网络安全技术 project
实验报告

516030910451 柯晓荣

环境配置：

（1） part1 和 part3 我选择在 win10 系统上使用 python3.7shell 运行；
（2） 由于 part2 部分 1）需要使用 crypto 库，在 win10 系统上安装失败，在 linux 系统中安装成功 2)在 linux 系统中进行通信更为方便，故我选择在 linux 系统中运行测试 part2，python 版本为 2.7；
（3） 没有将 part1 和 part3 统一在 linux 系统中运行的原因是在 linux 中，import hashlib 库后，使用 shake_256()函数时，会报 shake_256()函数不在 hashlib 库中的错误，我尚未找到原因，因此先在 win10 系统中运行；

python 文件：

part1.py 为 part1 部分的代码，为 textbook RSA 的实现；
part2.py 为 part2 部分的代码，为 textbook RSA + AES 的功能测试；
part2_server.py 和 part2_client.py 为 part2 部分的代码，分别为 server 端和 client（attacker）端的代码，测试时先运行 part2_server.py 使其处于监听模式，再运行 part2_client.py 进行交互；
part3.py 为 part3 部分的代码，为 RSAOAEP 的实现；

一、Implement the textbook RSA algorithm

1、textbook RSA 实现细节

（1）快速幂

RSA 涉及到一些幂较大的指数运算，需要快速幂算法来优化时间复杂度和降低空间复杂度。关于快速幂，我的做法是将指数看作二进制数，从最低为开始查是否为 1，同时维持更新一个保存平方模的变量，若为 1，则乘上相应的平方模变量；

```
def fem(b, e, m):
    result = 1
    while e != 0:
        if (e & 1) == 1:
            result = (result * b) % m
        e >>= 1
        b = (b*b) % m
    return result
```

（2）素性测试

素性测试我没有使用第三方库，而是实现了 miller_rabin 素性测试；

Miller_rabin 素性测试的原理为：如果 p 是素数，x 是小于 p 的正整数，且 x^2 mod p = 1，那么要么 x=1，要么 x=p-1。因为 x^2 mod 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)。

在具体操作时，我们先对指数 num - 1 进行因式分解；

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

求得 num - 1 = s * 2 ^ t。接着进行平方模的判断，对如下操作执行 18 次（18 次是因为 miller_rabin 素性测试为概率算法，错误率小于 1/4，当为 18 次时，可将错误概率降至 1/2^36，为可接受范围）：随机取一个数 v，求得 v = v ^ s mod num 的结果，若为 1 或 num - 1，则通过素性测试；若不是，则迭代求 v = v ^ 2 mod num 的结果，直至 v = num - 1（此时通过素性测试）或 v = 1（此时说明存在一个数，其不是 1 或 num - 1，但其平方模 num 却是 1，若 num 为素数，这与上述关于素数的论证矛盾，因此 num 不是素数）或指数已经达到了 num - 1 且 v 不是 1 且 v 不是 num - 1（此时说明 v ^（num - 1）mod num != 1，因此 num 不是素数）；

```
    for trials in range(18):
        a = random.randint(2, num - 1)
        v = fem(a, s, num)
        if v != 1:
            i = 0
            while v != (num - 1):
                if i == t - 1:
                    return False
                else:
                    i = i + 1
                    v = fem(v, 2, num)
    return True
```

（3）判断互素和求逆元

判断是否互素我使用了欧几里得算法，求逆元使用了扩展欧几里得算法，原理在上课时已经讲过，练习中也做过，我便不再赘述；

（4）e，d 的选择

在本项目中，我将 e 固定为 65537，再通过随机出的 p,q,n 值，求出公钥 d；

（5）keysize 的选择

这里我选择实现了 1024bit 的 RSA 算法，因此需要将 n 固定为 1024bit。我的做法时先将 p 和 q 的长度总和定为 1024，并随机生成两个大素数 p 和 q，再将 p 和 q 相乘，由于 p 和 q 的长度总和为 1024bit，因此 n = p * q 有概率落在 1024bit，此时再判断 n 是否真实落在 1024bit，若为真，则输出相应 p，q，n；若为假，则重复寻找 p 和 q 即可；由于 p 和 q 的长度总和为 1024bit，经测试后可以发现有大约 60%的概率 n 会落在 1024bit，因此在重复该操作 4 次后，就有超过 95%的概率能得到合法的 p,q,n，属于可以接受的范畴；

```
def keyGeneration():
    n = 0
    a = fe(2, 1023)
    b = fe(2, 1024)
    while (n < a or n > b - 1):
        p = get_prime(p_length)
        q = get_prime(q_length)
        n = p * q

    fn = (p-1) * (q-1)
    e = 65537
    d = computeD(fn, e)
    return (n, e, d, p, q)
```

（6）字符串与十进制数的相互转化

考虑到之后在对 AES 的密钥进行加密时，AES 的密钥是字符串，但 RSA 计算的是十进制数，因此我在 part1 预先实现了字符串与十进制数间的相互转换；具体操作为，在字符串转十进制数时将字符串每个字符的 ascii 码乘上 2^8*i（其中 i 为字符长度减 1 再减该字符在字符数组中的序号），本质上也就是将 256 进制的数转为 10 进制的做法，由于每个字符为 8 个 bit，因此这种转换方式不会出现重复映射的问题；在十进制转字符串时，将十进制数不断模 256 再除以 256，得到的一个个数便是对应字符的 ascii 码，由字符串来接收即可；

```
def str2int(m):
    tmp = 0;
    for i in range (len(m)):
        tmp = tmp + ord(m[len(m) - 1 - i]) * fe(2, 8 * i)
    return tmp

def int2str(t):
    #print ("t:",t)
    tmp = t
    c = 0
    while (tmp != 0):
        tmp = tmp // fe(2, 8)
        c = c + 1
    m = ""
    m2 = ""
    for i in range(c):
        m += chr(t % fe(2, 8))
        t = t // fe(2, 8)
    for i in range(c):
        m2 += m[len(m) - 1 - i]
    return m2
```

（7）加密解密

Textbook RSA 的加密解密本质上是快速幂的过程，调用已经写好的快速幂算法即可；

```
def encryption(M, e, n):
    M = str2int(M)
    return fem(M, e, n)

def decryption(C, d, n):
    m = fem(C, d, n)
    m = int2str(m)
    return m
```

2、textbook RSA 实验结果

```
================== RESTART: C:\Users\kxr\Desktop\RSA\part1.py ==================
key_size: 1024

p: 13226929122302690942891010606693106313949952368633775749720795545072785175049
06408910681621

q: 78917654475528056923819991939027155873916217185435016244746753139558228776988
10164163808465117994222586477065697018782096989061383642274761676491479713937072
23372958806380923008795157168341158621644114185636847550897 37

public key: 65537

secret key: 75185594826023597535574707510836982865660827056136460068813794518329
47806899175540089145534944061526882791439198436431647973836290389846672774139600
93168113361218429972233123478981452653127176788862482372626042050504624049054514
51521177801367390609781378556942802483185805872655097750084784158195474635402273


n: 10438382222461833516976929575548614227447968695642442926659992906149261739662
13891263260949679397835481511906068510116235760937637506159528099974516159957169
23665006720312725661899940794449154760463232983861781181444309781089784871466663
03324211931439429326411676198579184674179360133538523375083280569162367 7

t: 12110110708306906509511411711109511510509511610 5
PlainText: it_is_our_AESkey

Encryption of plainText: 80043328742625123892870609901627070544287577288844923562
25645626705670136620822554008894466703553550761292658173624267696087689564894716
82501112660220166041769757798817222926003287114887645578835408499575981876306635
21682930450077241635189424806880183040594238265572130546866588369286495861510026
6609158893348

Decryption of cipherText: it_is_our_AESkey

The algorithm is correct: True
```

成功输出各个值并且算法验证通过；

二、perform a CCA2 attack on textbook RSA

1、设置 Server-client 通信格式

通信部分由于我的虚拟机无法连接网络，并且找不到原因，因此借用同学的电脑进行了最终测试。预先设定好 server 端和 client 端，并设置好端口和 ip，运行 part2_server.py，使 server 端一开始处于监听模式，直至运行了 part2_client，监听到 client 端发送了消息，然后正式开始交互；

Client 端：

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

halfkeyLength = 512
p_length = 300
q_length = 724
host = "192.168.146.128"
port = 25535
addr = (host, port)
byte = 1024
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Server 端：

```
byte = 1024
port = 25535
host = ""
addr = (host, port)


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

sock.bind(addr)
```

一开始时，由 server 端先发送，并将 n，d，C，和用 AESKEY 加密过的 message（我在该项目中预设的 message 为"I_AM_GROOT_GROOT"）发给 client，client 成功接收后，开始计算 Cb127 以及对应的 message127 并输出给 server，此后进入循环；

```
lynx@lynx-virtual-machine:~$ python part2_server.py
1024252510200998069811402253879892219546605914529974833666639085645706801134623061809654596432461676611575238567832920629142051082583310980823197145202435986962178074576281778648511275579355569309608158742984562547106903324506881416307887286180181421652080976998704681939420627519600150433693124841793490754111
65537
6163198594274476077473775739914267995976493148861602584300307490687107365388655286388762649644132909458027285015376672362691140408626401341070825490503237539328980514055315089901900207204665988612565152973173778417762324293075015470196406749379905470347023557577481354646342916893383660260924477912814029901
it_is_my_request
m:
what's_your_next
m:
it's_invalid_one
m:
it's_invalid_one
m:
what's_your_next
```

```
lynx@lynx-virtual-machine:~$ python part2_client.py
('count:', 127)
('key:', '\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '\xe1\xe9\xbao\xba\x9c\x90\xb0T\x1a\x93K2\xb3Xn')
Cb
506634555829889857390062854365364236813779705777203065777352370223667060960508158298407829207385272967340563969782535843924360530801137792271859191055445379396817141543508331549302157619806364441025463273156188825235507269092148478399795359325092591359010721248234962144476183391607540178872257334142328288688
msg:
ca4cff964c302115cd87983b6e43d8e4
('count:', 127)
('key:', '\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '\xe1\xe9\xbao\xba\x9c\x90\xb0T\x1a\x93K2\xb3Xn')
Cb:
123372201015612172341156675609560879901112798985773612201641211127033236234496344983817348627012386211524059805616243359659239405155230178650260933267514803373896870880727145306991123384727080968030037610243805312037656218397527558497894640551787970721425875684301737563112763862227952779938422804577037316156
msg:
6d0964d643a7728b587e170bb79d5622
('count:', 126)
('key:', '@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '\xf3\x1b\x80d\x11J\xb1\x8b\xcd\xee\x99=\xec\xfct\xa7')
Cb:
369850197550673285961158979152758795738391336794592746522417006435979673982707365707590368863629540760963412193185585984454106315979567185106731177987684115579734148429858308393208925397444380669948089868995755551583578595118168976367405144318094288225747836965389511177300952231792156103566509420382605180336
msg:
8c22eb27c9988615fddedd32e9c57594
('count:', 125)
('key:', ' \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '\nU\x87\xea\xdcd\xbf\x9d\xd2\x84\xf4\x8a\n}2\xa0')
Cb:
```

2、CCA2 attack 实现细节

CCA2 attack 本质上是一个猜测的过程，我选择每次猜测对应 bit 为 1；

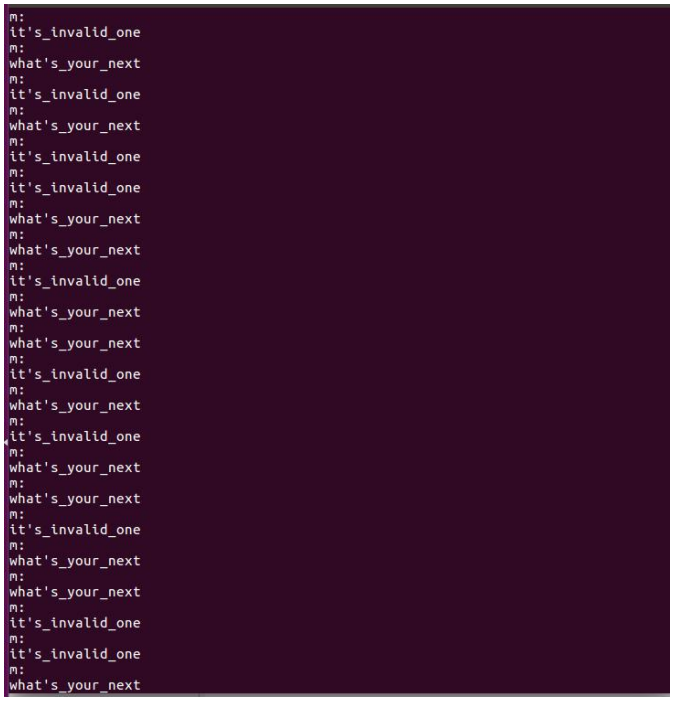
server 端需要将得到的 Cbx 进行解密，而后对 2^128 取模得到低 128 位的 keyx，再用 keyx 解密得到的 messagex 后与预先设置好的合法 request 进行比对，若有合法的 request 匹配成功，则输出“what's_your_next”询问下一条 request；若没有合法的 request 匹配成功，则输出“it's_invalid_one”表示该指令为非法指令；我在其中预设的合法指令为"it_is_my_request"；server 端本身不复杂，调用对应功能函数即可，不再赘述；

Client 端需要接收 server 端输出的“what's_your_next”或“it's_invalid_one”，若为“what's_your_next”，则说明该 bit 猜测正确，应该是 1，对此时的 key 作右移一位再加上 2^127 即可得到下一次猜测的 key；若为“it's_invalid_one”,则说明该 bit 猜测错误，应该是 0，由于此时 key 保存的还是上一次猜测的 key，即此时最高为是应该为 0 的 1，因此将现在的 key 异或 2^127，再右移一位后加上 2^127 即可得到下一次猜测的 key；接着再用相应的 key 计算 Cbx 的值和加密"it_is_my_request"后，输出给 server 端即可；如此往复 128 次，即可将 key 完全猜出；

```
if(m == m2):
        print("key:", key)
        if(len(key) == 16):
                decipher = AES.new(key, AES.MODE_ECB)
                print("message:", decipher.decrypt(mm))
        key = str2int(key)
        key = key / 2
        key = key + tmp
if (m == m3):
        key = str2int(key)
        key = key ^ tmp
        key = int2str(key)
        print("key:", key)
        if(len(key) == 16):
                decipher = AES.new(key, AES.MODE_ECB)
                print("message:", decipher.decrypt(mm))
        key = str2int(key)
        key = key / 2
        key = key + tmp
```

3、CCA2 attack 实验结果

Server 端部分实验结果：



Client 端部分实验结果：

可以在 count 为 8 的倍数时，看到 key 中的字符被一一破解出来；

```
942802537629160932398251585042854039086820674487416848221396505788096329387799303507045246525842815228717250151066684173667640864054343077697853349656087420153751239732524769312434803441560909545965576748859841056283880139398684
msg:
fba1adc492488f2608f85ca96b466efc
('count:', 96)
('key:', 'Skey\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '^\xdc\xd1\xb3\x98\xbe\x84\xed\x85\xf1\xd8\x12\xbd(\x15#')
Cb:
325814597011028230621381744116774698342702754361289749218391689323294139339430228983738987388342498901166145651185972791481832503391595711514069803229564763136195018702559584122533115388718094011744087792015158182848804759223622186346623720691989682632764961361038711113121662719063179421962741426204334207408970f48a3d5d672fec6f4634079cb7
('count:', 95)
('key:', '\xa9\xb5\xb2\xbc\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', '\xa9\xf7\xdb\x1d\x93\xda\xa3\x0b\x04Ne\xeb\x00\xf9\x16C')
Cb:
844933075988956511690404051318697437545307651739605127135547194382623750272669205243783498766912983671306871768127958096241691664155330928308214509921546276136975331200782264579260491989882113353580505901270183367470424722037506949368714341693076765454561028785465034701750945560425427113757011978427893821394d336fac3de457a099bcc9d0e5a2841
('count:', 94)
('key:', 'T\xda\xd9^@\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
('message:', 'Y\x0e1l\xf7\x10\xca\xd7\xe3z=Hq\xf2\xb1\xe8')
Cb:
637572464534315445481355512898407534045157828594654872238588932976193833869987091695910214979365556900417224017949596333580040018966545103574340429992654078184060247437597765889322127591051884224013048508589967728879084681392516124307134182816153891662933194615532324816635281017155191755569723279473976314916d4ef96126a32c09595cf854e
('count:', 93)
('key:', '\xaaml\xaf \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

```
250526480231780188738755869973642115840692307225697104855525584672101702468187882477994629050915386938280117699654815046549517824986725615274842484118394730579966882051801907554898453879875293779695889992365788184595962422766910
msg:
512ecd4ce5803e942cf0dd32824056e8
('count:', 42)
('key:', '}\xbd\xd5\xc9}\x05\x15M\xad\x95\xe4\x00\x00\x00\x00\x00')
('message:', '\x909\xfaz\xa7\x1eH\x92\x9a*\x10\xa3\x9f\x9f\xaf\x01')
Cb:
784658228220645782736987740911200496121607920853672926151585803772670439294151410850498331018557123037868742594500056542341447388034634159394427406283604857380715598886648283930590190943245543524777534212436364148753296406480109572521678865236016159025061050446474625051853806199310437217611079783190705918090msg:
10e591676a2c624e1b41a7e26a3654e9
('count:', 41)
('key:', '\xbe\xde\xea\xe4\xbe\x82\x8a\xa6\xd6\xca\xf2\x00\x00\x00\x00\x00')
('message:', '\xbc\x8b\x8b\x0f|\x8fQ.\xc7\x972Q\\\xe9\x8f\xde')
Cb:
188550649326950223634843482793826568481357487148433155748674451815635458193481178365857353393105377825887069122717369541322217353205454420959292317305234361547619934242024020230281615515175800666646777532831593820028397667396426325642625834562002973809415085561493403488315195246207148421994576155998423232510be879720cca7ffd917bae00d498fe4
('count:', 40)
('key:', '_our_AESkey\x00\x00\x00\x00\x00')
('message:', 'IK0:\xebd\xfa\xbbr\x07d-\xa7\xc4\xfa\x97')
Cb:
255785849909312970796537688790820397788284573529713378313600477631758362760206515781421220475261845563645378067082703029472300240214976690668056441858422916131299818468715868538927595237640340482703444736373837688557614100833812131271928683598872883603496604428808269463861768090711031136074748940842715546887msg:
59ec1f1bdb1bafe53a477ae2f23a8a44
('count:', 39)
('key:', '\xaf\xb7\xba\xb9/\xa0\xa2\xa9\xb5\xb2\xbc\x80\x00\x00\x00\x00')
```

Client 端：最终运行结果，当 count 为 0 时，成功破解 AESkey 为"it_is_our_AESkey"，并成功用该密钥破解出明文为"I_AM_GROOT_GROOT"

```
000535101043658113909446871528877736795641485252019241332461742188 4
msg:
79cbf8e785f80064b7806e93d908f016
('count:', 3)
('key:', 'K\xa2\xfbK\x9a\xfb{\xab\x92\xfa\n*\x9b[+\xc8')
('message:', '\x8ct\xb8QS\xf6\x01\xf80>\xaf<^Hz\x97')
Cb:
860671630172271503642483796427164933476286288285810862376453698150547109175263660
016235289819683190460035019344752647982570896425598338347593443713880374093014530
242954165438840413754615572529590511009127841717162920528351900219464083380163140
342775091573632944000589692694076460383720261059284820402099057671230
msg:
451b512f7a814772867e733860ab589c
('count:', 2)
('key:', '\xa5\xd1}\xa5\xcd}\xbd\xd5\xc9}\x05\x15M\xad\x95\xe4')
('message:', '#U\x0c\x8cQF\xfb\x9aY\xa9Z3\x00Uk\xc1')
Cb:
145707743860161979374627685828316254631748573479646081687882733258039987100824765
344639133626483394751027615302607169323302273426158939512744796481208760280137 4
730016261611462984719180971675047592719483238097172974622605488410204785217771320
897976751372628226770256588215876532698869575489129554419193630717060
msg:
d7fefbdf88e53ca9da5d533dccceb8b2
('count:', 1)
('key:', '\xd2\xe8\xbe\xd2\xe6\xbe\xde\xea\xe4\xbe\x82\x8a\xa6\xd6\xca\xf2')
('message:', '\xeb)\x99\x96{\x18\xe7x\x86\x07v\x84NCQ\xdb')
Cb:
616319859427447607747377573991426799597649314886160258430030749068710736538865520
863887626496441329094580272850153766723626911404086264013410708254905032375393280
980514055315089901900207204665988612565152973173778417762324293075015470196406740
937990547034702355757748135464634291689338366026092447791281402990 1
msg:
6d3efb7bc74588696e3924fa227cf65b
('count:', 0)
('key:', 'it_is_our_AESkey')
('message:', 'I_AM_GROOT_GROOT')
```

三、implement an RSA-OAEP algorithm and discuss why it can thwart such kind of attacks

1、OAEP 实现细节

OAEP 的目的是将消息加入随机数，作 padding 处理，形成一个新的消息，使得确定性的加密方案变成概率算法。加密算法的主体部分我沿用了 part1 的 textbook RSA。

（1）G，H 函数

关于 G，H 函数的选择，经过一定的搜索后我发现 G，H 函数可以直接使用 cryptography hash function 来发挥其 random orcales 的作用，但由于一般的 cryptography hash function 只能够输出定长的哈希值，因此在这里我们应该使用 mask generation function 来发挥 random orcales 的作用；mask generation function 就功能上来说类似于一般的哈希函数，具有单向性和抗碰撞性，且是确定性的算法，不同点在于 mask generation function 能够输出变长的值，在 OAEP 中正需要这个功能。

在 sha3 中，存在 sha3-keccak（）函数（简称 shake（）函数），该函数可以输出任意长度的哈希值，并具有单向性和抗碰撞性，且是确定性的算法；

我们进一步查阅 shake 函数的实现细节

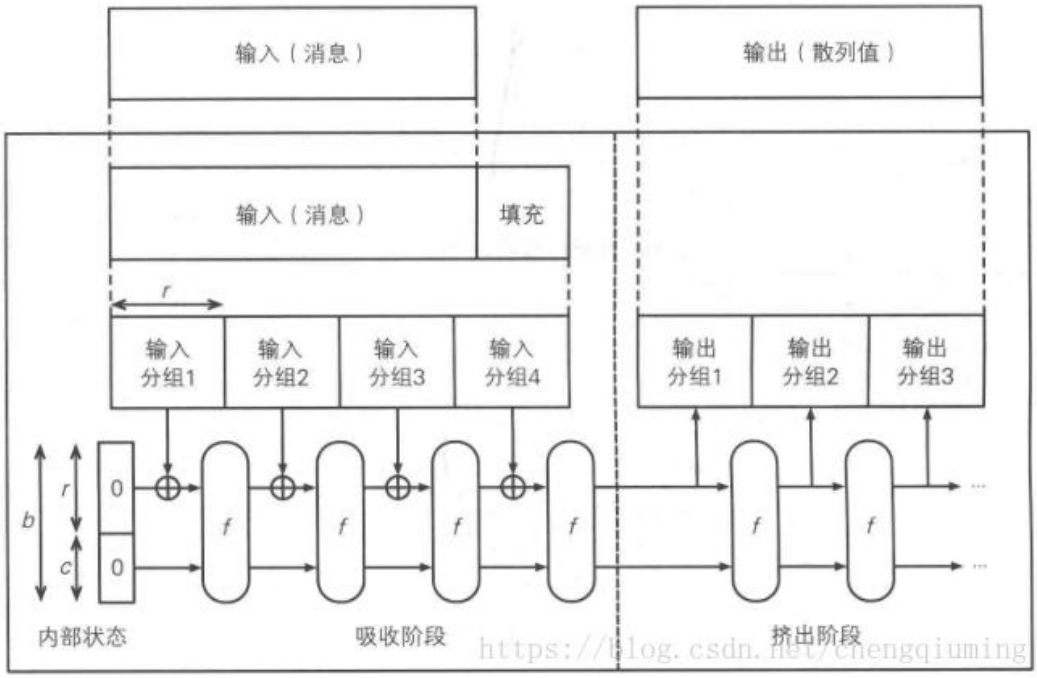
```c
/*************************************************
* Name:        shake256
*
* Description: SHAKE256 XOF with non-incremental API
*
* Arguments:   - unsigned char *output: pointer to output
*              - unsigned long long outlen: requested output length in bytes
*              - const unsigned char *input: pointer to input
*              - unsigned long long inlen: length of input in bytes
*************************************************/
void shake256(unsigned char *output,
              unsigned long long outlen,
              const unsigned char *input,
              unsigned long long inlen)
{
  unsigned int i;
  unsigned long nblocks = outlen/SHAKE256_RATE;
  unsigned char t[SHAKE256_RATE];
  uint64_t s[25];

  keccak_absorb(s, SHAKE256_RATE, input, inlen, 0x1F);
  keccak_squeezeblocks(output, nblocks, s, SHAKE256_RATE);

  output += nblocks*SHAKE256_RATE;
  outlen -= nblocks*SHAKE256_RATE;

  if(outlen) {
    keccak_squeezeblocks(t, 1, s, SHAKE256_RATE);
    for(i = 0; i < outlen; ++i)
      output[i] = t[i];
  }
}
```

上图中为 shake256 的 c 语言实现；



我们可以看到，shake 函数是通过一种 absorb-squeeze 结构，在 absorb 时，每次将消息异或后放入 f（）函数进行混淆和扩散，在 squeeze 时，通过定义第一个分组输出的长度来保证函数达到一定级别的抗碰撞性，此后若有长度需要，

则将该哈希值放入 f（）函数进行混淆和扩散后作为后续的长度输出，由此可以实现 mask generation function 能够输出变长的值的功能；

因此关于 G，H 函数，我选择使用 hashlib 库中的 shake_256（）函数，能够满足所需要的功能；

```
s2 = hashlib.shake_256()
s2.update(x.encode("utf8"))
tmp1 = str2int(s2.hexdigest(k0//8))
```

（2）k0，k1 选择

关于 k0 和 k1 的选择，由于我选择将 m 的长度定在 128bit（与 AESKEY 相对应），k0 定为 512bit，k1 定为 384bit；

2、OAEP 实验结果

```
================= RESTART: C:\Users\kxr\Desktop\RSA\part3.py =================
AES_KEY: 0000111122223333

x: t^ᏣᴮB'lfɔcᴬğƃƀʏµ5Ŭʒɖʃspȉs˝ɬ' Ó        ŸИ˜Đ1ↄ/ˎ`ðɨǿzˌ ŝÕ'□ɗɋɕf9c74134a79114312
82ce23541e982d50777b151f642356ad111d93b7a2d33fb471f8697b1

 y: ɱãuBŐ½É¼Üư□ɓŬ¯ɾ□ↄʉiåɲǯƃɪƃǴɸ&Ķ∑ʸ:ˌʌŸ·ˈğĘ1ÛÖᴹɬ₸□ɁJQᵃƛ℈±ᵛ'ᐟC9e06ba60a35
6e8b6d18f1acb0087919683c8d0d8f5b3ff4b2da0a5868e91eb70

m: 0000111122223333

check correctness: True
>>>
```

可以成功对 128bit 的 AESKEY 作 padding 并还原；

3、OAEP 安全分析

RSAOAEP 相比与 textbook RSA，对消息做了 padding 处理，使得消息具有了随机性，这样使得 RSAOAEP 成为了概率加密算法，而不是 textbook RSA 的固定加密算法，也就意味着同一明文在被加密时可能产生不同密文，更能保护明文的信息。

对 OAEP 模式下的 RSA 使用 CCA2 攻击是行不通的，原因在于当用 OAEP 对消息做 padding 后，原本 C = m ^ e mod n（m 为 AESkey），此时 C = (X||Y) ^ e mod n。攻击者在 CCA2 攻击中，猜测的是 m 或者 X||Y 的值，攻击者可以通过猜测 m 的每一 bit，并用相应的 key 来加密消息发送给 server 端，通过消息是否被成功解密来验证是否猜对，但在 OAEP 模式下，攻击者猜测 X||Y 的每一 bit 是没有意义的，因为在做 padding 时，消息 m（此处为 AESkey）已经被随机值 r 混淆了，猜测 X||Y 的每一 bit 为 0 还是 1，并不能得到相应的 key 来加密消息发给 server 端验证，因此无法恢复出消息 m（此处为 AESkey），因此对 OAEP 模式下的 RSA 使用 CCA2 攻击是行不通的。