BYKE

May 10, 2021

1 TP2 - Estruturas Criptográficas

1.1 Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

1.2 BYKE-KEM

[1]:
```
import random as rn
from cryptography.hazmat.primitives import hashes
import numpy as np

class BYKE:                                                              #␣
 ↪Geração dos parâmetros
    def __init__(self):
        self.n = 514
        self.r = 257
        self.w = 6
        self.t = 16

        self.K = GF(2)
        self.um = self.K(1)
        self.zero = self.K(0)

        self.R = PolynomialRing(self.K,name='w')                         #␣
 ↪Geração dos polinómios, Matrizes e vetores
        self.x = self.R.gen()
        self.Rr = QuotientRing(self.R,self.R.ideal((self.x^self.r)-1)) #␣
 ↪Matrizes circulantes de tamanho r com r primo
        self.Mr  = MatrixSpace(self.K,self.n,self.r)

        self.Vn  = VectorSpace(self.K,self.n)
        self.Vr  = VectorSpace(self.K,self.r)
        self.Vq  = VectorSpace(QQ,self.r)


    def rot(self,h):
```

```
        v = self.Vr() ; v[0] = h[-1]
        for i in range(self.r-1):
            v[i+1] = h[i]
        return v

    def Rot(self,h):
        M = Matrix(self.K,self.r,self.r) ; M[0] = self.expand(h)
        for i in range(1,self.r):
            M[i] = self.rot(M[i-1])
        return M


    def expand(self, f):                                        # Função␣
↪poly para passar para 1 vetor
        fl = f.list(); ex = self.r - len(fl)
        return self.Vr(fl + [self.zero]*ex)

    def expand2(self, code):                                    # Função␣
↪para passar para 2 vetores
        (f0,f1) = code
        f = self.expand(f0).list() + self.expand(f1).list()
        return self.Vn(f)


    def unexpand2(self, vec):                                   #␣
↪Transforma um vetor num poly
        u = vec.list()
        return (self.Rr(u[:self.r]),self.Rr(u[self.r:]))


    def sparse_pol(self):                                       # Gerar␣
↪um polinómio mónico sparse
        sparse= int(self.w//2)
        coeffs = [1]*sparse + [0]*(self.r-2-sparse)
        rn.shuffle(coeffs)
        return self.Rr([1]+coeffs+[1])

    def geng(self):                                             # Gerar␣
↪um polinómio denso
        sparse= int(self.r//2)
        coeffs = [1]*(self.r-sparse) + [0]*sparse
        rn.shuffle(coeffs)
        return self.Rr(coeffs)

    def hash_key(self, a):
        uu  = np.packbits(list(map(lift,self.expand2(a))))
        hsh =  hashes.Hash(hashes.SHAKE256(int(256)))
```

```
        hsh.update(uu)
        return hsh.finalize()


    # produz um par de polinomios dispersos de tamanho "r" com um dado número total de erros "t"
    def noise(self,t):
        el = [self.um]*t + [self.zero]*(self.n-t)
        rn.shuffle(el)
        return self.Rr(el)

    def mask(self,u,v):                                    # Produto componente a componente
        return u.pairwise_product(v)

    def hamm(self,u):                                      # Peso de Hamming
        return sum([1 if a == self.um else 0 for a in u])



    def BF2(self, H,code,synd, errs=0):                    # Bit-Flip com ruído até t/2

        mycode = code
        mysynd = synd
        cnt_iter= self.r

        while cnt_iter > 0 and self.hamm(mysynd) > self.t//2:
            cnt_iter   = cnt_iter - 1

            unsats     = [self.hamm(self.mask(mysynd,H[i])) for i in range(self.n)]
            max_unsats = max(unsats)

            for i in range(self.n):
                if unsats[i] == max_unsats:
                    mycode[i]  += self.um
                    mysynd     += H[i]

        if cnt_iter == 0:
            raise ValueError("BF: limite de iterações ultrapassado")

        return mycode


    def keygen(self):                                      #Gera Chave privada(h) e pública(f)
```

```
        while True:
            h0 = self.sparse_pol(); h1 = self.sparse_pol()
            if h0 != h1 and h0.is_unit() and h1.is_unit():      # Garantir que
↪os h são invertíveis
                break
        h = (h0,h1)

        g = self.geng()

        f0 = h1 + g*h0
        f1 = g
        f = (f0,f1)

        return h , f

    def encaps(self, f):                                        # Encapsular
        f0,f1 = f

        e = self.noise(self.t//2)
        e0 = self.noise(self.t//2)
        e1 = self.noise(self.t//2)

        c0 = e + e1 * f0
        c1 = e0 + e1 * f1
        c = (c0,c1)

        key = self.hash_key((e0,e1))

        return key , c

    def decaps(self,h,c):                                       # Desencapsular
        h0,h1 = h
        c0,c1 = c
        code = self.expand2(c)                                  # Converter para
↪vetor

        H = block_matrix(2,1,[self.Rot(h0),self.Rot(h1)])       # Matriz de
↪paridade
        synd = c0 + c1 * h0                                     # Calcula o
↪sindroma


        cw = self.BF2(H,code,self.expand(synd))                 # Descodifica
↪usando BitFlip em vetores

        (cw0,cw1) = self.unexpand2(cw)
```

```
        c0, c1 = c
        return self.hash_key((cw0 + c0 , cw1 + c1))
```

1.3 Teste KEM

[2]:
```
byke = BYKE()
#Geração das chaves
h,f =byke.keygen()
#Encapsular
key,c = byke.encaps(f)
#Desencapsular
k = byke.decaps(h,c)
#Verificação
k == key
```

[2]:
```
True
```

1.4 BYKE-PKE

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


    def BF_PKE(self, H,synd, errs=0):

        mycode = self.Vn([self.zero]*(self.n))
        mysynd = synd
        cnt_iter= self.r

        while cnt_iter > 0 and self.hamm(mysynd) > self.t//2:
            cnt_iter   = cnt_iter - 1

            unsats     = [self.hamm(self.mask(mysynd,H[i])) for i in range(self.
↪n)]
            max_unsats = max(unsats)

            for i in range(self.n):
                if unsats[i] == max_unsats:
                    mycode[i]  += self.um               ## bit-flip
                    mysynd     += H[i]

        if cnt_iter == 0:
            raise ValueError("BF: limite de iterações ultrapassado")


        return mycode
```

```
def keygen_pke(self):
    while True:
        h0 = self.sparse_pol(); h1 = self.sparse_pol()
        if h0 != h1 and h0.is_unit() and h1.is_unit():
            break
    h_=  (h0,h1)                          # chave privada

    h = (h1/h0)
    return h_ , h

def encode(self,s):
    r = []
    t0 = 0
    t1 = 0
    a = list(map(lift,list(s)))
    print(len(a))
    for i in range(self.r/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):
    lst = vec.list()
    res = bytearray()
    for poly in lst:
        print(len(bytearray(self.encode(poly))))
        res+=(bytearray(self.encode(poly)))
    return res

def enc(self, h, msg):
    m = self.encode(msg)

    f0,f1 = f

    e0 = self.noise(self.t//2)
    e1 = self.noise(self.t//2)

    c0 = e0 + e1 * h

    e1_bits  = np.packbits(list(map(lift,self.expand(e1))))
```

```
        c1 = bytes(a ^^ b for (a, b) in zip(m, bytes(self.hash_key((e0,e1)))))

        c = (c0,c1)

        return c

    def dec(self, h_, c):
        h0,h1 = h_
        c0,c1 = c
        #code = self.expand2(c)                    # converter para vetor

        H = block_matrix(2,1,[self.Rot(h0),self.Rot(h1)])
        synd = c0 * h0                             # calcula o sindroma

        cw = self.BF_PKE(H,self.expand(synd))                    # descodifica
↪usando BitFlip em vetores

        (cw0,cw1) = self.unexpand2(cw)


        x = self.decode_vec(c1)

        print(cw1)
        print(type(cw1))
        print(x)
        print(type(x))

        m = bytes(a ^^ b for (a, b) in zip(c1, (self.hash_key((cw0 + c0, cw1 +
↪x )))))

        return m
```

[4]:
```
byke = BYKE()
#Geração de uma mensagem
msg = byke.Rr.random_element()
#Geração das chaves
h_, h = byke.keygen()
#Cifrar da mensagem
c = byke.enc(h, msg)
#Decifrar
m = byke.decaps(h_,c)
#Verificação
m == msg
```

257

```
TypeError                                 Traceback (most recent call last)
<ipython-input-4-38fd74b23b1b> in <module>
      5 h_, h = byke.keygen()
      6 #Cifrar da mensagem
----> 7 c = byke.enc(h, msg)
      8 #Decifrar
      9 m = byke.decaps(h_,c)

<ipython-input-3-d8ebeebaf4a5> in enc(self, h, msg)
     61
     62     def enc(self, h, msg):
---> 63         m = self.encode(msg)
     64
     65         f0,f1 = f

<ipython-input-3-d8ebeebaf4a5> in encode(self, s)
     42         a = list(map(lift,list(s)))
     43         print(len(a))
---> 44         for i in range(self.r/Integer(2)):
     45             t0 = a[Integer(2)*i]
     46             t1 = a[Integer(2)*i+Integer(1)]

~/anaconda3/envs/sage/lib/python3.8/site-packages/sage/rings/rational.pyx in␣
↪sage.rings.rational.Rational.__index__ (build/cythonized/sage/rings/rational.::
↪6732)()
    556             return int(self)
    557
--> 558         raise TypeError(f"unable to convert rational {self} to an␣
↪integer")
    559
    560     cdef __set_value(self, x, unsigned int base):

TypeError: unable to convert rational 257/2 to an integer
```

[ ]: