《现代密码学》实验报告

实验名称：2048-bitRSA 加解密	**实验时间**：2025 年 12 月 1 日
学生姓名：谢润文	**学号**：23336262
学生班级：23 计科	**成绩评定**：

一、实验目的

通过实现大数运算的加法、乘法、模乘、模幂，进而实现 2048bit-RSA 的加解密，理解公钥密码学中单向陷门函数、RSAP 问题与 IFP（整数分解）问题、非对称消息填充以防止选择明文攻击的思想，提高大数运算及块填充的处理能力，熟练掌握大小端数据的转化。

二、实验内容

用 C/C++实现 RSA-2048 的加密和解密，不能使用大数运算的第三方库。

参数要求：

1. 加密,输入由 uint8_t[16]需要跳过的数据,uint8_t[256]以大端序表示的 2048 bit 的 n,uint8_t[256]以大端序表示的 2048 bit 的 e,uint8_t[256]需要跳过的数据,uint8_t 加密的消息长度（因为是 RSA-2048，所以消息长度不能超过 256-1- 32-32-1=190 bytes）以及 uint8_t[]加密的消息内容构成。输出只有最终的加密结果，即大端序表示的 c。

2. 解密，输入由 uint8_t[128]以大端序表示的 1024 bit 的 p，uint8_t[128]以大端序表示的 1024 bit 的 q，uint8_t[256]以大端序表示的 2048 bit 的 n，uint8_t[256]以大端序表示的 2048 bit 的 d，uint8_t[128]以大端序表示的 1024 bit 的 d mod (p-1)，uint8_t[128]以大端序表示的 1024 bit 的 d mod (q-1)，uint8_t[128]以大端序表示的 1024 bit 的 q-1 mod p 及 uint8_t[256]密文内容构成。输出解密结果，也就是 PS 的一连串 0x00 之后的第一个 0x01 之后的内容。

三、实验原理

0. 前置知识：大数运算

大数指的就是大小超过了 uint64_t 这样的基础类型的数。RSA-2048 中采用的是 2048bit 的模数 n（n 一般为两个 1024-bit 素数的乘积），2048bit 的指数 e，这是为了防止 n 太小进行的模数分解攻击，和 e 很小的低加密指数攻击。在 RSA 中需要计算 c=pow(m,e)mod n，其中 m 是明文消息。所以对应的大数运算需要计算模乘、模幂，这涉及到基本的加法运算和乘法运算。

0.1 大数存储、读入与写出

比较基本的方式是一位一位用 char 存储，但是这样计算效率会比较慢。相对来说比较快的方式是按照一个基数（BASE）进行存储，在十进制下可以取 10 的 n 次方（在 long long

或 int 范围内）进行表示，相当于把十进制转化为 10 的 n 次方为基底表示。

关于读入与写出，一种比较直接的方式是利用 cout 的<<和>>进行重载。但是众所周知，cout 的速度是比较慢的。稍微好一点的方式是在结构体里写一个 print，利用 printf 对于每个基于 BASE 里的数输出。更好的方式是利用 getchar 和 putchar 进行快读处理。

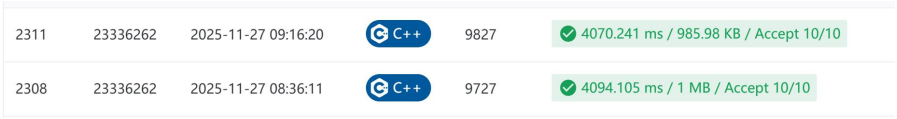

图 1 利用 getchar 和 putchar 进行的快读加速，提速了约 24ms

0.2 大数加（减）法运算

基本的思想是先取符号，再根据符号决定绝对值是相加还是相减。最基本的绝对值加法仍然是按位相加，从右（最低位）到左（最高位）进行运算，进位每次需要%10，最后考虑进位。稍微快一点的是基于 BASE 下的加法，这里以 BASE=100000000（10 的 8 次方），大数运算的第二个样例的第一组数据加法为例：

考虑 128014342154867+73305753838378，第一个数分为 1280143*BASE+42154867，第二个数分为 733057*BASE+53838378。对于 BASE 的 0 次方上的数相加，得到 95993245<BASE，不产生进位，BASE 的 1 次方上的数相加，得到 2013200<BASE，也不产生进位，所以最后的结果为 2013200*BASE+95993245=201320095993245，发现小于模数 n，不需要取模。

减法相对来说要处理更多的情况。我们希望的是大数减小数，所以执行计算前首先比较减数(num1)和被减数(num2)的大小，如果 num1>num2,那么就模拟 num1-num2 的过程，如果 num1<num2，那么结果就为-(num2-num1)。和加法类似，需要考虑借位。仍然可以用 BASE 下的数据表示。

整体复杂度是 O（k）级别的，其中 k=logn。

0.3 大数乘法运算

比较 trivial 的仍然是模拟按数位从右到左算。相对好一点的是利用 BASE 进制下的运算仍然考虑 128014342154867 和 73305753838378 的例子。对于第一个数的 BASE1 次方部分与第二个数相乘得到 1280143*733057*pow(BASE,2)+1280143*53838378*BASE,BASE0 次方部分与第二个数相乘得到 42154867*733057*BASE+42154867*53838378。每一部分考虑进位（/BASE）和 mod BASE 的结果，得到 9384*pow(BASE,3)+18785378*pow(BASE,2)+65761969*BASE+64085726，最终输出即可。

在实验中用两个 for 循环从低位到高位模拟这一过程即可。最后需要去除前导零。

更快的方式是利用 Karatsuba 算法进行分治：

1）将两个大数 x 和 y 分别拆分成两部分：x=x1*pow(10,m)+x0,y=y1*pow(10,m)+y0，其中 x0 和 y0 小于 pow(10,m)。

2）计算 z2=x1*y1，z0=x0*y0，z1=(x1+x0)*(y1+y0)-z2-z0。如果 x0、y0、x1、y1 仍然比较大，把 x0、y0 或 x1、y1 作为 x 和 y 执行 1），直到它们比较小（比如乘积小

于 long long 的界限）为止。

3）最终结果为：xy=z2*pow(10,2m)+z1*pow(10,m)+z0。

这样可以从双重 for 循环的 O(k*k)复杂度降低为 O(pow(k,log_base2_3))。

0.4 大数除法与取模运算

计算 a/b，a%b=a-a/b。一个显然超时的思路是，通过枚举将除法变成减法，从被除数中不断减除数，一直到不能减为止。稍微想一下，可以加速寻找次数，减少减法的次数，减法次数减小的一个方案就是扩大除数 b。如果 b 后面加个'0'，那么算出来的结果就乘以 10，减法的次数变成原来十分之一。根据这个思想我们可以一直每次找到 b 的最大 10 的倍数(小于 a)。即便是用 BASE 进制下进行运算，还是很慢，或者说常数比较大。

有没有什么办法提速呢？一种方法是二分试商，这样大概需要 k 次乘法，如果做普通的乘法，总体时间复杂度为 O（k*k*k)。事实上，在 Donald E. Knuth 的 The Art of Computer Programming 中提到了更好的算法。给定 $u=u_{m+n-1}\cdots\cdots u_1u_0$，$v=v_{n-1}\cdots\cdots v_1v_0$，如果 $v_{n-1}\geq BASE/2$，且 $q\hat{}=min((u_n*base+u_{n-1})/v_{n-1},BASE-1)$，则 $q\hat{}-2\leq q\leq q\hat{}$，其中 $q\hat{}$ 是 q 的估计值。这个定理表明，最多只需要计算三次乘法，就能得到商的准确值。[1]具体的说，算法如下：

1）定理要求 $v_{n-1}\geq BASE/2$，置 $d=BASE/(v_{n-1}+1)$，然后令 u=u*d，v=v*d。

2）初始化循环变量 j=m，第一次循环实际上是 $u_{j+n}\cdots\cdots u_{j+1}$ 除以 v。

3）计算 q 的估计值 $q\hat{}$

4）判断 $u_{j+n}\cdots\cdots u_j-q\hat{}*v$ 是否大于 v，若大于 v 则 q 减一（修正）。

5）j--，若 j 不等于 0，返回第三步。

组合起来得到商，而余数通过除数第一步相反的规格化得到。

不难证明时间复杂度 O(3*k*k)。

0.5 大数模逆

计算 $a^{-1} \bmod n$。注意到 gcd(a,n)=1，由贝祖定理可知存在 s,t 使得 as+nt=1，同模 n 可知 s 为 a 在 n 下的逆。于是可以用经典的 exgcd（Extended gcd，扩展欧几里得算法）求解模逆。具体算法的代码如下：

```
void ex_gcd(int a, int b, int& x, int& y) {
    if (!b) { x = 1; y = 0; }
    else { ex_gcd(b, a % b, y, x); y -= a / b * x; }
}
```

这是一般的模逆求法，递归深度 O(log b)。实际大数运算中 b 很大(比如 RSA 的 2048-bit)，如此深的递归深度很容易导致爆栈。因此，要使用非递归的形式，具体在代码中展现，此处不再赘述。

记 k=max{log a,log b}，循环不超过 k 次，每一次循环需要 1 次除法，故时间复杂度为 O（k）* O（k*k)=O（k*k*k）。

0.6 大数模幂与蒙哥马利规约

计算 a 的 b 次方 mod n。一个相对自然的想法是利用快速幂，利用 b 的二进制展开，这样大概需要计算 log b 次的平方、乘法和取模。利用蒙哥马利归约能够高效地实现模乘而无须显式执行传统的取模归约操作，从而快速实现模幂。具体的算法如下：

1）选取 R=BASE^k（其中 k=p.len）且 gcd(p,R)=1。这里采用 BASE 采用 10 的次幂，是因为 n 或为大素数或为两个大素数的乘积，不可能有 2 和 5 这两个因子，故与 10 是互素的。

2）预计算 n'满足 $p*n' \equiv -1 \pmod R$，通过 ex_gcd(p,R,p_inv0,y)得到 $p\_inv0 = p^{-1} \bmod R$，随后 p_inv=-p_inv0 即 n'。

3）计算 mont_mul（蒙哥马利乘法）：

a. 计算 T=a*b，T mod R（这里实际上不需要取模，取基于 BASE 的低数位即可）。

b. 计算 m=(T_mod_R*n') mod R。先计算 m=T_mod_R*p_inv，然后只保留低 k 个基于 BASE 的数位。

c. 计算 t=(T+m*p)/R。实现上等价于把(T+m*p)的低 k 个基于 BASE 的数位并取高位部分。

d. 若 t>=p 则 t-=p，返回 t（即 $(abR^{-1}) \bmod p$）。

4）利用平方-乘算法进行约 bitlen(b) 次 mont_mul（每位一次平方，若当前位为 1 再乘一次），最后再利用一个 mont_mul 还原成正常表示。

显然当 b 与 n 同阶（均为 k-bit）时，时间复杂度为 O（k*k*k）。

1. 公钥密码学引言

公钥密码学（public key cryptography）又称非对称密码学，是采用公钥与私钥配对机制的密码体系，与仅使用单一密钥的对称密码学相对应。其核心技术包括公钥加密算法和数字签名算法，典型代表有 RSA 算法与 Diffie-Hellman 密钥交换协议。[2]

公钥密码体制的设计最终归结为一个陷门单向函数。陷门单向函数是满足以下条件的函数 f：

1）正向计算容易。即如果知道公钥 Pk 和消息 M，容易计算 $C=f_{Pk}(M)$。

2）不知道密钥 Sk 的情况下，反向计算 $M=f_{Pk}^{-1}(C)$非常困难。

2. RSA 密码体制

2.1 RSA 的主算法

1） 密钥对的生成。选择两个大素数 p,q，计算 n=pq 和$\phi(n)=(p-1)(q-1)$。随机选取整数 e，满足 $1<e<\phi(n)$，且 gcd（$e,\phi(n)$）=1 作为公钥，计算 $d=e^{-1} \bmod \phi(n)$ 。此时公钥为（n,e），私钥为（p,q,d）或者（n,d）。

2） 加解密。加密利用公钥 e，将明文 m 加密为 $c=m^e \bmod n$；解密则利用私钥 d，将密文 c 解密为 $m=c^d \bmod n$。

3） 参数选择。为避免椭圆曲线因子分解法，p 和 q 的长度相差不能太大；也不能差值

太小（否则可以通过枚举 sqrt(n)附近的奇素数进行分解）。另外还要求 p-1、q-1 都有大的素因子（Strong RSA，SRSA）以防止 p-1 攻击。

即便如此，教科书式的 RSA 由于可以利用乘法群的性质对其进行延展，造成 CPA 攻击伪造特定消息；以及由于是确定性加密，不满足 IND-CPA。故其仍是不安全的。实际应用中需要利用最优非对称加密填充（OAEP）对消息进行处理。

2.2 RSAP 问题与 IFP 问题

RSA 问题（RSAP）是在 RSA 密码系统中核心的计算难题。RSAP 的目标是在不知道 n 的分解（即 p 和 q）以及 φ(n) 的情况下，确定明文 m，也就是计算 c 模 n 的 e 次根。显然，RSAP 可以在多项式时间内归约到 IFP，这意味着如果能够解决 IFP，就可以对 RSA 函数进行求逆。然而目前还不清楚是否存在比解决 IFP 更简单的方法来对 RSA 函数求逆。

2.3 RSA 的加密、OAEP 填充与 MGF 函数

图 2 给出了 OAEP 填充的方法，其中本次实验默认标签 L 为空，Hash 函数采用 SHA-256。Seed 是随机种子，长度与 Hash 输出的长度相同（本次实验中为 32 字节，代码中使用了 RDRAND 指令）。PS 是由 00 组成的填充部分。M 为二进制表示的消息。

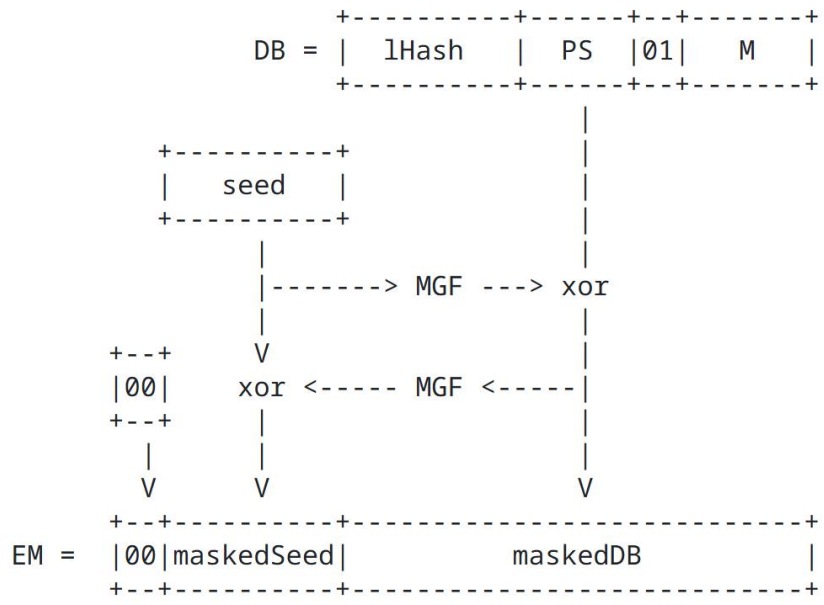

图 2 OAEP 填充

MGF 函数相当于可变长度的 hash，具体算法如下：1）设置一个从 0 开始、用 4 bytes 的大端序表示的计数器，将输出设为空数据。2）不断在输出上拼接 H(m||c)并增加计数，直到输出达到指定的长度为止，然后截断多余的部分。

然后把得到的 EM 转换为大整数 m，计算 $c=m^e \bmod n$，再按照大端序转化为二进制即可完成加密。

2.4 RSA 的解密

解密首先要计算 $m=c^d \bmod n$，然后再去除 OAEP 填充。假如知道 n 的分解 n=pq，则可以

用中国剩余定理（CRT）进行加速：

1）取 c 为密文，分别在小模上计算：m1=c^{d_p} mod p 和 m2=c^{d_q} mod q。

2）h=(m1-m2)*qInv mod p

3）m=m2+q*h

不难验证 m 的唯一性。设原模 n=pq 的位长为 k，不加速下单次模幂时间约 Θ(k*k*k)。CRT 加速做两次模为 k/2 位的幂：代价≈2 *Θ((k/2)^2 * bitlen(d_p))≈2 * (1/4) *Θ(k*k*(k/2))，通常能带来约 4 倍的加速。

然后得到 m 也即 EM 后，由于 EM=00||maskedSeed||maskedDB，首先检测开头是否为 00，若否，则认为密文不合法，输出空内容。然后先通过 maskedSeed 反向 MGF 得到 Seed，进一步得到 DBmask，与 maskedDB 异或得到 DB，进行检测：假如头 32 字节不是空标签对应的 hash 值，则密文不合法，输出空内容。否则遍历 DB 找到第一个 0x01 出现的位置（若未出现，则密文不合法）。0x01 后面的就是明文。至此解密完成。

四、实验步骤（源代码）

首先是 2048-bitRSA 加密的代码：

```
#include<iostream>
#include<vector>
#include<cstring>
#include<cstdint>
#include<fstream>
#include<fcntl.h>
#include<wmmintrin.h>
#include<immintrin.h>
#include<emmintrin.h>
#include<algorithm>
using namespace std;
const int MAXN=1000;
const long long BASE=100000000;
const int WIDTH=8;
/*
FILE *fp=fopen("dump (10).bin","rb");
```

```
    FILE *p=fopen("output.bin","wb");
    */

    #define fp stdin
    #define p stdout

    const uint32_t h[9]={0,0x6a09e667,0xbb67ae85,0x3c6ef372,0xa54ff53a,
                         0x510e527f,0x9b05688c,0x1f83d9ab,0x5be0cd19};
    uint32_t _hash[10]={0};
    __m128i
state0,state1,msg,msgtmp0,msgtmp1,msgtmp2,msgtmp3,msgtmp4,abef,cdgh;
    __m128i shuf_mask=_mm_set_epi32(
        0x0c0d0e0f,
        0x08090a0b,
        0x04050607,
        0x00010203
    );
    void sha_256(uint8_t tmp[],uint32_t _hash[],int start){
        if(!start){
            state0=_mm_set_epi64x(0x6a09e667bb67ae85,0x510e527f9b05688c);
            state1=_mm_set_epi64x(0x3c6ef372a54ff53a,0x1f83d9ab5be0cd19);
        }
        abef=state0;
        cdgh=state1;
        msg=_mm_loadu_si128((__m128i*)(tmp+0));
        msgtmp0=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0xe9b5dba5b5c0fbcf,0x71
374491428a2f98));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
```

```
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msg=_mm_loadu_si128((__m128i*)(tmp+16));
        msgtmp1=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0xab1c5ed5923f82a4,0x59
f111f13956c25b));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
        msg=_mm_loadu_si128((__m128i*)(tmp+32));
        msgtmp2=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0x550c7dc3243185be,0x12
835b01d807aa98));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
        msg=_mm_loadu_si128((__m128i*)(tmp+48));
        msgtmp3=_mm_shuffle_epi8(msg,shuf_mask);
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0xc19bf1749bdc06a7,0x80
deb1fe72be5d74));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
        msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
        msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);

msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x240ca1cc0fc19dc6,0xefbe4786e
```

```
49b69c1));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
        msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
        msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x76f988da5cb0a9dc,0x4a
7484aa2de92c6f));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
        msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
        msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
        msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0xbf597fc7b00327c8,0xa8
31c66d983e5152));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
        msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
        msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0x1429296706ca6351,0xd5
a79147c6e00bf3));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
```

```
        msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
        msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);
        msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x53380d134d2c6dfc,0x2e
1b213827b70a85));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
        msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
        msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x92722c8581c2c92e,0x76
6a0abb650a7354));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
        msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
        msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp0=_mm_sha256msg1_epu32(msgtmp0,msgtmp1);
        msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0xc76c51a3c24b8b70,0xa8
1a664ba2bfe8a1));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
        msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
        msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
        msg=_mm_shuffle_epi32(msg,0x0e);
```

```
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp1=_mm_sha256msg1_epu32(msgtmp1,msgtmp2);
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0x106aa070f40e3585,0xd6
990624d192e819));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp3,msgtmp2,4);
        msgtmp0=_mm_add_epi32(msgtmp0,msgtmp4);
        msgtmp0=_mm_sha256msg2_epu32(msgtmp0,msgtmp3);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp2=_mm_sha256msg1_epu32(msgtmp2,msgtmp3);

msg=_mm_add_epi32(msgtmp0,_mm_set_epi64x(0x34b0bcb52748774c,0x1e376c081
9a4c116));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp0,msgtmp3,4);
        msgtmp1=_mm_add_epi32(msgtmp1,msgtmp4);
        msgtmp1=_mm_sha256msg2_epu32(msgtmp1,msgtmp0);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msgtmp3=_mm_sha256msg1_epu32(msgtmp3,msgtmp0);
        msg=_mm_add_epi32(msgtmp1,_mm_set_epi64x(0x682e6ff35b9cca4f,0x4e
d8aa4a391c0cb3));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp1,msgtmp0,4);
        msgtmp2=_mm_add_epi32(msgtmp2,msgtmp4);
        msgtmp2=_mm_sha256msg2_epu32(msgtmp2,msgtmp1);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msg=_mm_add_epi32(msgtmp2,_mm_set_epi64x(0x8cc7020884c87814,0x78
```

```
a5636f748f82ee));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msgtmp4=_mm_alignr_epi8(msgtmp2,msgtmp1,4);
        msgtmp3=_mm_add_epi32(msgtmp3,msgtmp4);
        msgtmp3=_mm_sha256msg2_epu32(msgtmp3,msgtmp2);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        msg=_mm_add_epi32(msgtmp3,_mm_set_epi64x(0xc67178f2bef9a3f7,0xa4
506ceb90befffa));
        state1=_mm_sha256rnds2_epu32(state1,state0,msg);
        msg=_mm_shuffle_epi32(msg,0x0e);
        state0=_mm_sha256rnds2_epu32(state0,state1,msg);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        state0=_mm_add_epi32(state0,abef);
        state1=_mm_add_epi32(state1,cdgh);
        //cout<<hex<<state0[0]<<"        "<<state0[1]<<endl<<state1[0]<<"
"<<state1[1]<<endl;
        _hash[0]=(uint32_t)(state0[1]>>32);
        _hash[1]=(uint32_t)(state0[1]);
        _hash[4]=(uint32_t)(state0[0]>>32);
        _hash[5]=(uint32_t)(state0[0]);
        _hash[2]=(uint32_t)(state1[1]>>32);
        _hash[3]=(uint32_t)(state1[1]);
        _hash[6]=(uint32_t)(state1[0]>>32);
        _hash[7]=(uint32_t)(state1[0]);
        //for(int i=0;i<=7;++i) cout<<hex<<(int)_hash[i]<<" ";
        //cout<<endl;
    }
    struct bign{
```

```
    int len;
    long long s[MAXN];
    bool sign;
    bign(){memset(s,0,sizeof(s));len=1;sign=false;}
    bign(int num){*this=num;}
    bign(const char *num){*this=num;}
    bign operator=(const int num){
        char s_tmp[20];
        sprintf(s_tmp,"%d",num);
        *this=s_tmp;
        return *this;
    }
    bign operator=(const char *num){
        if(num[0]=='-'){
            sign=true;
            num++;
        }
        else sign=false;
        while(*num=='0') num++;
        int l=strlen(num);
        len=0;
        memset(s,0,sizeof(s));
        if(l==0){
            len=1;
            s[0]=0;
            sign=false;
            return *this;
        }
        int k=0;
        long long cur=0,w=1;
```

```
        for(int i=l-1;i>=0;i--){
            cur+=(num[i]-'0')*w;
            w*=10;
            k++;
            if(k==WIDTH){
                s[len++]=cur;
                cur=0;
                w=1;
                k=0;
            }
        }
        if(k>0)s[len++]=cur;
        return *this;
    }
    bool abs_less(const bign &b)const{
        if(len!=b.len) return len<b.len;
        for(int i=len-1;i>=0;i--) if(s[i]!=b.s[i]) return s[i]<b.s[i];
        return false;
    }
    bign abs_add(const bign &b)const{
        bign c;
        c.len=0;
        long long g=0;
        for(int i=0;g||i<max(len,b.len);i++){
            long long x=g;
            if(i<len) x+=s[i];
            if(i<b.len) x+=b.s[i];
            c.s[c.len++]=x%BASE;
            g=x/BASE;
        }
```

```
            return c;
        }
        bign abs_sub(const bign &b)const{
            bign c;c.len=0;
            long long g=0;
            for(int i=0;i<len;i++){
                long long x=s[i]-g;
                if(i<b.len)x-=b.s[i];
                if(x>=0)g=0;
                else{g=1;x+=BASE;}
                c.s[c.len++]=x;
            }
            c.clean();
            return c;
        }
        bign abs_mul(const bign &b)const{
            bign c;
            c.len=len+b.len;
            for(int      i=0;i<len;i++)      for(int      j=0;j<b.len;j++)
c.s[i+j]+=s[i]*b.s[j];
            for(int i=0;i<c.len;i++){
                c.s[i+1]+=c.s[i]/BASE;
                c.s[i]%=BASE;
            }
            c.clean();
            return c;
        }
        bign operator*(long long x)const{
            if(x==0) return 0;
            bign c;
```

```
        c.len=len;
        long long g=0;
        for(int i=0;i<len||g;i++){
            long long cur=g+(i<len?s[i]*x:0);
            c.s[i]=cur%BASE;
            g=cur/BASE;
            if(i>=c.len) c.len=i+1;
        }
        return c;
    }
    pair<bign,bign> div_mod(const bign &b)const{
        bign q,rem;
        q.len=len;
        rem.len=0;
        for(int i=len-1;i>=0;i--){
            if(!(rem.len==1&&rem.s[0]==0)){
                for(int j=rem.len;j>0;j--) rem.s[j]=rem.s[j-1];
                rem.s[0]=0;
                rem.len++;
            }
            rem.s[0]=s[i];
            if(rem.abs_less(b)){
                q.s[i]=0;
                continue;
            }
            long long q_est=0,rem_top=rem.s[rem.len-1];
            if(rem.len>b.len) rem_top=rem_top*BASE+rem.s[rem.len-2];
            long long b_top=b.s[b.len-1];
            if(b_top!=0) q_est=rem_top/(b_top+1);
            long long l=q_est,r=BASE-1,ans=0;
```

```
            while(l<=r){
                long long mid=(l+r)>>1;
                if(b*mid<=rem){
                    ans=mid;
                    l=mid+1;
                }
                else r=mid-1;
            }
            q.s[i]=ans;
            rem=rem-b*ans;
        }
        q.clean();
        rem.clean();
        return make_pair(q,rem);
    }
    void clean(){
        while(len>1 && !s[len-1]) len--;
        if(len==1 &&s[0]==0) sign=false;
    }
    bign operator+(const bign &b)const{
        if(sign==b.sign){
            bign c=abs_add(b);
            c.sign=sign;
            return c;
        }
        else{
            if(abs_less(b)){
                bign c=b.abs_sub(*this);
                c.sign=b.sign;
                return c;
```

```
            }
            else{
                bign c=abs_sub(b);
                c.sign=sign;
                return c;
            }
        }
    }
    bign operator-(const bign &b)const{bign t=b;t.sign=!t.sign;return
*this+t;}
    bign operator*(const bign &b)const{bign
c=abs_mul(b);c.sign=(sign!=b.sign);c.clean();return c;}
    bign operator/(const bign &b)const{pair<bign,bign>
res=div_mod(b);res.first.sign=(sign!=b.sign);res.first.clean();return
res.first;}
    bign operator%(const bign &b)const{pair<bign,bign>
res=div_mod(b);res.second.sign=sign;if(res.second.sign)res.second=res.s
econd+b;res.second.clean();return res.second;}
    bign operator+=(const bign &b){*this=*this+b;return *this;}
    bign operator*=(const bign &b){*this=*this*b;return *this;}
    bign operator-=(const bign &b){*this=*this-b;return *this;}
    bign operator/=(const bign &b){*this=*this/b;return *this;}
    bign operator%=(const bign &b){*this=*this%b;return *this;}
    bool operator<(const bign &b)const{if(sign!=b.sign)return
sign;if(sign)return b.abs_less(*this);return abs_less(b);}
    bool operator>(const bign &b)const{return b<*this;}
    bool operator<=(const bign &b)const{return !(*this>b);}
    bool operator>=(const bign &b)const{return !(*this<b);}
    bool operator==(const bign
&b)const{return !(*this<b)&&!(*this>b);}
```

```
    bool operator!=(const bign &b)const{return !(*this==b);}
    void print(){
        if(sign)putchar('-');
        printf("%lld",s[len-1]);
        for(int i=len-2;i>=0;--i) printf("%08lld",s[i]);
        putchar('\n');
    }
    static bign from_bytes(const uint8_t* buf,int len){
        bign res=0;
        for(int i=0;i<len;++i) res=res*256+buf[i];
        return res;
    }
    bign operator/(long long x)const{
        bign c;c.len=len;
        long long r=0;
        for(int i=len-1;i>=0;i--){
            long long cur=r*BASE+s[i];
            c.s[i]=cur/x;
            r=cur%x;
        }
        c.clean();
        return c;
    }
    long long operator%(long long x)const{
        long long r=0;
        for(int i=len-1;i>=0;i--) r=(r*BASE+s[i])%x;
        return r;
    }
    void to_bytes(uint8_t* buf,int len)const{
        bign temp=*this;
```

```
        for(int i=len-1;i>=0;--i){
            buf[i]=(temp%256);
            temp=temp/256;
        }
    }
};
bign zero=0;
void ex_gcd(bign a,bign b,bign& x,bign& y){
    if(b==zero){
        x=1;
        y=0;
        return;
    }
    bign x0=1,x1=0,y0=0,y1=1;
    bign r,q,tmp;
    while(b!=zero){
        q=a/b;
        r=a%b;
        tmp=x0-q*x1;
        x0=x1;
        x1=tmp;
        tmp=y0-q*y1;
        y0=y1;
        y1=tmp;
        a=b;
        b=r;
    }
    x=x0;
    y=y0;
}
```

```
bign mont_mul(bign a,bign b,bign p,bign p_inv,int k){
    bign T=a*b,T_mod_R;
    T_mod_R.len=0;
    for(int i=0;i<min(T.len,k);++i) T_mod_R.s[T_mod_R.len++]=T.s[i];
    bign m=T_mod_R*p_inv,m_mod_R;
    m_mod_R.len=0;
    for(int i=0;i<min(m.len,k);++i) m_mod_R.s[m_mod_R.len++]=m.s[i];
    m=m_mod_R;
    bign val=T+m*p,t;
    t.len=0;
    if(val.len>k) for(int i=k;i<val.len;++i) t.s[t.len++]=val.s[i];
    if(t>=p) t=t-p;
    return t;
}
bign mont_pow(bign a,bign b,bign p){
    int k=p.len;
    bign R,y,p_inv0;
    memset(R.s,0,sizeof(R.s));
    R.s[k]=1;
    R.len=k+1;
    R.sign=false;
    ex_gcd(p,R,p_inv0,y);
    bign p_inv=zero-p_inv0;
    p_inv=p_inv%R;
    if(p_inv.sign) p_inv=p_inv+R;
    bign a_tmp=(a*R)%p,res=R%p,b_tmp=b;
    while(b_tmp>zero){
        if(b_tmp.s[0]&1) res=mont_mul(res,a_tmp,p,p_inv,k);
        a_tmp=mont_mul(a_tmp,a_tmp,p,p_inv,k);
        int r=0;
```

```
        for(int i=b_tmp.len-1;i>=0;--i){
            long long cur=b_tmp.s[i]+(long long)r*BASE;
            b_tmp.s[i]=cur/2;
            r=cur%2;
        }
        b_tmp.clean();
    }
    return mont_mul(res,1,p,p_inv,k);
}

char s[6000],s2[6000],sp[6000];
uint8_t tmp[1024],seed[32];

/*
uint8_t seed[32]={0x84,0xEE,0x1D,0x92,0xBE,0xFC,0x55,0x96,
                  0x6F,0x20,0xB6,0xFD,0x18,0xFC,0x45,0x20,
                  0xCF,0x17,0x0B,0xD8,0x92,0xE2,0xE0,0xD3,
                  0x4F,0xE7,0xA0,0x10,0x17,0xC8,0x6B,0x67};
                  */
const uint8_t null_hash[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};

int t;
bign x,y;
vector<uint8_t> MGF(uint8_t seed[],int seedlen,int len){
    vector<uint8_t> v,data;
    v.reserve(len+32);
    uint32_t cnt=0,hash_res[8];
```

```
    while(v.size()<len){
        data.clear();
        data.insert(data.end(),seed,seed+seedlen);
        data.push_back((cnt>>24)&0xFF);
        data.push_back((cnt>>16)&0xFF);
        data.push_back((cnt>>8)&0xFF);
        data.push_back(cnt&0xFF);
        uint64_t currentLen=data.size(),bitLen=currentLen*8;
        data.push_back(0x80);
        while((data.size()%64)!=56) data.push_back(0x00);
        for(int i=7;i>=0;--i) data.push_back((bitLen>>(i*8))&0xFF);
        for(int i=0;i<data.size();i+=64){
            sha_256(data.data()+i,hash_res,(i==0?0:1));
        }
        for(int i=0;i<8;++i){
            uint32_t val=hash_res[i];
            v.push_back((val>>24)&0xFF);
            v.push_back((val>>16)&0xFF);
            v.push_back((val>>8)&0xFF);
            v.push_back(val&0xFF);
        }
        cnt++;
    }
    v.resize(len);
    return v;
}
int main(){
    #ifdef _WIN32
    setmode(fileno(stdin), O_BINARY);
    setmode(fileno(stdout), O_BINARY);
```

```
        #endif
        for(int i=0;i<32;i+=8){
            unsigned long long t;
            while(_rdrand64_step(&t)==0) ;//wait for randrom
            memcpy(seed+i,&t,8);
        }
        uint8_t buf[256],skip[256];
        fread(skip,1,16,fp);
        fread(buf,1,256,fp);
        bign n=bign::from_bytes(buf,256);
        fread(buf,1,256,fp);
        bign e=bign::from_bytes(buf,256);
        fread(skip,1,256,fp);
        uint8_t mLen;
        fread(&mLen,1,1,fp);
        uint8_t M[256];
        fread(M,1,mLen,fp);
        int psLen=190-mLen;
        int dbLen=33+psLen+mLen;
        vector<uint8_t> DB;
        DB.insert(DB.end(),null_hash,null_hash+32);
        for(int i=0;i<psLen;++i) DB.push_back(0x00);
        DB.push_back(0x01);
        DB.insert(DB.end(),M,M+mLen);
        vector<uint8_t> dbMask=MGF(seed,32,dbLen),maskedDB(dbLen);
        for(int i=0;i<dbLen;++i) maskedDB[i]=DB[i]^dbMask[i];
        vector<uint8_t>
seedMask=MGF(maskedDB.data(),dbLen,32),maskedSeed(32);
        //for(int i=0;i<32;++i) cout<<hex<<(int)seedMask[i]<<' ';
        //cout<<endl;
```

```
        for(int i=0;i<32;++i) maskedSeed[i]=seed[i]^seedMask[i];
        //for(int i=0;i<32;++i) cout<<hex<<(int)maskedSeed[i]<<' ';
        vector<uint8_t> EM;
        EM.push_back(0x00);
        EM.insert(EM.end(),maskedSeed.begin(),maskedSeed.end());
        EM.insert(EM.end(),maskedDB.begin(),maskedDB.end());
        //for(int i=0;i<EM.size();++i) cout<<hex<<(int)EM[i]<<' ';
        bign m=bign::from_bytes(EM.data(),256);
        bign c=mont_pow(m,e,n);
        uint8_t C[256];
        c.to_bytes(C,256);
        fwrite(C,1,256,p);
        return 0;
    }
```

然后是解密，前面部分与 RSA 加密部分完全相同，仅展示 main 函数：

```
    int main(){
        #ifdef _WIN32
        setmode(fileno(stdin), O_BINARY);
        setmode(fileno(stdout), O_BINARY);
        #endif
        uint8_t
p0[128],q[128],n[256],d[256],d_mod_pm1[128],d_mod_qm1[128],q_inv[128],C
[256];
        fread(p0,1,128,fp);
        fread(q,1,128,fp);
        fread(n,1,256,fp);
        fread(d,1,256,fp);
        fread(d_mod_pm1,1,128,fp);
        fread(d_mod_qm1,1,128,fp);
```

```
        fread(q_inv,1,128,fp);
        if(fread(C,1,256,fp)!=256) return 0;
        bign p_bn=bign::from_bytes(p0,128);
        bign q_bn=bign::from_bytes(q,128);
        bign d_mod_pm1_bn=bign::from_bytes(d_mod_pm1,128);
        bign d_mod_qm1_bn=bign::from_bytes(d_mod_qm1,128);
        bign qInv_bn=bign::from_bytes(q_inv,128);
        bign C_bn=bign::from_bytes(C,256);
        //CRT
        bign cp=C_bn%p_bn,cq=C_bn%q_bn,m1,m2;
        m1=mont_pow(cp,d_mod_pm1_bn,p_bn);
        m2=mont_pow(cq,d_mod_qm1_bn,q_bn);
        bign ttmp=((m1-m2)*qInv_bn)%p_bn;
        bign m=m2+ttmp*q_bn;
        uint8_t EM[256];
        m.to_bytes(EM,256);
        if(EM[0]!=0x00) return 0;
        uint8_t* MaskedSeed=EM+1;
        uint8_t* MaskedDB=EM+33;
        vector<uint8_t> seedMask=MGF(MaskedDB,223,32);
        uint8_t Seed[32];
        for(int i=0;i<32;++i) Seed[i]=MaskedSeed[i]^seedMask[i];
        vector<uint8_t> dbMask=MGF(Seed,32,223);
        uint8_t DB[256];
        for(int i=0;i<223;++i){
            DB[i]=MaskedDB[i]^dbMask[i];
            //cout<<hex<<(int)DB[i]<<' ';
        }
        for(int i=0;i<32;++i) if(DB[i]!=null_hash[i]) return 0;
        int idx=-1;
```

```
        for(int i=32;i<223;++i){
            if(DB[i]==0x01){
                idx=i;
                break;
            }
            if(DB[i]!=0x00) return 0;
        }
        if(idx==-1) return 0;
        //cout<<idx<<endl;
        int mStart=idx+1;
        int mLen=223-mStart;
        if(mLen>0) fwrite(DB+mStart,1,mLen,p);
        return 0;
    }
```

五、实验结果

首先是加密过程，以样例 1 固定 seed 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 为例，保留中间结果输出，

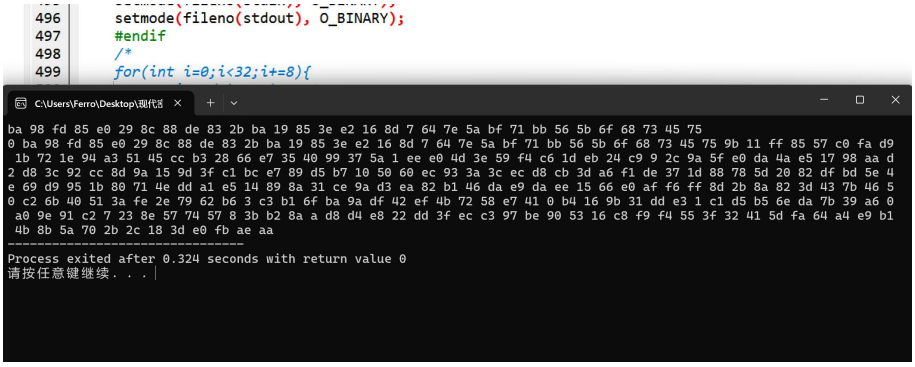

图 3 中间结果（maskedSeed 与 EM）

不难验证与样例展示的中间结果相同。这说明填充部分正确。

检查 output.bin：

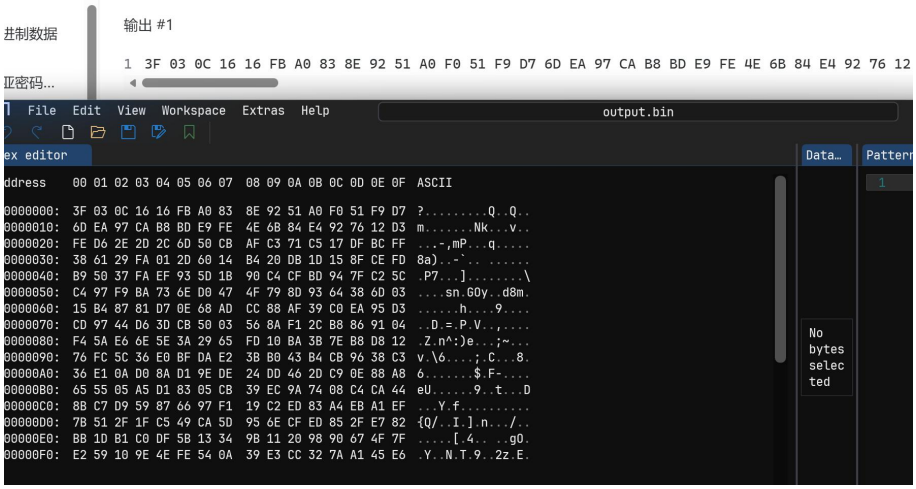

图 4 RSA 加密最终结果

逐一对比字节可以验证完全正确，这验证了大数运算部分的正确性。

其次是解密过程，仍然以第一个样例为例：

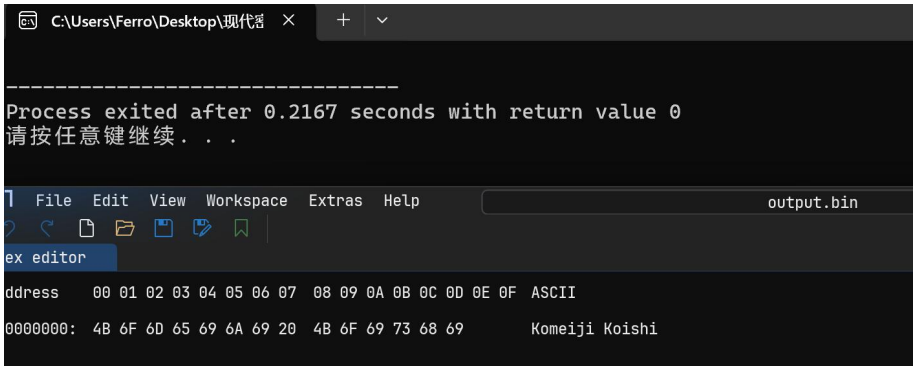

图 5 RSA 解密结果

得到有意义的明文 Komeiji Koishi（古明地恋），~~车万人狂喜~~！！

在 OJ 上的提交记录分别如下：

测试数据点全部通过，说明特判和加解密都没有问题。

六、思考题

1. RDRAND 的工作原理？

Apache RDRAND 的随机数生成器基于 Linux 内核中的/dev/urandom 接口，这个接口可以直接访问硬件随机数生成器。具体的说，硬件随机数生成器（HRNG）是一种基于物理原理的随机数生成器，如放射性衰变、热噪声等，这些物理过程具有随机性，可以产生高质量的随

机数。因此，它不同于“伪随机数”，是密码学意义下的安全随机函数。

2.为什么不用 Karatsubu 算法或者更快的 FFT 而只是朴素乘？

参考 https://www.luogu.com/article/h4jctljl，如果只是单纯拆分的话，常数比较大，甚至不如朴素乘，如果设置一个下限的话代码量比较大，而且实现复杂。（事实上，在 java 的 BigInt 中利用了普通乘法与 Karatsuba 算法、Toom-Cook 乘法的结合，根据位数决定采用哪种乘法，这在未来可以参考并进一步加速）

至于 FFT，由于是大整数的乘法，利用 FFT 容易丢失精度，最后需要将浮点结果四舍五入并处理进位，如果中间累计的误差较大，容易出现错误。

综上，实际代码实现中仍然采用朴素乘。

七、实验总结

本次实验是第一次公钥密码学的实验。通过前置的大数运算加法、乘法、模乘、模幂，进而实现 2048bit-RSA 的加解密，考察了大小端转化、二进制到 BASE 进制转换的能力以及对公钥密码学的理解（比如不对称消息填充包括随机数的产生），同时应用了上次 SHA-256 作为 hash 函数。大数运算的优化，能够使 RSA 加解密等比例的加速，因此选取合适的算法进行加速非常重要。此外，利用中国剩余定理对 RSA 进行解密也带来了一定程度的加速，但是实际应用中可能会带来侧信道攻击的风险，因此需要一些冗余计算防止侧信道攻击。

八、参考文献

【1】C++大整数运算（四）：除法，kedixa，
https://blog.kedixa.top/2017/cpp-bigint-div/#
【2】公钥密码学（使用一对公钥和私钥的密码学）_百度百科，
https://baike.baidu.com/item/%E5%85%AC%E9%92%A5%E5%AF%86%E7%A0%81%E5%AD%A6/24810