BIKE

Imports

In [42]:
```
import random as rn
import numpy as np
from hashlib import sha256
```

BIKE

Definição de uma *classe* que implemente encapsulamento de chaves e cifragem/decifragem de mensagens, com recurso à operação de *bit flip*.

key_gen

Método que gera as chaves públicas e privadas, a partir de parâmetros de segurança previamente estabelecidos na própria classe BIKE. Recorrendo ao método *sparse_cols*, geram-se polinómios que serão posteriormente utilizados na geração das chaves públicas e privadas, como definido na documentação do BIKE. Assim, a chave privada deve ser (h0, h1) e a pública (gh1, gh0).

PKE

enc

Este será o método de cifragem, que se servirá da chave pública para computar um criptograma, a partir da cifragem de uma mensagem recebida como argumento. Utilizará também na cifragem polinómios gerados pelo método *noise*. Assim, o criptograma resultante será dado por (m*f0 + e0, m*f1 + e1).

dec

É o método de decifragem, que decifrará um criptograma através do algoritmo de *bit flip*, com recurso à chave privada anteriormente gerada. Utiliza-se a chave privada para computar um síndroma, que será posteriormente decifrado no método de *bit flip*, resultando na mensagem original

KEM

encapsulate

Método em que se gera uma chave encapsulada e um criptograma, a partir dos quais posteriormente será possível obter a chave desencapsulada. Semelhante ao método de cifragem, apenas tem o acrescento de gerar uma chave a partir dos polinómios e0 e e1, sobre os quais se aplica o algoritmo *sha256*.

decapsulate

Método em que se obtém a chave desencapsulada a partir da chave privada e do criptograma gerado no método *encapsulate*. O método é bastante similar ao método de decifragem, com a diferença que no final a chave é obtida por aplicar o algoritmo *sha256* sobre o resultado de (cw0+c0,cw1+c1), em que cw0 e cw1 são o resultado do decode do criptograma e c0 e c1 são as duas partes do criptograma original.

Class BIKE

In [48]:
```
class BIKE:
    def __init__(self, r, t):
        self.K = GF(2)
        self.um = self.K(1)
        self.zero = self.K(0)

        self.r = r
        self.n = 2*r
        self.t = t

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

        self.Mr  = MatrixSpace(self.K,self.n,self.r)

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



    def key_gen(self):
        h0 = self.sparse_pol()
        h1 = self.sparse_pol()
        while(not (h0 != h1 and h0.is_unit() and h1.is_unit())):
            h0 = self.sparse_pol()
            h1 = self.sparse_pol()

        g = self.sparse_pol()
        while(not g.is_unit() and h0 != g and h1 != g):
            g = self.sparse_pol()

        private = (h0, h1)
        #public = (g*h1, g*h0)
        public = (1, h0/h1)

        return(private, public)


    def enc(self, public, m):
        (f0, f1) = public
        (e, e0, e1) = self.noise(self.t)
        return (m * f0 + e0, m * f1 + e1)
        #return (e + e1*f0, e0 + e1*f1)


    def dec(self, private, cripto):
        vec = self.expand2(cripto)
        (c0, c1) = cripto
        (h0, h1) = private
        H = block_matrix(2,1,[self.Rot(h0),self.Rot(h1)])
        #aux = block_matrix(2,1,[self.Rot(1),self.Rot(h0)])
        synd = vec * H
        #synd = vec * aux

        cw = self.BF(H,vec,synd,0)

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

        return cw0


    def encapsulate(self, public):
        (e,e0,e1) = self.noise(self.t)
        (f0, f1) = public
        ms = self.Rr.random_element()

        uu  = np.packbits(list(map(lift,self.expand2((e0,e1)))))
        m = sha256()
        m.update(uu)
        key = m.digest()

        return (key, (ms*f0 + e0, ms*f1 + e1))


    def decapsulate(self, private, cripto):
        vec = self.expand2(cripto)
        (c0, c1) = cripto
        (h0, h1) = private
        H = block_matrix(2,1,[self.Rot(h0),self.Rot(h1)])
        #aux = block_matrix(2,1,[self.Rot(1),self.Rot(h0)])
        synd = vec * H
        #synd = vec * aux

        cw = self.BF(H,vec,synd,0)

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


        uu  = np.packbits(list(map(lift,self.expand2((cw0+c0,cw1+c1)))))
        m = sha256()
        m.update(uu)
        key = m.digest()

        return key







    ################################################ AUXILIARES ##########################
    ######################################


    def BF(self, H,code,synd, errs=0):
        cnt_iter=self.r
        mycode = code
        mysynd = synd

        while cnt_iter > 0 and self.hamm(mysynd) > errs:
            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 sparse_pol(self, sparse=3):
        coeffs = [1]*sparse + [0]*(self.r-2-sparse)
        rn.shuffle(coeffs)
        return self.Rr([1]+coeffs+[1])


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


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


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


    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):
        f1 = f.list(); ex = self.r - len(f1)
        return self.Vr(f1 + [self.zero]*ex)


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


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

Testing PKE

In [49]:
```
b = BIKE(257, 16)

(private, public) = b.key_gen()
m = b.Rr.random_element()
cr = b.enc(public, m)
d = b.dec(private, cr)
print(cr == m)
print(d == m)
```

```
False
True
```

Testing KEM

In [50]:
```
b = BIKE(257, 16)

(private, public) = b.key_gen()
k1, c = b.encapsulate(public)
k2 = b.decapsulate(private, c)
print(k1 == k2)
```

```
True
```

Bibliografia:

https://bikesuite.org/files/BIKE.pdf

In [ ]:
```

```