bike

May 2, 2022

1 BIKE

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

[ ]:
```
K = GF(2)
um = K(1)
zero = K(0)

r = 257
n = 2*r
t = 16
```

[ ]:
```
Vn  = VectorSpace(K,n)
Vr  = VectorSpace(K,r)
Vq  = VectorSpace(QQ,r)

Mr  = MatrixSpace(K,n,r)


R = PolynomialRing(K,name='w')
w = R.gen()
Rr = QuotientRing(R,R.ideal(w^r+1))
```

[ ]:
```
def mask(u,v):
                                    ##
    return u.pairwise_product(v)

def hamm(u):
                                  ## peso de Hamming
    return sum([1 if a == um else 0 for a in u])
```

[ ]:
```
# Matrizes circulantes de tamanho r com r primo


def rot(h):
```

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

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

def expand(f):
    fl = f.list(); ex = r - len(fl)
    return Vr(fl + [zero]*ex)

def expand2(code):
    (f0,f1) = code
    f = expand(f0).list() + expand(f1).list()
    return Vn(f)

def unexpand2(vec):
    u = vec.list()
    return (Rr(u[:r]),Rr(u[r:]))

def mk_key(a):
    uu  = np.packbits(list(map(lift,expand2(a))))
    hsh =  hashes.Hash(hashes.SHAKE256(int(256)))
    hsh.update(uu)
    return hsh.finalize()
```

2 BGF

Na tentativa de ser fiel à documentação foi feita a tentativa de aplicar o decoder como sendo o Black-Gray-Flip. Esta encontra-se explicita no próximo bloco.

[ ]:
```
#Todas estas definições foram criadas
#  baseadas na documentação do algoritmo BIKE


#recebe como input um sindrome,
# ciphertext(code),uma mascara,
# um treshold e uma matriz para confirmar a paridade
def BitFlipMaskedIter(s,code,mask1,th, H):
    mysynd = s
    mycode=code
```

```
    #aplica-se a tranformaçã H[i]*s
    unsats = [hamm(mask(
        mysynd,H[i])) for i in range(n)]
    for i in range(n):
        if unsats[i] == th:
            #dá flip a um bit de erro
            mycode[i]  += mask1[i]

            #dá update ao sindrome
            mysynd     += H[i]

    return mysynd,mycode


#recebe como input um sindrome,
# ciphertext(code),uma mascara,
# um treshold e uma matriz para confirmar a paridade
def BitFlipIter(s,code,th,H):
    mysynd = s
    mycode=code

    #gera-se os arrays black e gray
    black     = [0 for i in range(n)]
    gray      = [0 for i in range(n)]

    #aplica-se a tranformaçã H[i]*s
    unsats     = [hamm(mask(mysynd,H[i])) for i in range(n)]
    for i in range(n):
        if unsats[i] == th:
            #dá flip a um bit de erro
            mycode[i]  += um
            #altera o vetor black para um no indice atual
            black[i]=um

            #atualiza o sindrome
            mysynd     += H[i]
        else:
            #caso se confirme a outra condição
            if unsats[i] == th-3:
                #altera o vetor gray  para um no indice atual
                gray[i]=um
                #atualiza o sindrome
                mysynd     += H[i]

    return (mysynd,mycode,black,gray)
```

3 BG

[ ]:
```
def BG(H,code,synd,cnt_iter=r, errs=0):
    mysynd=synd
    mycode=code
    p=4
    while cnt_iter > 0 and hamm(mysynd) > errs:
        cnt_iter   = cnt_iter - 1

        unsats     = [hamm(mask(mysynd,H[i])) for i in range(n)]
        max_unsats = max(unsats)
        (mysynd,mycode,black,gray)=BitFlipIter(mysynd,mycode,max_unsats,H)

        (mysynd,mycode)=BitFlipMaskedIter(mysynd,mycode,black,(t+1)/2,H)
        (mysynd,mycode)=BitFlipMaskedIter(mysynd,mycode,gray,(t+1)/2,H)


    return mycode
```

[ ]:
```
#kem
# Uma implementação do algoritmo Bit Flip sem quaisquer optimizações

def BF(H,code,synd,cnt_iter=r, errs=0):

    mycode = code
    mysynd = synd

    while cnt_iter > 0 and hamm(mysynd) > errs:
        cnt_iter   = cnt_iter - 1

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

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

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

    return mycode
```

[ ]:
```
#sparse polynomials of size r

# produz sempre um polinómio mónico com o último coeficiente igual a 1
```

```
# o parametro "sparse > 0"  é o numero de coeficientes não nulos sem contar com
 ↪o primeiro e o ultimo

def sparse_pol(sparse=3):
    coeffs = [1]*sparse + [0]*(r-2-sparse)
    rn.shuffle(coeffs)
    return Rr([1]+coeffs+[1])

## Noise
# produz um par de polinomios dispersos de tamanho "r" com um dado número total
 ↪de erros "t"

def noise(t):
    el = [um]*t + [zero]*(n-t)
    rn.shuffle(el)
    return (Rr(el[:r]),Rr(el[r:]))
```

[ ]:
```
## Bike

def bikeKG():
    while True:
        h0 = sparse_pol(); h1 = sparse_pol()
        if h0 != h1 and h0.is_unit() and h1.is_unit():
            break

    h = (h0,h1)                              # chave privada
    g = (1, h0/h1)                           # chave pública para um código
 ↪sistemático
    return (g,h)

def bikeEncrypt(g,mess=None):
    (g0,g1) = g
    (e0,e1) = noise(t)
    if mess != None:
        m = mess
        return (m * g0 + e0, m * g1 + e1)        # Modelo McEliece PKE
    else:
        m = Rr.random_element()
        key = mk_key((e0,e1))
        enc = (m * g0 + e0, m * g1 + e1)
        return (key,enc)                          # Modelo KEM

def bikeDecrypt(h,crypt,kem=False):
    code = expand2(crypt)                    # converter para vetor

    (h0,h1) = h                              # a partir da chave privada gera a
 ↪matriz de paridades
```

```
    H = block_matrix(2,1,[Rot(h0),Rot(h1)])
    synd = code * H                          # calcula o sindroma

    cw = BG(H,code,synd)                     # descodifica usando BitFlip em␣
 ↪vetores

    (cw0,cw1) = unexpand2(cw)                # passar a polinómios
                  # confirmação

    if not kem:
        return cw0                  # como é um código sistemático a primeira␣
 ↪componente da cw é a mensagem
    else:                           # modelo KEM
        enc0 , enc1 = crypt
        return mk_key((cw0 + enc0 , cw1 + enc1))
```

[ ]:
```
## gera o par de chaves

(g,h) = bikeKG()

## gera uma mensagem arbitrária
m = Rr.random_element()

# Cifra
cr = bikeEncrypt(g,m)

# Decifra
m1 = bikeDecrypt(h,cr)

# Verifica
m == m1
```

[ ]: False

[ ]:
```
k1,enc = bikeEncrypt(g)
k2 = bikeDecrypt(h,enc,kem=True)
k1 == k2
```

[ ]: False