

RSA实验报告

指导教师：王娜

学生信息：于珈尉-信息安全-21371048

报告目录

- [报告目录](#)
- [*1 实验基本原理*](#)
 - [1.1 RSA密码体系](#)
 - [1.1.1 加解密原理](#)
 - [1.1.2 数字签名原理](#)
 - [1.2 算法加速原理](#)
 - [1.2.1 利用 \$p\$ 、 \$q\$ 加速解密](#)
 - [1.2.2 等效模数](#)
 - [1.2.3 数学加速](#)
- [*2 实验结果与完整代码解读*](#)
 - [2.1 实验结果截图](#)
 - [2.2 核心模块](#)
 - [2.2.1 密钥与参数生成模块](#)
 - [2.2.2 加密与解密模块](#)
 - [2.2.3 签名与验签模块](#)
 - [2.3 基础模块](#)
 - [2.3.1 扩展欧几里得算法](#)
 - [2.3.2 快速模幂算法](#)
 - [2.3.3 素性检测算法](#)
 - [2.3.4 中国剩余定理](#)
 - [2.3.5 获取新的安全素数](#)
- [*3 思考：RSA参数选择与安全性优化*](#)

- [3.1 RSA参数选择](#)
 - [3.1.1 模数不能共用](#)
 - [3.1.2 p和q差值不能太小](#)
 - [3.1.3 私钥d过小 \(e过大\)](#)
 - [3.1.4 p-1和q-1都应该有很大的素因子.](#)
 - [3.1.5 公钥e不可以太小](#)
 - [3.1.6 避免特殊情况密文和明文相似度过高](#)
 - [3.2 大作业心得体会](#)
 - [*4 针对RSA安全漏洞的攻击实验*](#)
 - [4.1 Level-1](#)
 - [4.1.1 实验原理](#)
 - [4.1.2 实验结果与完整代码](#)
 - [4.2 Level-2](#)
 - [4.2.1 实验原理](#)
 - [4.2.2 实验结果与完整代码](#)
 - [4.3 Level-3](#)
 - [4.3.1 实验原理](#)
 - [4.3.2 实验结果与完整代码](#)
-

1 实验基本原理

RSA是被广泛使用的一种非对称密码体系，其安全性建立在大素数的难分解性上。本实验报告探究学习RSA基本实现原理，并在此基础上探究其实现、应用、安全漏洞等扩展内容。

1.1 RSA密码体系

RSA公钥体系基于下面的事实：

对于大素数 N ,

如果其可被分解为两素数(记为素数 p 、 q)；并且存在 e 和 $\phi(N)$ 互素（记， d 是 e 关于

模 $\phi(N)$ 的逆元)。

那么 $x^e \equiv c \pmod{N}$ 有解 $x \equiv c^d \pmod{N}$

1.1.1 加解密原理

- 场景：A需要给B发送加密信息，但A似乎不方便保存密码。
- 加密通讯流程：
 1. B方生成密钥，在公共频道公开公钥对 (e, N) ，自己保留私钥 d
 2. A方接收公钥对 (e, N) ，对自己要发送给B的消息 msg 进行加密： $cipher \equiv msg^e \pmod{N}$ ，并发送给B
 3. B接收到密文 $cipher$ ，进行解密： $msg \equiv cipher^d \pmod{N}$

1.1.2 数字签名原理

- 场景：A需要给B发送消息，消息不怕泄露，但害怕被篡改。换言之，通过数字签名，可以防止消息被篡改，但消息本身对攻击者也是可见的。
- 签名通讯流程：
 1. A方生成密钥，在公共频道公开公钥 (e, N) ，自己保留私钥 d
 2. A方对消息 msg 进行加签，以防篡改： $signature \equiv msg^d \pmod{N}$ ，并将签名和消息本身 $(signature, msg)$ 都发送给B
 3. B通过公钥 e 进行验签，即检验 $signature$ 验签结果是否和 msg 一致，如果一致就说明 msg 未被篡改： $msg' \equiv signature^e \pmod{N}$ ， $check\ msg' == msg$

1.2 算法加速原理

1.2.1 利用p、q加速解密

利用已知信息 p 、 q 加速算法解密信息的过程。这是因为通常 d 相对 e 来说比较大，因此解密成本比加密成本高。

优化步骤：

1. 利用 $N=q \times p$ ，将解密等式化简为方程组

$$m \equiv c^d \pmod{N}$$

$$\Longleftrightarrow \begin{cases} m_1 \equiv c^d \pmod{p} \\ m_2 \equiv c^d \pmod{q} \end{cases}$$

$$\Longleftrightarrow \begin{cases} m_1 \equiv (c \pmod{p})^{d \pmod{p-1}} \pmod{p} \\ m_2 \equiv (c \pmod{q})^{d \pmod{q-1}} \pmod{q} \end{cases}$$

2. 利用中国剩余定理计算方程组

中国剩余定理应用见[2.2.2](#)

实现见[2.3.4](#)

1.2.2 等效模数

求解d时，可以利用等效的模数加速求逆。这基于以下的推论：

$$d * e \equiv 1 \pmod{(p-1) * (q-1)} \Longleftrightarrow d * e \equiv 1 \pmod{\frac{(p-1) * (q-1)}{g}}$$

其中， $g = \text{Gcd}(p-1, q-1)$

两者d等效，但比正常方法求得的d小，计算时速度更快。这也说明了，d不是对模数唯一的。

(当然，代价是密钥d安全性也下降了，所以当g过大时，需要重新选择合适的pq，防止d过小)

1.2.3 数学加速

1. 快速模幂算法

利用快速平方乘算法，可以提高计算大素数高次平方模数的速度，对加密和解密过程的速度有较大提升。详细实现见代码部分[2.3.2 快速模幂算法](#)。

2. 扩展欧几里得算法求最大公因子与逆元

利用扩展欧几里得算法，不仅可以快速计算出最大公因子，还可以通过同时返回矩阵更多参数来实现求逆元功能。

- 扩展欧几里得算法依赖于将下式运算过程矩阵化，保留更多可用信息用于扩展：

$$a = b + k * N \Longleftrightarrow a \equiv b \pmod{N} \Longleftrightarrow (a, N) = (b, N)$$

- 求逆元依赖于以下事实：

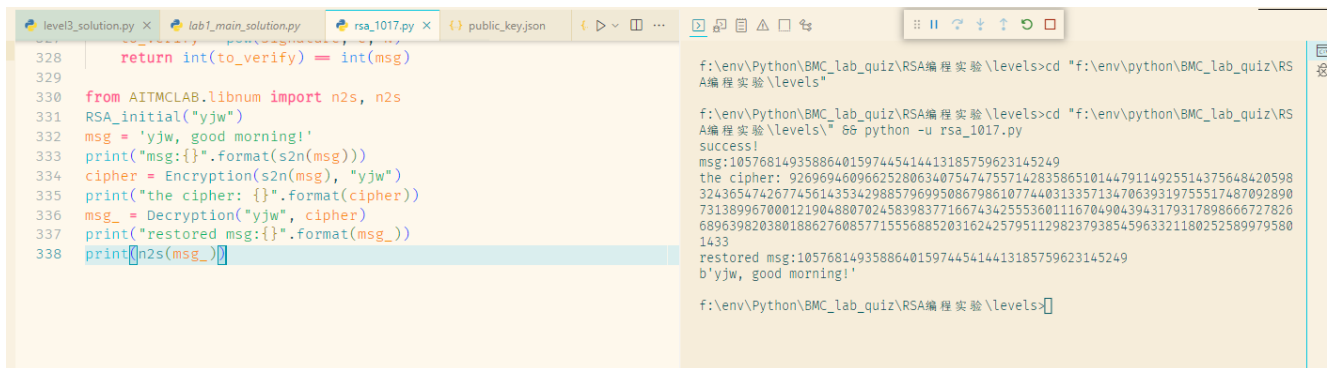
若 $\text{Gcd}(x, y)=1$ ，那么存在整数 a, b ，满足 $a * x + b * y = 1$ 。

变化该式： $a * x = 1 - b * y \Rightarrow a * x \equiv 1 \pmod{y} \Rightarrow a \equiv x^{-1} \pmod{y}$

2 实验结果与完整代码解读

实验各功能模块：

2.1 实验结果截图



```
level3_solution.py x lab1_main_solution.py rsa_1017.py x {} public_key.json
328     return int(to_verify) == int(msg)
329
330 from ITMCLAB.libnum import n2s, n2s
331 RSA_initial("yjwt")
332 msg = 'yjwt, good morning!'
333 print("msg:{}".format(s2n(msg)))
334 cipher = Encryption(s2n(msg), "yjwt")
335 print("the cipher: {}".format(cipher))
336 msg_ = Decryption("yjwt", cipher)
337 print("restored msg:{}".format(msg_))
338 print([n2s(msg_)])

f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>cd "f:\env\python\BMC_lab_quiz\RSA编程实验\levels"
f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>cd "f:\env\python\BMC_lab_quiz\RSA编程实验\levels" 66 python -u rsa_1017.py
success!
msg:10576814935886401597445414413185759623145249
the cipher: 9269694609662528063407547475571428358651014479114925514375648420598
3243654742677456143534298857969950867986107744031335713470639319755517487092890
7313899670001219048807024583983771667434255536011167049043943179317898666727826
6896398203801886276085771555688520316242579511298237938545963321180252589979580
1433
restored msg:10576814935886401597445414413185759623145249
b'yjwt, good morning!'

f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>
```

2.2 核心模块

2.2.1 密钥与参数生成模块

- 密钥生成步骤：

1. 保证 n 是较大素数(1024bit)，且安全 p 和 q 大小应拉开差距，所以分别取 p 和 q 位数为516和508。
2. 检验 p 和 q 最大公因子是否过大，最好是2
3. 取公钥为安全素数 d ，保证 d 大于 p 和 q ，并且较为靠近 $\Phi(n)$ 。
4. 求出等效公钥 e ，并检验 e 是否过小
5. 公开 (e, n) ，保留 d ，必要时保留 p 和 q

- 密钥的本地存取设计：

用json文件存储信息，按人员姓名分为私人文件夹和公共root文件夹(rsa)，用来模拟用户加密解密过程。

rsa文件夹负责存储各个用户的密码信息。其中，rsa下的个人文件夹负责存放个人的私钥部分，rsa下的public_key.json存储各个用户公布的公钥数据。json处理过程涉及的一系列检验都在代码注释中。

- 各算法实现语言：python

```
def RSA_initial(usr):
    '''RSA密钥生成函数'''
    # 生成p、q部分
    p = new_safe_prime(516)
    while 1:
        q = new_safe_prime(508) #通过位数不同（516+508=1024），和p拉开大小差距。
        _,_,gcd= Egcd(p-1, q-1)
        if gcd == 2:
            #检测p-1, q-1的最大公因子是否过大。
            break;
    N = p * q

    #利用文件操作，给不同人不同N，防止共模攻击。 同时，若可能，读取公钥信息
    public_path = "./rsa/public_key.json"
    try:
        with open(public_path, mode = 'r') as f_in:
            old_info = json.load(f_in) #此处读取了公钥文件的数据！
    except json.decoder.JSONDecodeError:
        """文件如果空时会出错"""
        print("empty existed public keys file, continuing ... ")
        f_in.close()
    else:
        for dict in old_info.values():
            #由于前面没有设置跳转，使用递归实现当N重复时的重置
            if dict['mod']==N:
                RAS_initial(usr)
                return
        f_in.close()

    #生成d、e部分
    phi = (p - 1) * (q - 1)
    d = random.randint(max(p, q), phi-1)
    while judge_if_prime(d)==False
```

```

        or judge_if_relatively_prime([d, phi//gcd])==False #保证两
者互素可求逆

        or judge_if_prime((d-1)/2)==False: #保证是安全素数
        # d比较重要先选d, d要大于p、q, 并且接近phi, 并且是安全素数。这样也同
时保证了d足够大(大于N**0.25)
        d = next_prime(d) #在初始随机数d附近寻找素数
        e = FindInverse(d, phi//gcd)
        if e==1 or e==2: #同样利用递归, 实现当e不合要求时的重置
            RSA_initial(usr)
            return

        #test部分, 检验密钥有效性, 令msg=12345678
        msg = 12345678
        assert judge_if_prime(p)
        assert judge_if_prime(q)
        assert d*e % phi == 1
        assert judge_if_relatively_prime([d, phi])
        assert pow(pow(msg, e, N), d, N)==msg, "invalid key"

        #将公钥写入公共文件保存, 私钥单独保存在用户文件夹
        with open(public_path, mode = "w") as f_pub_out:
            #公钥要检验是否已经该人已经生成了一对密钥了, 如果已经存在需要覆盖原本密
            钥。

            info = {'owner': usr, 'public_key':e, 'mod':N}
            print('public keys:{}'.format(info)) #mark
            if not 'old_info' in dir(): #防止是空文件导致读取失败
                json.dump({usr:info}, f_pub_out)
            else:
                for existed_usr in old_info.keys():
                    if (existed_usr == usr):
                        old_info[usr] = info
                    else :
                        old_info.update({usr: info})
                json.dump(old_info, f_pub_out); #f_pub_out.write('\n')
            f_pub_out.close()

        private_path = './rsa/{}/private_key.json'.format(usr)
        with open(private_path, mode = "w") as f_pri_out: #json文件不支持:
        newline="\n"
            # 私钥由于每人只允许生成一对, 由于保存结构, 会直接覆盖写
            info = {'owner': usr, 'private_key':d, 'q':q, 'p':p, 'mod':N}
            json.dump(info, f_pri_out); #f_pri_out.write('\n')
            print('private keys:{}'.format(info)) #mark

```

```
f_pri_out.close()
print("success!")
```

2.2.2 加密与解密模块

- 加密过程

1. 打开公钥存储文件(公共频道), 根据要发送消息给谁(to_who), 选择该人公布的公钥(e, N)
2. 加密: $cipher = msg^e \pmod N$, 加密过程使用[2.3.2 平方乘算法优化](#)
3. 检测明文msg和密文cipher相似度是否过高, 过高则报错(该人密钥选择不当, 可向其发送警示信息)
4. 将密文cipher发送给to_who

```
def Encryption(msg, to_who):
    '''加密函数, 需要指明目标发送者'''
    with open("./rsa/public_key.json", "r") as f_in:
        try:
            public_keys = json.load(f_in)
        except json.decoder.JSONDecodeError:
            print("error, empty public key! where to find e?")
            return
        for public_key in public_keys.values():
            if public_key['owner'] == to_who:
                e, N = public_key['public_key'], public_key['mod']
                break;
        assert 'e' in vars(), "error, no matched public_key_for:
to_who"
        f_in.close()
        cipher = fast_power_mod(msg, e, N)
        assert msg != cipher, "error, coincidence equal" #防止e=log(kn+m)
        时, 密文明文相同
        return cipher
```

- 解密过程

1. 去私人文件夹读取私钥d

2. 解密: $msg = cipher^d \pmod N$

```
def Decryption(usr, cipher):
    '''解密函数, 需要指明解密者'''
    private_path = './rsa/{}/private_key.json'.format(usr)
    with open(private_path, "r") as f_in:
        try:
            key = json.load(f_in)
        except json.decoder.JSONDecodeError:
            print("error, empty private key, where to find d?")
            return;
        d, N = key['private_key'], key['mod']
        assert key['owner']==usr, "error, wrong owner for private key"
        f_in.close()
    return pow(cipher, d, N)
```

- 用中国剩余定理优化解密过程

因为 d 一般都较大, 解密时间成本相应较高. 如果知道 p, q , 可以使用数学化简手段加速解密过程.

具体数学原理请见[1.2.1 利用 \$p\$ 和 \$q\$ 加速解密](#)

但是, 因为保存了 p 和 q , 相应的泄露安全风险也增加了. 故一般而言都是使用后, p 和 q 即销毁

```
def Decryption_accelerate_with_pq(c, usr):
    '''利用中国剩余定理加速解密过程'''
    private_path = './rsa/{}/private_key.json'.format(usr)
    with open(private_path, "r") as f_in:
        try:
            key = json.load(f_in)
        except json.decoder.JSONDecodeError:
            print("error, empty public key, where to find d?")
            return
        d = key['private_key'] #暂且不检验是否赋值成功啦
        q, p = key['q'], key['p']
        assert key['usr']==usr, "error, wrong owner for private key"
        f_in.close()
    eq = []
    eq.append( [pow((c%p), (d%(p-1))), p], p )
    eq.append( [pow((c%q), (d%(q-1))), q], q )
    return chinese_remainder_theorem(eq)
```

2.2.3 签名与验签模块

- 签名过程：

1. 去个人私人文件夹读取私钥d
2. 利用私钥d加签： $signature \equiv msg^d \pmod{N}$
3. 将签名sign和消息msg一起发送给接收方

```
def signature(msg, usr):  
    '''签名函数'''  
    private_path = './rsa/{}/private_key.json'.format(usr)  
    with open(private_path, "r") as f_in:  
        try:  
            key = json.load(f_in)  
        except json.decoder.JSONDecodeError:  
            print("error, empty private key, where to find d?")  
            return  
        d, N = key['private_key'], key['mod']  
        assert key['owner'] ==usr, "error, wrong woner for private  
key"  
        f_in.close()  
        return pow(msg, d, N)
```

- 验签过程：

1. 接收方去读取发送者的公钥
2. 利用公钥解签： $msg' \equiv signature^e \pmod{N}$
3. 检验 $msg=msg'$ ，如果相等，说明消息未被篡改。

```
def verification(msg, signature, sender):  
    '''验签函数'''  
    with open("./rsa/public_key.json", "r") as f_in:  
        try:  
            public_keys = json.load(f_in)  
        except json.decoder.JSONDecodeError:  
            print("error, empty public key")  
            return  
        for public_key in public_keys.values():  
            if public_key['owner'] == sender:
```

```

        e, N = public_key['public_key'], public_key['mod']
        break;
    assert 'e' in vars(), "error, no matched public_key_for:
to_who"
    f_in.close()
    to_verify = pow(signature, e, N)
    return int(to_verify) == int(msg)

```

2.3 基础模块

2.3.1 扩展欧几里得算法

- 算法主体：求最大公因子
数学原理见[1.2.3 数学模块](#)

```

def Egcd(a, b):
    '''扩展欧几里得求最大公因子算法'''
    #matrix calculation
    r, s, t = a, 1, 0
    r1, s1, t1 = b, 0, 1
    #gcd
    while r1 != 0:
        q = r // r1
        tmp1, tmp2, tmp3 = r - r1*q, s - s1*q, t - t1*q
        r, s, t = r1, s1, t1
        r1, s1, t1 = tmp1, tmp2, tmp3
    gcd = r
    assert gcd > 0, "gcd error"
    assert s*a + t*b == gcd, "s*a+t*b=gcd error"
    return s, t, gcd

```

- 求逆元算法
利用欧几里得方法计算逆元。

```

def FindInverse(a, p):
    '''欧几里得方法计算逆元'''
    b, _, gcd=Egcd(a, p)
    assert gcd==1, "error, p and a is not relative prime"
    return b %p

```

2.3.2 快速模幂算法

- 平方乘模块：
利用位运算实现，从低位到高位

```
def fast_power_mod2(num, exp, mod):  
    '''平方乘算法'''  
    ans = 1  
    base = num  
    while exp != 0:  
        if exp & 1 != 0:  
            ans = (base*ans) %mod  
        base = (base*base) %mod  
        exp >>= 1  
    return ans %mod
```

2.3.3 素性检测算法

- 米勒-拉宾法素性检测模块：
 - 先检验奇偶进行二分排除，排除大于2偶数
 - 再进行素性探测，该过程重复循环十次(可设定)，因为每次循环正确率为0.75，故总错误率控制在 $10e-60$ 数量级

```
def judge_if_prime(n):  
    '''素性检测: Miller-Rabin + 偶数排除'''  
    if n==2:  
        return True  
    elif n %2==0 or n<=1 :  
        return False  
    for i in range(10):  
        #test 10 times, wrong possibility is 10^(-60)  
        a = random.randint(2, n-1)  
        #或者a直接选用强序列: 2, 325, 9375, 28178, 450775, 9780504,  
        1795265022, ...  
        *b, gcd = Egcd(a, n)  
        if 1!=gcd:  
            return False  
        #n - 1 == 2^k * q  
        q, k = n - 1, 0  
        while q %2 == 1:
```

```

        q = q/2
        k = k+1
    a = pow(a, q, n)
    if a % n == 1:
        continue;
    for j in range(0, k-1, 1):
        if a % n == n-1:
            continue;
        a = (a*a) % n
    return False
return True

```

- 互素检测模块

调用米勒拉宾检测模块

```

def judge_if_relatively_prime(list):
    '''判断是否互素'''
    for i in range(0, len(list)):
        for j in range(i+1, len(list)):
            _, gcd = Egcd(list[i], list[j])
            if gcd != 1:
                return False
    return True

```

2.3.4 中国剩余定理

中国剩余定理解一次同余方程组

```

def chinese_remainder_theorem(eq):
    '''eq: nested list of (b, m), 中国剩余定理'''
    m, m_inverse, b = [], [], []
    for pair in eq:
        b.append(pair[0])
        m.append(pair[1])
    assert judge_if_relatively_prime(m) == True, "m not relatively prime"
    M = numpy.prod(m)
    sum = 0
    for i in range(len(m)):
        sum += (FindInverse(M/m[i-1], m[i-1]) * (M/m[i-1]) * b[i-1]) % M
    return sum

```

2.3.5 获取新的安全素数

先选取一个规定长度的随机数，然后在其附近寻找安全素数，检测素性调用米勒-拉宾检测模块。

通过调用random库实现随机整数，使用randint控制随机数长度(参数)。

```
import random
def next_prime(n):
    '''寻找下一个相邻素数'''
    n = (n + 1) | 1
    while not judge_if_prime(n):
        n += 2
    return n

def new_safe_prime(bin_length):
    '''获取一个指定二进制长度的素数'''
    rand = random.randint(
        2**(bin_length-1), 2**(bin_length)-1)
    while judge_if_prime(rand)==False
        or judge_if_prime((rand-1)/2)==False:
        #通过检验(p-1)/2，来检验rand是否是安全素数以增加分解N的困难性
        rand = next_prime(rand)
    return rand
```

3 思考：RSA参数选择与安全性优化

3.1 RSA参数选择

3.1.1 模数不能共用

- 模数N不能多人使用

假想一种情景：A需要同时想B和C群发某个消息，他找到了B和C公布的公钥e1和e2，并用这两个公钥分别加密消息msg，并将密文发送给B和C。此时如果B和C使用的是同一个N，那么存在共模攻击，使得hacker可以根据两个密文破解msg内容。

共模攻击原理见[4.1.1](#)

- 尽量不要用同一模数生成多组密钥

因为如果模N的一个密钥d泄露后，能根据其他该N生成的公钥e'，求得其对应密钥d'。因此会导致所有基于该N的加密都不再安全。

详细原理见[4.1.2](#)

3.1.2 p和q差值不能太小

p和q差值过小时，可以通过"费马算法"线性时间破解。因此，本作业主体函数实现时，为了获得1024bit的N，分别取p和q位数为516bit和508bit

- 费马算法原理：

假设p和q相差较小。由等式：

$$N = p * q = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$$

知，可以尝试不断循环尝试寻找关于N的两个平方数。即，让 $\left(\frac{p+q}{2}\right)^2$ 从 \sqrt{N} 开始递增，直到 $(a^2 - N)$ 是平方数为止。因为求出了 $\left(\frac{p+q}{2}\right)^2$ 和 $\left(\frac{p-q}{2}\right)^2$ ，因此p和q是可解的。

需要注意该方法只对奇合数有用，因为偶合数分解后无法整除2，因此不能验证平方数。算法复杂度为 $O(|p-q|)$ 因此该算法是否高效取决于p和q的取值是否足够接近。

3.1.3 私钥d过小（e过大）

如果d过小，存在针对某特殊情况的维纳(wiener)攻击，可在多项式时间破解d。

维纳攻击原理见[4.3.1](#)

3.1.4 p-1和q-1都应该有很大的素因子。

即p和q都应该是安全素数。

- 选择p和q的方法：

选择素数p1和q1，使得 $p = 2 * p1 + 1$ ， $q = 2 * q1 + 1$ 。并且p和q也是素数。在

本作业RSA密钥初始化实现中，为了代码简洁性，采用的方法则是检测 $p-1$ 和 $q-1$ 最大公因子gcd是否是2，若不是2则不断循环取新直到满足gcd=2.

- 为什么需要安全素数？

因为如果 $p-1$ 和 $q-1$ 有较大的公因子g，由1.2.2中等效d原理：

$d * e \equiv 1 \pmod{\frac{(p-1)*(q-1)}{g}}$ ，会存在相对很小的等效密钥d，其可能被快速枚举出来。

另外，取安全素数也增加了分解N的难度，因为N不会存在很多整数因子。

3.1.5 公钥e不可以太小

不能是1和2

但e相对小的情况，并不会明显降低安全性。相反，e小可以方便加密操作，并提高密钥d的安全性。通常选 $e=3$ ，或者 $17(= 2^4 + 1)$ ，或者 $65537(= 2^{16} + 1)$

3.1.6 避免特殊情况密文和明文相似度过高

如果 $e \neq \log(k * n + m)$ 时，密文和明文可能出现相同。在本作业加密过程实现中，手动检测了明文和密文的相似度

3.2 大作业心得体会

现代密码体系大多是基于数学上的困难(NP)问题，但数学问题终究有各种各样的特殊情况和解法，导致了安全漏洞始终存在。数学问题本身的难解性没有问题，但是将该严谨的数学问题转化为具体的密码模型时，就会出现各种细节与实现上的漏洞，比如该RSA实验中各种参数的选择和密钥的分配，都需要谨慎考虑各种可能的错误。从理论转到实践实现，任何细节的欠考虑都会极大拖累算法性能和安全性，甚至最终导致最终结果的失败。就比如生成大素数算法，原本我虽然也是生成随机数验证是否是素数，后续的处理上，我原本的程序会不断取随机数直到猜中一个强素数为止，但这样的效率是远远慢于在上一个非素数的随机数附近接着寻找下一个素数的，这导致了我一开始的程序选素数模块运行得非常慢。优化后，虽然正确率有所下降，但速度提升非常明显。因此我测试了每一个模块的运行速度，并查阅各种资料进行不断优化。

另一个感想是关于安全性和性能之间的平衡，一般情况下，追求最佳安全性和稳定性时，就会牺牲算法的性能。比如为了保证p和q的安全性，需要选比素数更严格的强素数，这导致了选素数算法运行速度有一定的下降(没针对性优化时，可能是几倍的

下降)。两者通常不能两全其美，只能在两者之间追求一种动态的平衡，在保证一定安全性正确性的前提下，也不让性能有太明显下降。当然，密码学的保密性可能更多是对绝对安全的追求，有时候会将性能的下降低转嫁到其他方面的代价，比如空间复杂度升高换取时间性能，或者协议方替保密方承受更多时间代价(RSA中，加密者的公钥 e 小，加密速度快，而解密方 e 大，解密速度可能比较慢)

4 针对RSA安全漏洞的攻击实验

4.1 Level-1

4.1.1 实验原理

- 实验思想

生成密钥的过程中使用了相同的模数 n ，此时用多组密钥加密同一信息 m 是不安全的。基于这一原理生成的RSA攻击方法称为共模攻击。

- 共模攻击原理

$$c1 = m^{e1} \pmod N$$

$$c2 = m^{e2} \pmod N$$

若两个密钥 e 互素根据扩展的欧几里得算法则存在 $s1$ ， $s2$ 有：

$$e1 * s1 + e2 * s2 = \gcd(e1, e2) = 1$$

结合以上所有信息，可以得到：

$$\begin{aligned} & (c1^{s1} * c2^{s2}) \pmod N \\ &= (m^{e1} \pmod N)^{s1} * (m^{e2} \pmod N)^{s2} \pmod N \\ &= m^{(e1*s1 + e2*s2)} \pmod N \\ &= m \pmod N \\ &= m \end{aligned}$$

也就是在完全不知道私钥的情况下，得到了明文 m

$$m = (c1^{s1} * c2^{s2}) \pmod N$$

4.1.2 实验结果与完整代码

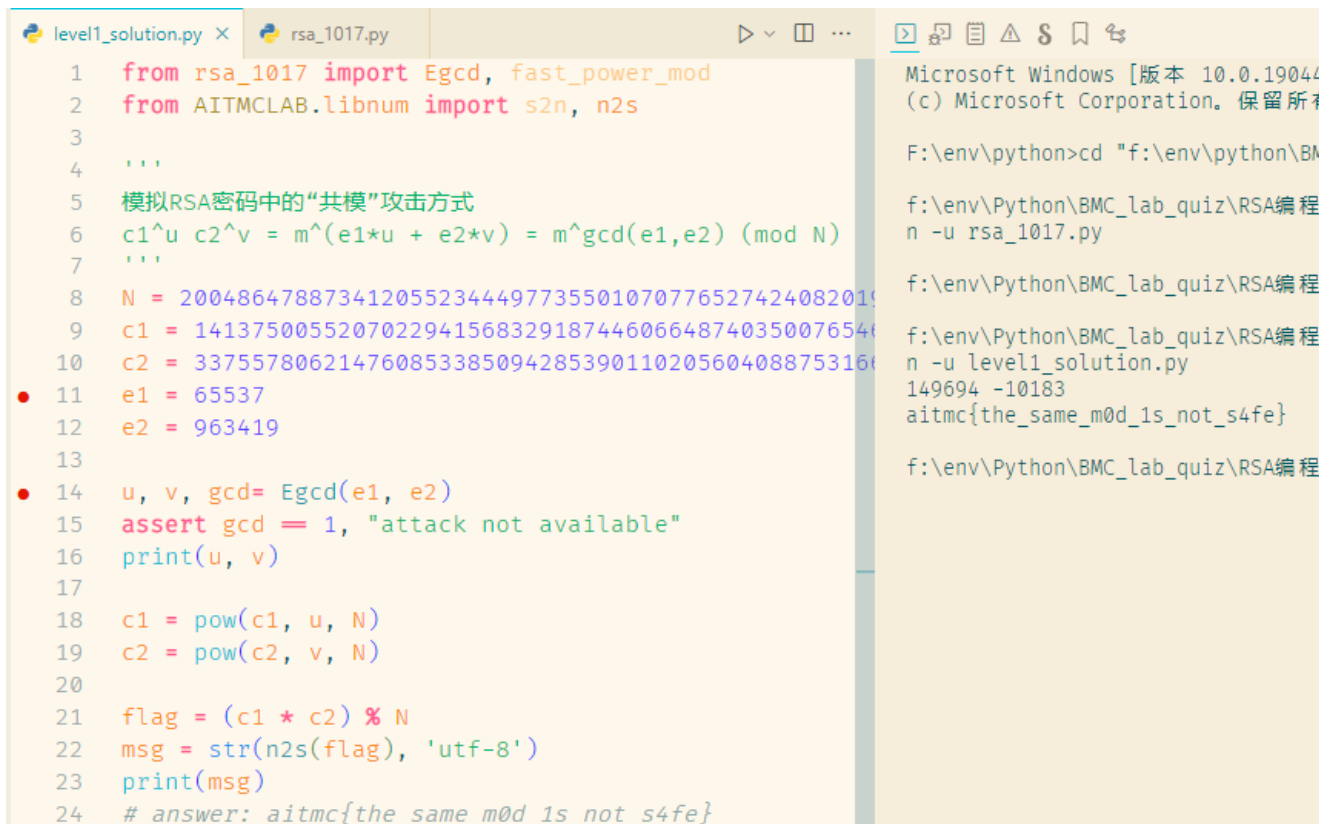
- 实验代码

```
u, v, gcd= Egcd(e1, e2)
assert gcd == 1, "attack not available"

c1 = pow(c1, u, N)
c2 = pow(c2, v, N)

flag = (c1 * c2) % N
msg = str(n2s(flag), 'utf-8')
print(msg)
```

- 实验结果



```
level1_solution.py x rsa_1017.py
1 from rsa_1017 import Egcd, fast_power_mod
2 from AITMCLAB.libnum import s2n, n2s
3
4 '''
5 模拟RSA密码中的“共模”攻击方式
6  $c1^u \cdot c2^v = m^{(e1 \cdot u + e2 \cdot v)} = m^{\gcd(e1, e2)} \pmod{N}$ 
7 '''
8 N = 2004864788734120552344497735501070776527424082019
9 c1 = 141375005520702294156832918744606648740350076546
10 c2 = 337557806214760853385094285390110205604088753166
11 e1 = 65537
12 e2 = 963419
13
14 u, v, gcd= Egcd(e1, e2)
15 assert gcd == 1, "attack not available"
16 print(u, v)
17
18 c1 = pow(c1, u, N)
19 c2 = pow(c2, v, N)
20
21 flag = (c1 * c2) % N
22 msg = str(n2s(flag), 'utf-8')
23 print(msg)
24 # answer: aitm{the_same_m0d_1s_not_s4fe}
```

```
'shell:
aitmc{the_same_m0d_1s_not_s4fe}
```

4.2 Level-2

4.2.1 实验原理

- 实验思想

当基于 N 的一个密钥 d 泄露时，基于该 N 的所有密钥 d' 都会变得不安全

- 实验原理分析

由被泄露的密钥 d_1 ：

$$d_1 * e_1 \equiv 1 \pmod{(p-1) * (q-1)}$$

将该式改写为：

$$d_1 * e_1 - 1 = k * (p-1) * (q-1)$$

根据 e_2 ，能求出 d_2' ，使得：

$$d_2' * e_2 \equiv 1 \pmod{k * (p-1) * (q-1)}$$

而该式的解 d_2' 与真正的 d_2 是等价的，因为一定满足：

$$d_2' * e_2 \equiv 1 \pmod{(p-1) * (q-1)}, \text{ (as } \phi > 1 \text{)}$$

因此我们能够根据任意公开的 e 算出对应的 d

4.2.2 实验结果与完整代码

- 代码主体[1]

```
eq_phi = d*e -1
d2 = rsa.FindInverse(e2, eq_phi) #等效Phi
msg, msg2 = str(n2s(pow(c2, d2, N))), str(n2s(pow(c, d, N)))
print(msg+msg2)
```

- 实验结果

```
level1_solution.py | level2_solution.py x | level3_solution.py | ▶ ▼ □ ... | 🔍 📄 📌 📁 📧
```

```
1 import rsa_1017
2 from AITMCLAB.libnum import n2s
3
4 #Data:
5 N = 259707266103386592679026714514647737567121558638
6 c = 110645789629810363091724964245872234647467722286
7 d = 126318934577758576068742725760343120423218032554
8 e = 131866378633366089093488433261645718304883698493
9 c2 = 77459007490844230495965801373210671888192469851
10 e2 = 0x20211011
11
12 '''解题分析:
13     d1*e1 == 1 mod phi
14 将该式改写为:
15     d1*e1-1 == phi * k
16 能求出d2, 使得:
17     d2*e2 == 1 mod k * phi
18 而该式的解d2也同样满足:
19     d2*e2 == 1 mod phi , (as phi > 1)
20 '''
21
22 print(len(str(bin(d*e))))
23 eq_phi = d * e - 1
24 d2 = rsa_1017.FindInverse(e2, eq_phi) #用等效phi算逆即可
25 # msg = str(n2s(rsa_1013.fast_power_mod(c2, d2, N)))
26 msg = str(n2s(pow(c2, d2, N)))
27 msg2 = str(n2s(pow(c, d, N)))
28 print(msg) #It_is_important_to_
29 print(msg2) #store_your_private_key}
30
```

```
'shell:
aitmc{It_is_important_to_
store_your_private_key}
```

4.3 Level-3

4.3.1 实验原理

- 实验思想:
当d过小且满足一定条件时, 使用连分数原理在多项式时间内破解密钥d
- 实验原理分析
前提: $3d < n^{\frac{1}{4}}$, 且 $q < p < 2q$
由前提条件可证明不等式:
1. $n = p * q > q^2$, 即 $q < \sqrt{n}$

$$2. \quad 0 < n - \Phi(n) = p + q - 1 < 2q + q - 1 < 3q < 3\sqrt{n}$$

由于 $d * e \equiv 1 \pmod{\Phi(n)}$, 所以 $\exists k$, 满足

$$d * e - k * \Phi(n) = 1$$

两边同除 $d * n$, 有

$$\frac{e}{n} - \frac{k * \Phi(n)}{d * n} = \frac{1}{d * n}$$

即

$$\frac{e}{n} - \frac{k}{d} = \frac{1 - k * (n - \Phi(n))}{d * n}$$

由不等式(2)知:

$$\left| \frac{1 - k * (n - \Phi(n))}{d * n} \right| < \frac{3k\sqrt{n}}{d * n} = \frac{3t}{d\sqrt{n}}$$

因为 $k \mid (d * e - 1)$, d 取素数, 所以 $k < d$, 因此

$$\left| \frac{e}{n} - \frac{k}{d} \right| < \frac{1}{3d^2}$$

由连分数理论可知, $\frac{k}{d}$ 是 $\frac{e}{n}$ 的一个渐进分数. 此时不断通过 $\frac{e}{n}$ 的连分数商不断向后遍历其渐进分数, 直到遍历到符合条件的 $\frac{k}{d}$ (即 k 和 d), 依此可以求出 N 的分解.

Wiener 进一步证明了该方法在前提条件下可以在多项式时间破解 RSA.

4.3.2 实验结果与完整代码

- 代码主体

```
def rational_to_contfrac(x,y):
    ...
    将有理数转化为连分数商列表 → [a0, ..., an]
    ...
    a = x//y
    pquotients = [a]
    while a * y != x:
```

```

        x,y = y,x-a*y
        a = x//y
        pquotients.append(a)
    return pquotients

def convergents_from_contfrac(frac):
    '''
        利用连分数商列表，计算其渐进分数
    '''
    convs = [];
    for i in range(len(frac)):
        convs.append(contfrac_to_rational(frac[0:i]))
    return convs

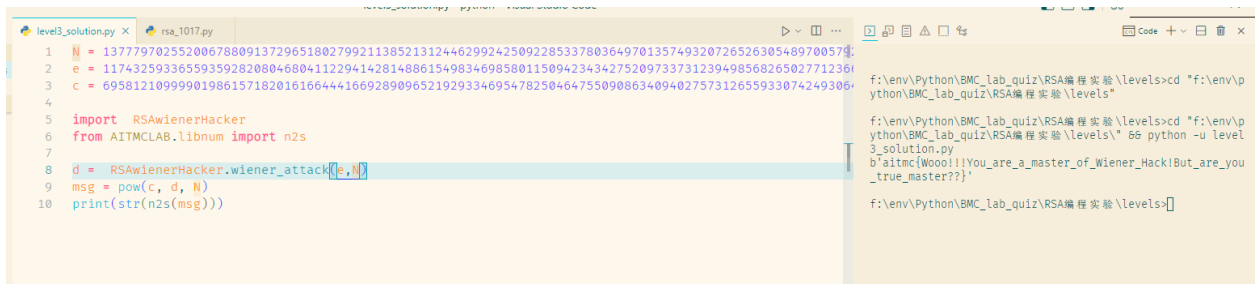
def contfrac_to_rational (frac):
    '''相反，将连分数列表转化为有理数分数形式，结果以元组表示分子分母
    '''
    if len(frac) == 0:
        return (0,1)
    num = frac[-1]
    denom = 1
    for _ in range(-2,-len(frac)-1,-1):
        num, denom = frac[_]*num+denom, num
    return (num,denom)

def wiener_attack(e,n):
    '''
        维纳攻击，利用连分数求解d
    '''
    frac = rational_to_contfrac(e, n)
    convergents = convergents_from_contfrac(frac)

    for (k,d) in convergents:
        #check if d is actually the key
        if k!=0 and (e*d-1)%k == 0:
            phi = (e*d-1)//k
            s = n - phi + 1
            delta = s*s - 4*n
            if(delta>=0):
                if is_square(delta) and (s+sqrt(delta))%2==0:
                    return d
    return -1

```

• 实验结果



```
1 N = 1377797025520067880913729651802799211385213124462992425092285337803649701357493207265263054897005794
2 e = 1174325933655935928208046804112294142814886154983469858011509423434275209733731239498568265027712361
3 c = 695812109999019861571820161664441669289096521929334695478250464755090863409402757312655933074249306
4
5 import RSAWienerHacker
6 from AITMCLAB.libnum import n2s
7
8 d = RSAWienerHacker.wiener_attack([e, N])
9 msg = pow(c, d, N)
10 print(str(n2s(msg)))
```

```
f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>cd "f:\env\p
ython\BMC_lab_quiz\RSA编程实验\levels"
f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>cd "f:\env\p
ython\BMC_lab_quiz\RSA编程实验\levels" && python -u level
3_solution.py
b'aitmc{Wooo!!!You_are_a_master_of_Wiener_Hack!But_are_you
_true_master??}'
f:\env\Python\BMC_lab_quiz\RSA编程实验\levels>
```

```
'shell:
aitmc{Wooo!!!You_are_a_master_of_Wiener_Hack!But_are_you_true_master??
}
```

Last edited in 10/23/2022，缺省主体实验结果，交互优化完成

1. 代码中rsa库即为大作业手搓功能主体，代码随附。下同 ↩