编号：_____


# 2019-2020 学 年

## 结课报告合集

学　　院：计算机与通信工程学院

专　　业：计算机科学与技术

班　　级：计 184

姓　　名：王丹琳

学　　号：41824179

指 导 老 师：王卫苹

学　　时：

装 订 时 间：2020 年 12 月 28 日

编号：_____

# RSA 基于身份的数字签名

【摘要】随着 Internet 的发展和普及，特别是电子商务的出现，使得数字签名得到越来越广泛的应用。本文介绍了数字签名背景，对如何用 RSA 实现数字签名的生成、验证算法进行了简单实验。

【关键词】RSA 算法，数字签名

## 一、概述

随着网络技术的飞速发展,信息安全性已成为亟待解决的问题。公钥密码体制中,解密和加密密钥不同,解密和加密可分离,通信双方无须事先交换密钥就可建立起保密通信,较好地解决了传统密码体制在网络通信中出现的问题。另外,随着电子商务的发展,网络上资金的电子交换日益频繁,如何防止信息的伪造和欺骗也成为非常重要的问题。数字签名可以起到身份认证,核准数据完整性的作用。目前关于数字签名的研究主要集中基于公钥密码体制的数字签名。

公钥密码体制的特点是:为每个用户产生一对密钥(PK 和 SK);PK 公开,SK 保密;从 PK 推出 SK 是很困难的;A,B 双方通信时,A 通过任何途径取得 B 的公钥,用 B 的公钥加密信息。加密后的信息可通过任何不安全信道发送。B 收到密文信息后,用自己私钥解密恢复出明文。

由于计算机网络通信和各类应用业务的需求，要求相应的数据文件加密方法应具有灵活性和快速的特点。

RSA 加密体制是 1978 年提出来的，它是第一个理论上最为成功的公开密钥加密体制，它的安全性基于数论中的 Euler 定理和计算复杂性理论中的下述论断：求两个大素数的乘积是很容易计算的，但要分解两个大素数的乘积，求出它们的素数因子却是非常困难的，它属于 NP 完全类，是一种幂模运算的加密体制。除了用于数据文件加密外，它还能用于数字签字和身份认证。要构造 RSA 加密算法和解密算法中的公钥和私钥，必须首先构造两个大素数，RSA 加密算法和解密算法的安全性与所使用的大素数有密切关系。因此。研究 RSA 公钥加密体制中的大素数生成，构造符合 RSA 安全体系要求的强素数，是 RSA 加密算法算法实用化的基础。

编号：_____

身份认证可以防止非法人员进入系统，防止非法人员通过违法操作获取不正当利益、访问受控信息和恶意破坏系统数据完整性的情况发生。同时，在一些需要具有较高安全性的系统中，通过用户身份的唯一性，系统可以自动记录用户所做的操作，进行有效的核查。在一个有竞争和争斗的现实社会中，身份欺诈是不可避免的，因此常常需要证明个人的身份。通信和数据系统的安全性也取决于能否正确验证用户或终端的个人身份。对于计算机的访问和使用、安全地区的出入也都是以精确的身份验证为基础的。网络中的各种应用和计算机系统都需要身份认证来确认一个用户的合法性，然后确定这个用户的个人数据和特定权限。

二、算法构成

一个基于身份的数字签名算法由以下 4 个子算法构成：

1. 初始化

输入：安全参数 k

输出：系统参数 params 和主密钥 master-key

2. 用户私钥生成

输入：系统参数 params、主密钥 master-key 和用户身份 ID

输出：用户私钥 gID

3. 签名

输入：系统参数 params，消息 M 和签名者的私钥 gID

输出：签名 $\sigma$

4. 验证

输入：系统参数 params、签名 $\sigma$ 、签名者公钥 ID 和消息 M

输出：验签结果 Accept or Reject

三、具体实现：

1. 选取两个大素数 p，q，计算它们的乘积 n

编号：_____

```java
public BigInteger[] getPrimeNumber(){
    BigInteger p=null;
    BigInteger q=null;
    BigInteger[] res=new BigInteger[2];
    Random random = new Random();
    p=BigInteger.probablePrime(64, random);//为了效率，此处设为64
    Random random1 = new Random();
    q=BigInteger.probablePrime(64, random1);
    //p=new BigInteger("473398607161");
    //q=new BigInteger("4511491");
    res[0]=p;
    res[1]=q;
    return res;
}
```

图 1-1 使用 BigInteger 下的 probablePrime 函数，随机生成两个大素数

```java
BigInteger n=p.multiply(q);
```

图 1-2 计算 n

```java
BigInteger sn=(p.subtract(new BigInteger("1"))).multiply(q.subtract(new BigInteger("1")));
```

图 1-3 计算 $\Phi(n)$

**2.** 选取与 $\Phi(n)$互素的整数 e，计算主私钥 d 满足 e*d=1 **mod** $\boldsymbol{\Phi(n)}$

```java
/**
 * 随机选取e
 * 0<e<sn && e和 sn互素
 * @param sn
 * @return
 */
public BigInteger getE(BigInteger sn){
    BigInteger e = null;
    //说明:此处把产生的e位数-2，是防止 nextProbablePrime()方法产生的素数大于sn
    int length = sn.toString().length()-2;// length为随机数位数
    e=new BigInteger(sn.toString().subSequence(0, length-2).toString()).nextProbablePrime();
    return e;
}
```

图 2-1 生成与 $\Phi(n)$互素的整数 e

编号：_____

```java
/**
 * 选取d
 * d同时与n和sn互素
 * @param n
 * @param sn
 * @return
 */
public BigInteger getD(BigInteger sn,BigInteger e){//Euclid算法
    BigInteger[] ret = new BigInteger[3];
    BigInteger u = BigInteger.valueOf(1), u1 = BigInteger.valueOf(0);
    BigInteger v = BigInteger.valueOf(0), v1 = BigInteger.valueOf(1);
    if (e.compareTo(sn) > 0) {
        BigInteger tem = sn;
        sn = e;
        e = tem;
    }
    while (e.compareTo(BigInteger.valueOf(0)) != 0) {
        BigInteger tq = sn.divide(e); // tq = sn / e
        BigInteger tu = u;
        u = u1;
        u1 = tu.subtract(tq.multiply(u1)); // u1 =tu - tq * u1
        BigInteger tv = v;
        v = v1;
        v1 = tv.subtract(tq.multiply(v1)); // v1 = tv - tq * v1
        BigInteger tsn = sn;
        sn = e;
        e = tsn.subtract(tq.multiply(e)); // e = tsn - tq * e
        ret[0] = u;
        ret[1] = v;
        ret[2] = sn;
    }
    return ret[1];
}
```

图 2-2 计算 d

**3.** 选取一个单向函数 h，这里选用的是 SHA256

```java
public static String getSHA256(String str){
    MessageDigest messageDigest;
    String encodeStr = "";
    try {
        messageDigest = MessageDigest.getInstance("SHA-256");
        messageDigest.update(str.getBytes("UTF-8"));
        encodeStr = bytesToHexString(messageDigest.digest());
    } catch (NoSuchAlgorithmException|UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    return encodeStr;
}
```

图 3-1 哈希函数实现

**4.** 输出系统参数 n、e、h，保存主密钥 n、d

```
1、产生的两个大素数是(保密)：p=14800987080335929591 q=15843226159191059593
2、计算的n是：234495385693027104771280611979933116463
  计算的Φ(n)是：234495385693027104740636398740406127280
3、选取的e是：23449538569302710474063639874040669
4、计算的d是：476104142572043640597854158240157457 49
5、公钥：n=234495385693027104771280611979933116463 e=23449538569302710474063639874040669
```

图 4-1 上述步骤计算生成的参数，为了方便查看全部输出

**5.** 输入用户 ID，计算用户私钥 gID=h(ID)$^d$ **mod n**

```
Scanner scanner=new Scanner(System.in);
String ID = scanner.nextLine();
BigInteger b = new BigInteger(SHAUtil.getSHA256(ID), 16);   //SHA256哈希
BigInteger gID = b.modPow(d, n);
System.out.println("gID:"+gID);
```

图 5-1 获取输入的用户 ID，计算 gID

```
7、请输入用户ID
123
gID:20916729765260524308666390719638647478O
```

图 5-2 输入输出结果（ID、gID）

**6.** 选取随机整数 r，计算 t=r$^e$ **mod n**

```
Random random = new Random();
BigInteger r = BigInteger.valueOf(random.nextInt(500000000000)+1);
```

图 6-1 随机生成 r

```
BigInteger t= r.modPow(e, n);
```

图 6-2 模幂计算 t

**7.** 计算 s=gID*r$^{h(t,m)}$ **mod n**，输出签名 $\sigma$ =<s,t>

```
BigInteger tm = new BigInteger(SHAUtil.getSHA256(t.toString()+ m),16);
```

图 7-1 哈希函数计算 h(t,m)

```
BigInteger s = gID.multiply(r.modPow(tm, n)).mod(n);
```

图 7-2 计算 s

```
9、输出签名 s:70538601829326393070767679845250394054  t:708736202640410663226858475586412834O1
```

图 7-3 输出的签名结果

**8.** 验证签名，if(s*e==h(ID)*t$^{h(t,m)}$ **mod n**) 验签成功输出 Accept，否则输出 Reject

$$
\begin{aligned}
S^e &= (gID \times r^{h(t,m)})^e \bmod n \\
&= (h(ID)^d \times r^{h(t,m)})^e \bmod n \\
&= h(ID)^{de} \times r^{eh(t,m)} \bmod n \\
\because\ & de = 1 \bmod n,\ r^e = t \bmod n \\
\therefore\ & S^e = h(ID) \times r^{h(t,m)}
\end{aligned}
$$

图 8-1 签名验证算法

```
BigInteger left = s.modPow(e, n);
BigInteger right = b.multiply(t.modPow(tm, n)).mod(n);
verify(left,right);
```

```
/**
 * 验签
 * @param l s^e mod n   //s=gIDx"r"^h(t, m) mod n
 * @param r h(ID)*t^h(t,m) mod n
 */
public static void verify(BigInteger l, BigInteger r) {
    int tem = l.compareTo(r);
    if(tem == 0)
        System.out.println("Accept");
    else
        System.out.println("Reject");
}
```

图 8-1&2 验证签名

```
9、输出签名 s:70538601829326393070767679845250394054  t:708736202640410663226858475586412834 21
Accept
```

图 8-3 验签结果

9. 使用本类计算好的值进行验签显得过于主动，之后编写了一个方法类，使用公开的参数进行计算验证。

```
public class yanqian {
    public static void main(String[] args) {

    }
    public void ver(BigInteger s, BigInteger n, BigInteger e, BigInteger t, BigInteger tm, String ID) {

        BigInteger hID = new BigInteger(SHAUtil.getSHA256(ID), 16);

        if(     (s.modPow(e, n).mod(n))   .compareTo(        hID.multiply(t.modPow(tm, n)).mod(n)     )    == 0      )
            System.out.println("Accept");
        else
            System.out.println("Reject");

    }
}
```

图 9-1 验签类方法实现

```
BigInteger left = s.modPow(e, n);
BigInteger right = b.multiply(t.modPow(tm, n)).mod(n);
verify(left,right);
//System.out.println(left);
//System.out.println(right);
yanqian y = new yanqian();
y.ver(s, n, e, t, tm, ID);
```

```
9、输出签名 s:70538601829326393070767679845250394054  t:708736202640410663226858475586412834 21
Accept
Accept
```

图 9-2 双重验签结果

对于效率问题，签名和验证需要模幂（Modular Exponentiations），模乘

（Modular Multiplication）和哈希运算，经测试对于百位以下的 ID 与密文在 Eclipse 也可以瞬间验签。

对于安全性，大数分解因子问题困难（IFP）使私钥安全性高，陷门函数逆向计算困难使信息保密性好。

## 四、总结

最后，RSA 算法是一种安全技术,但是 RSA 算法的安全性只是一种计算安全性,绝不是无条件的安全性,这是由它的理论基础决定的。因此,在实现 RSA 算法的过程中,每一步都应尽量从安全性考虑。

## 参考文献

[1]姚思帆.非对称加密技术分析[J].电子技术,2020,49(07):50-51.

[2]宋颖杰.非对称加密技术[J].信息网络安全,2004(01):48-49.