KYBER

May 10, 2021

1 TP2 - Estruturas Criptográficas

1.1 Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

1.1.1 KYBER-PKE

[1]:
```
import os
import math
import sys
import hashlib
import random as rn
import numpy as np


class KYBER:
    def __init__(self):                                         #Geração dos␣
↪parâmetros
        self.n = 256
        self.k = 2
        self.q = 3329

        self.n1 = 3
        self.n2 = 2
        self.du = 10
        self.dv = 4
        self.delta = 2^(-139)
        self.K = GF(self.q)
        self.zero = self.K(0)


        self.Z = PolynomialRing(ZZ,'x')                         #Geração dos␣
↪Anéis, Matrizes e Vetores
        self.F = GF(self.q)
        self.R_original = PolynomialRing(self.F,'x')
        Rq = self.R_original.quotient(x**(self.n) + 1,"a")
```

```
        self.Rq = Rq


        self.R = PolynomialRing(self.K,name='w')
        self.x = self.R.gen()
        self.Rr = QuotientRing(self.R,self.R.ideal((self.x^self.n)+1))

        self.Mr  = MatrixSpace(self.Rr,self.k,self.k)
        self.Vec = MatrixSpace(self.Rr,self.k,1)



    def rounding(self, x):                                              ␣
↪#Arredondamento para cima
        return math.ceil(x)


    def parse(self, bytray):                                            #Gera um␣
↪polinómio em Rq()
        i = 0
        j = 0
        a = 0
        Zx.<x>  = ZZ[]
        while j < self.n :
            d1 = bytray[i] + 256 * (bytray[i+1] % 16)
            d2 = math.floor(bytray[i+1]/16) + 16 * bytray[i+2]
            if d1 < self.q:
                a = a + d1 * x^j
                j = j + 1
            if d2 < self.q and j < self.n:
                a = a + d2 * x^j
                j = j + 1
            i = i + 3
        return a

    def access_bit(self, data, num):                                    #Passa de␣
↪Bytes para bits
        base = int(num // 8)
        shift = int(num % 8)
        return (data[base] & (1<<shift)) >> shift

                                                                        #Gera um␣
↪polinómio em Rq() de acordo
    def cbd(self, bytray, n):                                           #com a␣
↪distribuição polinomial
        bitray = [self.access_bit(bytray,i) for i in range(len(bytray)*8)]
        f = 0
```

```
        Zx.<x>  = ZZ[]
        for i in range(256):
            som = 0
            for j in range(n):
                som += bitray[2*i*n + j]
            a = som

            som = 0
            for j in range(n):
                som += bitray[2*i*n+ n + j]
            b = som

            f = f + (a - b) * x^i
        return self.Rr(f)


    def decode(self, bytray):                                    #Passa de␣
↪bytes para polinómio
        bitray = [self.access_bit(bytray,i) for i in range(len(bytray)*8)]
        f = 0
        Zx.<x>  = ZZ[]
        l = len(bytray)//32
        for i in range(256):
            som = 0
            for j in range(l):
                som += bitray[i*l+j] * 2^j
            f = f + som * x^i

        return self.Rr(f)



    def encode(self,s):                                          #Passa de␣
↪polinómio para bytes
        r = []
        t0 = 0
        t1 = 0
        a = list(map(lift,list(s)))
        for i in range(self.n/2):
            t0 = a[2*i]
            t1 = a[2*i+1]

            r.append((t0 >> 0)&0xff)
            r.append((t0 >> 8) | (t1 << 4)&0xff)
            r.append((t1 >> 4)&0xff)

        return r
```

```
    def encode_vec(self, vec):                                  #Passa um vetor␣
↪para bytes
        lst = vec.list()
        res = bytearray()
        for poly in lst:
            res+=(bytearray(self.encode(poly)))
        return res

    def decode_vec(self, bytarr):                               #Passa de bytes␣
↪para vetor
        size = len(bytarr)/384
        res = []
        for x in range(size):
            res.append(self.decode(bytarr[x*384:(x+1)*384]))

        return self.Vec(res)

    def hash_sha256(self, M):                                   #Função Hash␣
↪(SHA256)
        m = hashlib.sha256()
        m.update(M)
        return m.digest()

    def hash_sha512(self, M):                                   #Função Hash␣
↪(SHA512)
        m = hashlib.sha512()
        m.update(M)
        return m.digest()

    def xof(self, M, tam):                                      #Função XOF␣
↪(SHAKE-128)
        m = hashlib.shake_128()
        m.update(M)
        return m.digest(int(tam))

    def prf(self, M, tam):                                      #Função PRF␣
↪(SHAKE-256)
        m = hashlib.shake_256()
        m.update(M)
        return m.digest(int(tam))

    def kdf(self, M, tam):                                      #Função KDF
        m = hashlib.shake_256()
        m.update(M)
        return m.digest(int(tam))
```

```
    def compress(self,x,d):                                          #Função para␣
↪Comprimir
        h = list(x)
        l = []
        if d < self.rounding(log(self.q,2)):
            for i in h:
                res = self.rounding(((2^d)/self.q) * i.lift()) % 2^d
                l.append(res)
        return l

    def decompress(self, x, d):                                      #Função para␣
↪Descomprimir
        h = list(x)
        l = []
        for i in h:
            res = self.rounding((self.q / (2^d)) * (i.lift()))
            l.append(res)
        return l


    def keygen(self):                                                #Gera chave␣
↪privada e pública
        Zs.<x>  = ZZ[]

        d = os.urandom(32)

        # phi, ro
        phiro = self.hash_sha512(d)
        phi, ro = phiro[:32], phiro[32:]

        n = 0

        a = [None] * ((self.k)* (self.k))
        s = [None] * (self.k)
        e = [None] * (self.k)
                                                                     # Gera Matriz A␣
↪em dominio NTT
        for i in range(self.k):
            for j in range(self.k):
                jb = j.to_bytes((j.bit_length() + 7) // 8, 'big')
                ib = i.to_bytes((i.bit_length() + 7) // 8, 'big')
                seed = phi + jb + ib
                index = i*(self.k)+j
                a[index] = (self.parse(self.xof(seed,(i+1*j+1)*490)))

        A = self.Mr(a)
```

```
                                                        # Gera vetor␣
↪polinómio s em Rq()
        for i in range(self.k):
            nb = int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')
            s[i] = self.cbd(self.prf(ro+nb, 64*self.n1), self.n1)
            n = n + 1

        s_vec = self.Vec(s)

                                                        # Gera vetor␣
↪polinomio e em Rq()
        for i in range(self.k):
            nb = int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')
            e[i] = self.cbd(self.prf(ro+nb,64*self.n1),self.n1)
            n = n + 1
        e_vec = self.Vec(e)
                                                        # Multiplicação da␣
↪Matriz A com o vetor s
        As = A * s_vec
                                                        # Soma do vetor As␣
↪com o vetor e
        t = As + e_vec

        pk = self.encode_vec(t) + phi                   # Gera chave␣
↪pública
        sk = bytearray(self.encode_vec(s_vec))          # Gera chave␣
↪privada
        return(pk,sk)


    def enc(self, pk, m, r):
        Zs.<x>  = ZZ[]

        rr = [None] * (self.k)
        e1 = [None] * (self.k)

        a = [None] * ((self.k)* (self.k))
        n = 0

        poly_len = len(pk)//384
        phi_len = len(pk) - (poly_len * 384)

        phi = pk[(poly_len * 384):]
```

```
        t = self.decode_vec(pk[:-phi_len])

                                                            # Gera Matriz A␣
↪em NTT
        for i in range(self.k):
            for j in range(self.k):
                index = i*(self.k)+j
                jb = j.to_bytes((j.bit_length() + 7) // 8, 'big')
                ib = i.to_bytes((i.bit_length() + 7) // 8, 'big')
                seed = phi + jb + ib
                a[int(index)] = self.parse(self.xof(seed,(i+1*j+1)*490))

        A = self.Mr(a)                                      # Matriz A

        a_transpose = A.transpose()                         # Transposta da␣
↪Matriz A

                                                            # vetor r em Rq
        for i in range(self.k):
            nb = int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')
            rr[i] = self.cbd(self.prf(r+nb,64*self.n1),self.n1)
            n = n + 1
        r_vec = self.Vec(rr)                                # vetor r em Rq

        for i in range(self.k):
            nb = int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')
            e1[i] = self.cbd(self.prf(r+nb,64*self.n2),self.n2)
            n = n + 1
        e1_vec = self.Vec(e1)                               # vetor e1 em Rq


        nb = int(n).to_bytes((int(n).bit_length() + 7) // 8, 'big')
        e2 = self.cbd(self.prf(r+nb,64*self.n2),self.n2)      # e2 em Rq


        Ar = a_transpose * r_vec                           # Multiplicação da␣
↪matriz transposta a com o vetor r

        u = Ar + e1_vec                                    # Soma do vetor Ar␣
↪com o vetor e1

        Tr = t.transpose() * r_vec                         # Multiplicação do␣
↪vetor transposto r com o vetor r

        tr_e2 = (Tr[0][0]) + self.Rr(e2)                   # Soma de␣
↪Polinómios tr e e2
```

```
        a = self.Rr(self.decompress(self.decode(m),1))
        v = tr_e2 + a

        c1 = []
        for x in range(self.k):
            c1.append(self.compress(u[x][0],self.du))


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

        c1_lst_poly = list(map(self.Rr,c1))

        c1_vec = self.encode_vec(self.Vec(c1_lst_poly))
        c1_dec = self.decode_vec(c1_vec)

        c2_enc = bytearray(self.encode(self.Rr(c2)))
        c = c1_vec + c2_enc

        return c



    def dec(self, sk, c):
        c1_enc = c[:(12*self.k*(self.n/8))]                          # Separa o vetor c em c1 e c2
        c2_enc = c[(12*self.k*(self.n/8)):]

        c1 = self.decode_vec(c1_enc).list()                          # Decode nos vetores
        c2 = self.decode(c2_enc)
        u = []

        for x in range(self.k):
            res = self.decompress(self.Rr(list(c1[x])),self.du)
            u.append(self.Rr(res))
        u_vec = self.Vec(u)                                          # Gera o vetor u_vec

        v = self.decompress(self.Rr(c2), self.dv)

        s = self.Vec(self.decode_vec(sk))
        su = s.transpose() * u_vec

        m = bytearray(self.Rr(self.compress(self.Rr(v) - su[0][0],1)))
        return m
```

```
[2]: kyber = KYBER()

#Gerar as chaves
pk, sk = kyber.keygen()
#Gerar a seed
message = os.urandom(32)
#Decode da mensagem
a = kyber.decode(message)
#Gerar as coins
coins = os.urandom(32)
#Cifrar a mensagem
c = kyber.enc(pk, message, coins)
#Decifrar a mensagem
m = kyber.dec(sk, c)
#Gerar a mensagem em polinómio
m_poly = kyber.decode(message)
#Verificação
print((m) == (m_poly))
```

```
False
```

1.2 KYBER-KEM

```
[3]: class KYBER(KYBER):

    def key_gen_kem(self):                                              #Gera as chaves
        z = os.urandom(32)
        pk, sk = self.keygen()
        sk_ = sk + pk + self.hash_sha256(pk) + z
        return pk, sk_

    def encaps(self, pk):                                               #Encapsular a␣
 ↪chave
        m = os.urandom(32)
        m_hash = self.hash_sha256(m)
        pk_hash = self.hash_sha256(pk)
        kr   = self.hash_sha512(m_hash+pk_hash)
        k, r = kr[:32], kr[32:]
        c = self.enc(pk, m_hash, r)
```

```python
        k = self.kdf(k+self.hash_sha256(c),32)
        return c, k

    def decaps(self, sk_, c):                                   #Encapsular a␣
↪chave

        sk_tam = 12*self.k*(self.n/8)
        pk_tam = (12*self.k*(self.n/8)+32)

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


        m = self.dec(sk, c)

        kr = self.hash_sha512(m+h)
        k, r = kr[:32], kr[32:]

        c_ = self.enc(pk,m,r)
        if c == c_:
            return self.kdf(k+self.self.hash_sha256(c))
        else:
            print("Erro")
```

1.3 TESTE KYBER-KEM

[4]:
```python
kyber = KYBER()
#Geração da chave pública e privada
pk, sk = kyber.key_gen_kem()
#Encapsulamento da Chave
c, key = kyber.encaps(pk)
#Desencapsulamento da Chave
k = kyber.decaps(sk, c)
```

```
Erro
```

[ ]: