Trabalho prático 2

Grupo 5:

- Duarte Oliveira \<pg47157>
- Melânia Pereira \<pg47520>

Post-Quantum Cryptography na categoria de criptosistemas PKE-KEM

Criação de protótipo em Sagemath de uma técnica representativa da família de criptosistemas pós-quânticos KYBER ("LWE based").
Pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.
Para o desenvolvimento destas soluções foram seguidas as especificações ds documento oficial https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf.

PKE

Utilizamos, para parâmetros, os valores especificados no documento referenciado anteriormente, para o KYBER512, que são n = 256; k = 2; q = 3329; n_1 = 3; n_2 = 2; (d_u, d_v) = (10,4).
São ainda instânciadas as seguintes funções:

- XOF com SHAKE-128;
- H com SHA3-256;
- G com SHA3-512;
- PRF(s,b) com SHAKE-256($s||b$);
- KDF com SHAKE-256.

Começamos, então, pela implementação da função de geração das chaves.

Geração de chaves

A função **keygen()** não recebe parâmetros como *input* e produz um par de chaves (chave pública,chave privada) como *output*.

- Calculamos ρ e σ, usando a função **G** com um array de bytes **d** gerado aleatoriamente
- De seguida, geramos a matriz **A**, a partir de ρ, e a sua representação NTT **Â**
- Depois são gerados **s** e **e**, também a partir de ρ
- São calculadas as representações NTT dos arrays **^s** e **ê**
- Calculamos **^t**, a representação NTT da multiplicação da matriz **Â** com o vetor **^s** e adicionamos o vetor **ê**

- Finalmente, são calculadas as chaves:
  - a chave pública pk através do encode da concatenação de **^t** *modulo* q com ρ
  - a chave privada sk através do encode da concatenação de **^s** *modulo* q
- Devolvemos, então, o par de chaves

Cifragem

A função de cifragem recebe como argumentos a chave pública pk, uma mensagem m e um array r gerado aleatoriamente. Como output, devolve a mensagem cifrada c.

- Começamos por calcular o decode da chave pública pk e a sua representação NTT **^t**
- Obtemos ρ a partir da chave pública
- Geramos a matriz **Â** da mesma forma que na geração de chaves
- Geramos **r** e e_1, a partir de ρ
- Geramos ainda e_2, também a partir de ρ
- Calculamos a representação NTT de **r**, **^r**
- Multiplicamos a **Â** por **^r** e calculamos o NTT inverso do resultado, adicionando ao resultado o e_1, obtemos **u**
- Calculamos **decompressed_m**, fazendo o $decode_1(m)$ e o $decompress_q$ do resultado com 1
- Calculamos **v**, fazendo a multiplicação de **^t** com **^r** e calculando o NTT inverso do resultado, adicionando ainda e_2 e **decompressed_m**
- Finalmente, obtemos **c1** e **c2**:
  - **c1** é obtido do $encode_{du}$ de **u** $compress_q$ com d_u
  - **c2** é obtido do $encode_{dv}$ de **v** $compress_q$ com d_v
- Devolvemos **c**, que é a concatenação de **c1** com **c2**

Decifragem

A função de decifra recebe como argumentos a chave privada sk e o texto a decifrar c. Devolve a mensagem m original.

- Obtemos **u** a partir do $decompress_q$ do $decode_{du}$ de **c** com d_u
- Obtemos **v** a partir do $decompress_q$ do $decode_{dv}$ da 2ª componente de **c** (**c2**) com d_v
- Obtemos **^s** calculando o $decode_{12}$ de **sk**
- Calculamos **û**, a representação NTT de **u**
- Obtemos **mult**, que é o resultado da multiplicação de **û** com **^s**, e calculamos o seu NTT inverso (que podemos representar por **^mult**)
- Calculamos a diferença **dif** entre **v** e **^mult**
- Finalmente, obtemos **m** através do $encode_1$ do $compress_q$ de **dif** com 1

In [45]:
```
import math, os, numpy as np
from hashlib import sha3_512 as G, shake_128 as XOF, sha3_256 as H, shake_
```

In [50]:

```
class NTT(object):
# fornecida pelo professor
    def __init__(self, n=16, q=3329, base_inverse=False):
        if not  n in [16,32,64,128,256,512,1024,2048]:
            raise ValueError("improper argument ",n)
        self.n = n
        if not q:
            self.q = 1 + 2*n
            while True:
                if (self.q).is_prime():
                    break
                self.q += 2*n
        else:
            if q % (2*n) != 1:
                raise ValueError("Valor de 'q' não verifica a condição NTT'
            self.q = q

        self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen()

        g = (w^n + 1)
        x = g.roots(multiplicities=False)[-1]
        self.x = x
        if  base_inverse:
            rs = [x^(2*i+1)  for i in range(n)]
            self.base = crt_basis([(w - r) for r in rs])
        else:
            self.base = None



    def ntt(self,f,inv=False):
        def _expand_(f):
            if f.__class__ == list:
                u = f
            else:
                u = f.list()
            return u + [0]*(self.n-len(u))

        def _ntt_(x,N,f,inv=inv):
            if N==1:
                return f
            N_ = N//2 ; z =  x^2
            f0 = [f[2*i]   for i in range(N_)] ; f1 = [f[2*i+1] for i in ra
            ff0 = _ntt_(z,N_,f0,inv=inv) ; ff1 = _ntt_(z,N_,f1,inv=inv)

            s  = self.F(1) if inv else x
            ff = [self.F(0) for i in range(N)]
            for i in range(N_):
                a = ff0[i] ; b = s*ff1[i]
                ff[i] = a + b ; ff[i + N_] = a - b
                s = s * z
            return ff


        vec = _expand_(f)
        if  not inv:
            return self.R(_ntt_(self.x,self.n, vec, inv=inv))
        elif self.base != None:
            return sum([vec[i]*self.base[i] for i in range(self.n)])
        else:
```

```
            n_ = (self.F(self.n))^-1
            x_ = (self.x)^-1
            u = _ntt_(x_,self.n,vec, inv=inv)

            return self.R([n_ * x_^i * u[i] for i in range(self.n)])


    def random_pol(self,args=None):
        return (self.R).random_element(args)


# Teste
N=16

T = NTT()
#T = NTT(n=N,base_inverse=False)

f = T.random_pol(N//2)
#print(f)

ff = T.ntt(f)
#print(ff)

fff = T.ntt(ff,inv=True)

#print(fff)
print("Correto ? ",f == fff)
```

```
Correto ?  True
```

In [47]:

```
#definição dos parâmetros usados no kyber (KYBER512)
n = 256
q = 3329
Qq = PolynomialRing(GF(q), 'x')
y = Qq.gen()
RQ = QuotientRing(Qq, y^n+1)


'''
Funções auxiliares
'''

#definição dos parâmetros usados no kyber (KYBER512)
n = 256
q = 3329
Qq = PolynomialRing(GF(q), 'x')
y = Qq.gen()
RQ = QuotientRing(Qq, y^n+1)


'''
Definição da função mod+-
'''
def modMm(r,a) :
    _r = r % a
    # Testar se a é par
    if mod(a,2)==0 :
        # Cálculo dos limites -a/2 e a/2
        inf_bound, sup_bound = -a/2, a/2
    # a é ímpar
```

```
    else :
        # Cálculo dos limites -a-1/2 e a-1/2
        inf_bound, sup_bound = (-a-1)/2, (a-1)/2
    # Queremos garantir que o módulo se encontre no intervalo calculado
    while _r > sup_bound :
        _r-=a
    while _r < inf_bound :
        _r+=a
    return _r


'''
Função que converte um array de bytes num array de bits
'''
def bytesToBits(bytearr) :
    bitarr = []
    for elem in bytearr :
        bitElemArr = []
        # Calculamos cada bit pertencente ao byte respetivo
        for i in range(0,8) :
            bitElemArr.append(mod(elem // 2**(mod(i,8)),2))

        for i in range(0,len(bitElemArr)) :
            bitarr.append(bitElemArr[i])
    return bitarr

'''
Função que converte um array de bits num array de bytes
'''
def bitsToBytes(bitarr) :
    bytearr = []
    bit_arr_size = len(bitarr)
    byte_arr_size = bit_arr_size / 8
    for i in range(byte_arr_size) :
        elem = 0
        for j in range(8) : # Definir macro BYTE_SIZE = 0
            elem += (int(bitarr[i*8+j]) * 2**j)
        bytearr.append(elem)
    return bytearr


'''
Função parse cuja finalidade é receber como input
uma byte stream e retornar, como output, a representação NTT
'''
def parse(b) :
    coefs = [0]*n # O poly terá n=256 coeficientes
    i,j = 0,0
    while j<n :
        d = b[i] + 256*b[i+1]
        d = mod(d,2**13)
        if d<q :
            coefs[br(8,j)] = d
            j+=1
        i+=2
    return RQ(coefs)


'''
Implementação da função que implementa
o bit reversed order
```

```
Parâmetros:
    - _bits : nº de bits usados para representar nr
    - nr : valor a ser bitreversed
'''
def br(_bits, nr) :
    res = 0
    for i in range(_bits) :
        res += (nr % 2) * 2**(_bits-i-1)
        nr = nr // 2
    return res


'''
Definição da função compress
'''
def compress(q,x,d) :
    rounded = round((2**d)/q * int(x))
    r = mod(rounded,2**d)

    return r


'''
Definição da função decompress
'''
def decompress(q,x,d) :
    return round((q/(2**d)) * ZZ(x))


'''
Definição da função CBD. Recebe como
input o n (comprido) e o array de bytes
'''
def cbd(noise, btarray) :
    f = []
    bitArray = bytesToBits(btarray)
    for i in range(256) :
        a, b = 0, 0
        # Cálculo do a e do b
        for j in range(256) :
            a+=bitArray[2*i*noise + j]
            b+=bitArray[2*i*noise + noise + j]
        f.append(a - b)
    return RQ(f)


'''
Implementação da função decode
'''
def decode(l, btarray) :
    f = []
    bitArray = bytesToBits(btarray)
    for i in range(256) :
        fi = 0
        for j in range(l) :
            fi += int(bitArray[i*l+j]) * 2**j
        f.append(fi)
    return RQ(f)

'''
```

```
Implementação da função encode
'''
def encode(l, poly) :
    bitArr = []
    coef_array = poly.list()
    # Percorremos cada coeficiente
    for i in range(256) :
        actual = int(coef_array[i])
        for j in range(l) :
            bitArr.append(actual % 2)
            actual = actual // 2
    return bitsToBytes(bitArr)


'''
Função que implementa a multiplicação entre duas
entradas de vetores/matrizes de forma pointwise
(coeficiente a coeficiente)

Parâmetros :
    - e1 e e2 : elemento/entrada da matriz/vetor
'''
def pointwise_mult(e1,e2) :

    mult_vector = []
    for i in range(n) :
        mult_vector.append(e1[i] * e2[i])
    return mult_vector

'''
Função que implementa a soma entre duas
entradas de vetores/matrizes de forma pointwise
(coeficiente a coeficiente)

Parâmetros :
    - e1 e e2 : elemento/entrada da matriz/vetor
'''
def pointwise_sum(e1,e2) :

    sum_vector = []
    for i in range(n) :
        sum_vector.append(e1[i] + e2[i])
    return sum_vector

'''
Função que retorna um vetor resultante da
multiplicação entre uma matriz M e um vetor v
'''
def multMatrixVector(M,v,k) :
    T = NTT()
    As = []
    mult = []
    for i in range(k) :
        As.append([])
        for j in range(k) :
            mult.append([])
            mult[j] = pointwise_mult(M[i][j], v[j])
        for y in range(k-1):
            As[i] = pointwise_sum(mult[y],mult[y+1])
    return As
```

In [53]:

```
'''
Implementação da classe PKE
'''

class KyberPKE :

    def __init__(self,n,k,q,n1,n2,du,dv,RQ) :

        self.n = n
        self.k = k
        self.q = q
        self.n1 = n1
        self.n2 = n2
        self.du = du
        self.dv = dv
        self.Rq = RQ

    '''
    Função que permite a geração de
    uma chave pública
    '''
    def keygen(self) :
        d = os.urandom(32)
        h = G(d)
        digest = h.digest()
        ro,sigma = digest[:32], digest[32:]

        N = 0
        A, s, e = [], [], []
        T = NTT()

        # Construção da matriz A
        for i in range(self.k) :
            A.append([])
            for j in range(self.k) :
                xof = XOF()
                xof.update(ro + j.to_bytes(4,'little') + i.to_bytes(4,'lit
                A[i].append(parse(xof.digest(int(self.q))))
                A[i][j] = T.ntt(A[i][j])
        # Construção do vetor s
        for i in range(self.k) :
            prf = PRF()
            prf.update(sigma + int(N).to_bytes(4,'little'))
            s.append(cbd(self.n1, prf.digest(int(self.q+1))))
            N += 1
        # Construção do vetor e
        for i in range(self.k) :
            prf = PRF()
            prf.update(sigma + int(N).to_bytes(4,'little'))
            e.append(cbd(self.n1, prf.digest(int(self.q))))
            N += 1
        # Calculo do ntt de s
        _s = []
        for i in range(self.k) :
            _s.append(T.ntt(s[i]))
        # Calculo do ntt de e
        _e = []
        for i in range(self.k) :
            _e.append(T.ntt(e[i]))
```

```
        _As = multMatrixVector(A,_s,self.k)
        # Calculo do ntt da soma de A com e
        t = []
        for i in range(self.k):
            t.append(pointwise_sum(_As[i],_e[i]))
            t[i] = T.ntt(t[i])

        # Calculamos agora pk = (encode(12,t mod q)|| ro)
        for i in range(self.k) :
            # Para cada coeficiente do polinomio :
            lst = t[i].list()
            for j in range(len(lst)) :
                lst[j] = mod(t[i][j],self.q)
            t[i] = lst

        pk = []
        for i in range(self.k) :
            res = encode(12,self.Rq(t[i]))
            for j in range(len(res)) :
                pk.append(res[j])

        for i in range(len(ro)) :
            pk.append(ro[i])

        # Calculo de sk = encode(12,s mod q)
        # Para cada polinomio
        for i in range(self.k) :
            # Para cada coeficiente do polinomio
            lst = _s[i].list()
            for j in range(len(lst)) :
                lst[j] = mod(_s[i][j],self.q)
            _s[i] = lst
        # encode
        sk = []
        for i in range(self.k):
            _s[i] = self.Rq(_s[i])
            res = encode(12,_s[i])
            for bt in res :
                sk.append(bt)

        return(pk,sk)




    '''
    Função que implementa a cifragem de mensagens

    Parâmetros :
        - pk : Chave privada gerada
        - m : mensagem a ser cifrada
        - r : Random coins
    '''
    def encryption(self, pk, m, r) :
        N = 0
        T = NTT()

        '''
        Função auxiliar que permite transformar um
        array de bytes (representados por integers)
```

```
em bytes (python)
'''
def byteArrToBytes(btArray) :
    byts = b''
    for i in btArray :
        byts += i.to_bytes(1,'little')
    return byts

# Implementação do decode(dt,pk)
t = []
_t = []
for i in range(self.k) :
    _t.append([])
    t.append(decode(12,pk[i*32*12:i*32*12+32*12]))
    _t[i] = T.ntt(t[i])

ro = byteArrToBytes(pk[12*self.k*self.n/8:])

At = []
# Construção da matriz A
for i in range(self.k) :
    At.append([])
    for j in range(self.k) :
        xof = XOF()
        xof.update(ro + i.to_bytes(4,'little') + j.to_bytes(4,'lit
        At[i].append(parse(xof.digest(int(self.q))))
        At[i][j] = T.ntt(At[i][j])

rr, e1 = [], []
# Construção do vetor rr
for i in range(self.k) :
    prf = PRF()
    prf.update(r + int(N).to_bytes(4,'little'))
    rr.append(cbd(self.n1, prf.digest(int(self.q))))
    N += 1
# Construção do vetor e1
for i in range(self.k) :
    prf = PRF()
    prf.update(r + int(N).to_bytes(4,'little'))
    e1.append(cbd(self.n2, prf.digest(int(self.q))))
    N += 1

# Construção de e2
prf = PRF()
prf.update(r + int(N).to_bytes(4,'little'))
e2 = cbd(self.n2, prf.digest(int(self.q)))

# Cálculo do ^rr :
_rr = []
for i in range(self.k) :
    _rr.append(T.ntt(rr[i]))

# Cálculo do vetor em Rq u
mult = multMatrixVector(At,_rr,self.k)

u = []
for i in range(self.k) :
    mult[i] = T.ntt(mult[i],inv=true)
    u.append(pointwise_sum(mult[i],e1[i]))
    u[i] = self.Rq(u[i])
```

```
        # Calculamos o decompress(q,decode(1,m),1)
        decoded_m = decode(1,m)
        #print('decm',decoded_m);print()
        decompressed_m = []
        for i in range(len(decoded_m.list())) :
            decompressed_m.append(decompress(self.q, decoded_m[i], 1))
        #print('decomp',decompressed_m);print()

        mult_tt_r = []
        for i in range(self.k):
            mult_tt_r.append(pointwise_mult(_t[i],_rr[i]))
            mult_tt_r[i] = T.ntt(mult_tt_r[i],inv=true)

        # Calculo de v = NTT-1(NTT(t)T . _rr) + e2 + decompressed m
        v = [0] * self.n
        for i in range(self.k) :
            v = pointwise_sum(v,T.ntt(pointwise_mult(_t[i],_rr[i]),inv=true

        v = pointwise_sum(pointwise_sum(v,e2),decompressed_m)

        c1 = []
        # Calculo de compress(q,u,du)
        for i in range(self.k) :
            lst = u[i].list()
            for j in range(len(u[i].list())) :
                lst[j] = compress(self.q,lst[j],self.du)
            u[i] = self.Rq(lst)
        # Calculo de encode(du,compress(q,u,du))
        for i in range(self.k) :
            u[i] = encode(self.du,u[i])
            for bt in u[i] :
                c1.append(bt)

        # Cálculo de c2
        #print('vb4compress',v);print()
        for i in range(len(v)) :
            v[i] = compress(self.q,v[i],self.dv)
        v = self.Rq(v)
        #print('vb4encode',v);print()
        # Calculo de encode(dv,compress(q,v,dv)) :
        c2 = encode(self.dv,v)

        return c1+c2

    '''
    Função que implementa a decifragem de mensagens
    '''
    def decryption(self,sk,ct) :

        T = NTT()
        c1 = ct[:self.du*self.k*self.n/8]
        c2 = ct[self.du*self.k*self.n/8:]
        u = []

        # Calculo de decompress(q,decode(du,ct),du) :
        for i in range(self.k) :
            u.append(decode(self.du,ct[i*len(ct)/self.k:i*len(ct)/self.k+le
            lst = u[i].list()
            for j in range(len(lst)) :
                lst[j] = decompress(self.q,lst[j],self.du)
            u[i] = self.Rq(lst)
```

```
        # Calculo de v
        v = decode(self.dv,c2)
        lst = v.list()
        #print('c2decoded',v);print()
        for i in range(len(v.list())) :
            lst[i] = decompress(self.q,lst[i],self.dv)
        #print('c2decompressed',lst);print()
        v = self.Rq(lst)

        # Calculo de _s
        _s = []
        for i in range(self.k) :
            _s.append(decode(12,sk[i*32*12:i*32*12+32*12]))
            _s[i] = T.ntt(_s[i])

        # Calculo de NTT(u) :
        for i in range(self.k) :
            u[i] = T.ntt(u[i])
        # Calculo de sT . NTT(u) :
        mult = self.Rq([])
        for i in range(self.k) :
            mult = pointwise_sum(mult,pointwise_mult(_s[i],u[i]))

        mult = T.ntt(mult,inv=true)

        # Calculo de v - NTT-1(sT . NTT(u)) :
        dif = [0] * self.n
        for i in range(self.n) :
            dif[i] = v[i]-mult[i]

        #print('dif',dif);print()
        # Calculo de m = compress(q,v - NTT-1(sT . NTT(u)),1)
        m = []
        for i in range(self.n) :
            m.append(compress(self.q,dif[i],1))

        #print('mcompressed',self.Rq(m));print()

        m = encode(1,self.Rq(m))

        return m
```

In [54]:

```
k = KyberPKE(n=n,k=2,q=q,n1=3,n2=2,du=10,dv=4,RQ=RQ)
(pk,sk) = k.keygen()


print('Tamanho da chave publica: ',len(pk))
#print('\nChave privada: ')
#print(sk)

m = [32,4,35,78,64,45,2,35,64,45,2,35,53,34,54,32,32,4,35,78,64,45,2,35,64,

print('Mensagem a cifrar: ',m) ; print()

ct = k.encryption(pk,m,os.urandom(32))

dct = k.decryption(sk,ct)

print('Mensagem decifrada: ',dct) ; print('Mensages iguais?', m==dct)
```

```
Tamanho da chave publica:  800
Mensagem a cifrar:  [32, 4, 35, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 34, 5
4, 32, 32, 4, 35, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 34, 54, 32]

Mensagem decifrada:  [244, 125, 35, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 3
4, 54, 32, 32, 4, 35, 78, 64, 45, 2, 35, 64, 45, 2, 35, 53, 34, 54, 32]
Mensages iguais? False
```

KEM

Utilizamos, para parâmetros, os valores especificados no documento referenciado anteriormente, para o KYBER512, que são n = 256; k = 2; q = 3329; n_1 = 3; n_2 = 2; (d_u, d_v) = (10,4) e d_t = 2^{-139}.

Para a implementação da classe KEM, foram seguidos os algoritmos presentes no documento, que usam uma transformação Fujisaki-Okamoto da classe PKE já definida acima.

Geração de chaves

As chaves (pk, sk') são geradas recorrendo à função de geração de chaves do PKE, com a adição de um parâmetro **z**, que é um array de bytes gerado aleatoriamente.

A chave privada sk tem a adição de ser uma concatenação de:

- a chave privada gerada no PKE, sk'
- a chave pública gerada no PKE, pk
- o resultado da função **H** aplicada a sk
- o array z.

Encapsulamento

A função de encapsulamento recebe a chave pk a ser encapsulada.

- É gerada uma mensagem aleatoriamente e calculado o seu hash m
- Calculamos **(_K,r)** através da função $G(m||H(pk))$
- Obtemos c através da função de cifra do PKE, passando (pk, m, r) como parâmetros
- Obtemos **K**, calculando o $KDF(\_K||H(c))$
- Devolvemos o par (c,K)

Desencapsulamento

A função de desencapsulamento recebe um texto a decifrar c e a chave privada sk.

- Obtemos **pk**, **h** e **z** a partir da chave privada.
- Calculamos **m'** através da função de decifra do PKE, passando como argumento o texto **c**
- Obtemos **(_K',r')** através da função $G(m'||h)$
- Obtemos **c'** através da função de cifra do PKE, passando como argumento (pk, m', r')
- Se **c** for o mesmo que o **c'** obtido, devolvemos $K = KDF(\_K'||H(c))$
- Senão, devolvemos $K = KDF(z||H(c))$

In [71]:
```
class KyberKEM :
```

```
    def __init__(self,n,k,q,n1,n2,du,dv,dt,RQ) :

        self.n = n
        self.k = k
        self.q = q
        self.n1 = n1
        self.n2 = n2
        self.du = du
        self.dv = dv
        self.dt = dt
        self.Rq = RQ

        self.pke = KyberPKE(n,k,q,n1,n2,du,dv,RQ)


    def encode_list(self, lst):
        res = b''
        for poly in lst:
            res+=(bytearray(str(poly).encode()))
        return res


    def keyGen(self):
        z = os.urandom(32)

        (pk,_sk) = self.pke.keygen()

        sk = _sk+pk+list(H(self.encode_list(pk)).digest())+list(z)

        return (pk,sk)


    def enc(self,pk):
        _m = os.urandom(32)
        m = H(_m).digest()

        print('Mensagem: ',list(m))

        g = G(m+H(self.encode_list(pk)).digest()).digest()
        (_K,r) = (g[:32],g[32:])

        c = self.pke.encryption(pk,m,r)

        # PRF is same as KDF --> shake-256
        K = PRF(_K+H(self.encode_list(c)).digest()).digest(32)

        return (c,K)


    def dec(self,c,skfull):
        sk_tam = 12*self.k*self.n/8
        pk_tam = (12*self.k*self.n/8+32)

        pk = skfull[sk_tam:-64]
        h = skfull[sk_tam + pk_tam: - 32]
        z = skfull[sk_tam + pk_tam + 32:]
        sk = skfull[:-(pk_tam + 64)]

        _m = self.pke.decryption(sk,c)

        g = G(self.encode_list(_m+h)).digest()
```

```
        (_K,_r) = (g[:32],g[32:])

        _c = self.pke.encryption(pk,_m,_r)

        if c == _c:
            # PRF is same as KDF --> shake-256
            K_list = self.encode_list(_K)
            Khc = self.encode_list(K_list+list(H(c).digest()))
            K = PRF(Khc).digest(32)
        else:
            c_list =self.encode_list(c)
            hc = list(H(c_list).digest())
            zhc = self.encode_list(z+hc)
            K = PRF(zhc).digest(32)

        return K
```

In [72]:

```
n = 256
q = 3329
Qq = PolynomialRing(GF(q), 'x')
y = Qq.gen()
RQ = QuotientRing(Qq, y^n+1)



k = KyberKEM(n=n,k=2,q=q,n1=3,n2=2,du=10,dv=4,dt=11,RQ=RQ)
(pk,sk) = k.keyGen()

(c,K) = k.enc(pk)
print('Encapsulamento: ',K)


decaps = k.dec(c,sk)

print('Desencapsulamento: ',decaps)

print('Igual? ', K==decaps)
```

```
Mensagem:  [69, 179, 15, 178, 255, 20, 92, 195, 120, 186, 35, 22, 133, 15,
116, 152, 74, 165, 38, 179, 122, 163, 91, 130, 237, 154, 211, 178, 161, 192
, 134, 103]
Encapsulamento:  b'|\xe9k\x9bqC5yD!\x01\xd84\x85\x82sNy\xc8\x16-\x99\nO\xbf
\xf9\x18\xc0z\x92\x9dg'
Desencapsulamento:  b'\x88\xdf\x1b\x12\xf3\xb1\x19\x86R\x7f\x8cf\xedf&4\xe5
y\x1f\xcc&:\xdae\xebB\xbcE\x91\xcf\xba\x17'
Igual?  False
```