现代密码学实验报告

实验名称：RSA 加密的实现	实验时间：2024-12-17
学生姓名：庄云皓	学号：22336327
学生班级：22级保密管理	成绩评定：

实验5-1 RSA 加密的实现

实验目的

通过实现 2048 bit RSA 的加密函数，理解 RSA 密码体制的具体运作过程，进一步提高对于公钥密码体制的认识和理解。

实验内容

用C/C++ 实现2048 bit RSA 的加密函数
参数要求：

输入:

- `uint8_t[16]` 请忽略这个（应该是随机数种子）
- `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]` 请忽略这个(应该是以大端序表示的 2048 bit 的 d)
- `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）
- `uint8_t[]` 加密的消息内容

输出：

加密结果，也就是用大端序表示 256 bytes 的 $c = m^e \mod n$

实验原理

- 密钥生成
    1. 选择两个素数 p,q，计算 $n = pq$ 和 $\phi(n) = (p-1)(q-1)$
    2. 选择 e 满足 $gcd(e, \phi(n)) = 1$
    3. 计算 $d = e^{-1} \mod \phi(n)$
    4. 公钥$pk = (n, e)$私钥$sk = (p, q, d)$
- 将明文 m 加密为密文 $c = m^e \mod n$

由于textbook-RSA加密相同的明文一定会加密出相同的密文，这不符合语义安全（Semantic security）的要求。不满足选择明文攻击下的不可区分性（IND-CPA），所以要进行OAEP填充，参见 PKCS#1 v2.2 (RFC 8017)。填充过程在下面详细解释

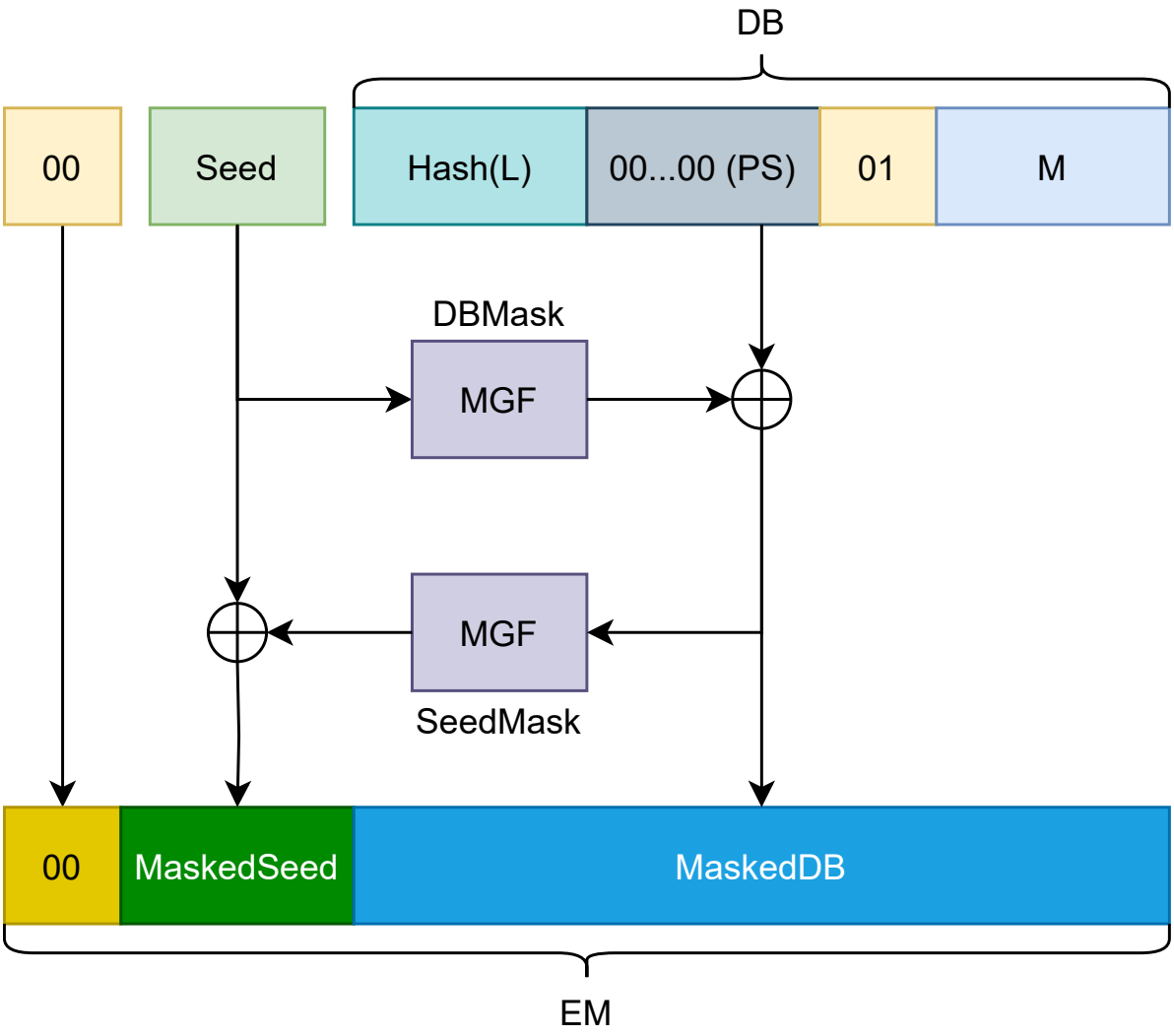
实验步骤（源代码）

- **预处理部分**：读取n,e，将原始的大端序转换为用`uint32_t[128]`的$radix = 2^{32}$进制表示,即$[a_0, a_1, a_2, \ldots, a_{127}]$表示$a_0 * radix^0 + a_1 * radix^1 + \ldots a_{127} * radix^{127}$,

  其中如果用A1,A2...分别表示依次读到的字符，$a_0 = [A3, A2, A1, A0]$，
  $a_1 = [A7, A6, A5, A4]$，以此类推

```
    uint8_t unknown [16];
    fread(unknown ,1,16,file);

    uint8_t n[256];
    uint8_t e[256];
    uint32_t N[128]={0};
    uint32_t E[128]={0};
    uint32_t M[128]={0};

    uint8_t skip[256];
    uint8_t len;


    BigInt cal0("1");

    for(int i = 0;i<64;i++){
        fread(n+i*4, 1, 4, file);
         uint32_t   value = ((uint32_t)n[4*i+0] << 24) |
            ((uint32_t)n[4*i+1] << 16) |
            ((uint32_t)n[4*i+2] << 8)  |
            ((uint32_t)n[4*i+3]);

         N[64-i-1]=value;

    }
    // printf("N=");
    // cout<<cal0.biToStr(N)<<endl;
    BigInt cal(cal0.biToStr(N));

    for(int i = 0;i<64;i++){
        fread(e+i*4, 1, 4, file);
         uint32_t   value = ((uint32_t)e[4*i+0] << 24) |
            ((uint32_t)e[4*i+1] << 16) |
            ((uint32_t)e[4*i+2] << 8)  |
            ((uint32_t)e[4*i+3]);
        E[64-i-1]=value;
    }
    // printf("E=");
    // cout<<cal.biToStr(E)<<endl;

    fread(skip, 1, 256, file);

    fread(&len, 1, 1, file);
```

- **OAEP填充**：使用最优非对称加密填充（Optimal Asymmetric Encryption Padding, OAEP），流程如下：



- 需要调用两个函数：
  - Hash：一个 hash 函数,在这个实验中采用sha256
  - MGF：掩码生成函数（Mask Generation Functions），相当于一种输出长度可以自定义的 hash 函数，后述
- m：需要加密的消息（Message）
- Seed：随机的二进制数据，长度和 Hash 的输出相同
- L：可选的标签（Label），可以为空字符串，在解密的时候也需要输入 L，并检查是否与 DB 中的 Hash(L) 部分相同
- PS：由 `0x00` 组成的填充部分（Padding String）
- DB：消息块（Data Block）
- EM：填充后的消息（Encoded Message），长度和 *n* 相同，与整数互相转换时使用大端序

图中上方部分：依次完成0x00,seed,hash(0),0x01,m,剩下部分都填0x00

```
    uint8_t m[260];

    m[0]=0x00;


    uint8_t seed[32]; // 存储随机字节的数组
    unsigned int random_value; // 用于接收32位随机数的变量
    int i = 0;
```

```
    while (i < 32) {

        if (_rdrand32_step(&random_value)) {

            seed[i++] = (uint8_t)(random_value & 0xFF); // 低8位
            seed[i++] = (uint8_t)((random_value >> 8) & 0xFF); // 次低8位
            seed[i++] = (uint8_t)((random_value >> 16) & 0xFF); // 次高8位
            seed[i++] = (uint8_t)((random_value >> 24) & 0xFF); // 高8位
        }
    }
    for(int i = 1;i < 33; i++){
        m[i]=seed[i-1];
    }

    for(int i = 33;i<65;i++){
        m[i]=hash_L[i-33];
    }

    m[256-len-1] = 0x01;
    fread(m+256-len,1,len,file);

    for(int i = 65;i< 256-len-1; i++){
        m[i]=0x00;
    }
```

MGF 函数

$$MGF_H(m, len) = (H(m \,||\, 0) \,||\, H(m \,||\, 1) \,||\, H(m \,||\, 2) \,||\, \cdots)[:len]$$

这里的0，1，2都是4字节的，如2表示0x00000002

H是我们前面实现的 SHA256 函数

```
int MGF1(const unsigned char *mgfSeed, unsigned int mgfSeedLen, unsigned int
maskLen, unsigned char *mask)
{
    unsigned char buf[MGF1_BUF_SIZE], *p;
    unsigned char digest[32];
    unsigned long digestLen=32;
    unsigned long counter, restLen;

    unsigned char buffer_sha[64];
    unsigned char data[MGF1_BUF_SIZE];
    memset(buffer_sha, 0, 64);

    uint32_t state[8];
    memcpy(state, H0, sizeof(H0));
    if (mgfSeedLen > MGF1_BUF_SIZE - 4)
    {
        printf("MGF1 buffer is not long enough!\n");
        return -1;
    }

    // copy mgfSeed to buffer
    memcpy(buf, mgfSeed, mgfSeedLen);
```

```
    // clear rest buffer to 0
    p = buf + mgfSeedLen;
    memset(p, 0, MGF1_BUF_SIZE-mgfSeedLen);


    counter = 0;
    restLen = maskLen;

    while (restLen > 0)
    {
        p[0] = (counter >> 24) & 0xff;
        p[1] = (counter >> 16) & 0xff;
        p[2] = (counter >>  8) & 0xff;
        p[3] = counter & 0xff;
        memcpy(data, buf, MGF1_BUF_SIZE);

        memset(buffer_sha, 0, 64);

        if (restLen >= digestLen)
        {

            sha256_update(data, state, buffer_sha, mgfSeedLen+4);

            sha256_final(buffer_sha, state, (unsigned char *)mask);

            // for(int i = 0; i < digestLen; i++)
            //     printf("%02x", mask[i]);
            // printf("\n");

            restLen -= digestLen;
            mask += digestLen;

            counter ++;
        }
        else // 剩余的不足单次哈希长度的部分
        {

            sha256_update(data, state, buffer_sha, mgfSeedLen+4);
            sha256_final(buffer_sha, state, (unsigned char *)digest);
            memcpy(mask, digest, restLen);

            restLen = 0;
        }
    }

    return 0;
}
```

调用MGF函数

```
    MGF1(seed,32,223,DB_mask);

    for(int i = 33;i<256; i++){
        m[i] = m[i] ^ DB_mask[i-33];
    }
    MGF1(m+33,223,32,seed_mask);
    for(int i = 1;i<33;i++){
        m[i] = m[i] ^ seed_mask[i-1];
    }
```

加密计算

将m转化为`uint32_t[128]`表示，计算得到最终加密结果， $c = M^E \mod N$ 输出.

```
    for(int i = 0;i < 64;i++){
        uint32_t   value = ((uint32_t)m[4*i+0] << 24) |
            ((uint32_t)m[4*i+1] << 16) |
            ((uint32_t)m[4*i+2] << 8)  |
            ((uint32_t)m[4*i+3]);
        M[64-i-1]=value;

    }
    // printf("M=");
    // cout<<cal.biToStr(M)<<endl;


    // fclose(file);
    uint32_t res[128] = {0};
    cal.modPow(res, M, E);
    // printf("res=");
    // cout<<cal.biToStr(res)<<endl;

    //以小端序输出
    for(int i = 0;i<64;i++){
        uint32_t   value = res[64-i-1];
        uint8_t bytes[4];
        bytes[3] = value & 0xFF;
        bytes[2] = (value >> 8) & 0xFF;
        bytes[1] = (value >> 16) & 0xFF;
        bytes[0] = (value >> 24) & 0xFF;
        fwrite(bytes, 1, 4, outputFile);
    }
```

关于SHA-256和大数运算部分已在之前实验中实现，这里不再赘述，大数运算部分参考了https://smallorange666.github.io/2024/11/29/%E5%A4%A7%E6%95%B0%E8%BF%90%E7%AE%97%E5%AE%9E%E7%8E%B0/。

实验总结

实验过程中，我成功实现了RSA-2048的加密，使用了SHA-256作为哈希函数，并遵循了OAEP填充方案。通过这次实验，我深刻理解了RSA算法的原理和操作步骤。

2048 位的 RSA 密码体制通过两个 1024 位的安全素数，构造一个 2048 位的大整数，并在模这个大整数的基础下进行操作。需要将可变长度的明文字符串转成对应的 2048 位的大整数，方便 RSA 密码体制的运行。这就需要用到上一个实验中实现的 SHA-256 函数，

在实验中碰到的问题:调用完一次sha256-update后再调用没有进行init导致错误，经过修改后解决了。