

密码学

平时实验报告

（ 2020 / 2021学年 第 二 学期）

题 目： RSA加密算法

|  |  |
| --- | --- |
| **专 业** | 信息安全 |
| **组 长 姓 名** | 任远哲 |
| **组 员 姓 名** |  |
| **班 级 学 号** | B190307,B19031614， |
| **指 导 教 师** | 李琦 |
| **指 导 单 位** | 信息安全系 |
| **日 期** | 2021年5月28日 |

|  |  |
| --- | --- |
| **成员分工** | |
| 组长（1） | 任远哲，B19031614，编写代码，撰写报告 |
| **简短评语：**  **指导教师：年月日** | |

1. **课题内容和要求**

**1.实验环境**

实验主机操作系统为Windows 7

**2.实验内容**

1.给定p，q，e，编写RSA的加解密算法

2.调研各个语言的加密算法包

1. **课题需求分析**

RSA算法的具体描述如下：

（1）任意选取两个不同的大素数p和q计算乘积n = p×q，φ(n) = (p-1)×(q-1)。

（2）任意选取一个大整数e，满足 ，整数e用做加密钥（注意：e的选取是很容易的，例如，所有大于p和q的素数都可用） ；

（3）确定的解密钥d，满足d\*e ≡ 1mod φ(n)，d为e的乘法逆元

（4）公开整数n和e，秘密保存d ；

（5）将明文m（m<n是一个整数）加密成密文c，加密算法为

C = M^e (mod n)

（6）将密文c解密为明文m，解密算法为

M = C^d (mod n)

然而只根据n和e（注意：不是p和q）要计算出d是不可能的。因此，任何人都可对明文进行加密，但只有授权用户（知道d）才可对密文解密。

具体的，求逆元采用扩展欧几里德算法和费马小定理+快速幂取模算法结合。（后者要求模逆元的模为素数，这里φ(n) = (p-1)×(q-1)不适用，但我还是加上了）。判断是否为质数采用了埃氏筛算法。

1.所谓扩展欧几里德算法，就在求gcd（a,b）的同时，顺带着求出x，y使贝祖等式ax+by= gcd（a,b）成立。在求模逆元a\*x=1 modb时，将原式化为ax+by=1= gcd（a,b）。运用扩展欧几里德算法即可求出a的模b逆元x。

2.所谓费马小定理/欧拉定理求逆元，就是

费马小定理：若p为素数，则有ap−1≡1(modp)ap−1≡1(modp)

ap−2∗a≡1(modp)ap−2∗a≡1(modp)

ap−2ap−2就是a在mod p意义下的逆元啦。

欧拉定理：若a、p互素，则有aφ(p)≡1(modp)aφ(p)≡1(modp)(费马小定理的一般形式)

aφ(p)∗a≡1(modp)aφ(p)∗a≡1(modp)

aφ(p)−1aφ(p)−1就是a在mod p意义下的逆元啦。

3. 快速幂取模就是在O(logn)内求出a^n mod b的值。算法的原理是ab mod c=(a mod c)(b mod c)mod c 即 (a^b) mod c = (a mod c)^b mod c。

**完整代码和注释：**

#include<cstring>

#include<iostream>

using namespace std;

typedef long long LL;

int p=11, q=23, e=7;

bool Is\_prime(int n)//普通判断素数

{

for(int i=1;i\*i<=n;++i)

{

if(n%i==0) return false;

}

return true;

}

LL exgcd(LL a,LL b,LL &x,LL &y)//扩展欧几里得算法 ax+by=gcd(a,b) 函数返回值ret为gcd(a,b)

{

if(b==0)

{

x=1,y=0;

return a;

}

LL ret=exgcd(b,a%b,y,x);

y-=a/b\*x;

return ret;

}

LL qkpow(LL a,LL p,LL mod) //求a^p %mod 思路是把p看成成二进制形式 ，(a^b) mod c = (a mod c)^b mod c

{

LL ans=1,base=a%mod;

while(p)

{

if(p&1) ans=ans\*base%mod;//如果 p最后一位是1

base=(base\*base)%mod; //每一次移位都要平方

p>>=1;

}

return ans;

}

LL getInv(int a,int mod)//求a在mod下的逆元，不存在逆元返回-1

{ if(Is\_prime(mod)) //如果mod为素数，用费马小定理+快速模指数算法求逆元

{return qkpow(a,mod-2,mod);

}

LL x,y;

LL d=exgcd(a,mod,x,y);

return d==1?(x%mod+mod)%mod:-1;

}

int main(){

// 计算 n = p×q，φ(n) = (p-1)×(q-1)

// 计算 d，满足 d\*e ≡ 1mod φ(n)

// 得到：公钥为 {e,n}, 私钥为 {d,n}

// 密文：C = M^e (mod n)

//明文：M = C^d (mod n)

int n = p\*q;

int d =getInv(e,(p-1)\*(q-1));

int plaintext=30;// plaintext < n

cout<<"明文为"<<plaintext<<endl;

int cipher=qkpow(plaintext,e,n);

cout<<"密文为"<<cipher<<endl;

int decode\_cipher=qkpow(cipher,d,n);

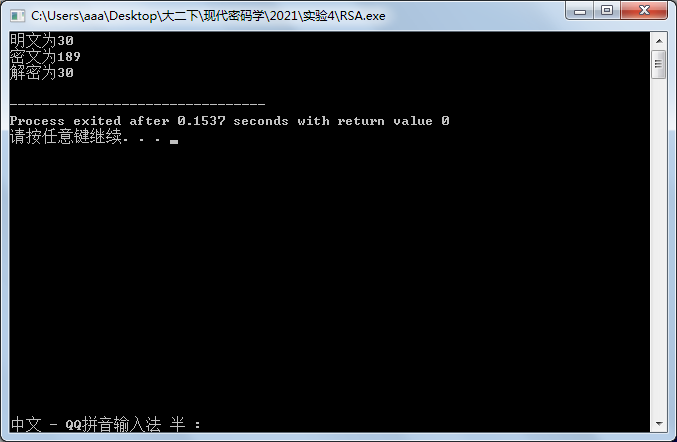
cout<<"解密为"<<decode\_cipher<<endl;

return 0;

}

**三.测试数据及其结果分析**

选择RSA参数：p=11, q=23, e=7

****

**四、课题完成过程中遇到的问题及解决方法**

问题：加解密失败。

解决方法：仔细检查算法规则，发现明文m的大小必须小于n。选择更大的素数p，q，解决问题。

**五、总结**

RSA的安全性依赖于大数分解。只要知道n的两个素因子p，q，就能根据φ(n) = (p-1)×(q-1)，d\*e ≡ 1mod φ(n)快速计算出私钥d。但是否等同于大数分解一直未能得到理论上的证明，也并没有从理论上证明破译。RSA的难度与大数分解难度等价。因为没有证明破解RSA就一定需要做大数分解。假设存在一种无须分解大数的算法，那它肯定可以修改成为大数分解算法，即RSA的重大缺陷是无法从理论上把握它的保密性能如何。

由于进行的都是大数计算，使得RSA最快的情况也比DES慢上好几倍，无论是软件还是硬件实现。速度一直是RSA的缺陷。一般来说只用于少量数据加密。RSA的速度比对应同样安全级别的对称密码算法要慢1000倍左右。

RSA在选择密码攻击面前显得很脆弱。

攻击者C欲破解B发给A的密文c；

C选择 r < n，计算 t = r-1 mod n；

x = re mod n, y = xc mod n；

C把y发送给A，要求数字签名；

A对y进行数字签名 u = yd mod n；

计算 tu = r^-1y^d = r^-1(x \* c)^d = r^-1 r m = m mod n，得到m的值。

一般攻击者是将某一信息进行下伪装，让拥有私钥的实体签名；然后，经过计算就可得到它所想要的信息。实际上，攻击利用的都是同一个弱点，即存在这样一个事实：乘幂保留了输入的乘法结构。前面已经提到，这个固有的问题来自于公钥密码系统的最基本的特征，即每个人都能使用公钥加密信息。从算法上无法解决这一问题，改进措施有两条：是采用好的公钥协议保证工作过程中实体不对其他实体任意产生的信息解密，不对自己一无所知的信息签名；二是决不对陌生人送来的随机文档签名，或签名时首先对文档作Hash处理，或同时使用不同的签名算法。

**六．补充python加密包的用法**

**1.AES加密解密**

from Crypto.Cipher import AES

from Crypto import Random

import binascii

key = '1234567890!@#$%^' #秘钥，必须是16、24或32字节长度

iv = Random.new().read(16) #随机向量，必须是16字节长度

cipher1 = AES.new(key,AES.MODE\_CFB,iv) #密文生成器,MODE\_CFB为加密模式

encrypt\_msg = iv + cipher1.encrypt('我是明文') #附加上iv值是为了在解密时找到在加密时用到的随机iv

print '加密后的值为：',binascii.b2a\_hex(encrypt\_msg) #将二进制密文转换为16机制显示

cipher2 = AES.new(key,AES.MODE\_CFB,iv) #解密时必须重新创建新的密文生成器

decrypt\_msg = cipher2.decrypt(encrypt\_msg[16:]) #后十六位是真正的密文

print '解密后的值为：',decrypt\_msg.decode('utf-8')

**2.DES加密解密**

from Crypto.Cipher import DES3

from Crypto import Random

import binascii

key = '1234567890!@#$%^'

iv = Random.new().read(8) #iv值必须是8位

cipher1 = DES3.new(key,DES3.MODE\_OFB,iv) #密文生成器，采用MODE\_OFB加密模式

encrypt\_msg = iv + cipher1.encrypt('我是明文必须是八')

#附加上iv值是为了在解密时找到在加密时用到的随机iv,加密的密文必须是八字节的整数倍，最后部分

#不足八字节的，需要补位

print '加密后的值为：',binascii.b2a\_hex(encrypt\_msg) #将二进制密文转换为16进制显示

cipher2 = DES3.new(key,DES3.MODE\_OFB,iv) #解密时必须重新创建新的密文生成器

decrypt\_msg = cipher2.decrypt(encrypt\_msg[8:]) #后八位是真正的密文

print '解密后的值为：',decrypt\_msg

**3.RSA加密解密**

import rsa

# rsa加密

def rsaEncrypt(str):

# 生成公钥、私钥

(pubkey, privkey) = rsa.newkeys(512)

print("公钥:\n%s\n私钥:\n%s" % (pubkey, privkey))

# 明文编码格式

content = str.encode("utf-8")

# 公钥加密

crypto = rsa.encrypt(content, pubkey)

return (crypto, privkey)

# rsa解密

def rsaDecrypt(str, pk):

# 私钥解密

content = rsa.decrypt(str, pk)

con = content.decode("utf-8")

return con

if \_\_name\_\_ == "\_\_main\_\_":

str, pk = rsaEncrypt("password")

print("加密后密文：\n%s" % str)

content = rsaDecrypt(str, pk)

print("解密后明文：\n%s" % content)

**4.哈希加密**

hash = hashlib.md5() #创建md5()加密实例

hash.update(bytes('admin', encoding='utf-8')) #对admin字符进行加密

print(hash.hexdigest()) #返回产生的十六进制的bytes

print(hash.digest())