✧ PART1 Textbook RSA

根据 Textbook RSA 的定义，首先需要选择两个大素数 p 和 q，用 p 和 q 的乘积 n 作为模数。之后选取与φ(n)互素的 e 作为另一部分的公钥，其中φ(n) = (p-1)*(q-1)。最后计算出 e 在模φ(n)意义下的逆元 d，与 n 一同作为私钥。加密时，$c \equiv m^e \bmod n$；解密时，$m \equiv c^d \equiv m^{ed} \bmod n$。

因为 Python 在处理大整数时很便捷、处理字符串时更灵活，故选用 Python3 实现。对上述 Textbook RSA 的定义进行总结，实现上可分为以下几个重点：①生成大素数 p 和 q；②选取与φ(n)互素的 e；③求逆元 d；④可读消息字符串到大整数的转换；⑤加解密时大整数的幂运算和模运算。

①生成大素数 p 和 q：根据素数分布定理，从不大于 n 的自然数中随机选取一个，它是素数的概率大约是 1/ln n；按这样计算，如果生成一个 n bit 的数，它是素数的概率大约是 2/n。这看起来不是一个很高的概率，但实现上来讲，我们的确大多采用随机选取的办法，再验证该数是否是素数，这就需要较快的素性检测算法来配合。此外，由于我们需要固定 bit 长度的 n，所以对于 p 和 q 的 bit 长度也应该能够控制，故先随机生成(len-1) bit 的比特串，再在最高位补 1，使 p 和 q 的长度可控。

```
def PQGenerate(len):
    isPrime = False
    while(not isPrime):
        number = random.getrandbits(len-1)
        number += 1<<(len-1)
        if(number % 2 == 0):
            continue
        else:
            isPrime = MillerRabin(number, 18)
    return number
```

采用 Miller-Rabin 算法进行素性检测，其原理是：如果 p 是素数，x 是小于 p 的正整数，且 $x^2 \bmod p = 1$，那么要么 x=1，要么 x=p-1。这是显然的，因为 $x^2 \bmod p = 1$ 相当于 p 能整除 x^2-1，也即 p 能整除(x+1)(x-1)。由于 p 是素数，那么只可能是 x-1 能被 p 整除(此时 x=1)或 x+1 能被 p 整除(此时 x=p-1)。Miller-Rabin 是概率性算法，该算法如果返回 False，证明被测数一定不是素数；但如果返回 True，被测数有 1/4 的可能也不是素数。基于此，我们连续进行 18 次 Miller-Rabin 素性检测，让其出错的概率降至 $1/2^{36}$，从而满足实际需要。

```
def MillerRabin(n, times):
    m = n - 1
    k = 0
    while m % 2 == 0:
        m = m // 2
        k += 1
    for i in range(0, times):
        isPrime = False
        a = random.randint(1, n - 1)
        b = pow(a, m, n)
        b = b % n
        if b == 1:
            isPrime = True
        for j in range(0, k):
            if b == n - 1:
                isPrime = True
                break
            b = (b * b) % n
        if not isPrime:
            return False
    return True
```

②e 的选取：事实上，我们一般将 e 的值固定，而非每次都重新选取。首先，e 不能选取过小的值，如 e=3 就很容易遭受广播场景中的小 e 攻击；其次，e 选取过大的值必然对运算效率造成影响。所以，基于对安全性和运算效率的折中考虑，也本着像实际应用中相关标准看齐的想法，选取 e = 65537 = 2^{16} + 1，其加密只需要 17 次模乘运算，且是天然的质数，与φ(n)互素的概率极大。

③求逆元 d：采用欧几里得扩展算法。

```
def extendEuclid(a,b):
    if(b==0):
        return 1,0, a
    x2, y2, remainder = extendEuclid(b,a%b) #Euclid for gcd

    x1 = y2
    y1 = x2 - (a//b)*y2

    return x1, y1, remainder
```

④可读消息字符串到大整数的转换：简单的 pack 和 unpack 函数，根据 ascii 码将字符和整数互相转换，要求 8 bit 即 1 byte 对齐，也是为后续 PART 的实现提供便捷统一的格式。

```
def pack(message):
    M = 0
    i = len(message) - 1
    for x in message:
        M += int(ord(x)) << (8 * i)
        i -= 1
    return M


def unpack(M):
    message = ""
    while(M != 0):
        x = M % (1 << 8)
        M = M // (1 << 8)
        message = chr(x) + message
    return message
```

⑤大整数的幂运算和模运算：采用快速幂算法。

```
def quickPowNMod(M, e, N):
    C = 1
    while(e!=0):
        if(e&1):
            C = (C*M)%N
        e >>= 1
        M = (M*M)%N
    return C
```

⑥主函数设计：将 key size 交给用户输入，希望应用上能满足灵活的需要，且这在实现上也不会额外增加难度。

选取 p 与 q 时，我们一般认为，p 和 q 不应该太接近（设置两个 512 bit 素数似乎是很自然的想法），否则攻击者很容易从根号 n 附近的整数开始逐个查验，解二次方程得到 p 和 q。基于这样的考虑，设置 p 和 q 的 bit 长度有所不同，这里的设置使得 p 和 q 的 bit 长度差异较大，实际上几个 bit 的差异就已经足够。

```
key_size = int(input("Please input your key size (1024 and 2048 recommended):"))

reGen = True
while(reGen):

    lenP = random.randint(key_size//4,key_size//7*3)
    lenQ = key_size - lenP
    P = PQGenerate(lenP)
    Q = PQGenerate(lenQ)

    N = P*Q
    phi_N = (P-1)*(Q-1)
    if(N.bit_length()!=key_size):
        continue #reGen

    e = 65537
    if(phi_N%e==0):
        continue #reGen

    d = (extendEuclid(phi_N,e)[1] + phi_N) % phi_N
    if((e*d)%phi_N!=1):
        continue #reGen

    reGen = False
```

此外，出现 n 的长度不足 1024 bit 时需要重新生成 p 和 q。编写测试，跑 100 组*5 次，按 n 的 bit 长度对 n 进行归类，发现合乎要求的 p 和 q 生成出不足 1024 bit 的 n 的概率约为 40%。测试如下：

```
cnt = 0
cnt1 = 0
cnt2 = 0
while(cnt<100):
    lenP = random.randint(300,450)
    lenQ = 1024 - lenP
    P = PQGenerate(lenP)
    Q = PQGenerate(lenQ)
    N = P*Q
    if(N.bit_length()==1024):
        cnt1 += 1
    elif(N.bit_length()==1023):
        cnt2 += 1
    cnt += 1
    print(cnt)

print("CNT 1024:", cnt1)
print("CNT 1023:", cnt2)
```

主函数中其余两个 reGen，一个是为了处理较为罕见但理论上可能出现的情况，即 e=65537 与本次的φ(n)不互素；另一个是为了验证 d 的确是φ(n)的逆元，这在实际测试中理应不会触发 reGen。

测试： Python3.5 @ Ubuntu 16.04 LTS

```
lynx@lynx-virtual-machine:~$ python3 rsa-textbook.py
Please input your key size (1024 and 2048 recommended):1024
public key = ( 65537 , 116246557213879195253094978769696263138381857672776055302
857240234236645447356913479693858846455465988911445392693228460473436543453777
738546115395369631571426487266547649476820672167919467421082048615494790792263
234032627418835431283626288754477884571545703051715634541599585042336498193371
922095536779 )

private key = ( 791112302713921294457068396189514782830304901668291933192354818
465719553225759033992745774769611423274618685181989059847428252856178671719867
026420274510621558144993312920144755712250865014763780838770862644893393277616
167764154482752962436960416233400413417232005245419171039324092120530190282138
9553 , 116246557213879195253094978769696263138381857672776055302857240234236645
473569134796938588464554659889114453926932284604734365434537775273854611539536
315714264872665476494768206721679194674210820486154947907922637623403262741883
312836262887544778845715457030517156345415995850423364981933715992209553677 )

Please input your message:
we pretend the night wont steal our youth

encrypted = 183318592822784020284260757396345253212241437741719094952578322112
675238562075575972647054109630632381548962468367142596652676951605139721454684
666075843441153065444033737989107327465337471582611665724689727390287434382228
290029695403692887682885932002843167641696457578104810105108405775726409084001

decrypted = we pretend the night wont steal our youth
```

✧ PART2 CCA2 Against Textbook RSA

采用 socket 用两个终端模拟 server 端和 client 端进行通信，选取一个 10000 以上的端口防止被占用，Client 端 host 设为本机 IP，Server 端监听并绑定。启动时需要先启动 server 端。上图为 Client 端，下图为 Server 端：

```
host = "192.168.146.128"

port = 25535
addr = (host, port)
byte = 1024
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


# sock
byte = 1024
port = 25535
host = ""
addr = (host, port)

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

sock.bind(addr)
print "waiting for a client..."
```

遵照论文 When Textbook RSA is Used to Protect the Privacy of Millions of Users 中提供的 CCA2 过程进行复现：在获得一个加密后的 AES key 后，从最低位开始对其进行逐 bit 猜测。这种攻击可行的一个原因是，Server 端每次解密后，直接选取最低 128 bit 作为 AES key，所以我们可以采用补 0 的方式，控制每次使用的 AES key 只有一位未知。该位只有 0 和 1 两种可能，我们便发送用该位为 0 的 AES key 加密的 wup 请求，如果被 Server 端判断为合法，证明该位为 0，否则为 1。这样，我们进行 128 轮猜测与验证，最终可以获得 128 bit 的真正的 AES key，并尝试用它解密历史消息。

Demo 中，先行启动 Server 端，选择本次 RSA 的公钥私钥并打印出来，这里也提供自行设置 AES key 的接口，但要求必须为 16 个字符，否则将被判断为不合法输入。Server 端此后等待一个 Client 的加入：

```
lynx@lynx-virtual-machine:~$ python server3.py
public key = ( 65537 , 152445007649765993438021029651218322775595938026049058337
176003397381346443325601563302275137776679016016320818846421799578139366943859 43
464207586948940916555181428103097103238695929002932771567427375303432019686828 74
235063667806358217242279371816363508929359780572137786639604808846227126800573 91
773175224417 )
private key = ( 761376015135288538873202060293220339821903084283479222041357545 0
818753119280610327555442070600891492671691125688513478407299124871365943002790 44
389183671542535920833531380840243827220655131441500823812462247096937394603881 09
374757581345669058155863891581037909035176696224376352342975785311030724739853 93
6449 , 152445007649765993438021029651218322775595938026049058337176003397381346 4
433256015633022751377766790160163208188464217995781393669438594346420758694894 09
165551814281030971032386959290029327715674273753034320196868287423506366780635 82
172422793718163635089293597805721377866396048088462271268005739177317522441 7 )

Feel free to set your aes key (16 characters):abc
I said we need 16 characters!
Input again:abcdefghijklmnop
aes key = abcdefghijklmnop

encrypted aes_key = 138386878315600920099882370860135282725152057023243674733648
895100860744257958179581620542809900300243205306518865264823825349578003196547 62
331243716707466940544125216632660161933359298983052607526932051329734721979742 16
316547026502842584885569558756228857006012878269950311008488463205835451964214 52
667483186

encrypted WUP request in utf-8:  a21548393616e59adb9347305495de82

waiting for a client...
```

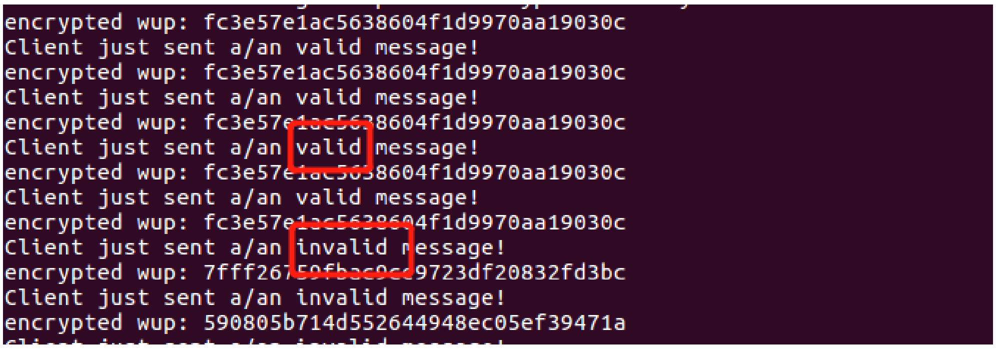
Client 端上线后，如果用户输入口令“Let's get it started”，Server 端将向 Client 端发送刚刚生成的 RSA 公钥和加密后的 AES key，它们在 Client 端被打印出来：

```
waiting for a client...
found a client. Sending rsa pk and encrypted aes key...
```

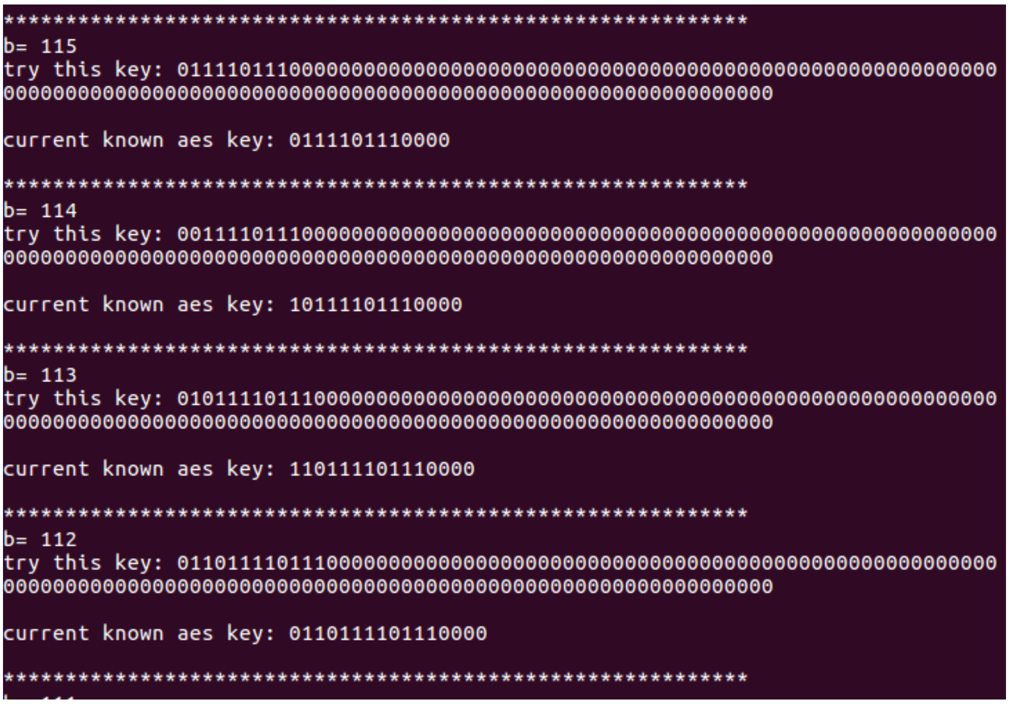
```
lynx@lynx-virtual-machine:~$ python client3.py
Shall we start?
Let's get it started
I got N: 152445007649765993438021029651218322775595938026049058337176003397381346
644332560156330227513777667901601632081884642179957813936694385944642075869489 4
091655518142810309710323869592900293277156742737530343201968682874235063667806 35
821724227937181636350892935978057213778663960480884622712680057391773175224417

I got C: 138386878315600920099882370860135282725152057023243674733648895100860744
257958179581620542809900300243205306518865264823825349578003196547623312437167
746694054412521663266016193335929898305260752693205132973472197974216316547026
284258488556955875622885700601287826995031100848846320583545196421452667483186
```

接下来，Attacker 即 Client 端，将根据密文 C（加密后的 AES key），实施 CCA2 攻击。Client 每次向 Server 端发送移位后的 C_b，以及用相应的自行补 0 后的 AES key（包含本次猜测的 1 bit，实现中猜测为 0）加密的合法 wup 请求，b 的范围从 127 到 0。Server 端将打印出 Client 每次发来的加密 wup 请求，判断它是否合法，合法则打印“Client just sent a/an valid message”，不合法则打印“Client just sent a/an invalid message”，并

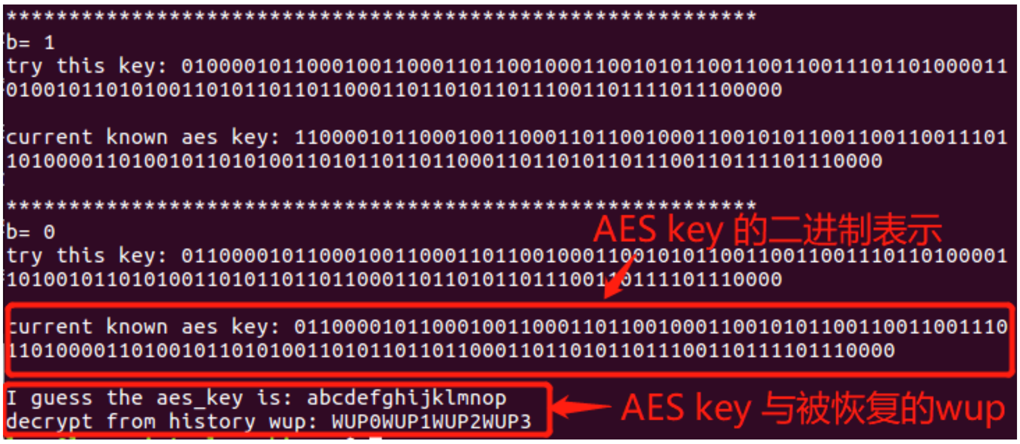
通知 Client 本次的消息是否合法:



Client 收到 Server 的消息，合法说明该 bit 是 0 的猜测正确，在已恢复的 AES key 串（current known）最高位写 0；不合法说明该 bit 是 1，则在最高位写 1。

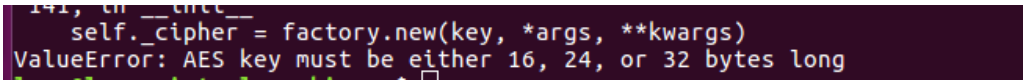


如此进行 128 轮，Client 将获得完整的 AES key，它尝试解出历史 wup 请求，结果成功:



一些问题：

①原本希望延续 PART1 使用 Python3，但实际测试中，总会在 AES 的加密处提示如下图的错误：



提示所用的 AES key 不是 16 bytes 长，但打印所用 AES key 的长度，确实又是 16。后发现，如果 AES key 中每个 byte 的比特串表示中的首位为 1，则会提示该错误。这也不无道理，毕竟自然情况下，没有哪个字符的 ascii 码比特串首位为 1。使用的测试如下，其中 aes_key3 和 aes_key5 不能成功解密：

```
import random
from Crypto.Cipher import AES
from binascii import b2a_hex, a2b_hex

def bin2str(b):
    s = ""
    for i in range(0,len(b),8):
        s = s + chr(int(b[i:i+8],2))
    return s

aes_key1 = "0123012301230123"
crypto1 = AES.new(aes_key1, AES.MODE_ECB)
c1 = crypto1.encrypt("messagemessage12")
d1 = crypto1.decrypt(c1)
print(d1,"\n")

aes_key2 = bin2str("0"*128)
crypto2 = AES.new(aes_key2, AES.MODE_ECB)
c2 = crypto2.encrypt("messagemessage12")
d2 = crypto2.decrypt(c2)
print(d2,"\n")

aes_key3 = bin2str("1"+"0"*127)
crypto3 = AES.new(aes_key3, AES.MODE_ECB)
c3 = crypto3.encrypt("messagemessage12")
d3 = crypto3.decrypt(c3)
print(d3,"\n")

aes_key4 = bin2str("0"+"1"+"0"*126)
crypto4 = AES.new(aes_key4, AES.MODE_ECB)
c4 = crypto4.encrypt("messagemessage12")
d4 = crypto4.decrypt(c4)
print(d4,"\n")

aes_key5 = bin2str("00000000"+"1"+"0"*119)
crypto5 = AES.new(aes_key5, AES.MODE_ECB)
c5 = crypto5.encrypt("messagemessage12")
d5 = crypto5.decrypt(c5)
print(d5,"\n")
```

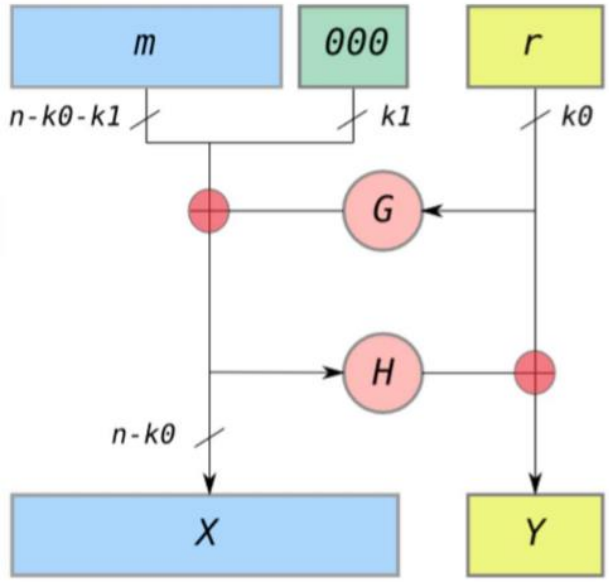
怀疑是 pyCrypto 库过旧，已经几年无人维护，与 Python3 存在兼容性问题。故而 PART2 的测试最后采用 Python2.7 进行。至于 Python3，可能应该使用 PyCryptodome 库。

②在 PART2 中，数据格式实际上是最大的问题，解决加解密前后消息的格式和 socket 传输过程中的格式化问题是本 PART 最大的难点。

③在完成 PART2 前，错误地认为可以针对这个场景进行 offline 解密，无需与 Server 端进行通信。后经助教指点，意识到不能仅考虑 AES key 本身，移位后的 C_b 加密的内容在 128 bit 向上仍有未知内容。也就是说，只能交由 Server 端在解密后比对 AES key，而不能在 Client 端通过比对 C_b 的方式获得 AES key。

✧ PART3 RSA-OAEP

RSA-OAEP 在 textbook RSA 之前，加入了 padding 后的消息和随机比特串的两轮 Feistel 结构。由于不涉及 AES，PART3 同 PART1 一样采用 Python3。实现上的重点有两个：①G 和 H 的选择。②k1 和 k0 的取值；



①G 与 H：稍稍思考后我们会发现，k1 和 k0 的取值与 G 和 H 对输入和输出的要求有关，故我们只能先考虑 G 和 H。简单地将 G 和 H 称为“哈希函数”是不严谨的，他们实际上应该是 MGF（Mask Generation Function），而非纯粹的“哈希函数”。MGF 是一类基于哈希函数，或者说内部用到了哈希函数的掩码生成函数，可以根据输入的 seed，输出用户给定长度的 Mask。这里不能直接使用“哈希函数”的另一个原因是，SHA 系列除了 SHA3 的中几个 SHAKE 函数，都只能输出定长的摘要 digest，且它们往往只能压缩，参照上图图示，让 r 的长度比 padding 后的 m 显然是不现实的。

我们试图寻找一些实际应用中相关标准的支持，发现 PKCS #1 的 appendix B 中 (https://tools.ietf.org/html/rfc3447#appendix-B)，有以下描述：

> Six hash functions are given as examples for the encoding methods in this document: MD2 [33], MD5 [41], SHA-1 [38], and the proposed algorithms SHA-256, SHA-384, and SHA-512 [39]. For the RSAES-OAEP encryption scheme and EMSA-PSS encoding method, only SHA-1 and SHA-256/384/512 are recommended. For the EMSA-PKCS1-v1_5 encoding method, SHA-1 or SHA-256/384/512 are recommended for new applications. MD2 and MD5 are recommended only for compatibility with existing applications based on PKCS #1 v1.5.

显然，由于王小云教授的贡献，现在再采用较易找到碰撞的 SHA1 显得不够合理，故最后基于 SHA-256 编写了 MGF 的函数。至于为什么不采用 SHA3 中的 SHAKE，一方面是认为对运算效率的兼顾性不够，另一方面，它们采用了与 SHA1 和 SHA2 截然不同的 Sponge 结构，偏离了真实的 RSA-OAEP+。

最后编写的 MGF 函数如下，I2OSP 函数的作用是将整数转化成指定长度的字符串：

```
def i2osp(integer, size = 4):
    return "".join([chr((integer >> (8 * i)) & 0xFF) for i in reversed(range(size))])


def mgf(input, length, hash = hashlib.sha256):#based on SHA256
    counter = 0
    output = ""
    while(len(output) < length):
        C = i2osp(counter, 4)
        output += hash((str(input)+C).encode('utf-8')).hexdigest()
        counter += 1
    return int(output[:length],16)
```

②k0 与 k1：（课程网站上的 reference 打不开啦）由于编写的 MGF 要求的输入是 256 bit，故将 k0 直接设置为 256。接下来，考虑消息过长要进行分块的情况，设置 block 长度为 512 bit，这样一来，留给 k1 的就应该是 256 bit。但直接将 k1 设置为 256 bit 会出现问题：如果对一个 512 it 的消息 padding 256 个 0，那么最终 X||Y 的长度也将和模数 n 一样，是 1024 bit，这样 X||Y 就有可能在数值上大于 n，这是不合规的，后续运算会出现问题。基于这个考虑，将 k1 减少一位，设置为 255。

加密时，先进行 pack，将字符串转换成整数；再进行 padding 和 Feistel 加密；最后进行 RSA 加密。解密的步骤顺序则正好相反。

```
def encrypt(messageBlock, e, N):
    #pack
    messageBlock = pack(messageBlock)

    #encode
    r = random.getrandbits(256) # k0 = 256
    messageBlock = messageBlock << 255 # k1 = 255
    digestG = mgf(r, 96)
    X = messageBlock^digestG
    digestH = mgf(X, 32)
    Y = r^digestH
    W =  (X << 256) + Y

    #encrypt
    CBlock = quickPowNMod(W, e, N)

    return CBlock

def decrypt(CBlock, d, N):
    #decrypt
    W = quickPowNMod(CBlock, d, N)

    #decode
    X = W >> 256
    Y = W % (1 << 256)
    digestH = mgf(X, 32)
    r = Y^digestH
    digestG = mgf(r, 96)
    messageBlcok = X^digestG
    messageBlcok = messageBlcok >> 255 # k1 = 255

    #unpack
    messageBlcok = unpack(messageBlcok)

    return messageBlcok
```

测试： Python3.5 @ Ubuntu 16.04 LTS

```
lynx@lynx-virtual-machine:~$ python3 rsa-oaep.py
key size = 1024

public key = ( 65537 , 109150355354919567115028732352316948459002023671261839983
436854202087463279407280239354998387121619230332440228482213343211067658489239
171415983589815566849280387493309989564178315315216601429051202843714441037477
047353103808195927497487183422565795045238806926747892381486240463479570770666
596299457371 )

private key = ( 746799812947585850655032784332192802371431518290336894404276750
907365587904332277329245789046942532743808616485994924567052871791524849551384
560506984134784345621974629174430376392771066221466974332395766989399928351354
711666346535954032424193208380679682025720242902115380308267382012479739446294
8913 , 109150355354919567115028732352316948459002023671261839983436854202087463
279407280239354998387121619230332440228482213343211067658489239631714159835898
668492803874933099895641783153152166014290512028437144410374778504735310380819
274974871834225657950452388069267478923814862404634795707706660259629945737 1 )

Please input your message:
rsa oaep is obviously better than its textbook version

encrypted = 550465559211867791836099524175717041294268238150818391816706161053
450879057337890358944138093142621695986281060213366666479203694917246746429359
810984491615431128608517767281152558197669530101876617566546796249807184431099
621412317285011670175620678372492228701350648129843665581206680811938334990695 11


decrypted = rsa oaep is obviously better than its textbook version
```

相比于 Textbook RSA，RSA-OAEP 加入了随机因素 r，相当于盐，使其变为了概率性的加密算法，即使明文相同，每次的密文也不会相同。此外，使用 Feistel 结构，起到了良好的混淆与扩散的效果，使明文和密文间、密文和密文间规律更不易察觉，且 Feistel 作为对称加密的方法，运算效率表现良好，无需解密算法，增加的运算代价非常有限。

回到 PART2 的场景中，如果 Server 端采用 RSA-OAEP，随机值 r 和 Feistel 的存在使得 C 与 AES key 的逐位对应关系被破坏了，攻击者再想逐位推测是不可能的。将 PART3 的实现加入 PART2 的 Server 端，可以发现最后得出的 AES key 与真实的 AES key 并不相同，这完全是没有意义的猜测。