# BIKE

April 30, 2023

## 0.1 Estruturas Criptográficas: Trabalho Prático 3

### 0.1.1 Criação de protótipos em Sagemath para os algoritmos KYBER e BIKE.

**Grupo 13, constituído por:** – Rodrigo Rodrigues, PG50726
– Rui Guilherme Monteiro, PG50739

## 0.2 BIKE

**Serão apresentadas duas implementações (com recurso ao SageMath) deste algoritmo:**
BIKE-PKE (IND-CCA seguro)
BIKE-KEM (IND-CPA seguro)

**Referências:** https://bikesuite.org/files/v4.2/BIKE_Spec.2021.09.29.1.pdf

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

### 0.2.1 BIKE-KEM

[1]:
```
from sage.all import *
from random import shuffle
import hashlib

class BIKE_KEM(object):

    def __init__(self, N, R, W, T, timeout=None):
        """
        Parâmetros de segurança
        -----------------------
        O parâmetro N define o tamanho da chave pública e privada em bits,␣
↪normalmente N = 2 * R, onde R é um número primo.
        O parâmetro R define o tamanho do anel polinomial usado no sistema.
        O parâmetro W define o peso de Hamming (número de bits iguais a 1)␣
↪usado para gerar as chaves.
        O parâmetro T é um inteiro usado na descodificação dos bits␣
↪criptografados.
        -----------------------
        K2: corpo finito de tamanho 2
        F: anel
        R: anel quociente
```

```
        """
        self.r = R    # r (número primo)
        self.n = N    # normalmente, n = 2*r
        self.w = W
        self.t = T        # inteiro usado na descodificação
        self.K2 = GF(2)    # Corpo finito de tamanho 2
        F.<x> = PolynomialRing(self.K2)
        R.<x> = QuotientRing(F, F.ideal(x^self.r + 1))
        self.R = R


    def Hash(self, e0, e1):
        """
            Recebe duas strings e0 e e1, concatena-as e aplica a função de hash␣
↪SHA3-256.

            Parâmetros
            ----------
            e0, e1: strings

            Output: valor `hashed` (32-byte digest).
        """

        m = hashlib.sha3_256()
        m.update(e0.encode())
        m.update(e1.encode())
        return m.digest()

    def rotateVector(self, h):
        """
            Roda os elementos de um vetor.
        """

        V = VectorSpace(self.K2, self.r)
        v = V()
        v[0] = h[-1]
        for i in range(self.r-1):
            v[i+1] = h[i]

        return v

    def rotationMatrix(self, v):
        """
            Gera a matriz de rotação a partir de um vetor
        """
```

```
        M = Matrix(self.K2, self.r, self.r)
        M[0] = self.polynomial_to_vector_r(v)

        for i in range(1, self.r):
            M[i] = self.rotateVector(M[i-1])
        return M

    def bitFlip(self, H, y, s, N):
        """
        Parâmetros
        ----------
        H: A matriz H = H0 + H1

        y: a palavra de código

        s: o sindrome s

        N: nº max iterações para descobrir os erros
        """

        x = y  # nova palavra de código
        z = s  # novo sindrome

        while N > 0 and self.hammingWeight(z) > 0:

            # Gerar um vetor com todos os pesos de hamming de |z . Hi|
            pesos = [self.hammingWeight(self.componentwise(z, H[i])) for i in
↪range(self.n)]
            maximo = max(pesos)

            for i in range(self.n):
                if pesos[i] == maximo:
                    x[i] += self.K2(1)  # flip the bit
                    z += H[i]           # update sindrome

            N = N - 1

        # Controlo das iterações
        if N == 0:
            raise ValueError("Limite de iterações ultrapassado.")

        return x

    def h(self):
        """
            Efetua o cálculo: |e0| + |e1| = t
```

```
            Gerar dois erros, e0 e e1, pertencentes a R, tal que a soma dos␣
↪pesos de hamming destes erros seja igual a t)
        """

        # (e0,e1) € R, tal que |e0| + |e1| = t.
        e = self.generateCoefP(self.t)
        # Gerar um m <- R, denso
        m = self.R.random_element()

        return (m,e)

    def f(self, public_key, m, e):
        """
            Calcular o par (k, c):
                K é o hash dos erros e0, e1
                c é obtido da multiplicação de m pela chave publica adicionando␣
↪os erros
        """

        # c = (c0, c1) <- (m.f0 + e0, m.f1 + e1)
        c0 = m * public_key[0] + e[0]
        c1 = m * public_key[1] + e[1]
        c = (c0,c1)

        # K <- Hash(e0, e1)
        k = self.Hash(str(e[0]), str(e[1]))

        return (k, c)



    def errorVector(self, secret_key, c):
        """
            Descobrir o vetor de erro para aplicar F.O. no PKE-IND-CCA, usando␣
↪bitFlip
        """

        # Converter criptograma (ou um tuplo de polinómios de tamanho n) para␣
↪um vetor
        V = VectorSpace(self.K2, self.n)
        f = self.polynomial_to_vector_r(c[0]).list() + self.
↪polynomial_to_vector_r(c[1]).list()
        code = V(f)

        # Formar a matriz H = (rot(h0)|rot(h1))
        H = block_matrix(2, 1, [self.rotationMatrix(secret_key[0]), self.
↪rotationMatrix(secret_key[1])])
```

```
        # s <- c0.h0 + c1.h1
        s = code * H

        # tentar descobrir s para recuperar (e0, e1)
        bf = self.bitFlip(H, code, s, self.r)

        # converter num par de polinómios
        (bf0, bf1) = self.coefToPol(bf)

        # visto ser um código sistemático, m = bf0
        e0 = c[0] - bf0 * 1
        e1 = c[1] - bf0 * secret_key[0]/secret_key[1]

        return (e0,e1)


    # Função recebe o vetor de erro e retorna o cálculo da chave (para permitir␣
↪aplicar F.O. no PKE-IND-CCA)
    def calculateKey(self, e0, e1):
        if self.hammingWeight(self.polynomial_to_vector_r(e0)) + self.
↪hammingWeight(self.polynomial_to_vector_r(e1)) != self.t:
            raise ValueError("Erro no decoding!")

        k = self.Hash(str(e0), str(e1))

        return k


    def generateKeyPair(self):
        """
            Gerar o par de chaves
        """

        h0 = self.generateCoef(self.w//2, self.r)
        h1 = self.generateCoef(self.w//2, self.r)

        # g <- R, com peso ímpar |g| = r/2.
        g = self.generateCoef(self.r//2, self.r)

        # (f0, f1) <- (gh1, gh0).
        f0 = g*h1
        f1 = g*h0

        return {'secret_key' : (h0,h1) , 'public_key' : (f0, f1)}

    def encaps(self, public_key):
```

```
        """
            Retorna a chave encapsulada k e o criptograma ("encapsulamento") c.
        """

        # Gerar um m <- R, denso
        (m,e) = self.h()

        return self.f(public_key, m, e)

    def decaps(self, secret_key, c):
        """
            Retorna a chave desencapsulada k ou erro
        """

        # Descodificar o vetor de erro
        (e0, e1) = self.errorVector(secret_key, c)

        # Calcular a chave
        k = self.calculateKey(e0, e1)

        return k


    ############################################ Métodos AUXILIARES␣
↪##############################################

    def hammingWeight(self, x):
        """
            Calcula o peso de hamming de um vetor
        """
        return sum([1 if a == self.K2(1) else 0 for a in x])

    def generateCoef(self, w, n):
        """
            Gera os coeficientes binários de um polinómio de tamanho n com w␣
↪1's.
        """

        res = [1]*w + [0]*(n-w-2)
        shuffle(res)
        return self.R([1]+res+[1])

    def generateCoefP(self, w):
        """
            Gera um par de polinómios de tamanho r, com total de w erros (1's)
        """
```

```
        res = [1]*w + [0]*(self.n-w)
        shuffle(res)
        return (self.R(res[:self.r]), self.R(res[self.r:]))

    def coefToPol(self, e):
        """
         Converte uma lista de coeficientes num tuplo de polinómios.
        """

        u = e.list()
        return (self.R(u[:self.r]), self.R(u[self.r:]))

    def polynomial_to_vector_r(self, p):
        """
            Converte um polinómio de tamanho r num vetor de tamanho r
        """

        V = VectorSpace(self.K2, self.r)
        return V(p.list() + [0]*(self.r - len(p.list())))

    def componentwise(self, v1, v2):
        """
            Calcula o produto componentwise de dois vetores
        """

        return v1.pairwise_product(v2)
```

[2]:
```
def main():
    # Parâmetros
    R = next_prime(1000)
    N = 2*R
    W = 6
    T = 32

    bike_kem = BIKE_KEM(N,R,W,T)

    # Gerar as chaves
    keys = bike_kem.generateKeyPair()

    # Gerar uma chave e o seu encapsulamento
    (k,c) = bike_kem.encaps(keys['public_key'])

    # Desencapsular
    k1 = bike_kem.decaps(keys['secret_key'], c)

    if k == k1:
```

```
        print("As chaves K e K1 são iguais!")


if __name__ == "__main__":
    main()
```

```
As chaves K e K1 são iguais!
```

### 0.2.2 BIKE-PKE

[63]:
```
class BIKE_PKE(object):
    """
        Utiliza BIKE_KEM como referência, aplicando uma transformação de␣
 ↪Fujisaki-Okamoto para obter um PKE que seja IND-CCA seguro.

    """

    def __init__(self, N, R, W, T, timeout=None):
        self.kem = BIKE_KEM(N,R,W,T)
        self.keys = self.kem.generateKeyPair()

    def xor(self, data, mask):
        """
            XOR de dois vetores de bytes (byte-a-byte).

            Parâmetros:
            ----------
            data: mensagem - deve ser menor ou igual à chave (mask). Caso␣
 ↪contrário, a chave é repetida para os bytes seguintes
        """

        masked = b''
        ldata = len(data)
        lmask = len(mask)
        i = 0
        while i < ldata:
            for j in range(lmask):
                if i < ldata:
                    masked += (data[i] ^^ mask[j]).to_bytes(1, byteorder='big')
                    i += 1
                else:
                    break

        return masked


    def encrypt(self, m, pk):
```

```
        """
            Cifra uma mensagem.
        """

        # Gerar um polinómio aleatório (denso): r <- R; e um par (e0,e1)
        (r,e) = self.kem.h()
        # Calcular g(r), em que g é uma função de hash (sha3-256)
        g = hashlib.sha3_256(str(r).encode()).digest()
        # Calcular y <- x (+) g(r)
        y = self.xor(m.encode(), g)
        # Transformar a string de bytes numa string binária
        im = bin(int.from_bytes(y, byteorder=sys.byteorder))
        yi = self.kem.R(im)
        # Calcular (k,w) <- f(y || r)
        (k,w) = self.kem.f(pk, yi + r, e)
        # Calcular c ← k  r
        c = self.xor(str(r).encode(), k)

        return (y,w,c)

    def decrypt(self, sk, y, w, c):
        """
            Decifra um criptograma.
        """

        e = self.kem.errorVector(sk, w)
        k = self.kem.calculateKey(e[0], e[1])
        # Calcula r <- c (+) k
        rs = self.xor(c, k)
        r = self.kem.R(rs.decode())

        # Transformar a string de bytes numa string binária
        im = bin(int.from_bytes(y, byteorder=sys.byteorder))
        yi = self.kem.R(im)

        # Verificar se (w,k) != f(y r)
        if (k,w) != self.kem.f(self.keys['public_key'], yi + r, e):
            # Erro
            raise IOError
        else:
            # Calcular g(r), em que g é uma função de hash (sha3-256)
            g = hashlib.sha3_256(rs).digest()
            # Calcular m <- y (+) g(r)
            m = self.xor(y, g)

        return m
```

[64]:
```python
def main():
    # Parâmetros para este cenário de teste
    R = next_prime(1000)
    N = 2*R
    W = 6
    T = 32

    bikePKE = BIKE_PKE(N,R,W,T)

    message = "Mensagem secreta do grupo 13."

    (y,w,c) = bikePKE.encrypt(message, bikePKE.keys['public_key'])

    message_decoded = bikePKE.decrypt(bikePKE.keys['secret_key'], y, w, c)

    if message == message_decoded.decode():
        print("Sucesso. Mensagem decifrada: " + message_decoded.decode())
    else:
        print("A decifragem falhou.")


if __name__ == "__main__":
    main()
```

```
Sucesso. Mensagem decifrada: Mensagem secreta do grupo 13.
```