密码学第七次实验报告

# RSA公钥密码体制

## 原理

RSA公钥密码体制是现行的一种公钥密码体制，主要基于求离散对数和大整数因子分解的困难性。它原理比较简单，实现相对方便，是基于模幂的原理。

## 伪代码

### 素数生成算法

static BigNum genPrime (int bitlength) {

do {

生成0到(1 << bitlength)之间的随机数;

} while (生成的随机数经过素性测试算法检验不通过);

return 生成的随机数;

}

### 生成公私钥对算法

static BigNum[][] genKeyPair(BigNum p, BigNum q) {

计算出n, phi(n), d

令 e = 65537

以 {d, n, p, q} 作为私钥，{e, n} 作为公钥

return 生成的公私钥对;

}

### 加密算法

static BigNum encrypt (BigNum plainText, BigNum[] publicKey) {

return (mod);

}

### 解密算法

static BigNum decrypt (BigNum cipherText, BigNum[] privateKey) {

return 方程组的解;

}

### OEAP加解密算法

按照文档中7.1.1和7.1.2节进行。

### 文件加密算法

static void encryptFile (Path plainTextPath, Path cipherTextPath, Path publicKeyPath) {

从publicKeyPath对应的文件中获得公钥;

令每个密文分组的长度为n所占的字节长度;

令每个明文分组的长度为n所占的字节长度-1;

读plainTextPath对应的文件，对每个分组加密，并写入cipherTextPath对应的文件中去,若最后一组长度不够，则后面用0字节填充;

}

### 文件解密算法

static void decryptFile (Path plainTextPath, Path cipherTextPath, Path privateKeyPath) {

从privateKeyPath对应的文件中获得私钥;

令每个密文分组的长度为n所占的字节长度;

令每个明文分组的长度为n所占的字节长度-1;

读cipherTextPath对应的文件，对每个分组解密，并写入cipherTextPath对应的文件中去,若最后一组长度不够，则后面用0字节填充;

}

### OAEP文件加解密算法

与普通文件加解密算法一样，不同的是每个明文分组长度固定，都是64个字节，而且用相应的OAEP算法加解密。

## 分析

### 素数生成算法和生成公私钥对算法

素数分布本身是有规律的，但是因为本次实验生成素数的范围确定，而且素数生成带有概率性，因此需要的时空复杂度都是。

### 加解密类算法

由于加解密算法数据规模确定，因此需要的时空复杂度都是。

### 文件加解密类算法

由于文件加解密类算法需要分块，所以时间复杂度与文件长度有关。每次加解密之间的数据中，有关联的都是常数级别大小。因此，时间复杂度是，空间复杂度是。

# 背包公钥密码算法

## 原理

背包公钥密码算法是最早产生的公钥密码算法，是根据背包问题是NP难问题的理论提出来的。

它的主要原理是生成一个超递增的背包，选择一个模数和与其互质的数。要比背包中所有的数之和大。由于超递增背包问题可以直接用贪心算法解决，而贪心算法在这里能以多项式时间结束，所以这不算一个困难问题。但是作为公钥的普通背包，是超递增背包中每个数乘以再模构成的新背包。这种在模意义下的背包问题，是一个NP难问题。可以看出，私钥提供了解决NP难问题的陷门，这里解决背包问题的算法可以当成一个陷门函数。加密时，可以加密位长度与背包长度相同的数据。只需要把每一位的数值乘以公钥，然后加起来即可。解密时把密文乘以模意义下的逆，再解决一个超递增背包问题。攻击者只知道公钥和密文，由于模意义下的背包问题是NP难问题，难以求出明文或私钥。这样，就建立了一个公钥密码体制。

背包公钥密码体制的破解，是通过LLL格约简算法进行的。它的主要目的，是找到向量，也就是相应长度的比特串的向量形式，使得它与公钥相乘，仍然能够得到密文。这样就能破解出明文。通过把密文和公钥适当排列成矩阵，再经过LLL算法，就相对有可能得到明文。LLL格约简算法，是一个多项式时间算法。所以，背包公钥密码体制可以说被成功破解了。

## 伪代码

### 密钥生成算法

**def** gen\_key(length: int) -> Tuple[Tuple[List[int]], int, int]:

检查长度

随机整数

对内每个:

随机整数

随机生成的与互素的整数

对内每个:

**return** (w, q, r), pk

### 加密算法

**def** encrypt(m: BitString, pk: List[int]) -> int:

**return**

### 解密算法

**def** decrypt(c: int, sk: Tuple[List[int], int, int]) -> BitString:

从中提取各, ,

:

解决相应的超递增背包问题

**return** 表示背包问题解法的比特串

### Gram-Schmidt算法

**def** \_gram\_schmidt(v: List[\_LLLVector]) -> List[\_LLLVector]:

u = []

对中各个向量:

的一个副本

对中各个向量:

在上的投影向量

**if** 不是零向量:

u.append()

**return** u

### LLL格约简算法

**def** lll(m: List[List[int]], delta: float) -> List[List[int]]:  
 n = m的行数

orthodox\_basis = m正交化后的结果

k = 1

**while** k < n:

**for** j **in** range(k – 1, -1, -1):

mu\_k\_j = 索引为的正交基在m的索引为的向量上的投影

**if** :

**if** 索引为的正交基的长度\*索引为的正交基的长度:

k += 1

**else**:

交换和

重新生成一遍orthodox\_basis

k = max(k – 1, 1)

**return** m

### 密码分析算法

**def** cryptanalysis\_lll(c: int, pk: List[int]) -> BitString:

delta = 0.75

n = pk的行数

用LLL算法约简

**if** 中有最后一个元素为0，其他元素不是0就是1的行向量:

**return** 相应的比特串

**else**:

**raise** 没有找到结果

## 分析

设密钥长度为。

### 密钥生成算法

这里密钥生成主要是生成随机数，并且把它们加起来。显然密钥生成需要的时间以及空间，都是和需要的密钥长度有关的。这里的“长度”指的是的个数。算法的空间复杂度是。真正计算时，需要随机找出与互素的，这里是一个随机化算法，但是最坏的时间复杂度是，因为找随机数的最差情况是和一个量级，每次判断互质又是的时间复杂度。所以算法的时间复杂度是。

### 加密算法

　　这里加密也是把相应的公钥中的数加起来和乘起来。其中计算时可以知道，不需要中间结果的储存，所以时间复杂度是，空间复杂度是。

### 解密算法

解密算法是先乘一遍模的逆元，然后再解密。这时需要求解一个超递增背包问题，需要的时间复杂度是。

### Gram-Schmidt算法

这个算法相对典型，对每个向量都要判断一遍。根据后面密码分析的步骤可以看到，密钥长度和要化简的向量个数一样。中间需要有的空间来储存正交化后的向量，所以空间复杂度是。对每个向量都要判断一遍，时间复杂度也是。

### LLL格约简算法

这个算法相对比较复杂，从维基百科上查到的时间复杂度是，在算法中是固定的，是向量个数，是各向量中最长的那个的长度。这样，显然。因此算法的时间复杂度为。算法的空间复杂度，可以通过中间变量的使用看出为。

### 密码分析算法

密码分析算法中用到了LLL格约简算法，但是也有把数据填入矩阵，以及判断向量的操作。容易看出，其中的时间复杂度还是，因为LLL格约简算法本身的时间复杂度就太大了。空间复杂度是，因为有保存矩阵的开销。

## 优化

### 密钥生成算法

其实密钥生成算法可以并行化，只要中间提前生成一个数，保证最后是超递增序列即可。但是这样意义不大，因为密钥生成是只做一次的事情。

### 加密算法

加解密算法其实也可以并行化，但是类似地，意义也不大。

### 解密算法

解密算法实际上就是一个解超递增背包问题的算法。其实优化的空间似乎不是很大。

### Gram-Schmidt算法和LLL格约简算法

实际上，Gram-Schmidt算法可以通过矩阵的方式进行，这样可能可以用得上SIMD类指令，或者其它向量指令集。LLL格约简算法在正交化的时候，也可以只更新相应的向量，不更新其他向量。

### 密码分析算法

实际上，这个算法可以提前缓存或者生成矩阵的一个模板，这样更方便使用，也能提高内存访问效率，因为通常的计算机体系结构会对内存的连续访问做优化。

## 测试

　　采用自动化测试工具进行测试。关于LLL算法的测试在lll\_test.py中，关于背包密码的测试在knapsack\_test.py中。运行测试，可以得到算法的结果正确，加解密也是可逆的。

# 总结

## 大整数操作

这次我用Java重写了一遍大整数操作，包含一些数论操作。大整数主要的特点是比较注重效率和细节。

## RSA相关算法

这次RSA相关算法数学上是比较简单的，但是关于大整数的细节确实有些卡人。我花了两个小时调一个bug，结果发现是大整数算法中的一个函数写错了。

## OAEP相关算法

OAEP相关算法其实是比较严格的，但是比较难以调试。我花了不少时间才把算法调试到正常水平。

## 背包密码相关算法

感觉其实背包密码相关算法数学性比较强，实际上就是考验对编程语言的熟练应用。但是破解还是挺有意思的，因为它用到了LLL算法。

## 算法评估与优化

这次算法评估与优化主要是在大整数算法上。我在大整数算法中使用了一定的优化技巧，比如乘法首先判断最高位，这使得乘法的复杂度大大降低。

## 系统设计与维护

这次良好的系统设计与维护起到了很好的作用。由于我实现大整数相关算法的时候考虑了模块化和系统设计，这使得我到后面执行复杂操作的时候越来越不需要考虑下层的需要，在最后实现OAEP的时候，也能比较快地发现bug。实现背包密码的LLL算法的时候，也是先实现向量，再实现矩阵，最后实现LLL算法。结果实现的速度也比较快，调试也容易很多。这也能体现良好系统设计的加速型回报。

## 对课程的建议

感觉以后可以把细节抽象出来，不要让我们写大整数运算了。我感觉大整数运算相对密码学可能有一点点细节。所以，我们经过一番细节的洗礼，不仅没法独立思考，更没法对这个课程有更深的兴趣。

## 总结

这次实验算一次从头重写大整数库的实验，RSA本身倒不难了。背包密码其实是建立在一个比较精密的理论基础上的，但是破解的理论也很精密，这一点值得思考。