BIKE

May 2, 2023

1 TP1

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 1.

1. Este problema é dedicado às candidaturas finalistas ao concurso NIST Post-Quantum Cryptography na categoria de criptosistemas PKE-KEM. Em Julho de 2022 foi selecionada para "standartização" a candidatura KYBER. Existe ainda uma fase não concluída do concurso onde poderá ser acrescentada alguma outra candidatura; destas destaco o algoritmo BIKE. Ao contrário do Kyber que é baseado no problema "Ring Learning With Errors" (RLWE) , o algoritmo BIKE baseia-se no problema da descodificação de códigos lineares de baixa densidade que são simples de implementar. A descrição, outra documentação e implementações em C/C++ destas candidaturas pode ser obtida na página do concurso NIST ou na diretoria Docs/PQC.
    1. O objetivo deste trabalho é a criação de protótipos em Sagemath para os algoritmos KYBER e BIKE.
    2. Para cada uma destas técnicas pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.

2 BIKE-KEM

Para tornar este KEM IND-CPA, seguimos as diretrizes de segurança do NIST e implementamos o algoritmo BIKE. Este algoritmo utiliza uma variação do esquema criptográfico de McEliece para gerar chaves de forma rápida.

Para gerar as chaves, foram fornecidos quatro parâmetros de segurança: N, R, W e T. Além disso, foi necessário gerar um corpo finito de tamanho 2 (K2) e o anel R, que é o quociente de polinômios F[X] / <X^r + 1 >.

A seguir, detalharemos os métodos implementados para este KEM:

2.0.1 Função KeyGen(): Geração da chave pública (f0, f1) e da chave privada (h0, h1)

Gerar os parâmetros h0 e h1. Ambos devem pertencer a R, com peso de Hamming igual a w/2 (o número de coeficientes do polinómio iguais a 1 deve ser w/2);

Gerar um novo polinómio (`g`). Este polinómio deve pertencer a R, com peso de Hamming igual a r/2;

Calcular a chave pública: (f0, f1) <- (gh1, gh0); e retornar tanto a chave privada (h0, h1) quanto a chave pública (f0, f1).

2.0.2 Função Encaps(): Encapsulamento e geração da chave

Calcular o par (k, c): `k` é a chave calculada, `c` é o encapsulamento da chave; recebendo a chave pública (f0, f1) como parâmetro. (NOTA: esta separação dos parâmetros foi implementada desta forma para facilitar a transformação de Fujisaki-Okamoto para a conversão para o PKE);

Definição da função h(), onde é efetuado 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); além disso, gera também um `m` pertencente a R, de forma aleatória e que deve ser denso;

Definição da função f() para efetivamente calcular o par (k, c), através dos parâmetros anteriormente referidos: a chave pública (f0, f1), o `m` e os erros (e0 e e1); c = (c0, c1) <- (m*f0 + e0, m*f1 + e1); k <- Hash(e0 + e1).

2.0.3 Função Decaps(): Desencapsulamento da chave

Calcular a chave `k`, através dos parâmetros: chave privada (h0, h1) e o encapsulamento da chave (`c`). Assim, tal como na função de encapsulamento, foram definidas duas funções auxiliares para ajudar neste processo:

Definição da função find_error_vec(), onde decodifica os vetores de erro `e0` e `e1`:

Começar por converter o encapsulamento da chave num vetor em n, sendo este o código usado aquando do bitFlip();

Depois, formamos a matriz `H` = (`rotation(h0)`|`rotation(h1)`);

Cálculo do síndrome: `s` <- `c0` * `h0` + `c1` * `h1` (multiplicação do código com a matriz `H`);

Depois, tenta-se decodificar `s` usando o algoritmo bitFlip() para recuperar o vetor (`e0`, `e1`);

Uma vez obtido o resultado do bitFlip(), converte-se esse resultado numa forma de par de polinómios (`bf0`, `bf1`);

Finalmente, tratando-se de um código sistemático, o `m` = `bf0` e o (`e0`,`e1`) é calculado como: e0 = c0 - bf0 * 1; e1 = c1 - bf0 * sk0/sk1.

Definição da função calculate_key() para efetivamente calcular a chave resultante.

[1]:
```python
# imports
import random, hashlib, sys
```

[2]:
```
class BIKE_KEM(object):

    def __init__ (self, N, R, W, T, timeout=None):
        # r (número primo)
        self.r = R

        # n = 2 * r
        self.n = N
        self.w = W

        # t (número inteiro utilizado na decifragem)
        self.t = T

        # Corpo finito de tamanho 2
        self.K2 = GF(2)

        # Anel Polinomial em x sobre Campo Finito de tamanho 2
        F.<x> = PolynomialRing(self.K2)

        # O anel polinomial cíclico F[X]/<X^r + 1>
        R.<x> = QuotientRing(F, F.ideal(x^self.r + 1))

        self.R = R

    # Calcular o peso de Hamming de um vetor (é um número de não zeros em␣
↪representação binária)
    def hammingWeight(self, x):
        return sum([1 if a == self.K2(1) else 0 for a in x])

    # Gerar aleatoriamente os coeficientes binários de um polinámio com w 1's e␣
↪de tamanho n
    def geraCoef(self, w, n):
        res = [1]*w + [0]*(n-w-2)
        random.shuffle(res)
        return self.R([1]+res+[1])

    # Gerar um par de polinómios de tamanho "r" com um número total de erros␣
↪(1's) "w"
    def geraCoefP(self, w):
        res = [1]*w + [0]*(self.n-w)
        random.shuffle(res)
        return (self.R(res[:self.r]), self.R(res[self.r:]))

    # Converte uma lista de coeficientes em dois polinómios
    def convertPolinomio(self, e):
        u = e.list()
        return (self.R(u[:self.r]), self.R(u[self.r:]))
```

```python
    #função para calcular o hash
    def Hash(self, e0, e1):
        m = hashlib.sha3_256()
        m.update(e0.encode())
        m.update(e1.encode())
        return m.digest()

    # Produto de vetores
    def componentwise(self, v1, v2):
        return v1.pairwise_product(v2)


    # Converter um polinómio de tamanho r para um vetor
    def vectorConverter_r(self, p):
        V = VectorSpace(self.K2, self.r)
        return V(p.list() + [0]*(self.r - len(p.list())))


    # Converter um tuplo de polinómios de tamanho n para um vetor
    def vectorConverter_n(self, pp):
        V = VectorSpace(self.K2, self.n)
        f = self.vectorConverter_r(pp[0]).list() + self.vectorConverter_r(pp[1]).list()
        return V(f)


    # Rodar os elementos de um vetor
    def vec_rotation(self, h):
        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


    # Função que gera a matriz de rotação a partir de um vetor
    def rotation(self, v):
        # Cria uma matriz binária de tamanho (r x r)
        M = Matrix(self.K2, self.r, self.r)
        # transforma v para vetor
        M[0] = self.vectorConverter_r(v)
        # Aplicar sucessivamente as rotações a todas as linhas da matriz
        for i in range(1, self.r):
            M[i] = self.vec_rotation(M[i-1])
```

```
        return M


    # Recebe como parâmetros:
    # a matriz H = H0 + H1
    # a palavra de código y
    # o sindrome s
    # n_iter: número de iterações máximas para descobrir os erros (questão de␣
↪eficiência)
    def bitFlip(self, H, y, s, n_iter):
        # Nova palavra de código
        x = y
        # Novo sindrome
        z = s

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

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

            for i in range(self.n):
                # Verificar se |hj . z|
                if pesosHam[i] == maxP:
                    # Efetua o flip do bit
                    x[i] += self.K2(1)
                    # atualiza o sindrome
                    z += H[i]
            # Decresce o número de iterações
            n_iter = n_iter - 1

        # Controlo das iterações
        if n_iter == 0:
            raise ValueError("Número máximo de iterações atingido!")

        return x


    # Função h() previamente descrita
    def h(self):
        # (e0,e1) € R, tal que |e0| + |e1| = t.
        e = self.geraCoefP(self.t)
        # Gerar um m <- R, denso
        m = self.R.random_element()

        return (m,e)
```

```
    # Função f() previamente descrita, de forma a permitir aplicar␣
↪Fujisaki-Okamoto no PKE-IND-CCA
    def f(self, pk, m, e):
        # c = (c0, c1) <- (m.f0 + e0, m.f1 + e1)
        c0 = m * pk[0] + e[0]
        c1 = m * pk[1] + e[1]
        c = (c0,c1)

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

        return (k, c)


    # Função para descobrir o vetor de erro (para permitir aplicar␣
↪Fujisaki-Okamoto no PKE-IND-CCA), com auxílio do bitFlip
    def find_error_vec(self, sk, c):
        # Converter o criptograma num vetor em n
        code = self.vectorConverter_n(c)
        # Formar a matriz H = (rotation(h0)|rotation(h1))
        H = block_matrix(2, 1, [self.rotation(sk[0]), self.rotation(sk[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.convertPolinomio(bf)
        # visto ser um código sistemático, m = bf0
        e0 = c[0] - bf0 * 1
        e1 = c[1] - bf0 * sk[0]/sk[1]

        return (e0,e1)


    # Função recebe o vetor de erro e retorna o cálculo da chave (para permitir␣
↪aplicar Fujisaki-Okamoto no PKE-IND-CCA)
    def calculate_key(self, e0, e1):
        # se |(e0,e1)| != t ou falhar
        if self.hammingWeight(self.vectorConverter_r(e0)) + self.
↪hammingWeight(self.vectorConverter_r(e1)) != self.t:
            # erro
            raise ValueError("Erro no decoding!")
        # K <- Hash(e0, e1)
        k = self.Hash(str(e0), str(e1))
```

```python
        return k


    # Função responsável por gerar o par de chaves
    def KeyGen(self):
        # h0,h1 <- R, ambos de peso ímpar |h0| = |h1| = w/2.
        h0 = self.geraCoef(self.w//2, self.r)
        h1 = self.geraCoef(self.w//2, self.r)

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

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

        return {'secretkey' : (h0,h1) , 'publickey' : (f0, f1)}


    # Retorna a chave encapsulada k e o criptograma ("encapsulamento") c.
    def Encaps(self, pk):
        # Gerar um m <- R, denso
        (m,e) = self.h()

        return self.f(pk, m, e)


    # Retorna a chave desencapsulada k ou erro
    def Decaps(self, sk, c):
        # Descodificar o vetor de erro
        (e0, e1) = self.find_error_vec(sk, c)

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

        return k
```

2.0.4 Exemplo de teste:

[3]:
```python
# Parâmetros para este cenário de teste
R = next_prime(1000)
N = 2*R
W = 6
T = 32

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

```python
# Gerar as chaves
keys = bike_kem.KeyGen()

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

# Desencapsular
k1 = bike_kem.Decaps(keys['secretkey'], c)

if k == k1:
    print("Chaves iguais!")
```

```
Chaves iguais!
```

3 BIKE-PKE

O algoritmo anterior apresentava uma vulnerabilidade de segurança CCA devido ao algoritmo de bitFlip, o que poderia levar a alguns erros. Para superar esse problema, a transformação de Fujisaki-Okamoto foi utilizada, exigindo a separação de alguns métodos anteriores para facilitar o processo.

A seguir, serão descritos os processos de certos métodos implementados:

3.0.1 Geração da chave pública (f0, f1) e da chave privada (h0, h1):

Para gerar ambas as chaves, basta-nos instanciar a classe anteriormente definida, BIKE-KEM. Assim, na inicialização desta nova classe, BIKE-PKE, basta-nos inicializar a outra classe, permitindo obter e gerar as chaves da mesma forma já definida.

3.0.2 Função cifragem(): Cifragem

A função de cifragem recebe como input a mensagem a cifrar e a chave pública. De seguida, é necessário o seguinte processo:

Gerar um polinómio aleatório (r <- R) denso, e um par (e0, e1);

Calcular g(r), onde g() é uma função de hash (sha3-256 neste caso);

Efetuar a operação de adição de polinómios entre a mensagem original e o hash de r (g(r)), que deve ser do mesmo tamanho do que a mensagem original: y <- m + g(r);

Converter a string de bytes em uma string binária, que, em seguida, será convertida em um polinômio em R;

Utilizar a função f() definida no BIKE-KEM para cifrar o polinômio convertido;

Finalmente, ofuscar a chave através da operação de XOR entre o `r` e o `k`: c <- r + k.

Retornar o triplo (y, w, c).

3.0.3 Função Decryption(): Decifragem

A função de decifragem recebe como input a ofuscação da mensagem original (y), o encapsulamento da chave (w) e a ofuscação da chave (c) e realiza as seguintes operações:

Desencapsular a chave através da chave privada (h0, h1) e do encapsulamento da chave (`w`), utilizando a função `Desencaps()` que calcula a chave `k`.

Calcular `r` como a operação de `c` XOR `k`: `r <- c (+) k`.

Converter a string de bytes `y` numa string binária e em seguida num polinómio em R.

Verificar se o encapsulamento da chave é igual a `(w,k)`, utilizando a condição `r == g(y)`, onde `g()` é uma função de hash (sha3-256 neste caso). Se a condição for verdadeira, calcular a mensagem original: `m <- y (+) g(r)`.

[4]:
```python
# Utiliza BIKE_KEM como referência, aplicando uma transformação de␣
↪Fujisaki-Okamoto
class BIKE_PKE(object):

    def __init__(self, N, R, W, T, timeout=None):

        # Inicialização da classe KEM do BIKE
        self.kem = BIKE_KEM(N,R,W,T)
        # Gerar as chaves
        self.chaves = self.kem.KeyGen()


    # XOR de dois vetores de bytes (byte-a-byte).
    # data: mensagem - deve ser menor ou igual à chave (mask).
    # Caso contrário, a chave é repetida para os bytes seguintes
    def xor(self, data, mask):

        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


    # Função usada para cifrar uma mensagem
    def cifragem(self, m, pk):
```

```python
    # 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)


# Função usada para decifrar um criptograma
def decifragem(self, sk, y, w, c):

    # Fazer o desencapsulamento da chave
    # k = self.kem.Desencaps(sk, w)
    e = self.kem.find_error_vec(sk, w)
    k = self.kem.calculate_key(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.chaves['publickey'], 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
```

4 Exemplo de Teste

[5]:
```python
# Parâmetros para este cenário de teste
R = next_prime(1000)
N = 2*R
W = 6
T = 32

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

message = "Estruturas Criptográficas - Grupo 7"

(y,w,c) = bike_pke.cifragem(message, bike_pke.chaves['publickey'])

message_decoded = bike_pke.decifragem(bike_pke.chaves['secretkey'], y, w, c)

if message == message_decoded.decode():
    print("Decifragem com sucesso.")
    print("Mensagem decifrada: " + message_decoded.decode())
else:
    print("Decifragem sem sucesso.")
```

```
Decifragem com sucesso.
Mensagem decifrada: Estruturas Criptográficas - Grupo 7
```