Trabalho Prático 3

Beatriz Oliveira pg50942

Bruno Jardim pg49997

1. Este problema é dedicado às candidaturas finalistas ao concurso NIST Post-Quantum Cryptography na categoria de criptosistemas PKE-KEM. Em Julho de 2022 foi selecionada para “standartização” a candidatura KYBER. Existe ainda uma fase não concluída do concurso onde poderá ser acrescentada alguma outra candidatura; destas destaco o algoritmo BIKE. Ao contrário do Kyber que é baseado no problema “Ring Learning With Errors” (RLWE) , o algoritmo BIKE baseia-se no problema da descodificação de códigos lineares de baixa densidade que são simples de implementar. A descrição, outra documentação e implementações em C/C++ destas candidaturas pode ser obtida na página do concurso NIST ou na diretoria Docs/PQC.

   1. O objetivo deste trabalho é a criação de protótipos em Sagemath para os algoritmos KYBER e BIKE.
   2. Para cada uma destas técnicas pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.

In [16]:
```
import os
from hashlib import shake_128, shake_256, sha256, sha512
from random import choice
```

In [17]:
```
n = 256

q = next_prime(3*n)
while q % (2*n) != 1:
    q = next_prime(q+1)
```

In [18]:
```
_Z.<w> = ZZ[]
R.<w> = QuotientRing(_Z, _Z.ideal(w^n + 1))

_q.<w> = GF(q)[]
_Rq.<w> = QuotientRing(_q, _q.ideal(w^n + 1))

Rq = lambda x : _Rq(R(x))
```

In [19]:
```
# Classe disponibilizda pelo professor
class NTT(object):

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

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

In [20]:
```
NTT = NTT(n,q)
```

In [21]:
```
def from_ntt(m):
    if type(m[0]) is list:
        r = []
        for i in range(len(m)):
            if type(m[i][0]) is list:
                r.append([])
                for j in range(len(m[i])):
                    r[i].append(NTT.ntt_inv(m[i][j]))
            else:
                r.append(NTT.ntt_inv(m[i]))
    else:
        r = NTT.ntt_inv(m)
    return r

def to_ntt(m):
    if type(m) is list:
        r = []
        for i in range(len(m)):
            if type(m[i]) is list:
                r.append([])
                for j in range(len(m[i])):
                    r[i].append(NTT.ntt(m[i][j]))
            else:
                r.append(NTT.ntt(m[i]))
    else:
        r = NTT.ntt(m)
    return r

def ntt_mult(ntt1,ntt2,size = n, modulo = q):
    r = []
    for i in range(size):
        r.append((ntt1[i]*ntt2[i]) % modulo)
    return r

def ntt_add(ntt1,ntt2,size = n, modulo = q):
    r = []
    for i in range(size):
        r.append((ntt1[i]+ntt2[i]) % modulo)
    return r

def ntt_sub(ntt1,ntt2,size = n, modulo = q):
    r = []
    for i in range(size):
        r.append((ntt1[i]-ntt2[i]) % modulo)
    return r
```

In [22]:
```
def Compress(X,d,q):
    coefs = []

    for coef in X.list():
        new_coef = round(2^d / q * int(coef)) % 2^d
        coefs.append(new_coef)
    return Rq(coefs)

def Decompress(X,d,q):
    coefs = []

    for coef in X.list():
        new_coef = round(q / 2^d * int(coef))
        coefs.append(new_coef)
    return Rq(coefs)

def Compress_DS(E,d,q):
    if type(E) is list:
        r = []
        for i in range(len(E)):
            if type(E[i]) is list:
                r.append([])
                for j in range(len(E[i])):
                    r[i].append(Compress(E[i][j],d,q))
            else:
                r.append(Compress(E[i],d,q))
    else:
        r = Compress(E,d,q)
    return r

def Decompress_DS(E,d,q):
    if type(E) is list:
        r = []
        for i in range(len(E)):
            if type(E[i]) is list:
                r.append([])
                for j in range(len(E[i])):
                    r[i].append(Decompress(E[i][j],d,q))
            else:
                r.append(Decompress(E[i],d,q))
    else:
        r = Decompress(E,d,q)
    return r
```

In [23]:
```
def xor(key, text):
    if len(text) > len(key):
        t1 = len(text) / len(key)
        key *= ceil(t1)
    return bytes(a ^^ b for a, b in zip(key, text))

def bytes_to_bits(bytes):
    bits = []
    for byte in bytes:
        bits8 = Integer(byte).digits(base=2,padto=8)
        bits8.reverse()
        bits += bits8
    return bits

def XOF(rho,i,j):
    return shake_128(str(i).encode() + str(j).encode() + str(rho).encode()).di

def H(s):
    return sha256(str(s).encode()).digest()

def G(a,b=""):
    digest = sha512(str(a).encode() + str(b).encode() ).digest()
    return digest[:32], digest[32:]

def PRF(s,b):
    return shake_256(str(s).encode() + str(b).encode()).digest(int(2048))

def KDF(a,b=""):
    return shake_256(str(a).encode() + str(b).encode()).digest(int(2048))
```

In [24]:
```
def Parse(bytestream,q,n):
    i = 0
    j = 0
    a = []
    while j < n and i + 2 < len(bytestream):
        d1 = bytestream[i] + 256 * bytestream[i + 1] % 16
        d2 = bytestream[i+1]//16 + 16 * bytestream[i + 2]
        if d1 < q:
            a.append(d1)
            j += 1
        if d2 < q and j < n:
            a.append(d2)
            j += 1

        i += 3
    return Rq(a)
```

In [25]:
```
def CBD(byte_array,eta):
    bits = bytes_to_bits(byte_array)
    f = []
    for i in range(0,256):
        a = sum([bits[2*i*eta+j] for j in range(eta)])
        b = sum([bits[2*i+eta+eta+j] for j in range(eta)])
        f.append(a-b)
    return R(f)
```

In [26]:
```
def multMV(M,V,k=2,n=n):
    for i in range(len(M)):
        for j in range(len(M[i])):
            M[i][j] = ntt_mult(M[i][j],V[j])

    r = [[0]*n]*k
    for i in range(len(M)):
        for j in range(len(M[i])):
            r[i] = ntt_add(r[i],M[i][j])
    return r

def multVV(V1,V2,n=n):
    for i in range(len(V1)):
        V1[i] = ntt_mult(V1[i],V2[i])

    r = [0]*n

    for i in range(len(V1)):
        r = ntt_add(r,V1[i])
    return r

def sumVV(V1,V2):
    for i in range(len(V1)):
        V1[i] = ntt_add(V1[i],V2[i])
    return V1

def subVV(V1,V2):
    for i in range(len(V1)):
        V1[i] = ntt_sub(V1[i],V2[i])
    return V1
```

Kyber KEM IND-CPA

In [27]:
```
class KyberKEM:
    def __init__(self):
        #256, 2, 7681, 3, 2, 10, 4
        self.n = n
        self.k = 2
        self.q = q
        self.eta1 = 3
        self.eta2 = 2
        self.du = 10
        self.dv = 4

    def KeyGen(self):
        d = _Rq.random_element()
        rho , sigma = G(d)
        N = 0
        A = [0,0]

        for i in range(self.k):
            A[i] = []
            for j in range(self.k):
                A[i].append(NTT.ntt(Parse(XOF(rho,j,i),self.q,self.n)))

        s = [0] * self.k
        for i in range(self.k):
            s[i] = NTT.ntt(CBD(PRF(sigma,N),self.eta1))
            N += 1

        e = [0] * self.k
        for i in range(self.k):
            e[i] = NTT.ntt(CBD(PRF(sigma,N),self.eta1))
            N += 1

        As = multMV(A,s)
        t = sumVV(As,e)

        self.pk = t,rho
        self.sk = s

        return self.pk,self.sk

    def enc(self,pk,m,coins):
        N = 0
        t, rho = pk
        A = [0,0]
        for i in range(self.k):
            A[i] = []
            for j in range(self.k):
                A[i].append(NTT.ntt(Parse(XOF(rho,i,j),self.q,self.n)))

        r = [0] * self.k
        for i in range(self.k):
            r[i] = NTT.ntt(CBD(PRF(coins,N),self.eta1))
            N += 1

        e1 = [0] * self.k
```

```
        for i in range(self.k):
            e1[i] = NTT.ntt(CBD(PRF(coins,N),self.eta2))
            N += 1

        e2 = NTT.ntt(CBD(PRF(coins,N),self.eta2))

        Ar = multMV(A,r)
        u = sumVV(Ar,e1)

        t = t[::]

        tr = multVV(t,r)
        v = ntt_add(tr,e2)
        v = ntt_add(v,NTT.ntt(m))

        u = from_ntt(u)
        v = from_ntt(v)

        c1 = Compress_DS(u,self.du,self.q)
        c2 = Compress_DS(v,self.dv,self.q)

        return c1,c2

    def dec(self,c):
        u, v = c
        u = Decompress_DS(u,self.du,self.q)
        v = Decompress_DS(v,self.dv,self.q)

        u = to_ntt(u)
        v = to_ntt(v)



        su = multVV(self.sk,u)

        m = ntt_sub(v,su)

        return Compress(NTT.ntt_inv(m),1,q)

    def KEM(self,pk):
        m = Rq([choice([0,1]) for _ in range(self.n)])
        coins = os.urandom(256)
        e = self.enc(pk,Decompress(m,1,q),coins)
        k = H(m)

        return e,k

    def KRev(self,e):
        m = self.dec(e)
        k = H(m)
        return k
```

In [28]:
```
kyber1 = KyberKEM()
kyber2 = KyberKEM()


pk1, sk1 = kyber1.KeyGen()
pk2, sk2 = kyber2.KeyGen()


e, key_sender = kyber1.KEM(pk2)
key_receiver = kyber2.KRev(e)

mitm_key = kyber1.KRev(e)

print(key_sender)
print(key_receiver)

print(key_sender == key_receiver)
print(mitm_key == key_receiver)
```

```
b'\x915\xf0\x06\xdee\xc0\xf2\x08[K\xd1FT?E?\x1c\x00\xc4\xcc\xa3\x12T\xff.Y\\\
xc4vt;'
b'\x915\xf0\x06\xdee\xc0\xf2\x08[K\xd1FT?E?\x1c\x00\xc4\xcc\xa3\x12T\xff.Y\\\
xc4vt;'
True
False
```

Kyber PKE IND-CCA (Transformação de Fujisaki-Okamoto)

A transformação FO original constrói um novo esquema de cifra assimétrica (E', D') , usando um novo "hash" aleatório g de tamanho igual ao da mensagem x .

O algoritmo de cifra é

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

As características essenciais deste algoritmo de cifra são

1. O parâmetro r , que na cifra original introduz a aleatoriedade, é nesta cifra transformado de duas formas diferentes
   A. Em primeiro lugar, via o "hash" g , é usado para construir uma ofuscação y da mensagem original.
   B. Em segundo lugar, r é misturado com y para construir via o hash h uma nova fonte de aleatoriedade $r' = h(r\|y)$
2. O par formado (y, c) pelo criptograma y e o criptograma que resulta de, com o f original, cifrar r com a aleatoriedade r' é o resultado da nova cifra.

O objetivo destas transformações da cifra original é construir um um algoritmo de decifra D' que permita recuperar a mensagem x mas também verificar a autenticidade do criptograma.

O algoritmo D' rejeita o criptograma se detecta algum sinal de fraude.

In [29]:
```
class KyberPKE:
    def __init__(self):
        self.kyberkem = KyberKEM()

    def KeyGen(self):
        self.pk,self.sk = self.kyberkem.KeyGen()
        return self.pk,self.sk

    def encrypt(self,pk,m):

        x = m.encode()
        r = Rq([choice([0,1]) for _ in range(self.kyberkem.n)])
        g = H(r)

        y = xor(g,x)

        ry = bytes(r) + y
        ry = H(ry)

        c = self.kyberkem.enc(pk,Decompress(r,1,self.kyberkem.q),ry)

        return y,c

    def decrypt(self,y,c):
        r = self.kyberkem.dec(c)


        ry = H(bytes(r) + y)
        new_c = self.kyberkem.enc(self.pk,Decompress(r,1,self.kyberkem.q),ry)

        if new_c != c:
            raise Exception("Could not decrypt")

        g = H(r)

        m = xor(g,y).decode()
        return m
```

In [30]:
```
kybercca = KyberPKE()
pk, sk = kybercca.KeyGen()

y,c = kybercca.encrypt(pk,"INDCCA-KyberPKE")

m = kybercca.decrypt(y,c)

print(m)

#m = kybercca.decrypt(y+b'0',c)
```

```
INDCCA-KyberPKE
```

In [ ]: