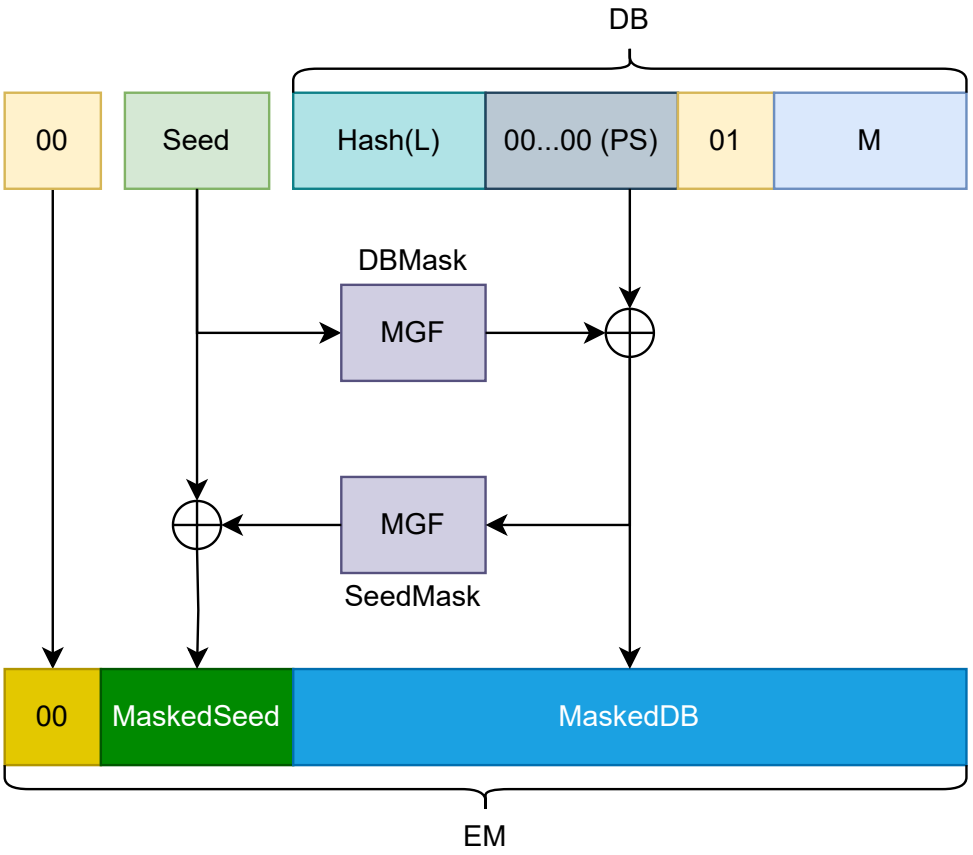
> 本人认为本次实验的实现共分为两个部分，一是大数运算的实现；另外一个是OAEP填充的实现，只要完成这两个部分的内容，剩下的加密原理就相当简单了。
>
> - 大数运算：
>
> 该部分算是本次实验的计算核心了，只要能够写出大数运算那么整个实验的难度其实就会降低很多。（虽然我添加到RSA的实现代码时，调了4、5个小时的bug）相比于样例0-3实验而说，本次实验输入的大数是以二进制流输入的，这样就少掉了将十进制数据转换为二进制的操作，并且能够更加适配 `<stdint.h>` 中的数据类型。（这里庆幸之前的大数运算就是用数组来实现的，不然这次实验又要动以前的框架了）
>
> - OAEP填充：
>
> 一开始看到这个填充的时候，我是一脸茫然。好在当时在课上询问了助教，搞懂了OAEP填充，其中每个填充部分的长度都是可计算的，重点实现逻辑集中到了MGF函数的实现。（这里需要用到上次实验的sha-256函数，上次实验纯纯手搓一点资料没看，导致本次实验完成的时候，可扩展性很差，导致我调了将近10个小时）
>
> $$MGF_H(m, len) = (H(m||0)||H(m||1)||H(m||2)||\dots)[:len]$$
>
> - 其中，数字（0，1.....）代表计数器用4bytes的大端序来表示。
>
> eg：比如现在计数为1，那么就要填充4个字节，这4个字节分别是 `0x00`，`0x00`，`0x00`，`0x01`

- `len`表示你最终想要的掩码长度，不断在增加计数并且进行拼接，直到输出达到指定的长度为止，然后截断多余的部分。

四、实验步骤

- OAEP填充：

填充的过程如图所示：



该填充中的每个部分的长度都是可以判断：

- 首位填充（1个字节）：0x00
- 随机的二进制数据（32个字节）：seed
- 可选的哈希函数（32个字节）：Hash（L），标签默认为空消息
- 填充长度（? 个字节）：可以通过计算得到256-1-32-32-1-消息长度
- 末位填充（1个字节）：0x01
- 消息信息（? 个字节）：由于填充的限制，最长不超过190字节（此时填充长度为0）

具体实现代码如下所示：

```
const uint8_t hash_0[32] =
    {
        0xe3, 0xb0, 0xc4, 0x42, 0x98, 0xfc, 0x1c, 0x14,
        0x9a, 0xfb, 0xf4, 0xc8, 0x99, 0x6f, 0xb9, 0x24,
        0x27, 0xae, 0x41, 0xe4, 0x64, 0x9b, 0x93, 0x4c,
        0xa4, 0x95, 0x99, 0x1b, 0x78, 0x52, 0xb8, 0x55};
void OAEP(uint8_t pad, uint8_t len, uint64_t *data, uint8_t *EM)
{
    uint8_t DB[DB_len]; // 数据块
    // 首先，填充Hash值，这里默认为SHA-256中无消息输入的初始值
    memcpy(DB, hash_0, 32);
```

```
    // 然后，填充0x00
    for (int i = 0; i < pad; i++)
    {
        DB[32 + i] = 0x00;
    }
    memset(DB + 32, 0, pad);
    memset(DB + 32 + pad, 0x01, 1);
    // 最后，将数据填充到数据块中
    memcpy(DB + 32 + pad + 1, data, len);
    // 生成随机数种子
    uint8_t seed[32];
    get_seed(seed);
    uint8_t DBmask[DB_len] = {0}; // 用于存储DB掩码
    MGF1_seed(DBmask, seed, 32);
    // 得到DB掩码之后，进行异或操作
    for (int i = 0; i < DB_len; i++)
    {
        DB[i] ^= DBmask[i];
    }
    // 生成seed掩码
    uint8_t seedmask[32] = {0}; // 用于存储seed掩码
    MGF1_DB(seedmask, DB, 223);
    // 得到seed掩码之后，进行异或操作
    for (int i = 0; i < 32; i++)
    {
        seed[i] ^= seedmask[i];
    }

    // 填充EM
    EM[0] = 0x00;
    memcpy(EM + 1, seed, 32);
    memcpy(EM + 33, DB, DB_len);
}
```

通过上述的代码，我们便实现了OAEP填充的全过程，过程的搭建并不复杂，真正复杂的则是在其中调用的函数实现。

生成随机数种子：

```
#include <immintrin.h>
void get_seed(uint8_t seed[32])
{
    uint16_t tmp;
    for (int i = 0; i < 16; ++i)
    {
        _rdrand16_step(&tmp);
        seed[i * 2] = tmp >> 8;
        seed[i * 2 + 1] = tmp & 0xFF;
    }
}
```

根据实验要求，需要我们利用密码学安全的随机数生成器来生成随机数，于是便利用这个函数生成对应的随机数（不过由于本次实验时采用special judge，其实将seed固定为样例一的seed，也是可以过的）

MGF1函数实现：

该函数的具体逻辑已经在上面阐述了，里面主体的函数需要调用sha-256函数，而该函数的实现已经在上次实验详细阐述了，这边便不再进行赘述。

```
void MGF1_seed(uint8_t *mask, uint8_t *seed, uint8_t len)
{
    // 由于我们的seed是36字节，所以直接进入sha256的填充环节
    // 需要将seed转换为uint8_t数组
    uint8_t seed_8[36]; // 因为后面要计数，所以多分配4个字节
    memcpy(seed_8, seed, 32);
    uint8_t times = (DB_len / 32) + 1; // 计算需要填充的次数
    uint32_t tmp[8] = {0};
    for (int i = 0; i < times; i++)
    {
        // 大端字节序
        seed_8[32] = (i >> 24) & 0xff;
        seed_8[33] = (i >> 16) & 0xff;
        seed_8[34] = (i >> 8) & 0xff;
        seed_8[35] = i & 0xff;
        sha256_final(seed_8, 36, len + 4, tmp);
        if (i == times - 1)
        {
            memcpy(mask + i * 32, tmp, 31);
            break;
        }
        memcpy(mask + i * 32, tmp, 32);
    }
}

void MGF1_DB(uint8_t *mask, uint8_t *meg, uint8_t len)
{
    // 一开始，仍是对512位数据进行处理
    uint8_t data[64] = {0};
    uint8_t times = (len / 64); // 需要进行3次循环
    for (size_t i = 0; i < times; i++)
    {
        memcpy(data, meg + i * 64, 64);
        sha256_DB(data);
    }
    // 最后一次循环
    memcpy(data, meg + times * 64, len % 64);
    // 进行计数填充
    data[31] = 0x00;
    data[32] = 0x00;
    data[33] = 0x00;
    data[34] = 0x00;

    sha256_final_DB(data, 35, len + 4);
    // 将hash值全部使用大数来存储
    for (int i = 0; i < 8; i++)
    {
        H[i] = toBigEndian(H[i]);
    }
    memcpy(mask, H, 32);
```

```
}
```

可以看到，我这里**分别**为生成DBMask和SeedMask写了对应的MGF1填充函数，这样做是为了适配我之前写的sha-256函数。由于填充的每个部分我们都可以计算，所以我们可以自己控制sha-256的每个部分。eg：如果我要生成DBMask，由于知道它的字节为223，而一次哈希函数可以生成32字节，所以我们只需要手动调用7次`final`函数即可。

综上所述，我们便完成了OAEP填充的完整过程。

- 大数运算：

本部分是这次实验的核心，分为许多函数的操作来具体实现。

- 模幂运算（本次实验需要重点实现的地方）

```
void mod_pow(uint32_t res[128], uint32_t a[128], uint32_t b[128])
{
    uint32_t temp[128] = {1};
    uint32_t base[128] = {0};
    for (int i = 0; i < 128; i++)
        base[i] = a[i];

    for (int i = 0; i < 128; i++)
    {
        for (int j = 0; j < 32; j++)
        {
            if (b[i] & pow2[j])
                mod_mul(temp, temp, base);

            mod_mul(base, base, base);
        }
    }
    for (int i = 0; i < 128; i++)
        res[i] = temp[i];
}
```

这个代码实现了“平方-乘”的算法来优化模幂计算，具体原理在课堂上老师已经讲解过了，这里就不在赘述，具体逻辑可以参考下图：

将指数a表示为 二进制形式 1010（$a_0\,a_1\,a_2\,a_3$）

1: $y <= x$; a0=1，初始化y=x

2: $y <= y^2 = x^2$; a1=0，平方

3: $y <= y^2 = (x^2)^2 = x^4$; a2=1，平方后补乘x

$y <= y * x = x^4 * x = x^5$

4: $y <= y^2 = (x^5)^2 = x^{10}$; a1=0，平方

通过观察看出，计算量对数下降。

注意<=符号代表y值是递推关系，即：$y_{i+1} = y_i$

- 模乘运算

```
void mod_mul(uint32_t res[128], uint32_t a[128], uint32_t b[128])
{
    mul(res, a, b);
    mod(res, res);
}
```

- 乘法运算（基础运算之一）

```
 void mul(uint32_t res[128], uint32_t a[128], uint32_t b[128])
 {
     uint32_t temp[256] = {0};

     for (int i = 0; i < 128; i++)
     {
         uint64_t carry = 0;
         for (int j = 0; j < 128; j++)
         {
             uint64_t sum = (uint64_t)a[i] * b[j] + temp[i + j] + carry;
             temp[i + j] = sum & 0xffffffff;
             carry = sum >> 32;
         }
         if (carry)
             temp[i + 128] += carry;
     }

     for (int i = 0; i < 128; i++)
         res[i] = temp[i];
 }
```

该大数乘法实现了“长乘法”。首先，使用一个大小为256的临时数组 `temp` 来存储计算过程中的部分和。接着，利用两层嵌套循环，遍历输入的两个128位大整数 `a` 和 `b`，对每一对32位数字 `a[i]` 和 `b[j]` 执行乘法，并加上之前的部分和和进位。每次计算后，将结果的低32位存入 `temp[i + j]`，并将高32位作为进位传递到下一轮。最后，处理所有的部分和并将低128位结果复制到输出数组 `res` 中。

- 模运算（基础运算之一）

```
void mod(uint32_t res[128], uint32_t a[128])
{
    uint32_t temp[128] = {0};
    div(temp, a, P);
    mul(temp, temp, P);
    sub(res, a, temp);
}
```

该运算的实现方式较为简单，主要是依赖于除法运算以及减法运算

- 减法运算（基础运算之一）

```
 const uint64_t BASE = 0x100000000;
 void sub(uint32_t res[128], uint32_t a[128], uint32_t b[128])
 {
     uint64_t br = 0; // 用做减法借位
```

```
    uint32_t temp[128] = {0};
    for (int i = 0; i < 128; i++)
    {
        uint64_t temp1 = b[i] + br;
        if (a[i] < temp1)
        {
            temp[i] = BASE + a[i] - temp1;
            br = 1;
        }
        else
        {
            temp[i] = a[i] - temp1;
            br = 0;
        }
    }

    for (int i = 0; i < 128; i++)
        res[i] = temp[i];
}
```

该算法实现了大整数的减法运算，具体是通过逐位计算并处理借位来实现。对于每一位，首先将减数 `b[i]` 与借位相加，然后与被减数 `a[i]` 相减。如果当前位的差为负，则需要从高位借1，并将结果进行调整，而借位在每一轮计算中被传递，直到整个128位数的减法完成。最终，计算结果存储在 `res` 数组中。

- 除法运算（基础运算之一）

```
void div(uint32_t res[128], uint32_t a[128], uint32_t b[128])
{
    uint32_t temp[128] = {0};
    int n = getBits(b) / 32;
    int m = getBits(a) / 32 - n; // 迭代次数

    uint32_t d[128] = {0}; // 用来计算缩放因子
    d[0] = BASE / (b[n] + (uint64_t)1);
    uint32_t u_[129] = {0}, v_[128] = {0};
    uint64_t cr = 0; // 存放进位
    for (int i = 0; i < 128; i++)
    {
        uint64_t temp = (uint64_t)a[i] * d[0] + cr;
        u_[i] = temp & 0xffffffff;
        cr= temp >> 32;
    }
    if (cr)
        u_[128] = cr;
    mul(v_, b, d);

    int j = m;
    while (j >= 0)
    {
        uint32_t tem[129] = {0};
        for (int i = 0; i <= n + 1; i++)
            tem[i] = u_[i + j];
        uint64_t tem2 = (tem[n + 1] * BASE + tem[n]) / v_[n];
        tem2 = min(tem2, BASE - 1);
```

```
            uint32_t q_hat = static_cast<uint32_t>(tem2 & 0xffffffff);  // 估商值
            uint32_t qv[128] = {0}; // 用来调整估商值
            for (int i = 0; i < 128; i++)
            {
                uint64_t temp = (uint64_t)v_[i] * q_hat + cr;
                qv[i] = temp & 0xffffffff;
                cr= temp >> 32;
            }

            while (bigger(qv, tem))
            {
                q_hat--;
                sub(qv, qv, v_);
            }

            sub(tem, tem, qv);
            for (int i = 0; i <= n + 1; i++)
                u_[i + j] = tem[i];

            temp[j] = q_hat;

            j--;
        }

        for (int i = 0; i < 128; i++)
            res[i] = temp[i];
    }
```

该算法实现了大整数除法，使用了类似于纸笔除法的长除法方法。首先通过缩放被除数和除数，使得除法运算变得更加简单。然后通过迭代地估算商并调整商，确保得到精确的结果。在每次迭代中，都会计算商的部分，更新余数，并处理进位和借位。最终得到的商存储在 `res` 数组中。（参考资料：https://blog.csdn.net/qq_37734256/article/details/86751853）

至此，大数运算全部实现完成。

- 输入处理

```
    // 进行数据读取
    {
        in.ignore(16); // 跳过16个字节
        in.read((char *)tmp_256, 256);
        for (int i = 0; i < 64; i++)
        {
            P[63 - i] = tmp_256[i * 4] | (tmp_256[i * 4 + 1] << 8) | (tmp_256[i * 4 + 2] << 16) | (tmp_256[i * 4 + 3] << 24);
        }
        in.read((char *)tmp_256, 256);
        for (int i = 0; i < 64; i++)
        {
            e[63 - i] = tmp_256[i * 4] | (tmp_256[i * 4 + 1] << 8) | (tmp_256[i * 4 + 2] << 16) | (tmp_256[i * 4 + 3] << 24);
        }
        in.ignore(256); // 跳过256个字
```

```
        in.read((char *)&len, 1);
    }
    // 这里为了方便，把输入的n和e的数组大小都设置为了128，但是我们需要的其实仅需64个，也就是读入的内容，需要把这个转换成小端序
    for (int i = 0; i < 64; i++)
    {
        P[i] = toBigEndian(P[i]);
        e[i] = toBigEndian(e[i]);
    }
```

这里再提一嘴对输入数据的处理，由于这里的大数运算实现是模拟人的运算来推断的，所以需要将输入的**大端序**二进制流修改为**小端序**，并且将字节头尾反转过来（相当于我们计算时从低位算到高位），这样最后就可以得到正确的实验结果。

五、实验结果

- 输入如下：

```
D2 F4 E4 BC 44 7B 97 96 C7 F9 53 63 EF 03 38 B2
6E 66 76 74 D3 C2 10 D6 35 68 20 6B 3A 20 BE 6F 7C 7E C4 16 B5 DF 89 23
9F 89 47 06 3F D9 3F 22 70 47 48 03 5D D7 0B 42 2A F1 8C 41 8C 3F 79 8E
BD 95 0D 2C 2F 93 FA D2 1C E8 5E 67 E3 A2 4B A7 D7 36 F9 4F 2B ED ED 7F
E5 DB 16 DD 09 E3 1A D4 9A 77 77 2D 28 02 81 50 3D DF 31 D0 29 3A E6 05
80 0D 2B 72 1E BE 53 3F 6C B6 24 B2 1C 80 31 3E 66 1C 2E BF 28 47 03 A2
09 72 6C 2E AD 60 B2 18 3F 23 88 49 32 83 B6 A2 1B 48 C3 53 3D 88 88 82
7E 8A 86 E3 51 6D 0E 72 FE 1C 62 42 E3 C9 28 B3 C0 E5 73 ED 43 10 6B 7F
90 F8 A2 EA 04 6E A5 88 68 35 C1 16 DE 05 F8 42 EF 69 FE 42 74 FA 6C 67
AB FA B4 E1 C0 1E 73 DD 06 39 78 1B 51 ED D6 7E B5 50 BE C2 59 A9 A2 C7
FC 9B 6C E1 28 2B 4E 17 D4 E9 1C 44 48 BE 61 5A 85 65 FC 50 DB 57 25 B7
E5 D7 8E C2 1B DB 45 F9 CE E8 08 4A 12 05 7A 37
1F C0 DF 26 1A 61 E6 87 51 18 34 DE AC E1 AA 66 10 5A 5B B7 53 B3 9B F2
1E 5C 51 3B 01 B9 DF AA DD BB 6B 12 16 D4 EA 02 69 6C AF 03 1B A9 B0 EE
F2 85 64 AD 4D 16 83 88 D5 68 40 0E 7D 2D 54 6F 55 48 8D FF ED 5D 64 97
2E 36 3C 2B 25 9A 60 C2 37 ED 94 F1 DF 3C BB B8 B7 D0 B7 C1 44 3A AE F6
53 8F B5 DE A3 A3 75 A6 75 24 3C 86 D5 80 70 C1 E2 4F 12 69 C7 21 41 2C
20 83 9B 4E 12 F9 EF 51 2E D7 0A E2 10 8E E9 EB 95 8F E0 9A E6 53 21 94
B8 7D B2 BA 54 14 46 DD 33 ED 45 60 7A 0F 4D E1 0F 9A 72 BC 3D 41 D4 51
FF 95 B6 4C DB CC 99 96 FC AF 24 65 BF 7E 63 26 CD 08 B1 EE D4 BF 42 0E
EE A6 34 6D 25 07 55 85 F3 80 55 B4 79 65 5A B9 DD D2 36 FB 26 B6 97 5F
5D 6B AE A8 57 D2 A8 96 53 8E 61 E1 A3 34 D3 9E C0 E5 5A 53 9A 26 AA E1
E4 A7 8F BC 45 65 F6 0A 12 67 3E F6 DC E2 89 FF
8F 35 49 7E 5D 35 16 5B 8F 0B F3 5A D9 B4 94 13 A1 09 DC 9B F2 B5 16 1B
4C 93 DD DB AD AE 92 B7 40 D2 3E 23 C7 46 F6 B2 A6 1F CF F0 54 98 4F F2
B3 4C EE E0 FF 65 51 64 2C 62 46 88 EA 58 AB AB 01 8D B8 BF 08 FA AE EB
49 C4 79 04 10 77 9F EC 3C A8 13 7E F3 47 57 E6 25 2B 31 9D 1E 82 6C CA
BD 24 6B C5 DE A1 97 06 F3 9A DD 2B 24 24 A6 3F 8C E6 62 7E 1F F0 83 35
AB B8 6D A0 87 9E AB 41 11 6D 0C A0 12 44 3C 42 B9 7C 62 76 6F B8 9D A2
5D E8 F2 25 AF FE 0D 7C CE EE 2E F3 76 94 33 7B 3A 98 04 F0 89 C5 41 F6
59 37 5C B4 94 73 9C CD 14 FA F3 CD 71 E5 B4 3C E5 52 AD D0 E0 A1 5D 96
20 12 EB 54 C2 7B 16 DE C0 2A 47 B0 39 A1 AF 2A D4 9C F7 75 30 B9 4D 57
AB F2 34 94 83 6B 13 40 50 0E C3 10 8E DA B8 D9 0A 34 A8 E0 3B 5E ED 6B
D4 D7 EC 50 A1 FB B9 D1 38 EA C1 1E 2B 12 F4 CA
0E
4B 6F 6D 65 69 6A 69 20 4B 6F 69 73 68 69
```

此时seed固定为：`84 EE 1D 92 BE FC 55 96 6F 20 B6 FD 18 FC 45 20 CF 17 0B D8 92 E2 E0 D3 4F E7 A0 10 17 C8 6B 67`

- 输出如下：

```
3F 03 0C 16 16 FB A0 83 8E 92 51 A0 F0 51 F9 D7 6D EA 97 CA B8 BD E9 FE 4E 6B 84 E4 92 76 12 D3
FE D6 2E 2D 2C 6D 50 CB AF C3 71 C5 17 DF BC FF 38 61 29 FA 01 2D 60 14 B4 20 DB 1D 15 8F CE FD
B9 50 37 FA EF 93 5D 1B 90 C4 CF BD 94 7F C2 5C C4 97 F9 BA 73 6E D0 47 4F 79 8D 93 64 38 6D 03
FE D6 2E 2D 2C 6D 50 CB AF C3 71 C5 17 DF BC FF 38 61 29 FA 01 2D 60 14 B4 20 DB 1D 15 8F CE FD
B9 50 37 FA EF 93 5D 1B 90 C4 CF BD 94 7F C2 5C C4 97 F9 BA 73 6E D0 47 4F 79 8D 93 64 38 6D 03
B9 50 37 FA EF 93 5D 1B 90 C4 CF BD 94 7F C2 5C C4 97 F9 BA 73 6E D0 47 4F 79 8D 93 64 38 6D 03
15 B4 87 81 D7 0E 68 AD CC 88 AF 39 C0 EA 95 D3 CD 97 44 D6 3D CB 50 03 56 8A F1 2C B8 86 91 04
15 B4 87 81 D7 0E 68 AD CC 88 AF 39 C0 EA 95 D3 CD 97 44 D6 3D CB 50 03 56 8A F1 2C B8 86 91 04
F4 5A E6 6E 5E 3A 29 65 FD 10 BA 3B 7E B8 D8 12 76 FC 5C 36 E0 BF DA E2 3B B0 43 B4 CB 96 38 C3
36 E1 0A D0 8A D1 9E DE 24 DD 46 2D C9 0E 88 A8 65 55 05 A5 D1 83 05 CB 39 EC 9A 74 08 C4 CA 44
8B C7 D9 59 87 66 97 F1 19 C2 ED 83 A4 EB A1 EF 7B 51 2F 1F C5 49 CA 5D 95 6E CF ED 85 2F E7 82
BB 1D B1 C0 DF 5B 13 34 9B 11 20 98 90 67 4F 7F E2 59 10 9E 4E FE 54 0A 39 E3 CC 32 7A A1 45 E6
```

六、实验总结

这次实验是我迄今为止做得最难受的一次实验。虽然在之前的实验中积累了一些基础代码，但这次的调试过程依然耗费了大量时间。或许可以归结为代码的扩展性不足，导致很多细节问题需要反复调整。特别是期末周有许多ddl，做这个密码学实验通常会占用我一周中的三天时间，导致其他作业堆积在一起，使得整个人感到非常焦虑。再加上调试时频繁出现的各种 bug，让我一度感到非常绝望和沮丧。

然而，尽管过程充满挑战，这次实验也让我受益匪浅。通过这次实验，我不仅更加深刻地掌握了调试的技巧和方法，还学会了如何面对和解决调试过程中可能出现的错误。每一个 bug 的解决都让我更加理解代码的细节，并让我对调试工具和技术有了更深入的认识。尤其是在处理大数运算和加密算法时，遇到的错误往往不是单纯的代码问题。

此外，通过这次实验，我对教科书式的 RSA 算法有了更加深入的理解。特别是学习了 **OAEP 填充**技术，这解决了传统 RSA 算法中存在的安全问题，使得加密过程更加安全和高效。

最后，通过实现大数运算，我加深了对课本上算法的理解和应用。大数运算不仅是密码学算法的基础，也是理解更复杂算法的关键。尽管实现大数加法、减法、乘法、除法等运算的过程繁琐，但它让我深刻理解了计算机如何处理大于标准数据类型范围的数字，如何优化算法以及如何进行高效的内存管理。

总的来说，这次实验让我学到了很多，不仅提升了自己的编程能力，还加深了对密码学原理的理解。