RSA

April 21, 2020

1 Exercício 1 - Esquema KEM- RSA-OAEP

Neste exercício temos de implementar um esquema KEM- RSA-OAEP que deve : * **Inicializar cada instância recebendo como parâmetro obrigatório o parâmetro de segurança (tamanho em bits do módulo RSA-OAEP) e gerando as chaves pública e privada * Conter funções para encapsulamento e revelação da chave gerada. * Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.**

Ao longo deste documento vamos explicar o algoritmo de cada um dos processos acima expostos.

1.1 Gerar Parâmetros

Neste processo vamos gerar os parâmetros essenciais que são a base do algoritmo do RSA: - 2 números primos p e q; - n é o módulo para a chave pública e a chave privada; - phi é co-prime com n;

[1]:
```
def rprime(l):
        return random_prime(2**l-1,True,2**(l-1))
```

[2]:
```
l = 1024
q = rprime(l)
p = rprime(l+1)


N = p * q
phi = (p-1)*(q-1)
```

1.2 Aplicação OAEP

[ ]:
```
G = IntegerModRing(phi)
R = IntegerModRing(N)


def generateKeys():
    e = G(rprime(512)) #public exponent
```

```
    s = 1/e #private exponent
    return (e,s)

e,s = generateKeys()
```

[57]:
```
def OAEP(pk,m): ##OAEP encrypt
    a = R(m)
    cm = a**pk
    return cm


def OAEPinv(sk,cm): ##OAEP decrypt
    b=R(cm)
    dm = b**sk
    return dm
```

1.2.1 Funções auxiliares para as classes KEM e FOT

[ ]:
```
def generateRandomString(size): # gera uma string de tamanho variável de 0 e 1␣
 ↪aleatórios.
    i = 0
    stream = ""
    while(i<size):
        j = randint(0,1)
        stream = stream + str(j)
        i+=1
    return stream

def generateZeroString(size): # gera uma string de tamanho variável de zeros.
    i = 0
    stream = ""
    while(i<size):
        stream = stream + str(0)
        i+=1
    return stream

xor = lambda x, y: x.__xor__(y)

def concat(i,j):
    return i +j
```

1.3 KEM

Esta classe contém funções para o encapsulamento e revelação da chave gerada.

[92]:
```
class KEM:
    def encrypt(self,pk,x,n):
        return power_mod(x,ZZ(pk),n)

    def decrypt(self,sk,enc,n):
        return power_mod(enc,ZZ(sk),n)

    def encapsulation(self,pk): #enc
        x = randint(1,N-1)
        #enc = self.encrypt(pk,x,N)
        enc = OAEP(pk,x)
        k = hash(x)
        return (k,enc)

    def reveal(self,sk,enc):
        #x = self.decrypt(sk,enc,N)
        x = OAEPinv(sk,enc)
        k = hash(x)
        return k

    def enc(self,pk):
        a = generateRandomString(l)
        zero = generateZeroString(l)
        (k,enc) = self.encapsulation1(pk,concat(a,zero))
        return (k,enc)

    def encapsulation1(self,pk,a0):
        enc = OAEP(pk,int(a0))
        print('encapsulation1- enc: ' + str(enc))
        k = hash(enc)
        print('encapsulation1- k: ' + str(k))
        return (k,enc)
```

[93]:
```
kem = KEM()
(k,enc) = kem.encapsulation(e)
print('k: ' + str(k))
print('enc: ' + str(enc))

k1 = kem.reveal(s,enc)
print('k1: ' + str(k1))

teste = kem.enc(e)
print('teste: ' +str(teste))
```

```
k: 6890470786718148014
enc: 4450759539630231728616291271908919186014212944379263381464373927648961320730
48295306038123299917799528663629753262102097276143880827390443361181321322141 51
```

```
28417578043706815637237018297714284333792423736445186599213450554143116999864053
11838854288314781592999579706352481336419532460909671890048112519387015404889506
86809287305588363647732134792049930934209132710131273347033025457619713923404471
55733713159190424445325699584052764671187516883438647363517370219925103708475480
66577829328394568229709430179777464526476822965681034762470723379844482230902469
2910441428878724069556405173837480528790746551091702794954335
k1: 6890470786718148014
encapsulation1- enc: 29734667258673771269005928691833388948606088607124330786626
52029454664601465802210177876361860738017022353484482705955081532887641519464175
04315271452717517163679205383698118093207340723691132619074102465767438740513617
80595234311754580064631347516550236798925347894533723887283619728237721497907857
49947918394490656846863498206440569652846071175916553440815060889370371641242962
29362514828831799748425280141752803339690368847970312230487947784423115995210345
78701760977465071150523485557206894022329100703050870689434204733639196582989748
576258509431725448769449653250479799627922338400356222764806641714407565808857
encapsulation1- k: 1702680818923933257
teste: (1702680818923933257, 297346672586737712690059286918333889486060886071243
30786626520294546646014658022101778763618607380170223534844827059550815328876415
19464175043152714527175171636792053836981180932073407236911326190741024657674387
40513617805952343117545800646313475165502367989253478945337238872836197282377214
97907857499479183944906568468634982064405696528460711759165534408150608893703716
41242962293625148288317997484252801417528033396903688479703122304879477844231159
95210345787017609774650711505234855572068940223291007030508706894342047336391965
82989748576258509431725448769449653250479799627922338400356222764806641714407565
808857)
```

1.4 PKE

Esta classe implementa um esquema PKE(public key encryption) a partir do esquema KEM

```python
class PKE:
    def __init__(self):
        self.kem = KEM()

    def encrypt(self,pk,m):
        (k,enc) = self.kem.encapsulation(pk)
        c = xor(m,k)
        return (enc,c)

    def decrypt(self,sk,c):
        (enc,m1) = c
        k = self.kem.reveal(sk,enc)
        m = xor(m1,k)
        return m
```

1.5 FOT

Esta classe contém funções para implementar a transformação de Fujisaki-Okamoto que transforma PKE's que possuem IND-CPA seguro em outros PKE's que possuem IND-CCA seguros

[7]:
```
class FOT:
    def __init__(self):
        self.kem = KEM()

    def encrypt(self,pk,m):
        a = generateRandomString(l)
        (enc,k) = self.encrypt1(pk,m,a)
        return (enc,k)

    def encrypt1(self,pk,a,m):
        (enc,k) = kem.encapsulation1(pk,concat(a,hash(m)))
        #print('encrypt1 - enc: ' + str(enc))
        #print('encrypt1 - k: ' + str(k))
        aux1 = concat(str(a),str(m))
        #print('encrypt1 - aux1: ' + str(aux1))
        aux2 = xor(int(aux1),int(k))
        #print('encrypt1 - aux2: ' + str(aux2))
        return (enc,aux2)

    def decrypt(self,sk,c):
        (enc,m1) = c
        k = kem.reveal(sk,enc)
        print('decrypt- k: ' + str(k))
        am = xor(m1,k)
        a = am[0:l]
        m = am[l:]
        if(c == encrypt1(pk,a,m)):
            return m
        else:
            return false
```

[103]:
```
pke = PKE()

fot = FOT()
pk = e
sk = s

m = 123
c = pke.encrypt(pk,m)
(enc,t) = c
print('t: ' + str(t))
m1 = pke.decrypt(sk,c)
```

```
print('m1: ' + str(m1))

c = fot.encrypt(pk,m)
enc,k = c
print('k: '+ str(k))
c2 = fot.decrypt(sk,c)
print(c2)
```

```
t: -546709695712292876
m1: 123
encapsulation1- enc: 341387359076587594860282210985608568171242401055143697572015
874117646491618100924264745115453066060258000119484320748650970685243937059122460
792431195252040079799255902555431793399473631794309657770050033598259080864864
123111012225122615116619020417383403837750521633498437799717706394175280201138066
885079631894216545533380629835061523928460895741317105454147792648177533018968171
985166013968887700766656911284922026972527502927090704372145477569176876182108
272726478063817749606750216773695406443406211399910056244446028967202979314829679
99582189496239382084770226010925677482616364955403125011590153900097352690945
encapsulation1- k: 8837127428590078470
k: 1231110000111101010111000001110100101000101100100001000000011011001011111000011
001001101001000000110100111011011000011111001111100100001010110110111100000100
111011110000110100100101100001100111010011100110111110100010000100000000001000111
110101010100100110101110101000001010011000010110100100110010110111011100001110001
010011110001011100001011100011011100001101000110111111100000011110001110110000101
0110101101143064584989492950044178848168543238914452582310370939846555403973129
238945937970376427303737247869189746366228651572983837116606520436024587354732890
014930303531655746079239570757527261613175622987332415562424942372368575686854
768671246017611159618342689565638475838824198162294448999562745638300379261131829
525435812777188090978318771843605959689481474835440808907113302088604404997348246
153956083693074257644083564089209259834013967977005946723087541085021361681006521
886709262301501991651120620472570350248745567079545122829591564048929346733171274
48651045840161173282253986450241189820964639468789080068910128227


↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call last)

      <ipython-input-103-9377166cdeef> in <module>()
       15 enc,k = c
       16 print('k: '+ str(k))
  ---> 17 c2 = fot.decrypt(sk,c)
       18 print(c2)


      <ipython-input-101-c8eb1c71241d> in decrypt(self, sk, c)
       22         k = kem.reveal(sk,enc)
```

```
     23         am = xor(m1,k)
---> 24         a = am[Integer(0):l]
     25         m = am[l:]
     26         if(c == encrypt1(pk,a,m)):


TypeError: 'long' object has no attribute '__getitem__'
```

[33]:
```
a = randstring(5)
```

```
    ␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-33-ef85094e3f4f> in <module>()
----> 1 a = randstring(Integer(5))


NameError: name 'randstring' is not defined
```

[49]:
```
stream = generateRandomString(4)
print(stream)
```

```
0100
```

1.5.1 Nota

De realçar que tivemos problemas na implementação de um PKE seja um IND-CCA seguro.Mesmo assim, colocámos o código da nossa tentativa.

[ ]:
```

```