BIKE

May 10, 2021

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

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

r = 257
#r = 12323
n = 2*r
t = 16
#t = 134
```

[3]:
```
Vn  = VectorSpace(K,n)
Vr  = VectorSpace(K,r)
Vq  = VectorSpace(QQ,r)

Mr  = MatrixSpace(K,n,r)
```

[4]:
```
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])
```

[5]:
```
# Matrizes circulantes de tamanho r com r primo

R = PolynomialRing(K,name='w')
w = R.gen()
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
```

```python
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:]))
```

0.1 O algoritmo de descodificação Bit-Flip

[6]:
```python
# 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
```

0.2 O PKE BIKE

[7]:
```
#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:]))
```

[8]:
```
## 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,m):
    (g0,g1) = g
    (n0,n1) = noise(t)
    return (m * g0 + n0, m * g1 + n1)       # Modelo McEliece

def bikeDecrypt(h,cr):
    code = expand2(cr)                       # 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 = BF(H,code,synd)                     # descodifica usando BitFlip em␣
↪vetores

    (cw0,cw1) = unexpand2(cw)                # passar a polinómios
    assert cw1*h1 == cw0*h0                  # confirmação
    return cw0                               # como é um código sistemático a␣
↪primeira componente da cw é a mensagem
```

0.3 TESTE

[10]:
```
## 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
```

[10]: True

1 TP 2 - Implementação BIKE

Este notebook tem a implementação do nosso BIKE KEM e BIKE-PKE-CCA usando a TFO. Antes de avancçarmos com a especificação do código há uns aspectos a salientar. O primeiro é que, citando o ponto 4.2.1, não há problema em usar o decoder fornecido pelo professor para implementar o resto do KEM. O segundo aspecto é que não sabemos a taxa de falha do decoder é impossível nós podermos dizer que o KEM é IND-CPA ou IND-CCA, no entanto podemos dizer que o PKE é IND-CCA porque é garantido pela Transformação Fujisaki-Okamoto. Por fim também convém salientar que seria muito melhor que o decoder tivesse sido implementado de raiz. É aconselhado que para a leitura deste documento também se tenha a especificação do BIKE de 22-10-2020 à mão.

1.1 KEM

1.1.1 KeyGen

Em primeiro geramos aleatóriamente um par de polinómios do espaço Hw ($|h0| = |h1| = w/2$). Em segundo calculamos um polinómio h que é igual a h1/h0. Em terceiro geramos uma sequência de 32 bytes aleatória usando um método apropiado para servir de sigma. No fim fazemos output destes 4 valores.

[ ]:
```
# KeyGen
# Void -> (Hw (h0 h1), B sigma, poly h)
def keyGen():
    # Passo 1 (h0, h1) random Hw
    while True:
        h0 = sparse_pol(); h1 = sparse_pol()
        if h0 != h1 and h0.is_unit() and h1.is_unit():
            break

    # Passo 2 h = h1 / h0
    h = h1/h0
    # Passo 3 sigma random B
    sigma = os.urandom(32)
    return (h0.lift(),h1.lift(),sigma,h.lift())
```

1.1.2 Encaps

Em primeiro geramos aleatóriamente uma sequência de 32 bytes para servir de m. Em segundo fazemos H(m) de forma a calcular dois polinómios e0 e e1. Em terceiro obtemos um c = (c0 e c1) que consiste no encapsulamento do h e de m com L(e0,e1). Em quarto calculamos a chave partilhada. No fim fazemos output da chave partilhada e do seu encapsulamento.

[ ]:
```
# Encaps
# poly h -> (B k, (poly c0, B c1))
def encapsulate(h):
    # Passo 1 m random B
    m = os.urandom(32)
    # Passo 2 (e0, e1) = H(m)
    (e0, e1) = hashH(m)
    # Passo 3 c = (e0 + (e1 * h), xor(m, L(e0,e1)))
    c = (e0 + (e1 * h), xor(m, hashL(e0, e1, r)))
    # Passo 4 K = K(m, c)
    k = hashK(m, c)
    return (k,c)
```

1.1.3 KEM_Encaps

Esta função é a anterior sem o primeiro passo de forma a ser usada na TFO.

[ ]:
```
# Encaps sem o passo 1 para a FOT
def kem_encapsulate(h, m):
    (e0, e1) = hashH(m)
    c = (e0 + e1 * h, xor(m, hashL(e0, e1, r)))
    k = hashK(m, c)
    return (k,c)
```

1.1.4 Decaps

Em primeiro fazemos o decode da primeira metade do criptograma. Em segundo fazemos m' = xor da segunda metade do criptograma com L(e1, e2). Em terceiro fazemos H do resultado do segundo passo. Em quarto verificamos se o resultado do H coincide com o resultado do decode. Se coincidir devolvemos k = K(m', c) senão devolvemos k = K(sigma, c).

O código deixa de correr aqui quando tentamos fazer pairwise product no decoder. AttributeError: 'sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X' object has no attribute 'pairwise_product'

[ ]:
```
# Decaps
# Hw (h0 h1) -> B sigma -> (poly c0, B c1) -> B k
def decapsulate(h0, h1, sigma, c):
    (c0, c1) = c

    # Passo 1 e' = decoder(c0 * h0, h0, h1)
    el = BF(c0*h0, h0, h1)
    (el1,el2) = unexpand2(el)

    # Passo 2 m' = xor(c1, L(e'))
    ml = xor(c1, hashL(el1, el2))
    (ml1, ml2) = hashH(ml)

    # Passo 3
    if el1 == ml1 & el2 == ml2:
        return hashK(ml, c)
    else:
        return hashK(sigma, c)
```

1.2 H

Nesta secção temos as funções necessárias para a função H. A função H é a aplicação do algoritmo 4 de forma a obter dois polinómios gerados aleatóriamente no espaço de erros, ou seja (|e0| + |e1| = t). Por uma questão de tempo e seguindo o que é dito no ponto 4.2.3 substituimos o gerador aleatório de strings de bytes pelo os.urandom mas tirando isso o resto da implementação é leal à especificação.

[ ]:
```
# Hash K geração aleatória de um par (e0, e1) no espaço de erros (|e0| + |e1| =
 ↪t)
def hashH(seed):
```

```
    (wlist, s) = alg4(seed, t, r)
    # Converter s noutra lista de coeficientes
    slist = bits_to_coeffs(s)

    # Devolver os polinómios criados com as listas de coeficientes
    return (Rr(wlist).lift(), Rr(slist).lift())

def alg4(s, wt, leng):
    # Passo 1
    wlist = []
    i = 0

    # Passo 2
    # s = aes_ctr_stream(seed, suf_large)
    s = bytes_to_bits(os.urandom(4*wt))

    # Passo 3
    maskI = (2 ** (ceil(log(r, 2)))) - 1
    mask = int_to_bits(maskI)

    # Passo 4
    for ctr in range (wt):
        # Passo 5
        posb = and_bits(s[((32 *(i + 1)) - 1) : (32 * i)], mask)
        pos = bits_to_int(posb)

        # Passo 6
        if (pos < leng) & (belongs(pos, wlist)):
            # Passo 7
            wlist.append(pos)
        i += 1

    # Passo 8
    return (wlist, s)

def bits_to_coeffs(s):
    new = []
    for i in range(0, len(s), 32):
        new.append(s[i : i+32])

    r = []
    for i in range(len(new)):
        tmp = 0
        for j in range(len(new[i])):
            tmp += (2 ** j)*new[i][j]
        r.append(tmp)
```

```
    return r

def belongs(n, l):
    return l.count(n) > 0

def bytes_to_bits(B1):
    B = []
    for i in range(len(B1)):
        B.append(Integer(B1[i]))
    b = []
    size = len(B)
    for i in range (size):
        tmp = B[i]
        for j in range (8):
            b.append(tmp % 2)
            tmp = tmp // 2
    stop = 0
    i = len(b) - 1
    while stop == 0:
        if b[i] == 1:
            stop = 1
        else:
            del(b[i])
        i -= 1
    return b

def int_to_bits(x):
    tmp = x
    b = []
    while tmp > 0:
        b.append(tmp%2)
        tmp = tmp//2
    return b

def bits_to_int(b):
    r = 0
    for i in range(len(b)):
        r += (2** i) * b[i]
    return r

def and_bits(b1, b2):
    r = []
    size = len(b1)
    if len(b2) < size:
        size = len(b2)

    for i in range (size):
```

```
        r.append(b1[i] & b2[i])

    return r
```

1.3 K

Esta função usa o SHA384 aplicada a um array de bytes, um polinómio (convertido num array de bytes) e outro array de bytes e devolve os 32 bytes menos significativos.

```
[ ]: # Hash K, output l = 256 bits menos significativos de hashing do m ||␣
     ↪b_to_B(c0) || c1
     def hashK(m, c):
         (c0, c1) = c
         c0b = coefs_to_bytes(c0)
         message = m + c0b + c1
         digest = hashes.Hash(hashes.SHA384())
         digest.update(message)
         r = digest.finalize()
         return r[:32]
```

1.4 L

Esta função usa o SHA384 aplicada a dois polinómios (convertidos em arrays de bytes) e devolve os 32 bytes menos significativos.

```
[ ]: # Hash L, output l = 256 bits menos significativos de hashing do b_to_B(e0) ||␣
     ↪b_to_B(e1)
     def hashL(e0, e1, r):
         e0b = coefs_to_bytes(e0)
         e1b = coefs_to_bytes(e1)
         message = e0b + e1b
         digest = hashes.Hash(hashes.SHA384())
         digest.update(message)
         r = digest.finalize()
         return r[:32]

     def coefs_to_bytes(e):
         l = e.list()
         return bytearray(l)
```

Funções Auxiliares

```
[ ]: # xor de um byte a com um byte b (o sagemath faz interferência com o operador␣
     ↪'^')
     def mini_xor(a, b):
         tmpa = a
         tmpb = b
```

```
    r0 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r1 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r2 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r3 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r4 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r5 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r6 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)
    r7 = tmpa % 2 + tmpb % 2
    tmpa = int(tmpa//2)
    tmpb = int(tmpb//2)

    soma = 0
    if r0 == 1:
        soma += 1
    if r1 == 1:
        soma += 2
    if r2 == 1:
        soma += 4
    if r3 == 1:
        soma += 8
    if r4 == 1:
        soma += 16
    if r5 == 1:
        soma += 32
    if r6 == 1:
        soma += 64
    if r7 == 1:
        soma += 128

    return soma

# xor de dois arrays de bytes
def xor(a, b):
```

```
    size = len(b)
    if len(a) < len(b):
        size = len(a)

    xored = bytearray(size)
    for i in range(size):
        xored[i] = mini_xor(a[i], b[i])
    return xored
```

1.5 Teste do KEM

[ ]:
```
## gera o par de chaves
(h0, h1, sigma, h) = keyGen()

## Encapsulamento da Chave
(k, c) = encapsulate(h)

# Desencapsulamento da Chave
kl = decapsulate(h0, h1, sigma, c)

# Verificação
k == kl
```

1.6 Transformação FO

Aqui temos o código do TP1 ligeiramente alterado tendo em conta que o BIKE não foi implementado numa classe. É de salientar para não haver confusão que o output da encapsulate está "trocado" em vez de devolver (e,k) devolve (k,e). Além disto o output da encapsulate não é apenas (k, e), mas sim (k, (e1, e)).

[ ]:
```
class PKE:
    # construtor da classe
    def __init__(self, x, h0, h1, sigma, h):
        self.x = x
        self.h0 = h0
        self.h1 = h1
        self.sigma = sigma
        self.h = h

    # hash h
    def hashh(self, message):
        digest = hashes.Hash(hashes.SHA256())
        digest.update(message)
        return digest.finalize()

    # hash g
    def hashg(self, message):
```

```
        digest = hashes.Hash(hashes.BLAKE2s(32))
        digest.update(message)
        return digest.finalize()

    # E'
    def cifrar(self):
        # primeiro passo, r <- h
        self.r = self.hashh(self.x)
        # segundo passo, y <- x XOR g(r)
        self.y = xor(self.x, self.hashg(self.r))
        # terceiro passo, r' <- y || r
        self.rl = self. y + self.r
        # quarto passo, KEM(r')
        (self.k, (self.e0, self.e)) = kem_encapsulate(self.h, 257, self.rl)
        self.k = bits_to_bytes(self.k)
        # finalmente c = k XOR r
        self.c = xor(self.k, self.r)

    # D'
    def decifrar(self):
        # k <- KREv(e)
        self.k = decapsulate(self.h0, self.h1, self.sigma, (self.e0, self.e),␣
↪257)
        # r <- c XOR k
        self.r = xor(self.c, self.k)
        # r' = y || r
        self.rl = self.y + self.r
        # (e, k) = f(rl)
        (self.k2, (self.e02, self.e2)) = kem_encapsulate(self.h, 257, self.rl)
        self.k2 = bits_to_bytes(self.k2)
        # verificação f(rl) == (e, k)
        if (self.k2 == self.k) & (self.e2 == self.e):
            # x == y XOR g(r)
            self.x = xor(self.y, self.hashg(self.r))
            print("True")
        else:
            print("False")
```

1.7 Teste do PKE IND-CCA

[ ]:
```
# Teste BIKE_FOT_PKE
(h0, h1, sigma, h) = keyGen()
# inicializamos a classe PKE
b = PKE(b'teste', h0, h1, sigma, h)
# fazemos E'(x)
b.cifrar()
# fazemos D' (yec)
```

```
b.decifrar()
```