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.

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;

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

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


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

Aplicação OAEP

In [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()
```

In [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
```

Funções auxiliares para as classes KEM e FOT

In [5]:
```
def generateRandomString(size): # gera uma string de tamanho variáv
el 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
```

KEM

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

In [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)
```

In [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: 271537983504884533935799470973946660199515510574792841540322112468406127415940111780264333346779288254565215776785050578840777786241814877996172269434690515778344089167141585043040169558357900920950074747646450116794258152857085043005693744435931887719498947582086346921089303243571433068794456845670162207943952806629299054533050383340056828169196058104965846085933385709331080696851948355063619594675492099873546616328160433720263385829476176332177615849328343824864770931784156054384151175805014205068877418475697859569420338268744407967556639652589830753705884706468944114249197473801780349822216412040150943814841
k1: 512998086484580947
encapsulation1- enc: 952308298281991089891020960425936882524619816682750296253597561896296168514547370018716357952242973582711682975702154199089587915772486529762574414826241607267983632514656042758039223371968006720002055075281640574170984656186928317621196249190013131037227026451616421564619642001001817812856947897407423862461372260959877974031614676766139784377006363486238259010067049797363261483229835547250928358840418415224861022768797437544069319989910115470990972729136015892114667811024409789326411591091891086897119333037383902843428935816630796043977535582045231441350948739392550469519958685769089523248974285317211388813
encapsulation1- k: 1166647538513584938
teste: (1166647538513584938, 952308298281991089891020960425936882524619816682750296253597561896296168514547370018716357952242973582711682975702154199089587915772486529762574414826241607267983632514656042758039223371968006720002055075281640574170984656186928317621196249190013131037227026451616421564619642001001817812856947897407423862461372260959877974031614676766139784377006363486238259010067049797363261483229835547250928358840418415224861022768797437544069319989910115470990972729136015892114667811024409789326411591091891086897119333037383902843428935816630796043977535582045231441350948739392550469519958685769089523248974285317211388813)
```

PKE

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

In [8]:
```
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
```

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

In [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
```

In [ ]:
```
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)
```

Nota

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