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

[3]:
```
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()
```

[4]:
```
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

[5]:
```
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.

[6]:
```
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)
```

[7]:
```
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: 512998086484580947
enc: 271537983504884533935799470973946660199515510574792841540322112468406127415
940111780264333346779288254565215776785050578840777786241814877996172269434690 51
```

```
57783440891671415850430401695583579009209500747476464501167942581528570850430056
93744435931887719498947582086346921089303243571433068794456845670162207943952806
62929905453305038334005682816919605810496584608593338570933108069685194835506361
95946754920998735466163281604337202633858294761763321776158493283438248647709317
84156054384151175805014205068877418475697859569420338268744079675566396525898830
75370588470646894411424919747380178034982221641204015094381484
k1: 512998086484580947
encapsulation1- enc: 95230829828199108989102096042593688252461981668275029625359
75618962961685145473700187163579522429735827116829757021541990895879157724865297
62574414826241607267983632514656042758039223371968006720002055075281640574170984
65618692831762119624919001313103722702645161642156461964200100181781285694789740
74238624613722609598779740316146767661397843770063634862382590100670497973632614
83229835547250928358840418415224861022768797437544069319989910115470990972729136
01589211466781102440978932641159109189108689711933303738390284342893581663079604
39775355820452314413509487393925504695199586857690895232489742853172113888135
encapsulation1- k: 1166647538513584938
teste: (1166647538513584938, 95230829828199108989102096042593688252461981668275029
62535975618962961685145473700187163579522429735827116829757021541990895879157724
86529762574414826241607267983632514656042758039223371968006720002055075281640574
74170984656186928317621196249190013131037227026451616421564619642001001817812856
94789740742386246137226095987797403161467676613978437700636348623825901006704979
73632614832298355472509283588404184152248610227687974375440693199899101154709909
72729136015892114667811024409789326411591091891086897119333037383902843428935816
63079604397753558204523144135094873939255046951995868576908952324897428531721138
88135)
```

1.4 PKE

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

[8]:
```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

[9]:
```
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
```

[ ]:
```
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)
```

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.