Exercicio 2

Com o objetivo de desenvolver uma key encapsulation mechanism baseado no RSA era necessário desde logo perceber o funcionamento do RSA, acompanhadas de funções de encapsulamento e desencapsulamento de chaves simétricas a partir de chaves publicas e privadas, geradas pelos parâmetros definidos no RSA - esquema assimétrico de criptografia.

Dada esta imposição foi necessário anuir alguns factos.

Em primeiro lugar espera-se a geração de uma chave pública e privada.

Estas chaves têm parâmetros definidos, dos quais a mesma depende para a sua geração.

Assim :

- n é o parâmetro de segurança
- Chave pública é definida por (n,e)
- Chave privada definida por (p,q,d)
  #### Tendo isto em conta, e consultando os apontamentos do docente e de alguns documentos online
- Gera-se um número primo aleatório para o parâmetro q do RSA
- Gera-se um número primo aleatório para o parâmetro p do RSA
- O parâmetro n é calculado pela expressão n= p * q
- O phi, que permite calcular o expoente da chave pública é calculado por phi(n) = (p-1) * (q-1)
  #### De forma a calcular o parametro d é necessário fazer algumas contas auxiliares
- Escolher um inteiro "e" entre 1 e phi(n) em que "e" e "phi(n) sejam relativamente primos
- Calcular o parâmetro auxiliar "d" tendo como base o algoritmo de euclides associado à formula d *e* - *k* phi(n) = 1
- Visto que é necessário descobrir um "k" e um "d" recorremos à identidade de bézout: gcd(x,y) = sx + ty a qual existe no sagemath com o nome xgcd(x,y) retornando um triplo (g,s,t) . A partir daqui infere-se que d será o resultado de d = mod(s,phi(n)), visto que 1 < d < phi(n)

Encapsulamento e geração da chave

Recebe-se a chave pública como input (e,n) e para gerar uma chave simétrica devidamente encapsulada foi necessário:

- gerar um número inteiro aleatório entre 1 n-1
- calcular o encapsulamente da chave através da atribuição c <- r^e mod n
- gerar a chave simetrica a partir da atribuição do valor resultante de uma KDF aplicado ao parâmetro r. Entendemos usar um KDF com uma função de hash SHA-256, sendo dessa forma necessário adicionar um salt! Isto levará a que tenhamos como resultado o par (w,salt + c)

Desencapsulamento da chave

Recebemos como argumento o resultado do encapsulamento da chave, contendo basicamente o (salt + c). De seguida:

- Obtém-se o valor de r através de:
  - r <- c^d mod n
- O r permite-nos gerar a chave simétrica usando mais uma vez um kdf cujo resultado é atribuido ao parametro w:
  - r <- KDF(r)

In [1]:

```
import math
import os, hashlib



class KEM_RSA(object):

    def __init__(self,sec_par):

        # Mersenne numbers
        # If p is prime and Mp=2p-1 is also prime,
        #  then Mp is called a Mersenne prime

        self.q = random_prime(pow(2,sec_par-1),pow(2,sec_par)-1)
        self.p = random_prime(pow(2,sec_par+1-1),pow(2,sec_par+1)-1)
        self.n = self.q * self.p


        self.phi = (self.p-1)*(self.q-1)

        self.e = ZZ.random_element(self.phi)

        while gcd(self.e, self.phi) != 1:
            self.e = ZZ.random_element(self.phi)
```

```
        self.bezout = xgcd(self.e, self.phi)



        s = self.bezout[1]
        self.d = Integer(mod(s, self.phi))



    def encapsula(self, e, n):

        r = ZZ.random_element(n)

        salt = os.urandom(16)

        key_encapsulation = Integer(power_mod(r, e, n))

        w = hashlib.pbkdf2_hmac('sha256', str(r).encode(), salt, 100000)

        k = (w,salt + str(key_encapsulation).encode())

        return k

    def revelacao(self, cs):

        salt = cs[:16]
        c = int(cs[16:].decode())

        r = Integer(power_mod(c, self.d, self.n))

        w = hashlib.pbkdf2_hmac('sha256', str(r).encode(), salt, 100000)

        return w
```

In [2]:

```
N = 1024

# Chave publica: (n,e)
# Chave privada: (p,q,d)
# Inicializacao da classe responsavel por implementar o KEM-RSA
kemrsa = KEM_RSA(N)

# Verificar que ed == 1 (mod φ(n))
#print(mod(kemrsa.e * kemrsa.d, kemrsa.phi))

# Procede-se ao encapsulamento
(w,c) = kemrsa.encapsula(kemrsa.e, kemrsa.n)

print("Chave devolvida pelo encapsulamento: ")
print(w)
print("\n'Encapsulamento' da chave: ")
print(c)

# Procede-se ao desencapsulamento
w1 = kemrsa.revelacao(c)
print("\nChave devolvida pelo desencapsulamento: ")
print(w1)

# Verificar se a chave devolvida pelo desencapsulamento é igual à que foi
if w == w1:
    print("OKAY, as chaves são iguais!!!")
else:
    print("NOP, as chaves são diferentes!!!")
```

```
Chave devolvida pelo encapsulamento:
b'E\xa8E\x1eZ\xfe\xcb\xbb\x80V\xdd8\xf2>\x92g\xce\xf57\x84\xa2\x7f0\xc5\x83
\x8e\x07\xd0\xd9\xb9\xeeD'

'Encapsulamento' da chave:
b'"\xc7\xec,\xf8p\xe6X\xac\xbf\xe9\x93+%\xaaT6012670229587620388241689824488
4521838426332050173894765986764852030396554391991984594678846527467069353 43
6680951565740882114009232875399902633959625757740826663050767142431289796 36
6035604957407825208550910812502511521572570735392001197634338089842111831 48
7437532334007753306016459473288846216907194326622586021699602522935218341 17
8730211027372438436835554616015968471274323993139304519267731716121960056 33
9943046081079595898034368734435185428342409497328722334819410172052637054 51
2063735175358563251936990960357228437548450072040830912349387504571643113 45
4994100872035622726336324188431380178660151411983357060868 25'

Chave devolvida pelo desencapsulamento:
b'E\xa8E\x1eZ\xfe\xcb\xbb\x80V\xdd8\xf2>\x92g\xce\xf57\x84\xa2\x7f0\xc5\x83
\x8e\x07\xd0\xd9\xb9\xeeD'
OKAY, as chaves são iguais!!!
```

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

Como indicado no enunciado, usamos a classe **KEM-RSA** criada anteriormente para a geração das chaves pública e privada.

A inicialização desta nova classe **PKE_IND_CCA** é, então, também a inicialização da classe **KEM_RSA** com um determinado N (que determina o tamanho dos números primos p e q no RSA) de forma a, posteriormente, obter as chaves.
Definimos também uma função para cifra e outra para decifra, para as quais necessitamos de umas outras auxiliares - d, h, xor - que vamos descrever de seguida:

A função ***h*** gera um inteiro aleatório entre 0 e n, sendo que n é recebido pela função como argumento.
A função ***f*** gera uma chave e o seu encapsulamento, precisando, para isso, de receber como argumento o seguinte:

- a *seed* para passar à KDF;
- um *salt* para a KDF;
- a chave pública (e, n).

Por fim, a função ***xor***, como o nome indica, realiza a operação binária de *exclusive or* entre dois *arrays* de bytes, byte a byte.

Finalmente, passamos a explicar as funções de cifra e decifra.

A função ***cifra*** é, como sugerido pelo nome, usada para cifrar uma mensagem. Então, esta função recebe como argumento a mensagem a cifrar e uma chave pública (e, n) a ser usada para a cifragem e efetua os seguintes passos:

- utiliza a função auxiliar h para gerar um número inteiro aleatório r em que $0 < r < n$;
- calcula o SHA-256 do número r gerado, colocando o resultado em g;
- efetua o XOR entre a mensagem a cifrar e g, com recurso à função auxiliar xor e guarda em y;
- é calculado yi, que é o *hash* de y;
- é gerada uma chave k e o seu encapsulamento w, usando a função f com os seguintes parâmetros:
  - a *seed* é a concatenação de r com yi;
  - e e n são o (e, n) recebidos como parâmetro;
  - $salt$ é um conjunto de 16 bytes gerados aleatoriamente.
- calcula o XOR entre r e k e coloca em c;
- finalmente, devolve y - a ofuscação da mensagem, w - a chave encapsulada, c - a ofuscação da chave.

A função ***decifra*** recebe como argumentos os resultados da cifra, a mensagem ofuscada y, a chave encapsulada w e a chave ofuscada c. O procedimento de decifra é o seguinte:

- desencapsular a chave w recorrendo à função da classe KEM-RSA e colocar em k;
- calcular o XOR entre k e c;
- obter o *salt* nos 16 primeiros bytes da chave w;

- calcular yi, o *hash* de y;
- utilizar a função f para gerar uma chave e o seu encapsulamento a partir dos parâmetros y e r;
- testar se o resultado da opração anterior é igual a (k, w);
- se sim:
  - calcula $g(r)$, sendo g a função de hash SHA-256 ;
  - calcula o XOR entre y e $g(r)$.
- se não, é gerada uma exceção.

In [3]:

```
import os, hashlib
# The Fujisaki-Okamoto (FO) transformation (CRYPTO 1999 and Journal of Cryp
# turns any weakly secure public-key encryption scheme into a strongly (IN
# the random oracle model.

# Basicamente usa-se a KEM_RSA para gerar as chaves publica e privada
# ind-cca é Chosen Ciphertext Attack


class PKE_IND_CCA(object):

    def __init__(self, N):
        # N é o tamanho usado para os primos p e q no RSA
        self.kem = KEM_RSA(N)


    def h(self, n):
        # Gerar um inteiro aleatorio r tal que 0 < r < n
        r = ZZ.random_element(n)
        return r


    # Funcao que gera a chave e o seu encapsulamento recebendo como parame
    # a seed do KDF,
    # a chave publica (e,n) e
    # um salt usado no KDF
    def f(self, seed, e, n, salt):
        # Criptograma usado para o encapsulamento da chave (c = seed^e mod
        c = Integer(power_mod(int(seed.decode()), e, n))

        # Geracao da chave simetrica a partir do seed (W = KDF(r))
        w = hashlib.pbkdf2_hmac('sha256', seed, salt, 100000)

        return (w, salt + str(c).encode())


    # XOR de 2 arrays de bytes, byte-a-byte!
    # A mensagem(data) deve ser menor ou igual á chave(mask)!
    # Caso contrario, a chave ou mask é 'repetida' para os bytes seguintes
    def xor(self, data, key):
        masked = b''
        ldata = len(data)
        lmask = len(key)
        i = 0
        while i < ldata:
            for j in range(lmask):
                if i < ldata:
                    masked += (data[i] ^^ key[j]).to_bytes(1, byteorder='b
```

```
                i += 1
            else:
                break
        return masked


    # Funcao usada para cifrar; recebe a mensagem m e uma chave publica (e,
    def cifra(self, m, e, n):
        # Gerar um inteiro aleatorio r tal que 0 < r < n
        r = self.h(n)
        # Calculo do g(r), em que g é uma função de hash (no nosso caso, si
        g = hashlib.sha256(str(r).encode()).digest()
        # Efetuar o calculo do XOR entre: x e g(r)
        y = self.xor(m, g)
        yi = Integer('0x' + hashlib.sha256(y).hexdigest())
        # Gerar o salt para derivar a chave
        salt = os.urandom(16)
        # Calcular (k,w)
        (k,w) = self.f(str(yi + r).encode(), e, n, salt)
        # Calcular o XOR entre k e r
        c = self.xor(str(r).encode(), k)

        return (y,w,c)


    # Funcao usada para decifrar; recebe a mensagem ofuscada, o encapsulam
    def decifra(self, y, w, c):
        # Fazer o desencapsulamento da chave
        k = self.kem.revelacao(w)
        # Calcula o XOR entre c  e  k
        r = self.xor(c, k)
        # Buscar os 16 primeiros bytes de w para obter o salt
        salt = w[:16]
        yi = Integer('0x' + hashlib.sha256(y).hexdigest())
        # Verificar se (w,k) != f(y+r)
        if (k,w) != self.f(str(yi + int(r)).encode(), self.kem.e, self.kem
            # Lancar excecao
            raise IOError
        else:
            # Calculo do g(r), em que g é uma função de hash (no nosso cas
            g = hashlib.sha256(r).digest()
            # Calcular o XOR entre y e g(r)
            m = self.xor(y, g)

        return m
```

In [4]:

```
pke = PKE_IND_CCA(1024)

#message = os.urandom(32)
message = b"TP1 de Estruturas Criptograficas"

print(b"Mensagem original: " + message)

(y,w,c) = pke.cifra(message, pke.kem.e, pke.kem.n)

print("\nTexto cifrado:")
print(b"Y = " + y)
print(b"W = " + w)
print(b"C = " + c)

print("\nTexto decifrado:")

try:
    message1 = pke.decifra(y,w,c)
    print(b"Texto decifrado: " + message1)
    if message == message1:
        print("[A mensagem decifrada é igual à original!!!!]")
    else:
        print("[A mensagem decifrada é diferente da original!!!!]")
except IOError as e:
    print("Erro ao decifrar a mensagem!!!!")
```

```
b'Mensagem original: TP1 de Estruturas Criptograficas'

Texto cifrado:
b'Y = X\xc1MVX\x17\xdd\xc1\x0f\x88n:"V%\xac;\x859\xab\xa8\xa3\xa00\x1a\x02\
x89_\xb8\xc6{\xd2'
b'W = a\xb6\x974\xab\xbd\xa0\xe5\xf32\xd9\x85\x98H\xbd\xc575945410147903844
544303231870253508191215767410234808336087859182268428276512748518403398745
027875151603276772073406631974174557338352715559876022459184599148782769340
388577569967262026184362257816785283447158982610111869507447081166067270237
386391565695623704308383590097505829691699609233729108909093175693309142928
433690015994768161959491711074414664097186056638808719225655521975105916498
665273485243788623953155279799600496803335639816677117161843725944558993255
311571502756178493134462140796758011892251212755375845485535215746526884549
940822981376859394424582170187404407498709288324392689334038433087577610 6'
b'C = Q\xa9F\xe9:=#em\xb6\xe5\x06\x9b\xea\x8fJ\xe0\xa3`\x0f\x17o\x9c\xd8\xf
1\xbf\xe9\xb4\xd6{\xf4.^\xadA\xe9;;+eh\xb0\xe3\t\x92\xe4\x8fH\xe5\xa7a\x03\
x1eo\x94\xdb\xf1\xb2\xeb\xb3\xd4u\xf2$[\xa6B\xef:=-gj\xb1\xe0\x00\x9d\xe4\x
84M\xef\xa4g\x08\x10b\x94\xd4\xfb\xbb\xeb\xbd\xd4v\xf2,X\xad@\xea2?#`l\xb7\
xe7\x01\x99\xed\x87A\xee\xada\x0c\x12e\x98\xd5\xf2\xbf\xea\xb7\xd3w\xf5.\\\
xa8@\xeb;<,aj\xb2\xef\x06\x98\xe5\x85O\xee\xa5i\x0f\x10g\x9e\xdb\xf0\xbb\xe
c\xbc\xddq\xfe-_\xa7E\xeb31+bk\xb1\xee\x02\x9c\xea\x8fN\xe7\xa1d\x0b\x12d\x
9d\xdb\xf1\xbb\xed\xb7\xd1z\xfe)[\xa7C\xee:>.fc\xb1\xe4\x08\x98\xee\x80L\xe
7\xaci\x0e\x10n\x9e\xdd\xfa\xba\xe5\xb5\xd7r\xf1%X\xa7@\xeb>1"bm\xb2\xe1\x0
3\x98\xe9\x81J\xe6\xa6f\x03\x16e\x95\xdf\xf5\xb9\xe4\xb1\xd7v\xfe$P\xadA\xe
a2>#do\xbe\xe6\x02\x9e\xef\x83K\xe5\xa7e\x0c\x12f\x94\xdd\xf7\xbb\xef\xb7\x
d6z\xf6/Z\xacA\xe628)fn\xbe\xe5\x08\x9d\xeb\x8fK\xe6\xa3c\t\x12f\x9e\xde\xf
2\xb8\xe4\xbc\xd6v\xf3*Z\xaaE\xee80/mh\xb1\xe7\t\x93\xe9\x81I\xe1\xa0c\x0f\
x17`\x9b\xdf\xf4\xbd\xee\xb0\xd1v\xf1/P\xadA\xe8?<*gn\xb5\xe4\x06\x93\xed\x
8eM\xe0\xa2a\x0c\x1ef\x9e\xdf\xf1\xbc\xee\xb4\xd5u\xf3%\\\xa7G\xef3<,ch\xb5
\xef\x01\x9d\xe5\x83@\xe5\xa5`\r\x13n\x9c\xdf\xfa\xb9\xe9\xb7\xd0q\xf3)_\xa
7O\xed3:"bl\xb6\xe5\x04\x98\xe5\x8fL\xe0\xa5g\t\x15g\x9f\xd8\xfb\xbc\xe5\xb
0\xdcp\xf1%_\xaf@\xe98>.dc\xb6\xe7\x02\x98\xed\x83N\xe6\xa4h\t\x1eb\x9b\xde
\xfb\xb8\xe9\xb5\xd4q\xf2$]\xa9D\xec3:)`o\xb4\xe0\x00\x99\xec\x81L\xe7\xa6d
\x0e\x15g\x9b\xdd\xf6\xbf\xe9\xb0\xd1q\xf6+Y\xaeB\xeb80#fl\xbe\xe6\x03\x98\
xeb\x84H\xe5\xa6h\r\x13a\x9c\xda\xf7\xba\xe5\xb4\xd6z\xfe([\xaeC\xe7<0"al\x
b4\xe6\x04\x9c\xec\x8fM\xe5\xa6c\r\x1ef\x94\xd4\xf1\xb9\xe9\xb1\xd7v\xf5-Y\
xabE\xee2;#ek\xb4\xe2\x00\x93\xe5\x81L\xe6\xa2g\x0e\x14n\x9e\xd9\xf2\xba\xe
9\xb6\xd3r\xf5%^\xaeC\xea30#'

Texto decifrado:
b'Texto decifrado: TP1 de Estruturas Criptograficas'
[A mensagem decifrada é igual à original!!!!]
```