南開大學
Nankai University

南开大学网络空间安全学院

《密码学》实验报告

学　　号：1813540

姓　　名：陈鸿运

年　　级：2018 级

专　　业：信息安全

完成日期：2020 年 12 月 22 日

目录

1 RSA 原理 …… 1
1.1 RSA 简介 …… 1
1.2 RSA 算法描述 …… 1
1.2.1 密钥的产生 …… 1
1.2.2 加密 …… 1
1.2.3 解密 …… 2
1.3 附加算法 …… 2
1.3.1 扩展欧几里得算法 …… 2
1.3.2 米勒-拉宾素性检验算法 …… 2
2 RSA 实现 …… 3
2.1 大整数类 …… 3
2.1.1 定义大整数类 …… 3
2.1.2 实现大整数类 …… 7
2.2 RSA 类 …… 8
2.2.1 定义 RSA 类 …… 8
2.2.2 实现 RSA 类 …… 10
2.3 加密与解密 …… 12
3 实验过程 …… 13
3.1 手工计算 …… 13
3.2 生成 512 比特素数 …… 13
3.3 明文加密与密文解密 …… 14
3.4 RSATool 程序 …… 15
A 素数生成程序框图 …… 17
B 加密解密程序框图 …… 18

第一章 RSA 原理

1.1 RSA 简介

RSA 算法是现今使用最广泛的公钥密码算法，也是号称地球上最安全的加密算法。在了解 RSA 算法之前，先熟悉下几个术语根据密钥的使用方法，可以将密码分为对称密码和公钥密码：对称密码：加密和解密使用同一种密钥的方式公钥密码：加密和解密使用不同的密码的方式，因此公钥密码通常也称为非对称密码。

1.2 RSA 算法描述

1.2.1 密钥的产生

- 选取两个保密的大素数 p 和 q；
- 计算 $n = p * q$，$\varphi(n) = (p-1) * (q-1)$，其中 $\varphi(n)$ 是 n 的欧拉函数值；
- 选取一整数 e，满足 $1 < e < \varphi(n)$，且 $gcd(\varphi(n), e) = 1$；
- 计算 d，满足

  $$d * e \equiv mod\varphi(n)$$

  即 d 是 e 在模 $\varphi(n)$ 下的乘法逆元，因 e 与 $\varphi(n)$ 互素，由模运算可知，它的乘法逆元一定存在；
- 以 e, n 为公钥，d, n 为密钥；

1.2.2 加密

加密时首先将明文比特串分组，使得每个分组对应的十进制数小于 n，即分组长度小于 $\log_2 n$。然后对每个明文分组 m，作加密运算：

$$c \equiv m^e modn$$

1.2.3 解密

对密文分组的解密运算为：

$$m \equiv c^d mod n$$

1.3 附加算法

1.3.1 扩展欧几里得算法

扩展欧几里得算法是欧几里得算法的扩展。已知整数 a、b，扩展欧几里得算法可以在求得 a、b 的最大公约数的同时，能找到整数 x、y（其中一个很可能为负数）. 使他们满足：

$$ax + by = gcd(a, b)$$

1.3.2 米勒-拉宾素性检验算法

要测试 N 是否为素数，首先将 $N-1$ 分解为 $2^s d$。在每次开始测试时，先随机选取一个介于 $[1, N-1]$ 的整数 a，之后如果对所有的 $r \in [0, s-1]$，若 $a^d mod N \neq 1$ 且 $a^{2^r d} mod N \neq -1$，则 N 是合数。否则，N 有 3/4 的概率是素数。

构成该算法的思想是，如果 $a^d \neq 1(mod n)$ 以及 $n = 1 + 2^s d$ 是素数，则值序列

$$a^d mod n, a^2 d mod n, a^4 d mod n, ..., a^{2^s d} modification$$

将以 1 结束，并且在头一个 1 的前面的值将是 $n-1$（当 p 是素数时，对于 $y^2 \equiv 1(mod p)$，仅有的解是 $y \equiv \pm 1(mod p)$，因为 $(y+1)(y-1)$ 必须是 p 的倍数）。注意，如果在该序列中出现了 $n-1$，则该序列中的下一个值一定是 1，因为 $(n-1)^2 \equiv n^2 - 2n + 1 \equiv 1(mod n)$。

第二章 RSA 实现

2.1 大整数类

2.1.1 定义大整数类

下面我们来看 RSA 算法的具体实现过程。因为算法所涉及的数相当大，而在 C++ 中并没有像 Java 中存在内置的大整数 BigInteger。因此，我们需要尝试自己实现，这里我们的实现参考了 Java 中的实现。主要代码如下：

```
class big_num
{
public:
    typedef long long long_t;
    typedef unsigned base_t;
    big_num() : is_negative(false) { data.push_back(0); }
    big_num(const big_num &);
    big_num(const string &);
    big_num(const long_t &);
    ~big_num();


    big_num add(const big_num &);
    big_num sub(const big_num &);
    big_num mul(const big_num &) const;
    big_num div(const big_num &);
    big_num remain(const big_num &);
    big_num mod(const big_num &);
    big_num div_rem(const big_num &, big_num &);

    big_num pow(const big_num &);
    big_num mod_pow(const big_num &, const big_num &)
```

```
const;
22        big_num mod_inverse(const big_num &);
23
24        big_num left_shift(const unsigned);
25        big_num right_shift(const unsigned);
26
27        int my_compare(const big_num &) const;

28        bool my_equal(const big_num &) const;
29        static big_num to_bignum(const long_t &);

30        string to_string() const;
31
32        big_num my_abs() const;
33
34     protected:
35        friend big_num operator + (const big_num &, const
    big_num &);
36        friend big_num operator - (const big_num &, const
    big_num &);
37        friend big_num operator * (const big_num &, const
    big_num &);
38        friend big_num operator / (const big_num &, const
    big_num &);
39        friend big_num operator % (const big_num &, const
    big_num &);
40        friend bool operator < (const big_num &, const big_num
     &);
41        friend bool operator > (const big_num &, const big_num
     &);
42        friend bool operator == (const big_num &, const
    big_num &);
43        friend bool operator <= (const big_num &, const
    big_num &);
44        friend bool operator >= (const big_num &, const
    big_num &);
45        friend bool operator != (const big_num &, const
```

```
    big_num &);
46
47
48
49      friend big_num operator + (const big_num &, const
    long_t &);
50      friend big_num operator - (const big_num &, const
    long_t &);
51      friend big_num operator * (const big_num &, const
    long_t &);
52      friend big_num operator / (const big_num &, const
    long_t &);
53      friend big_num operator % (const big_num &, const
    long_t &);
54      friend bool operator < (const big_num &, const long_t
    &);
55      friend bool operator > (const big_num &, const long_t
    &);
56      friend bool operator == (const big_num &, const long_t
     &);
57      friend bool operator <= (const big_num &, const long_t
     &);
58      friend bool operator >= (const big_num &, const long_t
     &);
59      friend bool operator != (const big_num &, const long_t
     &);
60
61      friend ostream & operator << (ostream &, const big_num
     &);
62      big_num operator = (const string & str) { return (*
    this) = big_num(str); }
63      big_num operator = (const long_t & num) { return (*
    this) = big_num(num); }
64
65
66   private:
67      big_num trim();
```

```
68          int hex_to_dec(char);
69
70      public:
71          static const int base_bit = 5;
72          static const int base_char = 8;
73          static const int base_int = 32;
74          static const int base_num = 0xffffffff;
75          static const int base_temp = 0x1f;
76          static const big_num ZERO;
77          static const big_num ONE;
78          static const big_num TWO;
79          static const big_num TEN;
80
81
82      private:
83          bool is_negative;
84          vector<base_t> data;
85
86          class bit {
87          public:
88              bit(const big_num &);
89
90              size_t size() { return length; }
91              bool at(size_t);
92
93          private:
94              vector<base_t> bit_vector;
95              size_t length;
96          };
97          friend class RSA;
98
99      };
```

这里我们主要定义了大整数类的加、减、乘、除、取模、取余等基本操作，除此之外，我们还重载了多种运算符以便对这一类进行操作。

2.1.2 实现大整数类

由于对其进行实现的代码量相当大，故不在此全部放出，我们只讨论几个较为重要的实现。

```
//扩展欧几里得算法求乘法逆元
//m代表求逆元时的模数
big_num big_num::mod_inverse(const big_num & m)
{
    assert(!is_negative); //当前大整数为负数时就报错
    assert(!m.is_negative); //m也要为正数，否则报错
    if (my_equal(ZERO) || m.my_equal(ZERO))
        return ZERO;  //二者之中有一个为0就不存在乘法逆元
    big_num a[3], b[3], t[3];

    // 以下进行初等变换
    a[0] = 0; a[1] = 1; a[2] = *this;
    b[0] = 1; b[1] = 0; b[2] = m;

    for (t[2] = a[2].mod(b[2]); !t[2].my_equal(ZERO); t[2] = a[2].mod(b[2])) {
        big_num temp = a[2].div(b[2]);
        for (int i = 0; i < 3; ++i) {
            t[i] = a[i].sub(temp.mul(b[i]));// 不超过一次a[2]-temp*b[2]就变为大数减小数
            a[i] = b[i];
            b[i] = t[i];
        }

    }
    if (b[2].my_equal(ONE)) {// 最大公约数为1,存在乘法逆元
        if (b[1].is_negative)// 逆元为负数
            b[1] = b[1].add(m);// 变为正数,使其在m的剩余集中
        return b[1];

    }
    return ZERO;// 最大公约数不为1,无乘法逆元
}
```

上面是扩展欧几里得算法的实现，主要用于求乘法逆元。下面的代码展示了模幂运算的实现，这一运算在我们算法实现过程中使用得较为频繁：

```
big_num big_num::mod_pow(const big_num & exponent, const
   big_num & m) const
{
    assert(!m.my_equal(ZERO));  //模数为0，报错
    big_num num(1);
    bit t(exponent);
    for (int i = t.size() - 1; i >= 0; i--)
    {
        num = num.mul(num).mod(m);
        if (t.at(i))
            num = mul(num).mod(m);
    }
    return num;
}
```

2.2 RSA 类

2.2.1 定义 RSA 类

在 RSA 类中，我们主要考虑实现使用公钥进行加密、使用私钥进行解密，以及生成大奇数，并判断其是否为素数等等。

```
class RSA
{
public:
    RSA(){}
    RSA(const unsigned len) { init(len); }       //利用len初始
   化对象
    ~RSA(){}

    void init(const unsigned);              //初始化，产生公私钥
   对
```

```
9
10      big_num encrypt_by_public(const big_num &);    //公钥加密
11      big_num decrypt_by_private(const big_num &);   //私钥解密
12
13      //可用于数字签名
14      big_num encrypt_by_private(const big_num &);   //私钥加密
15      big_num decrypt_by_public(const big_num &);    //公钥解密
16
17  protected:
18      friend ostream & operator << (ostream &, const RSA &);  //输出相关数据
19
20  private:
21      big_num create_odd_num(unsigned);            //生成一个大奇数，参数为其长度
22      bool is_prime(const big_num &, const unsigned);      //判断是否为素数
23      big_num create_rand_smaller(const big_num &);      //随机生成一个更小的数
24      big_num create_prime(unsigned, const unsigned);      //生成一个大素数，参数为其长度
25      void create_exponent(const big_num &);           //根据提供的欧拉数生成公钥、私钥指数
26
27  public:
28      big_num n, e;    //公钥
29
30  private:
31      big_num d;       //私钥
32      big_num p, q;    //大素数p、q
33      big_num eul;     //n的欧拉函数
34
35
36  };
```

2.2.2 实现 RSA 类

通大整数类，我们考虑几个重要的实现：

这是实现生成大奇数的代码，我们采用随机函数（系统时间提供随机数种子）不断生成介于 0 至 F（十六进制）的随机数，将其存入 string 变量中，随后采取我们在大整数类中所实现的将 string 变量转化为大整数的策略，获取我们需要的大整数。最后一位我们固定使其为 1，这样保证了生成的数为奇数。

```
//生成一个长度为len的奇数
//我们采取二进制，len表示二进制数的长度
big_num RSA::create_odd_num(unsigned len)
{
    static const char hex_table[] =
    { '0', '1', '2', '3', '4', '5', '6', '7',
    '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

    len >>= 2;  //十六进制数据每位占4位二进制
    if (len)
    {
        ostringstream oss;    //注意头文件sstream
        for (size_t i = 0; i < len - 1; i++)
        {
            oss << hex_table[rand() % 16];
        }
        oss << hex_table[1];
        return big_num(oss.str());
    }
    return big_num("F");
};
```

下面是实现素数判断的算法，这里我们采用了米勒-拉宾素性检验算法，代码如下：

```
//判断一个数是否为素数，使用米勒拉宾大素数检测算法
//num代表需要判断的数，k代表测试次数
bool RSA::is_prime(const big_num & num, const unsigned k)
{
    assert(num != big_num::ZERO); //若num为0，则报错
    if (num == big_num::ONE)     //1不为素数
        return false;
    if (num == big_num::TWO)     //2是素数
        return true;

    big_num t = num - 1;
    big_num::bit b(t);          //采用二进制进行处理
    if (b.at(0) == 1)          //减去1之后为奇数，说明原数为偶数，不为素数
        return false;

    //num-1 = 2^s*d
    size_t s = 0;           //用于统计二进制末尾有多少个0
    big_num d(t);
    for (size_t i = 0; i < b.size(); i++)
    {
        if (!b.at(i))
        {
            s++;
            d = d.right_shift(1);
        }
        else
            break;
    }

    for (size_t i = 0; i < k; i++)   //测试次数为k
    {

        big_num a = create_rand_smaller(num); //生成一个介于1至num-1之间的随机数
        big_num x = a.mod_pow(d, num);       //若a^(num-1)=1(
```

```
    mod num),
35
36          if (x == big_num::ONE)              //这次测试是素数
37              continue;
38          bool ok = true;
39
40          //二次检测
41          for (size_t j = 0; j < s&&ok; j++)
42          {
43              if (x == t)                 //若a^(num-1)=num-1(mod
    num);
44                  ok = false;
45              x = x.mul(x).mod(num);
46          }
47          if (ok)
48              return false;
49      }
50      return true;  //测试通过, 极有可能为素数
51
52  }
```

2.3 加密与解密

在实现了上述过程后，我们实现 RSA 的加密与解密是相当简单的：

```
1  //使用公钥进行加密
2  big_num RSA::encrypt_by_public(const big_num & m)
3  {
4      return m.mod_pow(e, n);
5  }
6  //使用私钥进行解密
7  big_num RSA::decrypt_by_private(const big_num & c)
8  {
9      return c.mod_pow(d, n);
10 }
```

第三章 实验过程

3.1 手工计算

题目：

根据已知参数：p = 3，q = 11，m = 2，手工计算公钥和私钥，并对明文 m 进行加密，然后对密文进行解密。

解答：

由 $p = 3$，$q = 11$，得 $\varphi(n) = \varphi(pq) = 20$，且 $n = 3 * 11 = 33$。

不妨取正整数 $e = 17$，得公钥 $(n, e) = (33, 17)$；

又 $e^{-1}(modn) = 13(mod20) \rightarrow d = 13$；

所以得私钥 $(n, d) = (33, 13)$；

下面执行加密解密过程：

明文 $m = 2$，那么密文 $c = m^e(modn) = 2^{17}(mod33) = 29$；

考虑所得密文 $c = 29$，那么明文 $m = c^d(modn) = 29^{23}(mod33) = 2$；

成功加密解密。

3.2 生成 512 比特素数

具体代码我们在前文已经讲述，这里我们来查看实现的效果：

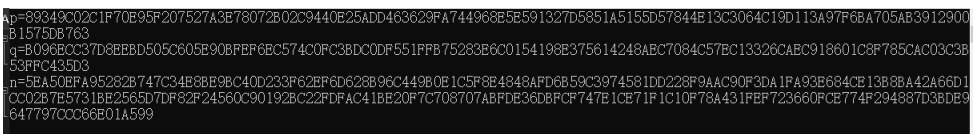

图 3.1: 素数生成

可以看到，程序成功生成了素数 p：

89349C02C1F70E95F207527A3E78072B02C9440E25ADD463629FA744968E5E59
1327D5851A5155D57844E13C3064C19D113A97F6BA705AB3912900B1575DB763

素数 q：

B096ECC37D8EEBD505C605E90BFEF6EC574C0FC3BDC0DF551FFB75283E6C0154
198E375614248AEC7084C57EC13326CAEC918601C8F785CAC03C3B53FFC435D3

大整数 n：

5EA50EFA95282B747C34E8BE9BC40D233F62EF6D628B96C449B0E1C5F8E4848
AFD6B59C3974581DD228F9AAC90F3DA1FA93E684CE13B8BA42A66D1CC02B7E5
731BE2565D7DF82F24560C90192BC22FDFAC41BE20F7C708707ABFDE36DBFCF
747E1CE71F1C10F78A431FEF723660FCE774F294887D3BDE9647797CCC66E01
A599

3.3 明文加密与密文解密

我们使用上文生成的大素数，对明文进行加密，密文进行解密，这里我们使用我的学号“1813540”，结果如下：

图 3.2: 执行结果

可以看到，程序成功进行了解密与解密。

3.4 RSATool 程序

下面我们来熟悉本次实验所提供的 RSATool 程序，这里我们按照默认参数，生成随机大素数，并对我的学号“1813540”进行加密，得到如下结果：

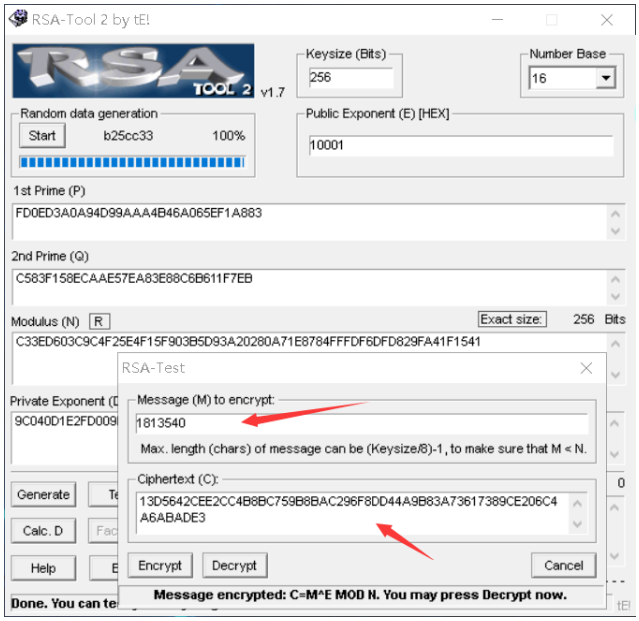

图 3.3: RSATool 程序加密效果

解密结果如下：

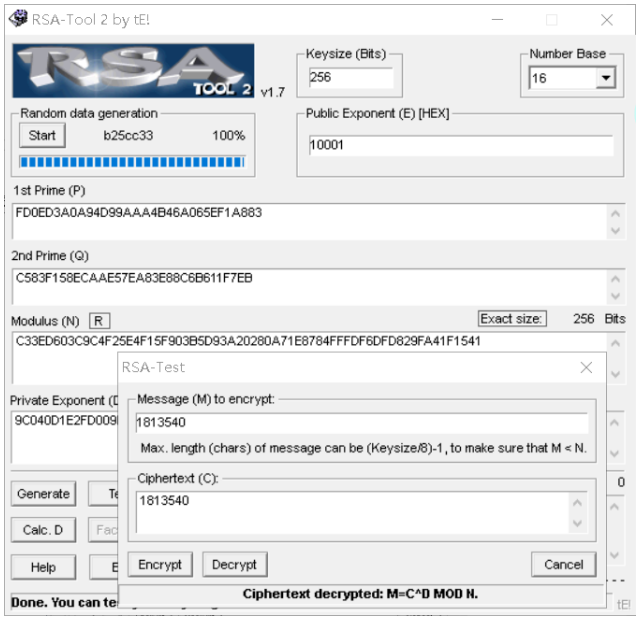

图 3.4: RSATool 程序解密效果

附录 A 素数生成程序框图

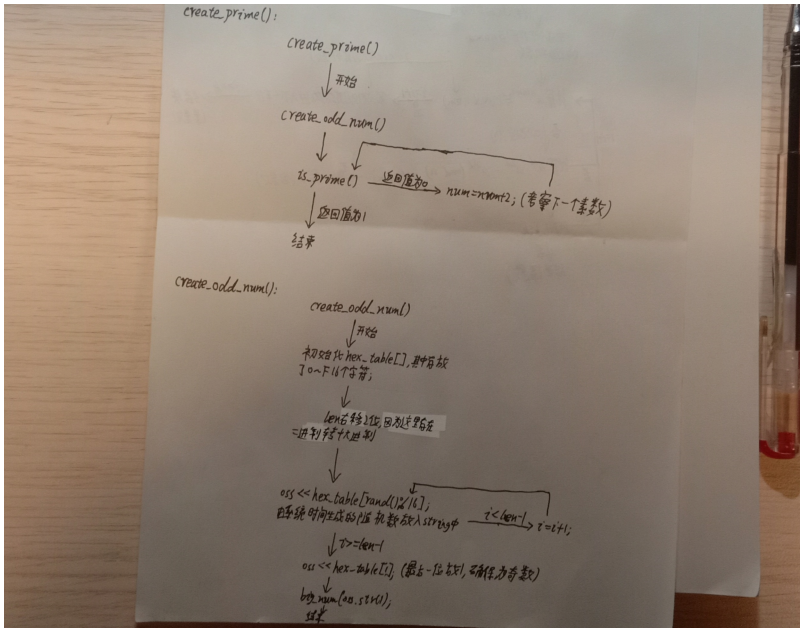

图 A.1: 素数生成程序框图 1(手绘)

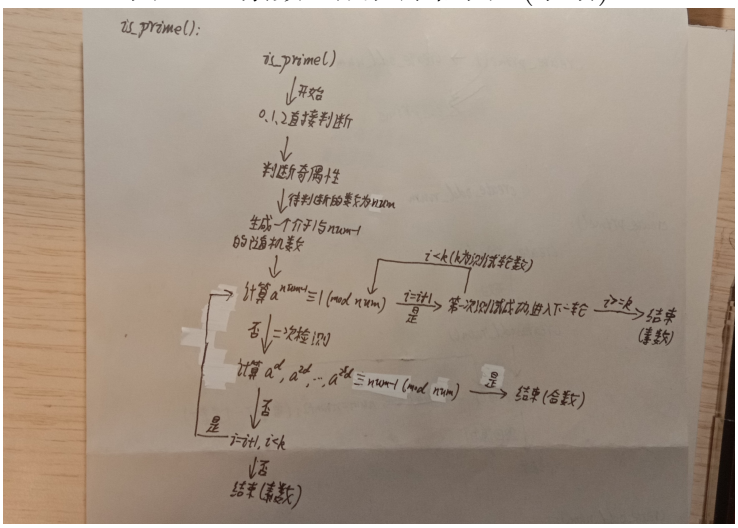

图 A.2: 素数生成程序框图 2(手绘)

附录 B 加密解密程序框图

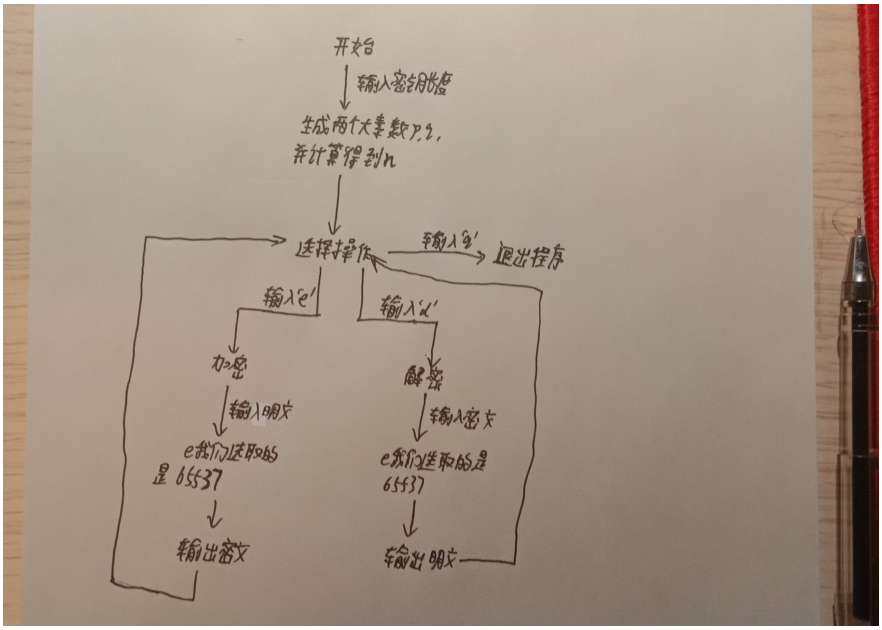

图 B.1: 加密解密程序框图 (手绘)