Network Security Project: RSA, CCA2 and OAEP

520030910281 肖真然

1 项目简介

本项目是SJTU CS3325课程期末项目。本项目包含三个Task：

- 在Task 1中，需要实现 Textbook RSA加解密算法；
- 在Task 2中，需要实现对Textbook RSA加解密算法的CCA2攻击[1]；
- 在Task 3中，需要实现OAEP-RSA算法，并讨论其能够防御CCA2攻击的原因。

2 环境

- OS：Win10
- 语言：Python 3.x
- 库：Crypto，binascii，hashlib
- IDE：VS Code Jupyter Notebook

3 Task 1：Textbook RSA

3.1 任务说明

在Task 1中，需要实现Textbook RSA加解密算法（无填充），包括：

- 生成具有给定密钥大小的随机RSA密钥对（比如1024-bit）；
- 用公钥加密明文；
- 用私钥解密密文。

3.2 Textbook RSA原理

RSA加密算法属于公钥加密算法，是一种非对称算法，它的安全性依赖于大整数分解的困难性。

其密码学语法如下：

- $KeyGen(1^\lambda) \to (pk, sk)$：
    1. Input：1^λ，指定密钥空间大小；
    2. 生成大素数 p、q，计算$N = pq$，$\phi(N) = (p-1)(q-1)$；
    3. 选择正整数e，满足$gcd(e, \phi(N)) = 1$且$e < \phi(N)$，公钥对即为$pk = (N, e)$；
    4. 计算$d = e^{-1} \mod \phi(N)$，私钥对即为$sk = (N, d)$；
    5. Output：(pk, sk)，公私钥对。
- $Enc(pk, m) \to c$：
    1. Input：$pk = (N, e)$，公钥对；$m \in Z_N^*$，为明文（Z_N^*为明文空间，表示从1到N之间与N互素的正整数集合）；
    2. 计算得到密文$c = m^e \mod N$；
    3. Output：c，密文。
- $Dec(sk, c) \to m$：

1. Input：$sk=(N,d)$，私钥对；$c\in Z_N^*$，为密文（Z_N^*为密文空间）；
2. 计算得到明文$m=c^d \mod N$；
3. Output：m，明文。

RSA正确性验证过程如下：

$$\begin{aligned} Dec(sk,Enc(pk,m)) &= Dec(sk,m^e \mod N) \\ &= (m^e \mod N)^d \mod N \\ &= m^{ed \mod \phi(N)} \mod N \\ &= m \mod N \end{aligned}$$

在上式中，第三个等号成立的前提条件是m与N互素。但从$[0,N-1]$中挑到与N不互素的m的概率为$\frac{1}{p}+\frac{1}{q}-\frac{2}{pq}$，由于$p$和$q$都很大，所以此概率可忽略不计[2]。

3.3 代码设计

3.3.1 Miller-Rabin素性检验[3]

Miller-Rabin素性检验是一种素数判定法则，利用随机化算法判断一个数是否是素数。

Miller-Rabin素性检验依赖以下定理：

> 如果p是素数，x是小于p的正整数，且$x^2=1 \mod p$，则x要么为1，要么为$p-1$。

简单证明：如果$x^2=1 \mod p$，则p整除x^2–1，即整除$(x+1)(x-1)$，由于p是素数，所以p要么整除$x+1$，要么整除$x-1$，前者则x为$p-1$，后者则x为1。

Miller-Rabin素性检验首先利用了因数分解式，将幂次n-1降低为以2为阶的各次幂，再利用中国剩余定理，推出强伪素数的满足条件。在算法优化中，采用模平方算法降低复杂度，这也是该算法的优势之一，大大提高了效率。

具体操作为：

1. 给定奇整数$n\geq 3$和安全参数k；
2. $n-1=2^st$，其中t为奇整数；
3. 随机选取整数b，$2\leq b\leq n-2$；
4. 计算$r_0=b^t \mod n$；
5. (a) 如果$r_0=1$或$r_0=n-1$，则通过检验，n可能为素数，回到3；
   (b) 否则有$r_0\neq 1$且$r_0\neq n-1$，计算$r_1=r_0^2 \mod n$，回到(a).
   (c) 以此类推直到$r_i=n-2$，若仍未通过检验，则n为合数。

在k次检测通过的情况下，n为合数的概率小于$\frac{1}{4^k}$。

代码实现如下：

```
# Miller Rabin算法 判断一个给定的大数是否为素数
def miller_rabin(num, safe_k=1024):
    # num - 1 = 2^s * t
    s = 0
    t = num - 1
```

```
    while t % 2 == 0:
        s += 1
        t //= 2

    for _ in range(safe_k):
        b = random.randint(2, num - 2)
        r = pow(b, t, num)
        if r == 1 or r == num - 1:
            continue
        for _ in range(s - 1):
            r = pow(r, 2, num)
            if r == num - 1:
                break
        else:
            return False

    return True
```

设置安全参数为1024，则num为合数的概率小于$\frac{1}{4096}$。

3.3.2 欧几里得算法求最大公约数[4]

欧几里得算法又称辗转相除法，用于计算两个非负整数a，b的最大公约数。

欧几里得算法依赖以下定理：

> 两个整数的最大公约数等于其中较小的那个数和两数相除余数的最大公约数。

代码实现如下：

```
# 欧几里得算法（辗转相除） 求两个数的最大公因数
def gcd(a, b):
    remainder = a % b
    while remainder:
        a = b
        b = remainder
        remainder = a % b
    return b
```

3.3.3 拓展欧几里得算法求模逆元[5]

拓展欧几里得算法用于计算模逆元。

拓展欧几里得算法依赖以下定理：

> 给予二个整数a、b，必存在整数x、y使得$ax + by = gcd(a, b)$。

有两个数a、b，对它们进行辗转相除法，可得它们的最大公约数。然后，收集辗转相除法中产生的式子，倒回去，可以得到$ax + by = gcd(a, b)$的整数解。

代码实现如下：

```
# 拓展欧几里得算法 求模逆元(a ^ -1) % b
def ext_euclid(a, b):
    if b == 0:
        return 1, 0
    else:
        x, y = ext_euclid(b, a % b)
        x, y = y, (x - (a // b) * y)
        return x, y

def mod_inverse(a, b):
    return ext_euclid(a, b)[0] % b
```

3.3.4 快速幂取余[6]

快速幂取余又叫蒙哥马利算法，用于更快地计算$a^b \mod n$。

原理公式较长，在此不做过多赘述，详见参考资料[6]。

代码实现如下：

```
# 快速幂取余算法 求(a ^ b) % n
def fast_exp_mod(a, b, n):
    res = 1
    while b != 0:
        if b & 1:
            res = (res * a) % n
        b >>= 1
        a = pow(a, 2, n)
    return res
```

3.3.5 素数生成算法

想要生成素数，首先要能判断一个数是否为素数。

代码实现如下：

```
# 判断一个给定的数是否为素数
def is_prime(num):
    if num < 2:
        return False
    if num in small_primes:
        return True
    for prime in small_primes:
        if num % prime == 0:
            return False
    return miller_rabin(num)
```

在以上代码中，使用小于1000的素数集加快了判断速度，对一般情况调用了Miller Rabin算法进行判断。

能够判断素性后，再进行指定位数的素数生成。

代码实现如下：

```
# 生成一个指定（二进制）位数的素数
def gen_prime(size=1024):
    while True:
        num = random.randrange(2 ** (size - 1), 2 ** size)
        if is_prime(num):
            return num
```

在以上代码中，根据指定的位数循环生成随机数，直到此随机数为素数即可返回。

3.3.6 RSA密钥生成

原理再3.2中已经阐明。

代码实现如下：

```
# RSA密钥生成
def RSA_KeyGen(size=1024):
    # 生成大素数p q，计算N
    p = 1
    q = 1
    N = p * q
    while N.bit_length() != size:
        p = gen_prime(size // 2)
        q = gen_prime(size - size // 2)
        N = p * q
    # 计算ϕ(N)
    phi_N = (p - 1)*(q - 1)
    # 选择与ϕ(N)互素的正整数e
    e = 1
    while True:
        e = random.randrange(3, phi_N)
        if gcd(e, phi_N) == 1:
```

```
            break
    # 生成公钥对
    pk = (N, e)
    # 计算d，生成私钥对
    d = mod_inverse(e, phi_N)
    sk = (N, d)

    # 写入文件
    write_file(str(N), 'RSA_Moduler.txt')
    write_file(str(p), 'RSA_p.txt')
    write_file(str(q), 'RSA_q.txt')
    write_file(str(pk), 'RSA_Public_Key.txt')
    write_file(str(sk), 'RSA_Secret_Key.txt')

    return pk, sk
```

实现RSA密钥生成算法利用了：

- 素数生成算法生成大素数；
- 欧几里得算法选取公钥e；
- 拓展欧几里得算法求e的模N逆元私钥d。

3.3.7 RSA加密解密算法

原理在3.2中已经阐明。

加密算法代码实现如下：

```
# 加密
def RSA_Enc(pk, m):
    m_bytes = bytes(m, encoding='utf-8')
    m_hex_str = int(b2a_hex(m_bytes), 16)
    c = fast_exp_mod(m_hex_str, pk[1], pk[0]) # c = (m ^ e) % N
    return c
```

注意首先进行了字符串明文到ascii编码的转换。

解密算法代码实现如下：

```
# 解密
def RSA_Dec(sk, c):
    m = fast_exp_mod(c, sk[1], sk[0])  # m = (c ^ d) % N
    m_int = a2b_hex(hex(m)[2:])
    m_str = str(m_int, encoding='utf-8')
    return m_str
```

注意最后进行了ascii编码到字符串明文的转换。

3.4 运行结果

调用RSA算法过程如下：

```
# 随机种子
random.seed(520030910281)
# 生成公私钥对
size = 1024
pk, sk = RSA_KeyGen(size)

# 明文消息
plain_msg = read_file('Raw_Message.txt')
print("Raw Message:", plain_msg)

# 加密
cipher_msg = RSA_Enc(pk, plain_msg)
print('Encrypted Message:', hex(cipher_msg))
write_file(hex(cipher_msg), 'Encrypted_Message.txt')

# 解密
decipher_msg = RSA_Dec(sk, cipher_msg)
print("Decrypted Message:", decipher_msg)
```

结果：

```
Raw Message: Hello RSA!
Encrypted Message: 0x1d94d3d354b81fd295ed3d83ae4a6a1619bec9179925c56a77afeceed0ad598e96175c1ffefe8b6bd1d2d9d7c25b51456380b713daf781e04645a078bdcf18a47c38bdc
Decrypted Message: Hello RSA!
```

对比原消息和解密结果可知，代码实现正确。

4 Task 2: CCA2

4.1 任务说明

在Task 2中，需要对Textbook RSA加解密算法进行CCA2攻击。

任务要求模拟一对Server-Client的通信模型[1]：

- Client：
    1. 为此次交互会话生成一个128-bit的AES会话密钥；
    2. 使用AES会话密钥加密WUP交互报文；
    3. 使用1024-bit的RSA公钥加密AES会话密钥；
    4. 将经过RSA加密后的AES会话密钥和经过AES加密后的WUP交互报文生成message发送给Server。
- Server：
    1. 使用RSA私钥解密被加密的AES会话密钥；
    2. 选择解密结果中有效的 128-bit 低位作为 AES 会话密钥；
    3. 使用AES会话密钥和原始长度信息解密接收到的 WUP 交互报文；
    4. 若WUP交互报文中的request合法，返回一个AES解密成功的响应。

而在此种CCA2攻击中，Server知道：

- RSA密钥对；
- AES密钥。

攻击者知道：

- RSA公钥；
- RSA加密的AES密钥；
- 一个AES加密的WUP交互报文。

攻击者想知道：

- AES密钥。

Task 2要求：

- 在代码中正确设计WUP交互报文格式、Server-Client通信模型等；
- 生成一个message，它应该包括一个RSA加密的AES密钥和一个AES加密的交互报文；
- 模拟从message获得AES密钥、并进一步解密交互报文的攻击过程；
- 可以使用第三方库实现AES加解密。

4.2 CCA2原理

RSA很优雅，但没有语义安全性。自适应选择密文攻击（缩写为CCA2）是一种交互式攻击，是选择密文攻击的一种形式。攻击者发送一些待解密的密文，然后使用这些解密的结果来选择随后的密文。该攻击的目标是逐渐解密一个被加密的消息或密钥本身。

“When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users[1]”一文阐述了CCA2的原理：

令C表示用RSA公钥(N,e)加密后的128-bit的AES密钥k，则有

$$C \equiv k^e \mod N$$

令$k_b = 2^b k$，表示k向左移动了b比特的移位，C_b表示RSA加密后的AES密钥，则

$$C_b \equiv k_b^e \mod N$$

因此，可以仅通过C和公钥e计算出C_b：

$$\begin{aligned} C_b &\equiv k_b^e \mod N \\ &\equiv (2^b k)^e \mod N \\ &\equiv k^e 2^{be} \mod N \\ &\equiv (k^e \mod N)(2^{be} \mod N) \mod N \\ &\equiv C(2^{be} \mod N) \mod N \end{aligned}$$

首先考虑k_{127}和C_{127}，k_{127}的最高位是k的最低位，其他位均为0。猜测k_{127}的最高位为1，并发送一条由k_{127}加密的WUP报文，与C_{127}组合成一条message发送给Server。如果Server利用此条message解密的WUP请求合法，则说明k的最高位确实为1，否则是0。依此类推，直到恢复k所有的信息。综上所述，只需迭代128次即可破解密钥k。对于更长的密钥，破解时间也与密钥长度同量级，时间成本完全可以接受。

4.3 代码设计

4.3.1 WUP类

根据4.1中要求设计WUP类如下：

```
# WUP类
class WUP:
    def __init__(self, request="", key=""):
        # AES加密的 WUP Request
        self.aes_en_wup = request
        # RSA加密的 AES密钥
        self.rsa_en_aes_key = key
```

4.3.2 Client

根据4.1中要求设计Client类如下：

```
# Client类
class Client:
    def __init__(self):
        # RSA公钥
        self.rsa_pk = eval(read_file('RSA_Public_Key.txt'))
        # 生成AES密钥
        self.aes_key = random.randrange(2 ** 127, 2 ** 128)
        write_file(hex(self.aes_key), 'AES_Key.txt')

    def send_message(self, msg):
        # WUP Request写入文件
        write_file(hex(int(msg.encode('utf-8').hex(), 16)), 'WUP_Request.txt')
        # 为AES加密补齐位数
        while len(msg) % 16 != 0:
            msg += '\0'

        # AES加密 WUP Request
        AES_Cryptor = AES.new(a2b_hex(hex(self.aes_key)[2:]), AES.MODE_ECB)
        aes_en_wup = b2a_hex(AES_Cryptor.encrypt(msg.encode('utf-8')))

        # RSA加密 AES密钥
        rsa_en_aes_key = RSA_Enc(self.rsa_pk, self.aes_key)

        # AES加密的 WUP Request写入文件
        write_file(hex(int(aes_en_wup.hex(), 16)), "AES_Encrypted_WUP.txt")
        # History Message（包括AES加密的 WUP Request和RSA加密的 AES密钥）写入文件
        write_file(hex(int(aes_en_wup.hex(), 16)) + '\n' + hex(rsa_en_aes_key),
'History_Message.txt')

        # 发送WUP报文
        w = WUP(aes_en_wup, rsa_en_aes_key)
        return w
```

4.3.3 Server

根据4.1中要求设计Server类如下：

```
class Server:
    def __init__(self):
        # RSA公钥
        self.rsa_pk = eval(read_file('RSA_Public_Key.txt'))
        # RSA私钥
        self.rsa_sk = eval(read_file('RSA_Secret_Key.txt'))
        # AES密钥，初始尚未收到
        self.aes_key = 0

    def receive_message(self, w):
        # RSA私钥解密 WUP报文中的 AES密钥
        aes_key = bin(RSA_Dec(self.rsa_sk, w.rsa_en_aes_key))[-128:]
        aes_key = int(aes_key, 2)

        # 补齐位数
        aes_key_string = ""
        for i in hex(aes_key)[2:]:
            aes_key_string += i
        while len(aes_key_string) < 32:
            aes_key_string = "0" + aes_key_string
        self.aes_key = a2b_hex(aes_key_string)

        # AES密钥解密 WUP报文中的 WUP Request
        AES_Decrypter = AES.new(self.aes_key, AES.MODE_ECB)
        plain_message = b2a_hex(AES_Decrypter.decrypt(a2b_hex(w.aes_en_wup)))

        return plain_message
```

4.3.4 CCA2攻击

根据4.2中原理模拟以下攻击：

```
# CCA2攻击过程
def CCA2_Attack(server, history_msg):
    # 已知RSA公钥
    rsa_pk = eval(read_file('RSA_Public_Key.txt'))

    # 要猜测的AES密钥，128-bit
    current_key = 0
    # 迭代128次
    for ite in range(128, 0, -1):
        # 本次猜测密钥
        test_key = int(current_key >> 1) + (1 << 127)
```

```
        # 填充用 WUP Request
        pad_request = 'whatever'
        while len(pad_request) % 16 != 0:
            pad_request += '\0'

        # 用猜测的AES密钥加密 WUP Request
        AES_Cryptor = AES.new(a2b_hex(hex(test_key)[2:]), AES.MODE_ECB)
        aes_en_wup = str(b2a_hex(AES_Cryptor.encrypt(pad_request.encode('utf-8'))),
encoding='utf-8')

        # 用C（RSA公钥加密后的AES密钥）和 RSA公钥 计算 C_b
        factor = fast_exp_mod(2, (ite - 1) * rsa_pk[1], rsa_pk[0])
        rsa_en_aes_key = fast_exp_mod(history_msg.rsa_en_aes_key * factor, 1,
rsa_pk[0])

        # 构造WUP发送给Server，得到回复
        w = WUP(aes_en_wup, rsa_en_aes_key)
        response = server.receive_message(w)

        # 如果WUP Request解密成功说明该位猜测正确，否则取反
        if response == b2a_hex(bytes(pad_request, encoding='utf-8')):
            current_key = test_key
        else:
            test_key = int(current_key >> 1)
            current_key = test_key

    # 用最终猜得的AES密钥解密History Message中的WUP Request
    AES_Decrypter = AES.new(a2b_hex(hex(current_key)[2:]), AES.MODE_ECB)
    plain_msg = str(AES_Decrypter.decrypt(a2b_hex(history_message.aes_en_wup)),
encoding='utf-8')
    plain_msg = plain_msg.rstrip('\0')

    # 验证，与client最初生成的AES密钥一致则攻击成功
    if current_key == client.aes_key:
        print("Attack Success!")
        print('History Message Request:', plain_msg)
        print('AES Key:', hex(current_key))
    else:
        print("Attack Fail!")
        print('Guessed AES Key:', current_key)
```

4.4 运行结果

模拟Client与Server间通信过程，得到历史消息：

```
# 随机种子
random.seed(520030910281)

# 初始化
client = Client()
```

```
server = Server()

# WUP Request
request = "Sample Request"
# Client发送WUP获得历史消息
history_message = client.send_message(request)

# Server收到消息，返回解密的WUP Request
print(str(a2b_hex(server.receive_message(history_message)), encoding='utf-8'))
```

根据历史消息实施攻击：

```
# 实施攻击
CCA2_Attack(server, history_message)
```

```
Attack Success!
History Message Request: Sample Request
AES Key: 0xb65160645fd3d71e7f21aed0e7d08e0f
```

`Attack Success!`说明攻击成功，对比Client发送的原消息和攻击者的解密结果也证明攻击成功。

5 Task 3：OAEP-RSA

5.1 任务说明

在Task 3中，需要实现OAEP-RSA算法，并讨论为什么它可以防御这种类型的攻击。

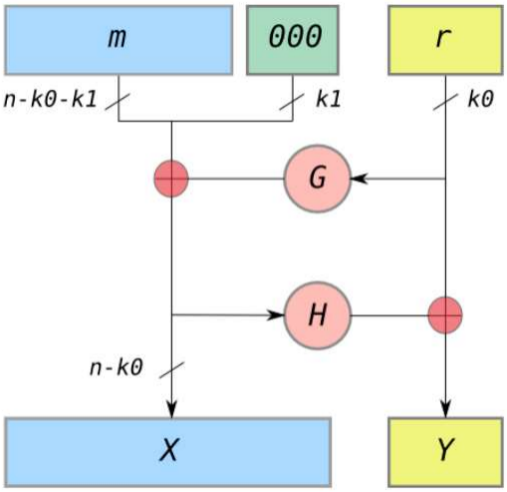
Task 3要求：

- 将OAEP填充模块添加到Textbook RSA的实现中。
- 讨论OAEP-RSA相对于Textbook RSA的优势。
- 可以进一步尝试向OAEP-RSA实施CCA2攻击，看看其是否可以防御在Task 2中实现的CCA2攻击。

5.2 OAEP原理

由于Textbook RSA易受攻击，在“When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users[1]”一文中，作者给出了一个解决方案：使用OAEP密钥填充算法。OAEP（Optimal Asymmetric Encryption Padding）是一种对消息进行随机填充的策略，在密码学中，最优非对称加密填充（OAEP）是一种填充方案通常与RSA加密一起使用。

OAEP填充过程如下图所示：



流程描述如下：

1. 由协议选定整数k_0、k_1，加密散列函数G和H；
2. 在明文m（$n-k_0-k_1$位）后填充k_1个0，填充后长度为$n-k_0$，n为RSA算法中模的位数（比如1024-bit）；
3. 随机生成一个位数为k_0的串r；
4. G将k_0位的r展开为$n-k_0$位；
5. 计算$X=(m||0^{k_1})\oplus G(r)$；
6. H将$n-k_0$位的X减少到k_0位；
7. 计算$Y=H(X)\oplus r$；
8. 输出为$X||Y$。

整合成一个公式：

$$OAEP(m,r)=X||Y=(m||0^{k_1})\oplus G(r)||r\oplus H((m||0^{k_1})\oplus G(r))$$

OAEP恢复方式为：

- $r=Y\oplus H(X)$;
- $m||0^{k_1}=X\oplus G(r)$。

OAEP具有“全有或全无”的安全性[7]：为了恢复m，必须恢复整个X和整个Y，从Y中恢复r需要X，从X中恢复m需要r。由于单向散列函数的雪崩效应——即对任何输入信息的变化，哪怕仅一位，都将导致散列结果的明显变化——因此必须一次性完全恢复整个X和整个Y。

OAEP满足了以下两个目标[8]：

- 增加一个随机元素，将一个确定性的加密方案（比如传统的RSA）转换为概率方案。
- 确保攻击者在无法反转陷门单向函数的情况下难以恢复明文的任何部分，以防止密文的任何部分被解密，或泄漏关于明文的任何信息，即提供了语义安全。即使攻击者拥有多个明文的加密结果，他们也无法通过对比这些密文获取关于明文的信息。

5.3 代码设计

5.3.1 OAEP填充

根据5.2中原理实现附加OAEP填充的RSA解密算法如下：

```
# OAEP-RSA加密
def OAEP_RSA_Enc(m, pk, k0=512):
    # 字符串转ascii码
    encoded_m = hex(int(m.encode('utf-8').hex(), 16))
    int_m = int(encoded_m, 16)

    # 计算k1，注意填充后结果不能超过N
    k1 = pk[0].bit_length() - int_m.bit_length() - k0 - 1

    # OAEP填充
    r = random.randrange(1 << (k0 - 1), (1 << k0) - 1)
    write_file(str(r), 'Random_Number.txt')
    g_r = hashlib.sha512(hex(r).encode('utf-8'))
    x = int(int_m << k1) ^ int(g_r.hexdigest(), 16)
    h_x = hashlib.sha512(hex(x).encode('utf-8'))
    y = r ^ int(h_x.hexdigest(), 16)
    padded_m = (x << k0) + y
    write_file(hex(padded_m), 'Message_After_Padding.txt')

    # RSA公钥加密填充后结果
    return RSA_Enc(pk, padded_m), k1
```

算法首先对明文进行了OAEP填充得到填充后的明文，然后进行RSA加密。

5.3.2 OAEP还原

根据5.2中原理实现附加OAEP还原的RSA解密算法如下：

```
# OAEP-RSA解密
def OAEP_RSA_Dec(c, sk, k1, k0=512):
    # RSA私钥解密填充后结果
    padded_m = RSA_Dec(sk, c)

    # OAEP还原
    y = padded_m % (1 << k0)
    x = padded_m >> k0
    h_x = hashlib.sha512(hex(x).encode('utf-8'))
    r = y ^ int(h_x.hexdigest(), 16)
    g_r = hashlib.sha512(hex(r).encode('utf-8'))
    int_m = x ^ int(g_r.hexdigest(), 16)

    # ascii码转字符串
    encoded_m = a2b_hex(hex(int_m >> k1)[2:])
    m = str(encoded_m, encoding='utf-8')
```

```
    return m
```

算法首先对密文进行了RSA解密得到OAEP填充后的明文，然后进行还原。

5.4 运行结果

调用OAEP-RSA算法过程如下：

```
# 随机种子
random.seed(520030910281)

# RSA公钥
pk = eval(read_file('RSA_Public_Key.txt'))
# RSA私钥
sk = eval(read_file('RSA_Secret_Key.txt'))

# 明文消息
plain_msg = "Sample Message"
print('Raw Message:', plain_msg)

# 加密
cipher_msg, k1 = OAEP_RSA_Enc(plain_msg, pk)
print('Encrypted Message:', hex(cipher_msg))
write_file(hex(cipher_msg), 'Encrypted_Message.txt')

# 解密
decipher_msg = OAEP_RSA_Dec(cipher_msg, sk, k1)
print("Decrypted Message:", decipher_msg)
```

```
Raw Message: Sample Message
Encrypted Message: 0x359b53c11e93c51f0677c0890a567d8c98053bd511a2263126d415fa5cca2fa6381e4f6ed7e0ad02ffd81d53753dae411702ada8c9535d7601a04d7bd0732185ef90f476
Decrypted Message: Sample Message
```

对比原消息和解密结果可知，代码实现正确。

5.5 OAEP-RSA对CCA2攻击的防御

回顾4.2中阐述的CCA2攻击的原理，在OAEP-RSA的场景下：

令C表示用OAEP-RSA加密后的128-bit的AES密钥k，则有

$$C \equiv OAEP^e(k) \mod N$$

令$k_b = 2^b k$，表示k向左移动了b比特的移位，C_b表示OAEP-RSA加密后的AES密钥，则

$$C_b \equiv OAEP^e(k_b) \mod N$$

然而，无法仅通过C和公钥e计算出C_b：

$$
\begin{aligned}
C_b &\equiv OAEP^e(k_b) \mod N \\
&\equiv OAEP^e(2^b k) \mod N \\
&\neq C(2^{be} \mod N) \mod N \\
&\equiv (OAEP^e(k) \mod N)(2^{be} \mod N) \mod N \\
&\equiv (2^{be} OAEP^e(k) \mod N) \mod N \\
&\equiv ((2^b OAEP(k))^e \mod N) \mod N
\end{aligned}
$$

即

$$
OAEP(kx) \neq kOAEP(x)
$$

所以攻击者无法构造合法的C_b，每次Server利用攻击者发送的message解密的WUP请求都不合法，导致攻击者猜测k的每一位都为1。若实际k不全为1，则攻击者攻击失败。

因此，Task2中实现的CCA2攻击对OAEP-RSA必然无效。

6 总结与致谢

在该项目中，学生复习了Textbook RSA的原理，并详细学习了其中用到的各个算法的细节，如欧几里德算法、Miller-Rabin算法等。还学习了“When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users[1]”一文提出的CCA2攻击和OAEP填充方案，理解了Textbook RSA不是CCA安全的，而可以OAEP弥补这一点，感受了一番密码学的奥妙。

在原理学习方面，非常感谢刘振老师[2]和朱浩瑾老师[1]的教导。而在代码设计方面，要感谢助教提供的作业文档[7]，为代码设计提供了很好的思路；撰写代码细节时，遇到最大的麻烦应该是各种进制、类型之间的转换，以及各个算法对数值位数的严格约束，在此特别感谢学长学姐们[9][8]的帮助！

7 参考资料

1. Knockel J , Ristenpart T , Crandall J .When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users[J]. 2018.DOI:10.48550/arXiv.1802.03367. ↩ ↩ ↩ ↩ ↩ ↩

2. Public Key Encryption Schemes Slides. Liu Zhen. ↩ ↩

3. Miller-Rabin大素数判断算法的原理及其实现 ↩

4. 欧几里得算法_百度百科 ↩

5. 扩展欧几里得算法_百度百科 ↩

6. 快速幂取余 - 简书 ↩ ↩

7. Network Security - Project Slides. Tian Dong ↩ ↩

8. AlchemistYe/CS3325_CCA2_Attack ↩ ↩

9. syun-fung/RSA-CS3325 ↩