

Estruturas Criptográficas 2022/23 — TP1. Problema 1

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

May 2, 2023

Pretende-se a criação de um protótipo em Sagemath para o algoritmo KYBER: implementação de um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.

Tal como apresentado [aqui](#), o KYBER é IND-CCA2 seguro. Uma vez que o CCA2-KEM foi concebido para proporcionar um nível de segurança mais elevado do que o CCA-KEM, também pode ser considerado IND-CPA seguro. Isto deve-se ao facto de que qualquer ataque que quebre a segurança CPA, quebra também a segurança CCA2. No entanto, uma vez que é apenas pretendido um protótipo cujo KEM seja IND-CPA seguro, com base num esquema PKE IND-CCA seguro, a transformação de Fujisaki-Okamoto (FO), tal como está nos apontamentos da disciplina, será aplicada no esquema PKE. Note-se que a implementação do KEM-CCA2 conforme o artigo já mencionado segue uma variante da transformação FO, mas neste trabalho a transformação FO é aplicada ao PKE.

Notação

R corresponde ao anel $\mathbb{Z}[X]/(X^n + 1)$ e R_q ao anel $\mathbb{Z}_q[X]/(X^n + 1)$

```
[10]: from cryptography.hazmat.primitives import hashes
      from pickle import dumps, loads
```

```
[11]: # Classe que implementa as multiplicações em  $R$  - number-theoretic transform
      ↪ (NTT)
class NTT:

    def __init__(self, n=128, q=None):

        if not n in [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)
xi = g.roots(multiplicities=False)[-1]
self.xi = xi
rs = [xi^(2*i+1) for i in range(n)]
self.base = crt_basis([(w - r) for r in rs])

def ntt(self,f):

    def _expand_(f):
        u = f.list()
        return u + [0]*(self.n-len(u))

    def _ntt_(xi,N,f):
        if N==1:
            return f
        N_ = N/2 ; xi2 = xi^2
        f0 = [f[2*i] for i in range(N_)] ; f1 = [f[2*i+1] for i in
↪range(N_)]
        ff0 = _ntt_(xi2,N_,f0) ; ff1 = _ntt_(xi2,N_,f1)

        s = xi ; 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 * xi2
        return ff

    return _ntt_(self.xi,self.n,_expand_(f))

def invNtt(self,ff):
    return sum([ff[i]*self.base[i] for i in range(self.n)])

# Operações sobre matrizes e vetores
# Soma de matrizes
def sumMatrix(e1, e2, n):
    for i in range(len(e1)):
        e1[i] = sumVector(e1[i], e2[i], n)
    return e1

# Subtração de matrizes
def subMatrix(e1, e2, n):
    for i in range(len(e1)):
        e1[i] = subVector(e1[i], e2[i], n)

```

```

    return e1

# Multiplicação de matrizes
def multMatrix(vec1, vec2, n):
    for i in range(len(vec1)):
        vec1[i] = multVector(vec1[i], vec2[i], n)
    tmp = [0] * n
    for i in range(len(vec1)):
        tmp = sumVector(tmp, vec1[i], n)
    return tmp

# Multiplicação de uma matriz por um vector
def multMatrixVector(M, v, k, n):
    for i in range(len(M)):
        for j in range(len(M[i])):
            M[i][j] = multVector(M[i][j], v[j], n)
    tmp = [[0] * n] * k
    for i in range(len(M)):
        for j in range(len(M[i])):
            tmp[i] = sumVector(tmp[i], M[i][j], n)
    return tmp

# Soma de vetores
def sumVector(ff1, ff2, n):
    res = []
    for i in range(n):
        res.append((ff1[i] + ff2[i]))
    return res

# Multiplicação de vetores
def multVector(ff1, ff2, n):
    res = []
    for i in range(n):
        res.append((ff1[i] * ff2[i]))
    return res

# Subtração de vetores
def subVector(ff1, ff2, n):
    res = []
    for i in range(n):
        res.append((ff1[i] - ff2[i]))
    return res

```

1 Implementação do esquema PKE IND-CCA seguro

Conforme já mencionado, o esquema PKE apresentado será sujeito à transformação FO de modo a ser IND-CCA seguro. A geração das chaves não sofre qualquer alteração, pelo que está de acordo

com o algoritmo 4. Os métodos para cifrar e decifrar serão alterados conforme a transformação FO.

KeyGen (algoritmo 4)

São geradas a chave privada e a chave pública. O método começa por gerar a matriz $\in R_q^{k \times k}$ no domínio NTT e os vetores $s, e \in R_q$. Para tal foram utilizadas os métodos *Parse* e *CBD* implementados de acordo com os algoritmos 1 e 2, respetivamente. Conforme apresentado no algoritmo 4, são geradas as variáveis necessárias para calcular a chave pública $pk = (*s + e, \rho)$, com $\rho = G(d)[32:]$ e d um valor pseudo-aleatório, e a chave privada $sk = s$.

Encrypt (algoritmo 5)

Cifra, com recurso à chave pública e um valor aleatório, uma mensagem m gerada a partir do anel R_q . O processo está descrito em pseudo-código no algoritmo 5 do artigo.

De modo a ser um PKE IND-CCA seguro, a transformação FO indica que o método de cifra passará a ser:

$$E'(x) \equiv \vartheta r \leftarrow h \cdot \vartheta (y, r') \leftarrow (x \oplus g(r), h(r\|y)) \cdot (y, f(r, r'))$$

Começa-se por gerar o valor aleatório $r \in R_q$. Calcula-se y como a operação de XOR entre o *plaintext* e o hash do valor r , com recurso à primitiva SHA3_256. Além disso, r é misturado com y para construir via o hash h (que neste caso será também a primitiva SHA3_256) uma nova fonte de aleatoriedade $r' = h(r\|y)$. Finalmente, o resultado é o par formado pela ofuscação y e o criptograma que resulta de, com o f original, i.e. o método *encrypt* original, cifrar r com a aleatoriedade r' .

Decrypt (algoritmo 6)

Decifra, com recurso à chave privada, o *ciphertext*. Novamente, o processo está descrito no artigo, agora no algoritmo 6.

Por fim, e de forma a garantir segurança IND-CCA do PKE, o método será transformado via a transformação FO de acordo com a expressão:

$$D'(y, c) \equiv \vartheta r \leftarrow D(c) \cdot \text{if } c \neq f(r, h(r\|y)) \text{ then } \perp \text{ else } y \oplus g(r)$$

Começa-se por decifrar o criptograma c com recurso ao método original *decrypt*. O resultado é usado para derivar a ofuscação da chave tal como no método de cifra, utilizando o método original *encrypt*, de forma a verificar se a ofuscação recebida é igual. Se for igual, a chave é válida e procede-se ao XOR do y com o hash do criptograma inicial já decifrado.

```
[44]: class KYBER_PKE:

    def __init__(self, pset):
        self.n, self.q, self.T, self.k, self.n1, self.n2, self.du, self.dv, self.
        ↪ Rq = self.setup(pset)

    def setup(self, pset):
        n = 256
        q = 7681
        n2 = 2
        if pset == 512:
            k = 2
            n1 = 3
```

```

        du = 10
        dv = 4
    elif pset == 768:
        k = 3
        n1 = 2
        du = 10
        dv = 4
    elif pset == 1024:
        k = 4
        n1 = 2
        du = 11
        dv = 5
    else: print("Error: Parameter set not valid!")

    Zq.<w> = GF(q) []
    fi = w^n + 1
    Rq.<w> = QuotientRing(Zq, Zq.ideal(fi))

    T = NTT(n,q)

    return n, q, T, k, n1, n2, du, dv, Rq

def bytes2Bits(self, byteArray):
    bitArray = []
    for elem in byteArray:
        bitElemArr = []
        for i in range(0,8):
            bitElemArr.append(mod(elem//2**(mod(i,8)),2))
            for i in range(0,len(bitElemArr)):
                bitArray.append(bitElemArr[i])
    return bitArray

def G(self, h):
    digest = hashes.Hash(hashes.SHA3_512())
    digest.update(bytes(h))
    g = digest.finalize()
    return g[:32],g[32:]

def XOF(self,b,b1,b2):
    digest = hashes.Hash(hashes.SHAKE128(int(self.q)))
    digest.update(b)
    digest.update(bytes(b1))
    digest.update(bytes(b2))
    m = digest.finalize()
    return m

def PRF(self,b,b1):

```

```

        digest = hashes.Hash(hashes.SHAKE256(int(self.q)))
        digest.update(b)
        digest.update(bytes(b1))
        return digest.finalize()

def Compress(self,x,d) :
    coefficients = x.list()
    newCoefficients = []
    for c in coefficients:
        new = mod(round( int(2 ** d) / self.q * int(c)), int(2 ** d))
        newCoefficients.append(new)
    return self.Rq(newCoefficients)

def Decompress(self,x,d) :
    coefficients = x.list()
    newCoefficients = []
    for c in coefficients:
        new = round(self.q / (2 ** d) * int(c))
        newCoefficients.append(new)
    return self.Rq(newCoefficients)

# Método XOR
def xor(self, b1, b2):
    return bytes(a ^^ b for a, b in zip(b1, b2))

# Algorithm 1
def Parse(self, byteArray):
    i = 0
    j = 0
    a = []
    while j < self.n:
        d1 = byteArray[i] + 256 * mod(byteArray[i+1],16)
        d2 = byteArray[i+1]//16 + 16 * byteArray[i+2]
        if d1 < self.q :
            a.append(d1)
            j = j+1
        if d2 < self.q and j<self.n:
            a.append(d2)
            j = j+1
        i = i+3
    return self.Rq(a)

# Algorithm 2
def CBD(self, byteArray, nn):
    f=[0]*self.n
    bitArray = self.bytes2Bits(byteArray)

```

```

    for i in range(256):
        a = 0
        b = 0
        for j in range(nn):
            a += bitArray[2*i*nn + j]
            b += bitArray[2*i*nn + nn + j]
        f[i] = a-b
    return self.Rq(f)

# Algorithm 3
def Decode(self, byteArray, l):
    f = []
    bitArray = self.bytes2Bits(byteArray)
    for i in range(len(byteArray)):
        fi = 0
        for j in range(l):
            fi += int(bitArray[i*l+j]) * 2**j
        f.append(fi)
    return self.Rq(f)

def Encode(self, f, l):
    coefficients = list(f)
    return coefficients

# Algorithm 4
def keyGen(self):
    d = bytearray(os.urandom(32))
    ro, sigma = self.G(d)
    N = 0

    # Generate matrix  $\hat{A}$  in  $R_q$  in NTT domain
    A = []
    for i in range(self.k):
        A.append([])
        for j in range(self.k):
            A[i].append(self.T.ntt(self.Parse(self.XOF(ro,j,i))))

    # Sample  $s$  in  $R_q$  from  $B_{\eta 1}$ 
    s = []
    for i in range(self.k):
        s.insert(i, self.CBD(self.PRF(sigma, N), self.n1))
    N = N+1

    # Sample  $e$  in  $R_q$  from  $B_{\eta 1}$ 
    e = []
    for i in range(self.k):
        e.insert(i, self.CBD(self.PRF(sigma, N), self.n1))

```

```

        N = N+1

    for i in range(self.k) :
        s[i] = self.T.ntt(s[i])
        e[i] = self.T.ntt(e[i])

    t = sumMatrix(multMatrixVector(A,s,self.k,self.n), e, self.n)

    pk = t, ro
    sk = s

    return pk, sk

# Algorithm 5
def encrypt(self, pk, m, r):
    N = 0
    t, ro = pk

    # Generate matrix  $\hat{A}$  in  $R_q$  in NTT domain
    transposeA = []
    for i in range(self.k):
        transposeA.append([])
        for j in range(self.k):
            transposeA[i].append(self.T.ntt(self.Parse(self.XOF(ro,i,j))))

    # Sample  $r$  in  $R_q$  from  $B\eta_1$ 
    rr = []
    for i in range(self.k):
        rr.insert(i,self.T.ntt(self.CBD(self.PRF(r, N), self.n1)))
        N += 1

    # Sample  $e_1$  in  $R_q$  from  $B\eta_2$ 
    e1 = []
    for i in range(self.k):
        e1.insert(i,self.CBD(self.PRF(r, N), self.n2))
        N += 1

    # Sample  $e_2$  in  $R_q$  from  $B\eta_2$ 
    e2 = self.CBD(self.PRF(r, N), self.n2)

    uAux = multMatrixVector(transposeA, rr, self.k, self.n)
    uAux2 = []
    for i in range(len(uAux)) :
        uAux2.append(self.T.invNtt(uAux[i]))
    uAux3 = sumMatrix(uAux2, e1, self.n)
    u = []
    for i in range(len(uAux3)) :

```



```

        u.append(self.Rq(uAux3[i]))

    vAux = multMatrix(t, rr, self.n)
    vAux1 = self.T.invNtt(vAux)
    vAux2 = self.Rq(sumVector(vAux1, e2, self.n))

    v = self.Rq(sumVector(vAux2, self.Decompress(m, 1), self.n))

    # Compress(u, du)
    c1 = []
    for i in range(len(u)):
        c1.append(self.Compress(u[i], self.du))

    # Compress(v, dv)
    c2 = self.Compress(v, self.dv)

    return c1, c2

# Algorithm 6
def decrypt(self, sk, c):
    c1, c2 = c

    u = []
    for i in range(len(c1)):
        u.append(self.Decompress(c1[i], self.du))

    v = self.Decompress(c2, self.dv)

    s = sk

    uNTT = []
    for i in range(len(u)) :
        uNTT.append(self.T.ntt(u[i]))

    mAux = subVector(v, self.T.invNtt(multMatrix(s, uNTT, self.n)), self.n)

    m = self.Compress(self.Rq(mAux), 1)

    return m

# hashes h e g
def hashFOT(self, b):
    r = hashes.Hash(hashes.SHA3_256())
    r.update(b)
    return r.finalize()

def encryptCCA(self, x, pk):

```

```

    r = self.Rq([choice([0, 1]) for i in range(self.n)])
    y = self.xor(x, self.hashFOT(bytes(r)))
    c = self.encrypt(pk, r, self.hashFOT(bytes(r)+y))
    return (y, c)

def decryptCCA(self, y, c, pk, sk):
    r = self.decrypt(sk, c)
    derived_c = self.encrypt(pk, r, self.hashFOT(bytes(r)+y))
    if c[0] != derived_c[0]:
        print("Error: key doesn't match!")
        return None
    else:
        return self.xor(y, self.hashFOT(bytes(r)))

```

1.1 Teste da transformação FO

```

[48]: kyber = KYBER_PKE(512)

pk, sk = kyber.keyGen()
m = b'Hello there!'
print("Original message:")
print(m)

y, c = kyber.encryptCCA(m, pk)
print("\nCiphertext:")
print(c)

plaintext = kyber.decryptCCA(y, c, pk, sk)
print("\nDecrypted ciphertext:")
print(plaintext)

```

Original message:
b'Hello there!'

Ciphertext:

$$\begin{aligned}
 & ([986*w^{255} + 312*w^{254} + 123*w^{253} + 798*w^{252} + 892*w^{251} + 777*w^{250} + \\
 & 508*w^{249} + 939*w^{248} + 168*w^{247} + 503*w^{246} + 136*w^{245} + 272*w^{244} + \\
 & 540*w^{243} + 625*w^{242} + 276*w^{241} + 687*w^{240} + 999*w^{239} + 375*w^{238} + \\
 & 482*w^{237} + 43*w^{236} + 1000*w^{235} + 919*w^{234} + 636*w^{233} + 685*w^{232} + 59*w^{231} \\
 & + 791*w^{230} + 192*w^{229} + 355*w^{228} + 612*w^{227} + 141*w^{226} + 843*w^{225} + \\
 & 294*w^{224} + 781*w^{223} + 434*w^{222} + 753*w^{221} + 829*w^{220} + 759*w^{219} + \\
 & 170*w^{218} + 857*w^{217} + 354*w^{216} + 468*w^{215} + 655*w^{214} + 306*w^{213} + \\
 & 733*w^{212} + 878*w^{211} + 651*w^{210} + 736*w^{209} + 421*w^{208} + 139*w^{207} + \\
 & 661*w^{206} + 87*w^{205} + 748*w^{204} + 640*w^{203} + 713*w^{202} + 866*w^{201} + 179*w^{200} \\
 & + 945*w^{199} + 644*w^{198} + 388*w^{197} + 982*w^{196} + 460*w^{195} + 152*w^{194} + \\
 & 762*w^{193} + 690*w^{192} + 247*w^{191} + 174*w^{190} + 179*w^{189} + 916*w^{188} + 78*w^{187} \\
 & + 425*w^{186} + 617*w^{185} + 448*w^{184} + 288*w^{183} + 999*w^{182} + 921*w^{181} +
 \end{aligned}$$

$587w^{180} + 964w^{179} + 898w^{178} + 41w^{177} + 153w^{176} + 818w^{175} + 69w^{174}$
 $+ 60w^{173} + 1001w^{172} + 620w^{171} + 581w^{170} + 345w^{169} + 744w^{168} +$
 $179w^{167} + 746w^{166} + 417w^{165} + 754w^{164} + 900w^{163} + 851w^{162} +$
 $324w^{161} + 936w^{160} + 602w^{159} + 383w^{158} + 318w^{157} + 261w^{156} +$
 $668w^{155} + 356w^{154} + 406w^{153} + 155w^{152} + 1013w^{151} + 494w^{150} +$
 $629w^{149} + 133w^{148} + 309w^{147} + 644w^{146} + 531w^{145} + 710w^{144} + 91w^{143}$
 $+ 328w^{142} + 234w^{141} + 236w^{140} + 812w^{139} + 977w^{138} + 697w^{137} +$
 $651w^{136} + 722w^{135} + 817w^{134} + 906w^{133} + 207w^{132} + 547w^{131} +$
 $699w^{130} + 109w^{129} + 171w^{128} + 24w^{127} + 409w^{126} + 1000w^{125} +$
 $170w^{124} + 349w^{123} + 353w^{122} + 117w^{121} + 48w^{120} + 659w^{119} + 348w^{118}$
 $+ 457w^{117} + 367w^{116} + 760w^{115} + 652w^{114} + 580w^{113} + 226w^{112} +$
 $409w^{111} + 549w^{110} + 78w^{109} + 487w^{108} + 300w^{107} + 489w^{106} + 493w^{105}$
 $+ 928w^{104} + 698w^{103} + 123w^{102} + 464w^{101} + 701w^{100} + 403w^{99} +$
 $486w^{98} + 802w^{97} + 686w^{96} + 216w^{95} + 692w^{94} + 199w^{93} + 532w^{92} +$
 $842w^{91} + 650w^{90} + 354w^{89} + 796w^{88} + 133w^{87} + 641w^{86} + 877w^{85} +$
 $521w^{84} + 649w^{83} + 327w^{82} + 711w^{81} + 370w^{80} + 260w^{79} + 934w^{78} +$
 $488w^{77} + 165w^{76} + 810w^{75} + 577w^{74} + 512w^{73} + 63w^{72} + 931w^{71} +$
 $704w^{70} + 612w^{69} + 234w^{68} + 815w^{67} + 652w^{66} + 819w^{65} + 536w^{64} +$
 $680w^{63} + 226w^{62} + 178w^{61} + 868w^{60} + 96w^{59} + 275w^{58} + 990w^{57} +$
 $924w^{56} + 400w^{55} + 694w^{54} + 523w^{53} + 145w^{52} + 866w^{51} + 782w^{50} +$
 $602w^{49} + 312w^{48} + 95w^{47} + 915w^{46} + 656w^{45} + 1013w^{44} + 262w^{43} +$
 $887w^{42} + 165w^{41} + 888w^{40} + 920w^{39} + 286w^{38} + 777w^{37} + 915w^{36} +$
 $396w^{35} + 981w^{34} + 779w^{33} + 340w^{32} + 252w^{31} + 7w^{30} + 949w^{29} +$
 $503w^{28} + 494w^{27} + 190w^{26} + 245w^{25} + 842w^{24} + 227w^{23} + 358w^{22} +$
 $346w^{21} + 718w^{20} + 308w^{19} + 993w^{18} + 732w^{17} + 751w^{16} + 203w^{15} +$
 $984w^{14} + 286w^{13} + 383w^{12} + 466w^{11} + 238w^{10} + 678w^9 + 877w^8 +$
 $454w^7 + 131w^6 + 999w^5 + 1007w^4 + 521w^3 + 537w^2 + 1005w + 571,$
 $171w^{255} + 602w^{254} + 62w^{253} + 859w^{252} + 537w^{251} + 294w^{250} + 336w^{249}$
 $+ 140w^{248} + 769w^{247} + 731w^{246} + 971w^{245} + 380w^{244} + 887w^{243} +$
 $674w^{242} + 428w^{241} + 997w^{240} + 606w^{239} + 794w^{238} + 263w^{237} +$
 $933w^{236} + 670w^{235} + 769w^{234} + 56w^{233} + 503w^{232} + 740w^{231} + 406w^{230}$
 $+ 792w^{229} + 747w^{228} + 150w^{227} + 112w^{226} + 496w^{225} + 156w^{224} +$
 $328w^{223} + 187w^{222} + 827w^{221} + 696w^{220} + 343w^{219} + 386w^{218} +$
 $285w^{217} + 56w^{216} + 160w^{215} + 228w^{214} + 563w^{213} + 764w^{212} + 372w^{211}$
 $+ 897w^{210} + 721w^{209} + 443w^{208} + 60w^{207} + 673w^{206} + 740w^{205} +$
 $56w^{204} + 343w^{203} + 966w^{202} + 638w^{201} + 851w^{200} + 916w^{199} + 831w^{198}$
 $+ 894w^{197} + 275w^{196} + 458w^{195} + 809w^{194} + 981w^{193} + 610w^{192} +$
 $535w^{191} + 202w^{190} + 484w^{189} + 78w^{188} + 913w^{187} + 387w^{186} + 308w^{185}$
 $+ 412w^{184} + 927w^{183} + 447w^{182} + 779w^{181} + 819w^{180} + 885w^{179} +$
 $877w^{178} + 937w^{177} + 212w^{176} + 507w^{175} + 1002w^{174} + 334w^{173} +$
 $623w^{172} + 540w^{171} + 973w^{170} + 135w^{169} + 225w^{168} + 117w^{167} +$
 $433w^{166} + 562w^{165} + 939w^{164} + 474w^{163} + 767w^{162} + 673w^{161} +$
 $274w^{160} + 741w^{159} + 787w^{158} + 2w^{157} + 841w^{156} + 416w^{155} + 964w^{154}$
 $+ 101w^{153} + 378w^{152} + 846w^{151} + 834w^{150} + 754w^{149} + 827w^{148} +$
 $813w^{147} + 698w^{146} + 880w^{145} + 287w^{144} + 13w^{143} + 214w^{142} + 12w^{141}$
 $+ 260w^{140} + 482w^{139} + 683w^{138} + 997w^{137} + 678w^{136} + 578w^{135} +$
 $559w^{134} + 172w^{133} + 730w^{132} + 384w^{131} + 1021w^{130} + 1005w^{129} +$
 $617w^{128} + 430w^{127} + 648w^{126} + 941w^{125} + 177w^{124} + 630w^{123} +$

$341w^{122} + 752w^{121} + 805w^{120} + 208w^{119} + 1004w^{118} + 599w^{117} +$
 $1000w^{116} + 829w^{115} + 889w^{114} + 535w^{113} + 3w^{112} + 562w^{111} + 67w^{110}$
 $+ 370w^{109} + 869w^{108} + 741w^{107} + 630w^{106} + 552w^{105} + 13w^{104} +$
 $335w^{103} + 217w^{102} + 640w^{101} + 686w^{100} + 26w^{99} + 444w^{98} + 855w^{97} +$
 $907w^{96} + 102w^{95} + 378w^{94} + 500w^{93} + 85w^{92} + 825w^{91} + 233w^{90} +$
 $78w^{89} + 991w^{88} + 538w^{87} + 18w^{86} + 832w^{85} + 176w^{84} + 264w^{83} +$
 $544w^{82} + 61w^{81} + 1011w^{80} + 823w^{79} + 811w^{78} + 692w^{77} + 299w^{76} +$
 $439w^{75} + w^{74} + 949w^{73} + 850w^{72} + 617w^{71} + 11w^{70} + 250w^{69} + 198w^{68}$
 $+ 202w^{67} + 105w^{66} + 794w^{65} + 933w^{64} + 890w^{63} + 410w^{62} + 32w^{61} +$
 $128w^{60} + 151w^{59} + 220w^{58} + 432w^{57} + 777w^{56} + 761w^{55} + 366w^{54} +$
 $699w^{53} + 37w^{52} + 585w^{51} + 813w^{50} + 933w^{49} + 146w^{48} + 350w^{47} +$
 $811w^{46} + 204w^{45} + 634w^{44} + 884w^{43} + 201w^{42} + 911w^{41} + 303w^{40} +$
 $551w^{39} + 64w^{38} + 633w^{37} + 613w^{36} + 267w^{35} + 141w^{34} + 527w^{33} +$
 $576w^{32} + 161w^{31} + 116w^{30} + 649w^{29} + 666w^{28} + 1017w^{27} + 141w^{26} +$
 $750w^{25} + 216w^{24} + 634w^{23} + 452w^{22} + 601w^{21} + 30w^{20} + 432w^{19} +$
 $135w^{18} + 626w^{17} + 180w^{16} + 683w^{15} + 412w^{14} + 880w^{13} + 621w^{12} +$
 $95w^{11} + 816w^{10} + 913w^9 + 700w^8 + 478w^7 + 460w^6 + 816w^5 + 668w^4 +$
 $472w^3 + 741w^2 + 250w + 347], 8w^{255} + 4w^{254} + 13w^{252} + 9w^{251} +$
 $5w^{249} + w^{248} + w^{246} + 13w^{245} + 11w^{244} + 6w^{243} + 13w^{242} + 12w^{241} +$
 $8w^{240} + 10w^{239} + 15w^{238} + 9w^{237} + w^{236} + 9w^{235} + 7w^{234} + 2w^{233} +$
 $10w^{232} + 6w^{231} + w^{229} + 6w^{228} + 6w^{227} + 11w^{226} + 10w^{225} + w^{224} +$
 $5w^{223} + 14w^{222} + 13w^{221} + 4w^{220} + 4w^{218} + 12w^{217} + 5w^{216} + 8w^{215}$
 $+ 2w^{214} + 4w^{213} + 8w^{212} + 14w^{211} + 4w^{210} + 6w^{209} + 7w^{208} +$
 $11w^{207} + 12w^{206} + 15w^{205} + 3w^{204} + 11w^{203} + 14w^{202} + 7w^{201} +$
 $14w^{200} + 6w^{199} + 7w^{198} + 8w^{197} + 5w^{196} + 3w^{195} + w^{194} + 8w^{193} +$
 $13w^{192} + 4w^{191} + 15w^{190} + 9w^{189} + 10w^{188} + 7w^{187} + 8w^{186} + 3w^{185}$
 $+ 9w^{184} + 4w^{183} + 11w^{182} + w^{181} + 5w^{180} + 14w^{179} + 14w^{178} +$
 $11w^{177} + 10w^{176} + 13w^{175} + 4w^{174} + 6w^{173} + 6w^{172} + 9w^{171} + 8w^{170}$
 $+ 15w^{169} + 9w^{168} + 6w^{167} + 7w^{166} + 6w^{165} + w^{164} + 9w^{163} + w^{162} +$
 $w^{161} + 13w^{160} + 2w^{159} + 4w^{158} + 15w^{157} + 5w^{156} + 3w^{155} + 2w^{154} +$
 $12w^{153} + 8w^{152} + 3w^{151} + 13w^{150} + 4w^{149} + 4w^{148} + 7w^{147} + 8w^{146}$
 $+ 3w^{145} + 10w^{144} + 2w^{143} + 11w^{142} + 3w^{141} + 11w^{140} + 13w^{139} +$
 $12w^{138} + 13w^{137} + 8w^{136} + 5w^{135} + 6w^{134} + 9w^{133} + 2w^{132} + 2w^{131}$
 $+ 11w^{130} + 3w^{129} + 5w^{128} + 7w^{127} + w^{126} + 2w^{125} + 5w^{124} + 9w^{123} +$
 $11w^{122} + 9w^{121} + 10w^{120} + 12w^{119} + 7w^{118} + 6w^{117} + 9w^{116} + w^{115} +$
 $13w^{112} + 12w^{111} + 2w^{110} + 8w^{109} + 13w^{108} + 14w^{107} + 5w^{106} +$
 $3w^{105} + 7w^{104} + 5w^{103} + 4w^{102} + 3w^{101} + 8w^{100} + 5w^{99} + w^{98} +$
 $14w^{97} + 14w^{96} + 2w^{95} + 5w^{94} + 3w^{93} + 4w^{92} + 12w^{91} + 6w^{90} +$
 $13w^{89} + 2w^{88} + 10w^{87} + 12w^{86} + 12w^{85} + 12w^{84} + 5w^{83} + 9w^{82} +$
 $13w^{81} + 15w^{80} + 10w^{79} + 14w^{78} + 7w^{77} + 13w^{76} + 5w^{75} + 3w^{74} +$
 $9w^{73} + w^{72} + 10w^{71} + 7w^{70} + 8w^{69} + 7w^{68} + 8w^{66} + w^{65} + 3w^{64} +$
 $4w^{63} + 2w^{62} + 11w^{61} + 11w^{60} + 11w^{59} + 14w^{58} + 13w^{57} + 14w^{56} +$
 $5w^{55} + 10w^{54} + 10w^{53} + 12w^{52} + 11w^{51} + 6w^{50} + 11w^{49} + 6w^{48} +$
 $7w^{47} + 7w^{46} + 8w^{45} + 3w^{44} + 6w^{43} + 6w^{42} + 11w^{41} + w^{40} + w^{39} +$
 $w^{38} + 9w^{36} + w^{35} + 2w^{34} + 14w^{33} + 11w^{32} + 12w^{31} + 14w^{30} + 8w^{29} +$
 $10w^{28} + 4w^{27} + 14w^{25} + 9w^{24} + 4w^{23} + w^{22} + 9w^{21} + 3w^{20} + 3w^{19} +$
 $8w^{17} + 13w^{16} + 6w^{15} + 15w^{14} + 4w^{13} + 8w^{12} + 12w^{11} + 5w^9 + 10w^8$
 $+ 10w^7 + 2w^6 + 2w^5 + 2w^4 + 3w^3 + 2w^2 + 10)$

```
Decrypted ciphertext:  
b'Hello there!'
```

2 Implementação do KEM IND-CPA-secure

KeyGen

Foi utilizada a função *keyGen* definida na classe `KYBER_PKE`.

Encapsulamento da chave

Começa-se por calcular o *hash* de um nonce aleatório $m \in R_q$ que irá ser a chave secreta. Por fim, segue-se a cifragem do m (convertido em bytes) pela função *encryptCCA* com recurso à chave pública, obtendo o criptograma e uma ofuscação da chave.

Desencapsulamento da chave

Utilizando a chave privada, utiliza-se a função *decryptCCA* com o criptograma e a ofuscação da chave de modo a obter o valor aleatório m . Por fim, calcula-se o hash do m .

```
[176]: class KYBER_KEM:  
    def __init__(self, pset):  
        self.pke = self.setup(pset)  
  
    def setup(self, pset):  
        pke = KYBER_PKE(pset)  
        return pke  
  
    def H(self, b):  
        r = hashes.Hash(hashes.SHA3_256())  
        r.update(b)  
        return r.finalize()  
  
    def keyGen(self):  
        return self.pke.keyGen()  
  
    def encapsulate(self, pk):  
        m = self.pke.Rq([choice([0, 1]) for i in range(self.pke.n)])  
        key = self.H(dumps(m)[:12])  
        y, c = self.pke.encryptCCA(dumps(m)[:12], pk)  
        return y, c, key  
  
    def decapsulate(self, y, c, pk, sk):  
        m = self.pke.decryptCCA(y, c, pk, sk)  
        return self.H(m)
```

2.1 Cenário de teste do KEM

```
[177]: kem = KYBER_KEM(512)
pk, sk = kem.keyGen()

print("Encapsulating public key...")
y, c, key = kem.encapsulate(pk)

print("Decapsulating public key...")
decapsulated_key = kem.decapsulate(y, c, pk, sk)

if(key == decapsulated_key):
    print("Encapsulate and Decapsulate work!")
else:
    print("Something went wrong...")
```

Encapsulating public key...

Decapsulating public key...

Encapsulate and Decapsulate work!