kyber

May 2, 2022

1 KYBER

[ ]:
```
from sage.all import *
import sys
import math
import hashlib
```

[ ]:
```
n=256
n1=3
n2 = 2
q=3329
k = 2
du = 10
dv = 4
K = GF(q)

#Definição dos anéis a serem usados

R = PolynomialRing(ZZ, 'a') ; a = R.gen()
RingRq   = R.quotient(a ^ n + 1, 'x')
x = RingRq.gen()
VectorRq = MatrixSpace(RingRq, k, 1)
MatrixRq = MatrixSpace(RingRq,k, k)
```

2 CLASSE NTT

[ ]:
```
#Classe NTT para a sua conversão
class NTT(object):
    def __init__(self, n=16, q=None, 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):
            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␣
↪range(N_)]
            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)
```

3 METODOS AUXILIARES

[ ]:
```
#Definição do parse segundo o algoritmo 1 da documentação do KYBER
#Esta função recebe um byte stream
#Retorna um polinomio
def parse(byte_array):
        i = 0 ; j = 0 ;
        a = [0] * n
        while j < n:
            d1 = byte_array[i] + 256 * (byte_array[i + 1] % 16)
            d2 = round(byte_array[i + 1] / 16) + 16 * byte_array[i + 2]
            if d1 < q:
                a[j] = d1
                j += 1
            if  d2 < q and j < n:
                a[j] = d2
                j += 1
            i += 3
        return RingRq(a)


#função que converte um array de bytes no array de bits
#sendo que cada byte são 8 bits são aplicadas as respetivas transformações
def bytes_to_bits( byte_array):
        bit_array = []
        byte_array_length = len(byte_array) * 8
        for i in range(byte_array_length):
            base  = int(i // 8)
            shift = int(i % 8)
            bit_array.append( byte_array[base] >> shift & 0x1 )
        return bit_array

#Definição do CBD segundo o algoritmo 2 da documentação do KYBER
#Recebe um array de bytes e um tamanho
# Retorna um polinomio que pertence ao RingRq
def CBD(byteArrray,n1):
    bitray=bytes_to_bits(byteArrray)

    f = []
```

```
    for i in range(256):
        a=0
        b=0
        for j in range(n1):
            a=a+bitray[2*i*n1 + j]
            b=b+bitray[2*i*n1 + n1 + j]
        f.append(a - b)
    return RingRq(f)

#função adaptada do código fornecido pelos autores do KYBER
#esta função recebe um elemento subtrai-lhe q e aplica
#um shift de 15 seguido de um xor com q
def csubq(a):
  a -= q
  a += (a >> 15) & q
  return a

#função adaptada do código fornecido pelos autores do KYBER
#função que aplica a csubq a todos os elementos de um polinomio
def poly_csubq(r):
    result=[]
    for i in range(n):
        a= csubq(r[i])
        result.append(a)
    return result

#função adaptada do código fornecido pelos autores do KYBER
#função que transforma um Polinomio num array de bytes
# esta aplica a csubq a todos os membros do polinomio em questão
# posteriormente realiza os shifts necessários
#e concatena tudo num array de bytes
def encode(p,l):
    byteArray= []
    result=poly_csubq(p.list())
    for i in range(n/2):
        t0=result[2*i]
        t1=result[2*i+1]
        byteArray.append(t0 >> 0)
        byteArray.append((t0 >> 8) | (t1 << 4))
        byteArray.append(t1 >> 4)
    return byteArray


#função que aplica o encode definido anteriormente
#  a todos os membros do vetor
def encode_vector(p,l):
    poli_list  = p.list()
```

```
    byte_array = []
    for i in range(k):
        byte_array += encode(poli_list[i],l)
    return byte_array



#função adaptada do código fornecido pelos autores do KYBER
#função que transforma um array de bytes num Polinomio
#Esta aplica as transformações ao array de bytes para polinomio
def decode(byteArray):
    f = [None] * n
    for i in range(n / 2):
        f[2 * i + 0] = ( ((byteArray[3 * i + 0] >> 0) | (byteArray[3 * i + 1]
 ↪<< 8)) & 0xff )
        f[2 * i + 1] = ( ((byteArray[3 * i + 1] >> 4) | (byteArray[3 * i + 2]
 ↪<< 4)) & 0xff )
    return RingRq(f)



#Realiza também o decode mas em vez de ser do byteArray
#é da mensagem
def poly_frommsg(msg):
    f = [None] * n
    for i in range (n/8):
        for j in range(8):
            mask = ((-msg[i] >> j) &1)
            f[8*i+j] = mask & (int((q+1)/2))
    return RingRq(f)



#Aplica a decode a todos os elementos do vetor
def decode_vector( byte_array,l):
    size = len(byte_array) // (32 * l)
    f=[]
    for i in range(0,size):
            f.append(decode( byte_array[i * 32 * l : (i + 1) * 32 * l]))
    return VectorRq(f)


#Aplica a poly_frommsg a todos os elementos do vetor
def decode_vector_32( byte_array, l):
    size = len(byte_array) // (32 * l)
    f    = [None] * k
    for i in range(size):
```

```
        f[i] = poly_frommsg( byte_array[i * 32 * l : (i + 1) * 32 * l] )
    return VectorRq(f)



def hash_G(data):
        h = hashlib.sha3_512()
        h.update(data)
        res=h.digest()
        p=res[0:32]
        sigma=res[32:]
        return (p,sigma)

def hash_G1(data):
        h = hashlib.sha3_512()
        h.update(data)
        return h.digest()

def hash_H( data):
        hash_funtion = hashlib.sha3_256()
        hash_funtion.update(data)
        return hash_funtion.digest()


def XOF(data,i,j):
    h = hashlib.shake_128()
    h.update(data + j.to_bytes(4,'little') + i.to_bytes(4,'little'))
    return h.digest(q)

def prf( data, N1):
    Nb = int(N1).to_bytes(4, "little")
    seed = data + Nb
    h= hashlib.shake_256()
    h.update(seed)
    return h.digest(q)

#pega num elemento do ringRq e retorna um elemento no espaço {0, ..., 2 ^ d -␣
 ↪1}. onde d < [log2(q)]
def compress_intl( x, d):
    return round( ((2 ^ d) / q) * x ) % (2 ^ d)

#função que comprime polinomios aplicando
# a função compress__intl a todos os seus membros
def compress_poly( x, d):
    res   = []
    coefs = x.list()
    for coef in coefs:
```

```
        res.append( compress_intl(coef, d) )
    return RingRq(res)


#função que comprime vetores aplicando
# a função compress_poly a todos os seus membros
def compress_vector( x, d):
    res   = []
    coefs = x.list()
    for coef in coefs:
        res.append( RingRq(compress_poly(coef, d)) )
    return VectorRq(res)

#Faz o inverso da compress_intl
def decompress_intl( x, d):
    return round( (q / (2 ^ d))  * x )



#Faz o inverso da compress_poly
def decompress_poly( x, d):
    res   = []
    coefs = x.list()
    for coef in coefs:
        res.append( decompress_intl(coef, d) )
    return RingRq(res)


#Faz o inverso da compress_vector
def decompress_vector( x, d):
    res   = []
    coefs = x.list()
    for coef in coefs:
        res.append( RingRq(decompress_poly(coef, d)) )
    return VectorRq(res)
```

4 PKE

[ ]:
```
#Definição da keygen segundo o algoritmo 4 da documentação do KYBER
#A keygen retorna uma chave privada e uma chave pública
def keygen():
    Zs.<x>  = ZZ[]
    d=os.urandom(32)
    (ro,sigma)=hash_G(d)
    N=0
    A = []
```

```
    #gera-se a matriz a
    for i in range(k):
        A.append([])
        for j in range(k):
            A[i].append(parse(XOF(ro,i,j)))
    A = MatrixRq(A)

    #gera-se o polinomio s
    s=[]
    for i in range(k):
        s.append(CBD(prf(sigma,N),n1))
        N = N + 1

    #gera-se o polinomio e
    e=[]
    for i in range(k):
        e.append(CBD(prf(sigma,N),n1))
        N = N + 1

    #aplicação da transformação NTT
    nTT = NTT()
    s_ntt = VectorRq( [nTT.ntt(s[i]) for i in range(k)] )
    e_ntt = VectorRq( [nTT.ntt(e[i]) for i in range(k)] )
    t = compress_vector((A * s_ntt) + e_ntt,11)

    #chave publica sendo a codificação
    # do vetor t e posterior concatenação com o ro
    pk=encode_vector(t,12) +list(ro)

    #chave privada sendo a codificação do vetor s
    sk=encode_vector(s_ntt,12)
    return pk,sk



#Definição da enc segundo o algoritmo 5 da documentação do KYBER
#recebe a chave publica,uma mensagem de 32 bytes e um coins de 32 bytes
#retorna uma mensagem cifrada
def enc(pk,message,coins):
    N=0
    #descodificação da chave publica
    t=decode_vector(pk,12)
    ro = pk[12 * k * n / 8:]
    ro1=bytes(ro)

    #gera-se a matriz A no dominio NTT
    A = []
```

```
    for i in range(k):
        A.append([])
        for j in range(k):
            A[i].append(parse(XOF(ro1,i,j)))
    A  = MatrixRq(A)
    At = A.transpose()

    #gera-se o polinomio r pertencente a RingRq
    r=[]
    for i in range (0,k):
        r.append(CBD(prf(coins,N),n1))
        N = N + 1

    #gera-se o polinomio e pertencente a RingRq
    e1=[]
    for i in range(0,k):
        e1.append(CBD(prf(coins,N),n2))
        N=N+1

    #gera-se e2
    e2=CBD(prf(coins,N),n2)

    #transformação de e1 para MatrixSpace
    e1=VectorRq(e1)

    #transformação ntt
    ntt=NTT()
    r_ntt = VectorRq( [ntt.ntt(r[i]) for i in range(k)] )
    u=At*r_ntt
    u_ntt=VectorRq( [ntt.ntt(u[i][0],inv=True) for i in range(k)] ) + (e1)

    vt=t.transpose()*r_ntt
    v_ntt=RingRq(ntt.ntt(vt[0][0], inv=True) ) +e2 +␣
↪decompress_poly(poly_frommsg(message),1)

    #concatenação de c1 que e a codificação do vetor u
    #com c2 que é a codificação do polinomio v
    c1 = encode_vector( compress_vector(u_ntt, du) ,du)
    c2 = encode( compress_poly(v_ntt, dv) ,dv)
    return c1+c2



#Definição da dec segundo o algoritmo 6 da documentação do KYBER
#recebe a chave privada e o texto cifrado
#retorna uma mensagem decifrada
def dec(sk,ct):
```

```
    #divisão do texto cifrado no u e no v
    c_u = ct[:du * k * n / 8]
    c_v = ct[du * k * n / 8:]

    #realiza-se o processo inverso a enc
    u=decompress_vector(decode_vector_32(c_u,du),du)
    v = decompress_poly(decode(c_v), dv )
    s=decode_vector(sk,12)
    #aplica-se a transformação ntt
    nTT = NTT()

    #Faz return da mensagem m
    u_ntt = VectorRq( [nTT.ntt(u.list()[i]) for i in range(k)] )
    m=encode(compress_poly(v-(u_ntt*s.transpose())[0][0],1),1)
    return m
```

5 TESTE PKE

[ ]:
```
pk,sk=keygen()
print("pk",len(pk))
print("sk",len(sk))
coins = os.urandom(32)
msg = os.urandom(32)

p=enc(pk,msg,coins)
m=dec(sk, p)
print(m==list(msg))
```

```
pk 800
sk 768
False
```

6 METODOS AUXILIARES - KEM

[ ]:
```
def byteArrToBytes( btArray) :
        byts = b''
        for i in btArray :
            byts += int(i).to_bytes(1,'little')
        return byts

def kdf( data, length):
    hash_funtion = hashlib.shake_256()
    hash_funtion.update(data)
    return hash_funtion.digest(length)

def __int_to_bytes( x):
```

```
        bt = b""
        for i in range( len(x) ):
            bt += int(x[i]).to_bytes((int(x[i]).bit_length() + 7) // 8, 'big')
        return bt
```

7 KEM

[ ]:
```
#Definição da keygen segundo o algoritmo 7 da documentação do KYBER
#A keygen_kem retorna uma chave privada e uma chave pública
def keygen_kem():
    z=os.urandom(32)
    (pk,sk)=keygen()
    sk1= __int_to_bytes(sk)+__int_to_bytes(pk)+z+hash_H(__int_to_bytes((pk)))
    return pk,sk1



#Definição da enc segundo o algoritmo 8 da documentação do KYBER
#recebe a chave publica
#retorna uma mensagem cifrada
def enc_kem(pk):
    #geração de uma mensagem de 32 bytes
    m=os.urandom(32)

    #aplicação do hash(sha3_256)
    m1=hash_H(m)


    kr=hash_G1(m1+hash_H((__int_to_bytes(pk))))
    k=kr[:32]
    r=kr[32:]

    #utiliza-se o enc da pke para gerar uma mensagem encriptada
    c=enc(pk,m1,r)

    #aplica-se outra transformação
    k1=kdf(k+hash_H(__int_to_bytes(c)),32)
    return (c,k1)


#Definição da enc segundo o algoritmo 9 da documentação do KYBER
#recebe a chave secreta e o texto cifrado
#retorna a chave partilhada
def dec_kem(c,sk):
    #retira-se os elementos necessários da chave secreta
```

```
    pk = sk[12 * k * n / 8:]
    h  = sk[24 * k * n / 8 + 32:]
    z  = sk[24 * k * n / 8 + 64:]
    sk1=list(sk)
    #aplica-se o dec da pke para obeter uma mensagem decifrada
    m=dec((sk1),c)

    #processo inverso da enc
    kr=hash_G1(m+h)
    kp, r = kr[:32], kr[32:]
    c1=enc(pk,m,r)
    if(c==c1):
        k1=kdf(kp+hash_H(c),32)
    else:
        k1=kdf(z+hash_H(c),32)
    return k1
```

8 TESTE KEM

Como tivemos problemas na dec_kem, nenhum teste foi feito.

[ ]:
```
pk, sk = keygen_kem()
c, K = enc_kem(pk)
```

[ ]:
```
K = dec_kem(c, sk)
```