bike

May 10, 2021

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)

def mask(u,v):
    return u.pairwise_product(v)

def hamm(u):                                        ## peso de Hamming
    return sum([int(a == um) for a in u])

# Matrizes circulantes de tamanho r com r primo

R.<w> = PolynomialRing(K)
Rr = QuotientRing(R,R.ideal(w^r+1))

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:]))

# 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:]))


class BIKE:
    def __init__(self):
        while True:
            h0 = sparse_pol()
            h1 = sparse_pol()
            if (h0 != h1 and h0.is_unit()) and (h1.is_unit()):
                break
        self.h = (h0,h1)
        self.g = (1, h0/h1)

    def encrypt(self, msg):
        (g0,g1) = self.g
        (e0,e1) = noise(t)
        # Modelo McEliece PKE
        m = Rr(msg)
        return (m * g0 + e0, m * g1 + e1)

    def decrypt(self,enc):
        code = expand2(enc)                     # converter para vetor
        (h0,h1) = self.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 = BF(H,code,synd)                    # descodifica usando BitFlip em␣
 ↪vetores
        (cw0,cw1) = unexpand2(cw)               # passar a polinómios
        return list(cw0)                        # como é um código sistemático␣
 ↪a primeira componente da cw é a mensagem

    def encapsulate(self):
        m = self.random_poly_binary()
        sha3 = hashes.Hash(hashes.SHAKE256(int(256)))
        sha3.update(bytes(m))
```

```
        shared_key = sha3.finalize()
        return (shared_key, self.encrypt(m))

    def decapsulate(self,enc):
        m = self.decrypt(enc)
        sha3 = hashes.Hash(hashes.SHAKE256(int(256)))
        sha3.update(bytes(m))
        return sha3.finalize()

    def random_poly_binary(self):
        return [ choice([0,1]) for i in range(r) ]

bike = BIKE()

# BIKE PKE Test
msg = bike.random_poly_binary()
enc = bike.encrypt(msg)
dec = bike.decrypt(enc)
print("BIKE PKE Test:",msg == dec)

# BIKE KEM Test
bob_shared_secret, enc = bike.encapsulate()
# print("bob",bob_shared_secret)
alice_shared_secret = bike.decapsulate(enc)
# print("alice",alice_shared_secret)

print("BIKE KEM Test:",bob_shared_secret == alice_shared_secret)
```