


《现代密码学》实验报告

实验名称: RSA	实验时间: 2022.11.21
学生姓名: 伍建霖	学号: 20337251
学生班级: 20网安	成绩评定:

一、实验目的

了解RSA密码体制的原理, 通过OAEP最有非对称加密填充实现RSA加密, 并且给出解密函数进行验证。

二、实验内容




实验5 RSA实现

- 实验内容:
实现2048RSA加密并给出解密函数验证。
- 要求:
 - p 和 q 是两个1024位的安全素数。(每个人按照自己的学号最后一位的数字选择, 例如学号20214876可选择第6组)
 - 密钥: 公钥为学号的下一个素数, 根据公钥生成私钥
 - 明文: 姓名全拼(高位补0填充到 m 比特的长度)并进行简单的OAEP填充

*填充时使用的函数 G 、 H 使用实验4实现的SHA256的扩展形式(迭代做四次SHA256, 得到4个256比特的结果, 按顺序排列, 并将前两个字节置0x00), 为方便可令 $k_0 = m = 1024$ 。

- **DDL: 11.22 实验课上课之前**



三、实验原理

RSA

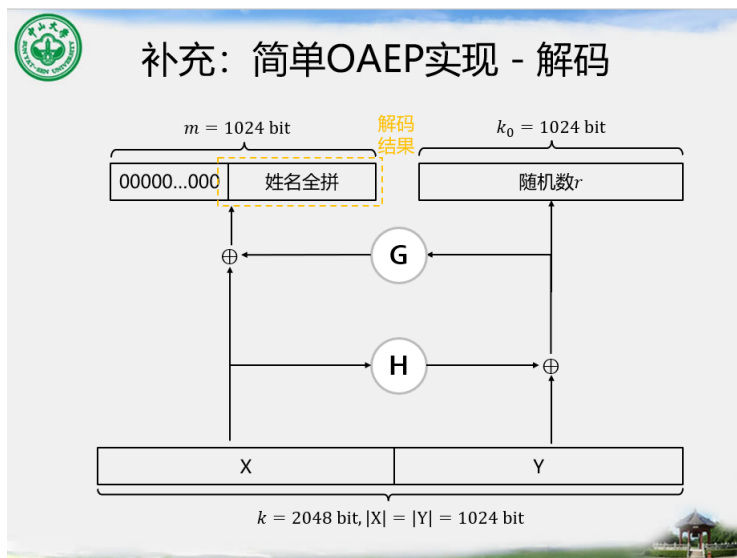
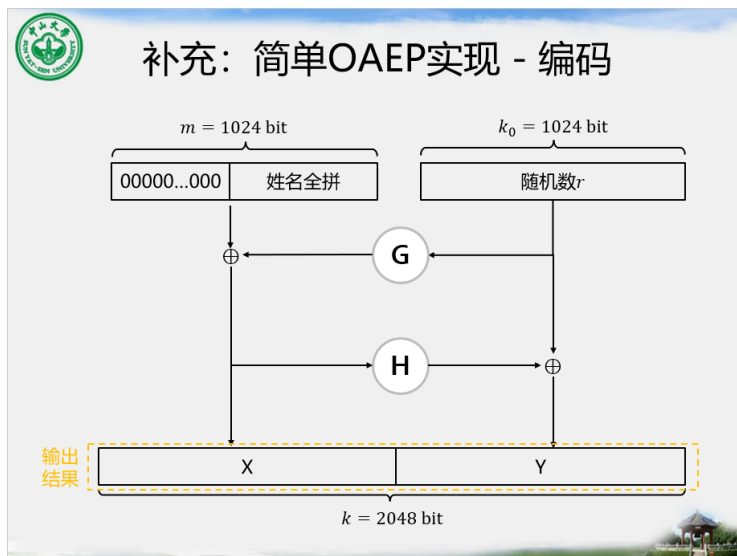
rsa的原理很简单, 发送者用公钥和 $n(n=pq)$ 做模幂运算加密, 接收方用私钥和 n 做模幂解密。其中 pq 都是大素数, 由 $\phi_n = (p-1)(q-1)$, 可由扩展欧几里得算法求逆得出私钥。

$$C = M^e \bmod n$$
$$M = C^d \bmod n$$

OAEP

OAEP填充是为了将明文扩展到2048位，完善RSA的安全性。先从高位补0使 m 达到1024位，并准备一个1024位的随机数，然后 $\text{OAEP_output} = ((\text{MGF}(m) \oplus r) \parallel (\text{MGF}(r) \oplus p_1))$ 。

OAEP去填充则和填充相反，先将 $\text{MGF}(p_1)$ 和 p_2 异或得到 r ，再将 $\text{MGF}(r)$ 和 p_1 异或获得填充前的消息，最后去掉高位补的0就得到了最初的明文。MGF(x): 先将 x 分别和0123拼接，再分别计算sha256，再将四次hash值拼接，最后输出。



四、实验步骤

大致流程：

初始化参数----->

明文-----oaepPadding----->明文_OAEP

-----rsaEncrypt----->密文

-----rsaDecrypt----->解密后的明文_OAEP

-----oaepUnpadding----->解密后的明文

GMP

这次实验中我使用了大数库GMP，先从gmp官网下载压缩包，解压，再下载msys2，安装msys2，在msys2中安装gcc，gdb等工具。接着将解压好了的gmp文件夹移动到msys2目录下的/home/username中，再打开msys2的控制台，以此输入./configure，make，make check，make install 即可，噢vscode中使用需要额外添加编译器指令。

RSA

生成参数

在实验的一开始，我们已经有了一对安全素数 p 和 q ，由此可算出 n 和 ϕ_n 。同时还有公钥（学号的下一个素数），由公钥和 ϕ_n 进行求逆运算我们可以得到私钥。至于求逆运算，gmp库提供了一个速度特别快的函数来实现，`mpz_invert`(私钥，公钥， ϕ_n)。

```
void initial()
{
    cout << "we have already got public key and phi_n," << '\n';
    cout << "now generate private key with them." << '\n';
    mpz_invert(privateKey.get_mpz_t(), publicKey.get_mpz_t(),
    phi_n.get_mpz_t());
}
```

加密

这一步中我们已经拿到了OAEP填充后的明文，使用gmp中的`mpz_powm()`进行模幂运算后就能拿到密文了。由于输入的数据为一个存着填充后明文的二进制格式的string，故需要将其转为十六进制或十进制才能存入`mpz_class`中。

```
void rsaEncrypt()
{
    string pt_oaep_hex = "";
    for (int i = 0; i < 2048;)
    {
        string temp = pt_oaep.substr(i, 4);
        if (temp == "0000")
        {
            pt_oaep_hex += '0';
        }
        ...
        else if (temp == "1110")
        {
            pt_oaep_hex += 'e';
        }
        else
        {
            pt_oaep_hex += 'f';
        }

        i = i + 4;
    }

    mpz_class pt_mc(pt_oaep_hex, 16);
    mpz_class ct_mc;
```

```

cout << "pt_oaep = " << pt_oaep << '\n';
mpz_powm(ct_mc.get_mpz_t(), pt_mc.get_mpz_t(), publicKey.get_mpz_t(),
n.get_mpz_t());
ct_oaep = ct_mc.get_str();
cout << "ct_mc = " << ct_mc << '\n';
}

```

解密

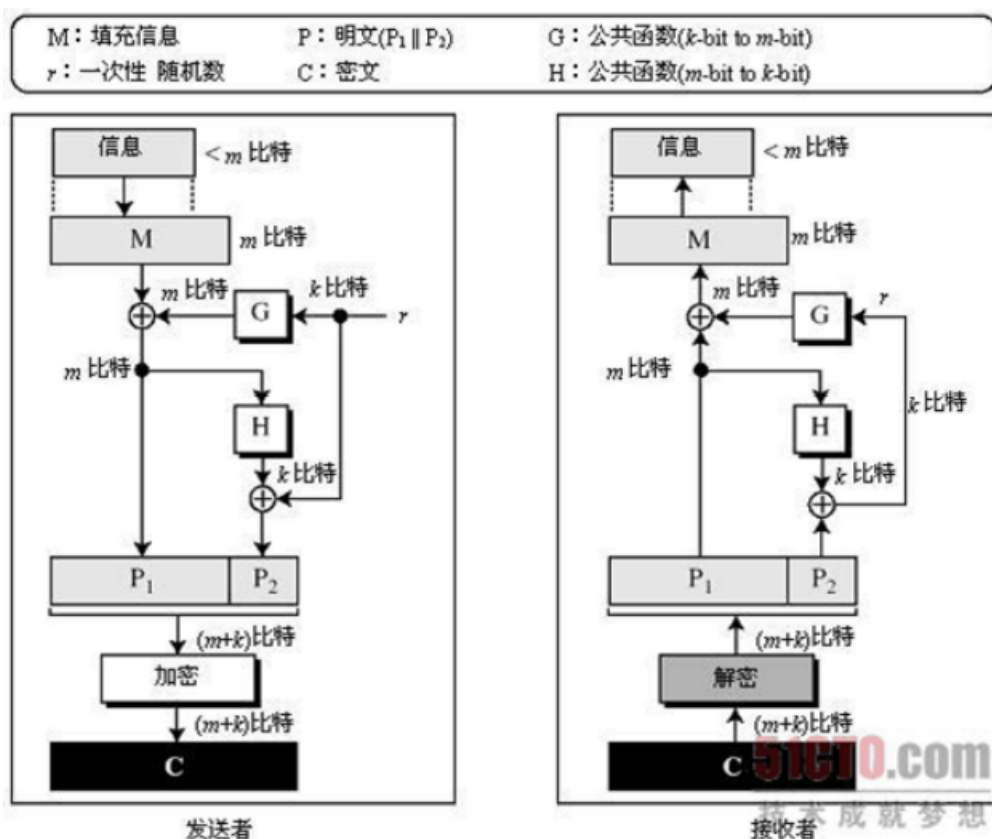
解密操作和加密是一样的，只不过少了格式转换的问题。

```

void rsaDecrypt()
{
    mpz_class dt_mc;
    mpz_class ct_mc(ct_oaep);
    mpz_powm(dt_mc.get_mpz_t(), ct_mc.get_mpz_t(), privateKey.get_mpz_t(),
n.get_mpz_t());
    dt_oaep = dt_mc.get_str();
}

```

OAEP



MGF

我先实现的MGF()函数，假设输入的数据为一个存放二进制数据的string，如“0001011101”，输出一个二进制数据的string。输入m先分别和4字节长度的0123拼接，再分别做sha256的哈希计算，接着将得到的4个hash值拼接，最后将前两个字节置0即可输出。由于上次实验实现的sha256输入的是像“wujianlin”这样的字符串，输出的是像“6ba7...9ace”这样的256个十六进制数，故需要进制转换，代码逻辑混乱了一点，有点冗余。

因此，最终 G, H 的计算过程如下：

- $h = H'(m||0) || H'(m||1) || H'(m||2) || H'(m||3)$

- 将 h 的前16个比特（即前两个字节）置0

- 输出 h

```
string MGF(string input)
{
    string m = input, ret = "", ret_hex;
    // 2. m||0, m||1, m||2, m||3
    string mm[4] = {m + "00000000000000000000000000000000", m +
"00000000000000000000000000000001", m + "00000000000000000000000000000010", m +
"00000000000000000000000000000011"};
    string input_ascii[4] = {"", "", "", ""};
    for (int i = 0; i < 1024; i = i + 8)
    {
        bitset<8> temp(mm[0].substr(i, 8));
        input_ascii[0] += (temp.to_ulong());
    }
    int bitlen = input_ascii[0].length() * 8;
    for (int i = 0; i < (1024 - bitlen) / 8; i++)
    {
        input_ascii[0] = "0" + input_ascii[0];
    }

    // 3. sha256 hash
    string input1 = input_ascii[0] + "0000";
    string Hash1 = sha256(input1);
    string Hash2 = sha256(input_ascii[0] + "0001");
    string Hash3 = sha256(input_ascii[0] + "0002");
    string Hash4 = sha256(input_ascii[0] + "0003");
    cout << "\ninput_ascii + 0000 = " << input1 << '\n';
    cout << "\nhash1 = " << Hash1 << '\n';
    if (Hash1 == Hash2)
    {
        cout << "\ninitiated....." << '\n';
    }

    // 4. H1 || H2 || H3 || H4
    string Hash = Hash1 + Hash2 + Hash3 + Hash4;
    // 5. highest 2 byte set 0
    ret_hex = "00" + Hash.substr(2, 256);
    // 6. hex -> bit
    for (int i = 0; i < 256; i++)
    {
        switch (ret_hex[i])
        {
            case '0':
                ret += "0000";
                break;
            ...
            ret += "1110";
        }
    }
}
```

```

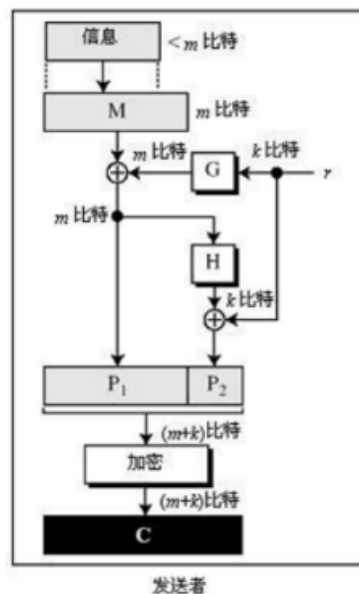
        break;
    case 'f':
        ret += "1111";
        break;
    }
}

return ret;
}

```

填充

接着是oaep填充部分：先将明文转成二进制的，再在高位和1024个“0”拼接，接着截后面的1024位，这样就实现了高位补0的操作，but这里我用的string来存储，一开始想着string可以很方便的拼接和初始化mpz_class以及bitset，但后来因此麻烦了不少。由于随机数我也是给的一个字符串，如“adfgghjkl”，所以也需要上述操作。补完0后，分别将两个1024位的二进制数的string传入MGF()，得到两个1024位的二进制string后，转成bitset来进行运算，如下图两两异或再拼接，就得到了我们需要的填充后的明文（2048位）。



```

// pt OAEP padding
void oaepPadding()
{
    // 1. expand pt and r to 1024 bit in bit
    for (int i = 0; i < pt_str.length(); i++)
    {
        bitset<8> temp(pt_str[i]);
        pt_bit += temp.to_string();
    }
    pt_bit = STR + pt_bit;
    pt_bit = pt_bit.substr(pt_bit.length() - 1024, 1024);

    string r_bit = "";
    for (int i = 0; i < r.length(); i++)
    {
        bitset<8> temp(r[i]);
        r_bit += temp.to_string();
    }
    r_bit = STR + r_bit;
}

```

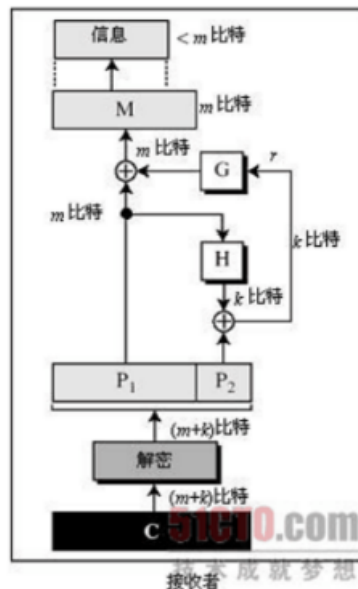
```

r_bit = r_bit.substr(r_bit.length() - 1024, 1024);
// 2. type exchange between bitset and string
string r_bit_mgf = MGF(r_bit);
string pt_bit_mgf = MGF(pt_bit);
bitset<1024> r_bit_bs(r_bit);
bitset<1024> pt_bit_bs(pt_bit);
bitset<1024> r_bit_mgf_bs(r_bit_mgf);
bitset<1024> pt_bit_mgf_bs(pt_bit_mgf);
// cout << "\npt_bit_bs = " << pt_bit_bs << '\n';
// cout << "\nr_bit_mgf_bs = " << r_bit_mgf_bs << '\n';
// cout << "\npt_bit_mgf_bs = " << pt_bit_mgf_bs << '\n';
// cout << "\nr_bit_bs = " << r_bit_bs << '\n';
// 3. generate oaepPadding's output
bitset<1024> p1 = pt_bit_bs ^ r_bit_mgf_bs;
bitset<1024> p2 = pt_bit_mgf_bs ^ r_bit_bs;
// cout << "\np1 when padding = " << p1 << '\n';
// cout << "\np2 = " << p2 << '\n';
pt_oaep = p1.to_string() + p2.to_string();
}

```

去填充

去填充和填充是一样的流程图，只不过箭头反着走。这里我们得到解密后的明文，是一个存放在string中的十进制数据，我们需要先将其转成二进制的string，再分割成两个1024位的string，接着如下图还原随机数 r ，最后得到高位补了0的解出来的明文，将补的0去掉就是我们最终解出来的明文。



```

void oaepUnpadding()
{
    // cout << "pt_oaep = " << pt_oaep << '\n';
    // cout << "ct_oaep = " << ct_oaep << '\n';
    // cout << "dt_oaep in Unpadding func = " << dt_oaep << '\n';
    string dt_oaep_bit = decToBit(dt_oaep);
    dt_oaep_bit = STR + dt_oaep_bit;
    dt_oaep_bit = STR + dt_oaep_bit;
    dt_oaep_bit = dt_oaep_bit.substr(dt_oaep_bit.length() - 2048, 2048);

    string p1 = dt_oaep_bit.substr(0, 1024);
    string p2 = dt_oaep_bit.substr(1024, 1024);
}

```



```

dt_oaep in Unpadding Func = 240232237161474033197402771816780344840987739529420041745645828361616285889565
7349421388194584274216256266676136283248905349531504345759344830215325686979099695936908149692750420223382
4327048908834957407007590387833856018475038986242324235268083795322820851855232379422625221101124524036379
6042652455655165656627010223906419130824644372648555223606676758271571765068929150466917043438870444063963
7121392631716000078545526798792720847497571458653864699156595264428463334022928595756658185739755874593380
8085575024519023273641208680849571086451642553401790507239517099282355519981954548919333785906557333750022
335387

now dt_oaep == pt_oaep
p1_mgf when unpadding = 000000000000000100101110110101100011010110011100010111011111000111011101111000101
001100010000110011100010101000000110011010101000111011000010001110011110011001000110001100001110011011100
111010000110101100110101100011111001010001001001101100010011010101111011001011010011110000111111011001110
11111001000010010110011001110011011111011110110010101101010111011001011010011110000110111010101101101000111
1110101011110110001011110100110010110001110001011001001100100001100110100101011001100100100011100101010
1111110011100001101110000111001010100110100011000110011111000001110110110110010001100100110111110000
0110110101101111100010111011111100011100011100001100010101101111100000011100000010110110110010100011010
0000001111100111100000100011101011100110101010000101000111100101010111000000010101101101010011100110000
10001110100000111010010101000100000100100100100011010100101010000010111111011100111111111110100000001
1010011100110101110100010000011111011011101000100100110011100100010101111110100111100000010101

```

接着判断dt_oaep转换为二进制后和pt_oaep是否相等，结果是相等，即rsa解密成功。p1_mgf为解密时的中间变量（调试时的输出）。

六、实验总结

这次实验算上补充理论知识到完成代码，花了我三个全天（从早10到晚12）。其中补充理论知识和安装gmp库花了我6个小时左右，主要是网上关于安装gmp的博客都有点过时了，导致有的地方需要靠猜来穷举试错，花了我不少时间。

至于代码部分，也遇到了不少麻烦。实现rsa部分倒是没什么麻烦，短短的七八十行就能解决，麻烦的是oaep部分。首先是oaep中sha256接口的问题，我上次实验设计的sha256输入一个字符串，我这次传入了一个1024位的string，需要额外转进制，就增加了一部分代码和逻辑。其次是有时候需要将string存放的大数转成mpz_class格式，或转成其他进制，需要捋逻辑和增加代码。

最后，也是最头痛的，由于使用到了gmp，我将这次实验的编译器换成了msys2中的g++，之前使用的是mingw64的g++。sha256在之前的编译器下能正确出结果，但是在这次试验的编译器下输出的结果不一样！！！这是我找了一晚上bug发现的，但实在不知道该怎么改，所以最后没能恢复出明文，只做出来个半成品