Exercício 2

2. Use o SageMath para,

1. Construir uma classe Python que implemente um KEM-RSA. A classe deve

i. Inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits do módulo RSA) e gere as chaves pública e privada.

ii. Conter funções para encapsulamento e revelação da chave gerada.

2. Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.

Parte 1

Vamos começar por implementar o algoritmo criptográfico RSA. Este algoritmo utiliza 2 tipos de chaves, uma chave pública usada para na cifra da mensagem e uma chave privada usada na decifra da mensagem. Para gerar estas chaves é necessário: gerar 2 números primos aleatórios (***p*** e ***q***) de tal forma grandes que a recuperação de um desses primos a partir do produto dos dois é difícil, depois é necessário definir o módulo ***n*** que correspode à multiplicação dos dois primos, após a geração destes calculamos o valor de ***n*** que corresponde à multiplicação dos dois. Calculamos o valor de *fi de n* que corresponde à multiplicação entre ***p-1*** e ***q-1***. De seguida definimos o valor de ***e*** como sendo um valor entre 0 e ***fi de n***, mas tem também de ser primo em relaçao a ***fi de n***. Por fim calculamos o valor de d que corresponde à inversa multiplicativa de ***e***. Após este processo definimos a funções: encrypt_rsa que gera um mensagem cifra através da exponenciação modular, a base é a mensagem, o expoente é o ***e*** e usamos o módulo de ***n***, obtidos pela chave pública, e decrypt_rsa que decifra uma mensagem utilizando também a exponenciação modular, a base é a mensagem cifrada, o expoente é a chave privada e usamos o módulo de ***n***, obtido pela chave pública. Por fim criamos as funções responsáveis por encapsular e revelar a chave partilhada. Para auxiliar estas funções criamos também um função qeu deriva uma chave a partir de outra. Para a função de encapsulamento começar por gerar um número aleatório entre 1 e ***n***, que corresponde ao ***n*** da chave pública do RSA, depois ciframos este número através do RSA e geramos a nossa chave partilhada. Por fim, retornamos o número cifrado, a chave e o valor do salt. Para a função que revela a chave, deciframos o número aleatório através do RSA e geramos a chave partilhada através deste número, esta função retorna a chave partilhada.

In [1]:
```
import os
import hashlib
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.hashes import Hash,SHA256


class KEM_RSA:
    def init(self,param):
        self.param = param
        #Cálculo do dois números primos
        p = random_prime(2^(param/2) - 1,false, 2^(param/2-1))
        q = random_prime(2^(param/2) - 1,false, 2^(param/2-1))

        #Cálculo de n
        n = p*q

        #Cálculo do fi de n
        fiN = (p-1)*(q-1)

        #Definição de um número entre 1 e fi de n, que seja primo do fi d
        e = self.calcE(fiN)

        #Cálculo de d
        d = self.calcD(e,fiN)

        self.public_key = (e,n)
        self.private_key = d

        #print(p,q,n,fiN,e,d, mod(d * e, fiN), mod(1, fiN))

    #Função que gera um número aleatório entre 1 e fi de n, tal que estes
    def calcE(self,fiN):
        e = ZZ.random_element(fiN)

        while(gcd(e,fiN)!= 1):
            e = ZZ.random_element(fiN)
        return e

    #Função que calcula o inverso multiplicativo de e em módulo de fi de
    def calcD(self,e,fiN):
        bezout = xgcd(e, fiN)
        d = Integer(mod(bezout[1], fiN))
        return d


    def encrypt_rsa(self,message,public_key):
        e,n = public_key
        cypherText = power_mod(message, e, n)
        return cypherText

    def decrypt_rsa(self,cypherText):
        e,n = self.public_key
        text = power_mod(cypherText, self.private_key, n)
        return text



    def kdf(self, m, salt):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt
```

```
        return r


    def encapsule(self,r,public_key,salt):
        r_bytes = int(r).to_bytes(int(r).bit_length() + 7 // 8, 'big')
        ct = self.encrypt_rsa(r,public_key)
        ct_as_bytes = int(ct).to_bytes(int(ct).bit_length() + 7 // 8, 'bi


        shared_key = self.kdf(r_bytes,salt)

        return shared_key,ct_as_bytes,salt

    def reveal(self,ct,salt):
        ct_int = int.from_bytes(ct, 'big')
        m = self.decrypt_rsa(ct_int)
        m_bytes = int(m).to_bytes(int(m).bit_length() + 7 // 8, 'big')
        return self.kdf(m_bytes,salt)
```

In [2]:

```
salt = os.urandom(16)
Bob_rsa = KEM_RSA()
Alice_rsa = KEM_RSA()
Bob_rsa.init(128)

alice_pub_key = Bob_rsa.public_key
r = Bob_rsa.random_Number(Bob_rsa.public_key)

k,ct,salt = Alice_rsa.encapsule(r,alice_pub_key,salt)
print("Chave partilhada: ", k)

key  = Bob_rsa.reveal(ct,salt)
print("Chave partilhada: ",key)
```

```
Chave partilhada:  b'jsT\r\x0e\xb1[\xb9o9p/x\xbdd\xb8\x82\xa7\x84DW\x0c\x
15R\xc2\x0ev\x9cS\xef\x802'
Chave partilhada:  b'jsT\r\x0e\xb1[\xb9o9p/x\xbdd\xb8\x82\xa7\x84DW\x0c\x
15R\xc2\x0ev\x9cS\xef\x802'
```

Parte 2

In [3]:
```
class PKE:
    def init(self,kem):
        self.kem = kem

    def hash_g(self, msg):
        h = hashes.Hash(hashes.SHA3_256())
        h.update(msg)
        return h.finalize()

    def hash_h(self, msg):
        h = hashes.Hash(hashes.SHA3_256())
        h.update(msg)
        return h.finalize()

    def encrypt(self,msg,pub_key,salt):
        r = self.hash_h(msg)
        y = bytes(a ^^ b for a, b in zip( msg, self.hash_g(r)))

        concat = y + r
        concat_int = int.from_bytes(concat,'big')

        cypher_r,shared_key,salt = self.kem.encapsule(concat_int,self.kem

        c = bytes([a ^^ b for a, b in zip(shared_key, r)])

        return y,cypher_r,c


    def decrypt(self, y, cypher_r, c,salt):
        #Obtemos a chave com o KEM definido antes
        shared_key = self.kem.reveal(cypher_r,salt)

        #Aplicamos o XOR com a chave simetrica de ambos para decifrar
        r = bytes([a ^^ b for a, b in zip(c, shared_key)])
        concat = y + r
        concat_int = int.from_bytes(concat, "big")

        new_cypher_r, new_shared_key,salt = self.kem.encapsule(concat_int

        if shared_key != new_shared_key:
            pass
        else:
            if cypher_r != new_cypher_r:
                print("Mensagem não coincide com a inicial!")
            else:
                message = bytes([a ^^ b for a, b in zip(y, self.__hash_fu
                print("Mensagem recebida: ", message)
```

In [4]:
```
    salt = os.urandom(16)

    first_RSA = KEM_RSA()
    first_RSA.init(1024)

    second_RSA = KEM_RSA()
    second_RSA.init(1024)


    first_PKE = PKE()
    first_PKE.init(first_RSA)
    y,cypher_r,c = first_PKE.encrypt(b"Teste da Mensagem TOP SeCreta", se

    second_PKE = PKE()
    second_PKE.init(second_RSA)
    second_PKE.decrypt(y,cypher_r,c,salt)
```

In [ ]: