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 trasnformação 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)
```

```python
    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, eR_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:

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

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. Este valor será que será utilizado juntamente, com a mensagem a cifrar x, na operação `xor` para formar a variariavel y. 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:

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

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 incial 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
```

```python
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 Â in Rq 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 Rq from Bη1
        s = []
        for i in range(self.k):
            s.insert(i,self.CBD(self.PRF(sigma,N), self.n1))
            N = N+1

        # Sample e in Rq from Bη1
        e = []
```

```python
        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 Â in Rq 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 Rq from Bη1
        rr = []
        for i in range(self.k):
            rr.insert(i,self.T.ntt(self.CBD(self.PRF(r, N), self.n1)))
            N += 1

        # Sample e1 in Rq from Bη2
        e1 = []
        for i in range(self.k):
            e1.insert(i,self.CBD(self.PRF(r, N), self.n2))
            N += 1

        # Sample e2 in Rq from Bη2
        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:
([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 + 587*w^180 + 964*w^179 + 898*w^178 + 41*w^177 + 153*w^176 + 818*w^175 + 69*w^174 + 60*w^173 + 1001*w^172 + 620*w^171 + 581*w^170 + 345*w^169 + 744*w^168 + 179*w^167 + 746*w^166 + 417*w^165 + 754*w^164 + 900*w^163 + 851*w^162 + 324*w^161 + 936*w^160 + 602*w^159 + 383*w^158 + 318*w^157 + 261*w^156 + 668*w^155 + 356*w^154 + 406*w^153 + 155*w^152 + 1013*w^151 + 494*w^150 + 629*w^149 + 133*w^148 + 309*w^147 + 644*w^146 + 531*w^145 + 710*w^144 + 91*w^143 + 328*w^142 + 234*w^141 + 236*w^140 + 812*w^139 + 977*w^138 + 697*w^137 + 651*w^136 + 722*w^135 + 817*w^134 + 906*w^133 + 207*w^132 + 547*w^131 + 699*w^130 + 109*w^129 + 171*w^128 + 24*w^127 + 409*w^126 + 1000*w^125 + 170*w^124 + 349*w^123 + 353*w^122 + 117*w^121 + 48*w^120 + 659*w^119 + 348*w^118 + 457*w^117 + 367*w^116 + 760*w^115 + 652*w^114 + 580*w^113 + 226*w^112 + 409*w^111 + 549*w^110 + 78*w^109 + 487*w^108 + 300*w^107 + 489*w^106 + 493*w^105 + 928*w^104 + 698*w^103 + 123*w^102 + 464*w^101 + 701*w^100 + 403*w^99 + 486*w^98 + 802*w^97 + 686*w^96 + 216*w^95 + 692*w^94 + 199*w^93 + 532*w^92 + 842*w^91 + 650*w^90 + 354*w^89 + 796*w^88 + 133*w^87 + 641*w^86 + 877*w^85 + 521*w^84 + 649*w^83 + 327*w^82 + 711*w^81 + 370*w^80 + 260*w^79 + 934*w^78 + 488*w^77 + 165*w^76 + 810*w^75 + 577*w^74 + 512*w^73 + 63*w^72 + 931*w^71 + 704*w^70 + 612*w^69 + 234*w^68 + 815*w^67 + 652*w^66 + 819*w^65 + 536*w^64 + 680*w^63 + 226*w^62 + 178*w^61 + 868*w^60 + 96*w^59 + 275*w^58 + 990*w^57 + 924*w^56 + 400*w^55 + 694*w^54 + 523*w^53 + 145*w^52 + 866*w^51 + 782*w^50 + 602*w^49 + 312*w^48 + 95*w^47 + 915*w^46 + 656*w^45 + 1013*w^44 + 262*w^43 + 887*w^42 + 165*w^41 + 888*w^40 + 920*w^39 + 286*w^38 + 777*w^37 + 915*w^36 + 396*w^35 + 981*w^34 + 779*w^33 + 340*w^32 + 252*w^31 + 7*w^30 + 949*w^29 + 503*w^28 + 494*w^27 + 190*w^26 + 245*w^25 + 842*w^24 + 227*w^23 + 358*w^22 + 346*w^21 + 718*w^20 + 308*w^19 + 993*w^18 + 732*w^17 + 751*w^16 + 203*w^15 + 984*w^14 + 286*w^13 + 383*w^12 + 466*w^11 + 238*w^10 + 678*w^9 + 877*w^8 + 454*w^7 + 131*w^6 + 999*w^5 + 1007*w^4 + 521*w^3 + 537*w^2 + 1005*w + 571,
171*w^255 + 602*w^254 + 62*w^253 + 859*w^252 + 537*w^251 + 294*w^250 + 336*w^249 + 140*w^248 + 769*w^247 + 731*w^246 + 971*w^245 + 380*w^244 + 887*w^243 + 674*w^242 + 428*w^241 + 997*w^240 + 606*w^239 + 794*w^238 + 263*w^237 + 933*w^236 + 670*w^235 + 769*w^234 + 56*w^233 + 503*w^232 + 740*w^231 + 406*w^230 + 792*w^229 + 747*w^228 + 150*w^227 + 112*w^226 + 496*w^225 + 156*w^224 + 328*w^223 + 187*w^222 + 827*w^221 + 696*w^220 + 343*w^219 + 386*w^218 + 285*w^217 + 56*w^216 + 160*w^215 + 228*w^214 + 563*w^213 + 764*w^212 + 372*w^211 + 897*w^210 + 721*w^209 + 443*w^208 + 60*w^207 + 673*w^206 + 740*w^205 + 56*w^204 + 343*w^203 + 966*w^202 + 638*w^201 + 851*w^200 + 916*w^199 + 831*w^198 + 894*w^197 + 275*w^196 + 458*w^195 + 809*w^194 + 981*w^193 + 610*w^192 + 535*w^191 + 202*w^190 + 484*w^189 + 78*w^188 + 913*w^187 + 387*w^186 + 308*w^185 + 412*w^184 + 927*w^183 + 447*w^182 + 779*w^181 + 819*w^180 + 885*w^179 + 877*w^178 + 937*w^177 + 212*w^176 + 507*w^175 + 1002*w^174 + 334*w^173 + 623*w^172 + 540*w^171 + 973*w^170 + 135*w^169 + 225*w^168 + 117*w^167 + 433*w^166 + 562*w^165 + 939*w^164 + 474*w^163 + 767*w^162 + 673*w^161 + 274*w^160 + 741*w^159 + 787*w^158 + 2*w^157 + 841*w^156 + 416*w^155 + 964*w^154 + 101*w^153 + 378*w^152 + 846*w^151 + 834*w^150 + 754*w^149 + 827*w^148 + 813*w^147 + 698*w^146 + 880*w^145 + 287*w^144 + 13*w^143 + 214*w^142 + 12*w^141 + 260*w^140 + 482*w^139 + 683*w^138 + 997*w^137 + 678*w^136 + 578*w^135 +

559*w^134 + 172*w^133 + 730*w^132 + 384*w^131 + 1021*w^130 + 1005*w^129 +
617*w^128 + 430*w^127 + 648*w^126 + 941*w^125 + 177*w^124 + 630*w^123 +
341*w^122 + 752*w^121 + 805*w^120 + 208*w^119 + 1004*w^118 + 599*w^117 +
1000*w^116 + 829*w^115 + 889*w^114 + 535*w^113 + 3*w^112 + 562*w^111 + 67*w^110
+ 370*w^109 + 869*w^108 + 741*w^107 + 630*w^106 + 552*w^105 + 13*w^104 +
335*w^103 + 217*w^102 + 640*w^101 + 686*w^100 + 26*w^99 + 444*w^98 + 855*w^97 +
907*w^96 + 102*w^95 + 378*w^94 + 500*w^93 + 85*w^92 + 825*w^91 + 233*w^90 +
78*w^89 + 991*w^88 + 538*w^87 + 18*w^86 + 832*w^85 + 176*w^84 + 264*w^83 +
544*w^82 + 61*w^81 + 1011*w^80 + 823*w^79 + 811*w^78 + 692*w^77 + 299*w^76 +
439*w^75 + w^74 + 949*w^73 + 850*w^72 + 617*w^71 + 11*w^70 + 250*w^69 + 198*w^68
+ 202*w^67 + 105*w^66 + 794*w^65 + 933*w^64 + 890*w^63 + 410*w^62 + 32*w^61 +
128*w^60 + 151*w^59 + 220*w^58 + 432*w^57 + 777*w^56 + 761*w^55 + 366*w^54 +
699*w^53 + 37*w^52 + 585*w^51 + 813*w^50 + 933*w^49 + 146*w^48 + 350*w^47 +
811*w^46 + 204*w^45 + 634*w^44 + 884*w^43 + 201*w^42 + 911*w^41 + 303*w^40 +
551*w^39 + 64*w^38 + 633*w^37 + 613*w^36 + 267*w^35 + 141*w^34 + 527*w^33 +
576*w^32 + 161*w^31 + 116*w^30 + 649*w^29 + 666*w^28 + 1017*w^27 + 141*w^26 +
750*w^25 + 216*w^24 + 634*w^23 + 452*w^22 + 601*w^21 + 30*w^20 + 432*w^19 +
135*w^18 + 626*w^17 + 180*w^16 + 683*w^15 + 412*w^14 + 880*w^13 + 621*w^12 +
95*w^11 + 816*w^10 + 913*w^9 + 700*w^8 + 478*w^7 + 460*w^6 + 816*w^5 + 668*w^4 +
472*w^3 + 741*w^2 + 250*w + 347], 8*w^255 + 4*w^254 + 13*w^252 + 9*w^251 +
5*w^249 + w^248 + w^246 + 13*w^245 + 11*w^244 + 6*w^243 + 13*w^242 + 12*w^241 +
8*w^240 + 10*w^239 + 15*w^238 + 9*w^237 + w^236 + 9*w^235 + 7*w^234 + 2*w^233 +
10*w^232 + 6*w^231 + w^229 + 6*w^228 + 6*w^227 + 11*w^226 + 10*w^225 + w^224 +
5*w^223 + 14*w^222 + 13*w^221 + 4*w^220 + 4*w^218 + 12*w^217 + 5*w^216 + 8*w^215
+ 2*w^214 + 4*w^213 + 8*w^212 + 14*w^211 + 4*w^210 + 6*w^209 + 7*w^208 +
11*w^207 + 12*w^206 + 15*w^205 + 3*w^204 + 11*w^203 + 14*w^202 + 7*w^201 +
14*w^200 + 6*w^199 + 7*w^198 + 8*w^197 + 5*w^196 + 3*w^195 + w^194 + 8*w^193 +
13*w^192 + 4*w^191 + 15*w^190 + 9*w^189 + 10*w^188 + 7*w^187 + 8*w^186 + 3*w^185
+ 9*w^184 + 4*w^183 + 11*w^182 + w^181 + 5*w^180 + 14*w^179 + 14*w^178 +
11*w^177 + 10*w^176 + 13*w^175 + 4*w^174 + 6*w^173 + 6*w^172 + 9*w^171 + 8*w^170
+ 15*w^169 + 9*w^168 + 6*w^167 + 7*w^166 + 6*w^165 + w^164 + 9*w^163 + w^162 +
w^161 + 13*w^160 + 2*w^159 + 4*w^158 + 15*w^157 + 5*w^156 + 3*w^155 + 2*w^154 +
12*w^153 + 8*w^152 + 3*w^151 + 13*w^150 + 4*w^149 + 4*w^148 + 7*w^147 + 8*w^146
+ 3*w^145 + 10*w^144 + 2*w^143 + 11*w^142 + 3*w^141 + 11*w^140 + 13*w^139 +
12*w^138 + 13*w^137 + 8*w^136 + 5*w^135 + 6*w^134 + 9*w^133 + 2*w^132 + 2*w^131
+ 11*w^130 + 3*w^129 + 5*w^128 + 7*w^127 + w^126 + 2*w^125 + 5*w^124 + 9*w^123 +
11*w^122 + 9*w^121 + 10*w^120 + 12*w^119 + 7*w^118 + 6*w^117 + 9*w^116 + w^115 +
13*w^112 + 12*w^111 + 2*w^110 + 8*w^109 + 13*w^108 + 14*w^107 + 5*w^106 +
3*w^105 + 7*w^104 + 5*w^103 + 4*w^102 + 3*w^101 + 8*w^100 + 5*w^99 + w^98 +
14*w^97 + 14*w^96 + 2*w^95 + 5*w^94 + 3*w^93 + 4*w^92 + 12*w^91 + 6*w^90 +
13*w^89 + 2*w^88 + 10*w^87 + 12*w^86 + 12*w^85 + 12*w^84 + 5*w^83 + 9*w^82 +
13*w^81 + 15*w^80 + 10*w^79 + 14*w^78 + 7*w^77 + 13*w^76 + 5*w^75 + 3*w^74 +
9*w^73 + w^72 + 10*w^71 + 7*w^70 + 8*w^69 + 7*w^68 + 8*w^66 + w^65 + 3*w^64 +
4*w^63 + 2*w^62 + 11*w^61 + 11*w^60 + 11*w^59 + 14*w^58 + 13*w^57 + 14*w^56 +
5*w^55 + 10*w^54 + 10*w^53 + 12*w^52 + 11*w^51 + 6*w^50 + 11*w^49 + 6*w^48 +
7*w^47 + 7*w^46 + 8*w^45 + 3*w^44 + 6*w^43 + 6*w^42 + 11*w^41 + w^40 + w^39 +
w^38 + 9*w^36 + w^35 + 2*w^34 + 14*w^33 + 11*w^32 + 12*w^31 + 14*w^30 + 8*w^29 +
10*w^28 + 4*w^27 + 14*w^25 + 9*w^24 + 4*w^23 + w^22 + 9*w^21 + 3*w^20 + 3*w^19 +

```
8*w^17 + 13*w^16 + 6*w^15 + 15*w^14 + 4*w^13 + 8*w^12 + 12*w^11 + 5*w^9 + 10*w^8
+ 10*w^7 + 2*w^6 + 2*w^5 + 2*w^4 + 3*w^3 + 2*w^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!
```