Criptografia pós-quântica -> BIKE

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 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.

```
1. O objetivo deste trabalho é a criação de protótipos em Sagemath para o algoritm
o BIKE.
2. Pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-C
CA seguro.
```

Este trabalho é dedicado às candidaturas finalistas do concurso NIST Post-Quantum Cryptography na categoria de sistemas criptográficos PKE-KEM. O objetivo deste trabalho consiste em criar um protótipo em Sagemath para o algoritmo BIKE, implementando um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.

Neste contexto, foram desenvolvidas classes Python/SageMath, que incluem as versões KEM-IND-CPA e PKE-IND-CCA. Estas implementações foram construídas com base nas especificações fornecidas em https://www.dropbox.com/sh/mx4bybl0d6e9g1m/AAC-k1JneClay7XKdj8KuSVWa/BIKE?dl=0&subfolder_nav_tracking=1 e em https://bikesuite.org/files/BIKE-presentation-NIST-04.13.2018.pdf.

Abordagem e código

Esta primeira parte do código do algoritmo BIKE começa por importar as bibliotecas necessárias e definir os parâmetros. Os espaços vetoriais e matriciais também são declarados, bem como funções auxiliares para a manipulação de bits e conversão de caracteres.

A função mask(u, v) calcula o produto escalar de dois vetores u e v. A função hamm(u) calcula o peso de Hamming do vetor u. As funções chr_to_byte, string_to_bits, string_to_poly, bits_to_string, hashs e xoring auxiliam na conversão entre diferentes representações de dados, como caracteres, bytes, bits e polinómios, além de calcular o hash e realizar a operação XOR entre duas strings.

O algoritmo Bike utiliza polinómios e matrizes circulantes, que são implementados através do anel polinomial R e do anel quociente Rr.

A função principal BF(H, code, synd, cnt_iter=r, errs=0) implementa o algoritmo Bit-Flip. Esta função recebe uma matriz H, um vetor de código code, um vetor de síndroma synd, o número máximo de iterações cnt_iter e o número máximo de erros permitidos errs. O algoritmo tenta corrigir os erros no vetor de código usando a matriz H e o vetor de síndroma synd. Se o número máximo de iterações for atingido, é lançada uma exceção.

Por fim, as funções sparse_pol e noise são usadas para gerar polinómios dispersos e pares de polinómios dispersos com um determinado número de erros.

In [54]:

```
import itertools
import os
# Importação das bibliotecas necessárias
import random as rn
from hashlib import sha256
```

In [55]:

```
# Declaração dos pârametros
K = GF(2)
um = K(1)
```

```
zero = K(0)

r = 257
n = 2 * r
t = 16
```

In [56]:

```
# Declaração dos espaços vetoriais e matriciais
Vn = VectorSpace(K, n)
Vr = VectorSpace(K, r)
Vq = VectorSpace(QQ, r)

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

In [57]:

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


# Peso de Hamming
def hamm(u):
    return sum([1 if a == um else 0 for a in u])


# Função que converte caracter em byte
def chr_to_byte(char):
    # Obtem código ascii do char
    c = ord(char)
    # Array de bits que representará o char
    b = []

    for i in range(0, 8):
        # Se c >= 2^7-i => 7-i resulta em [7, 6, 5, 4, 3, 2, 1, 0]
        if c >= 2 ** (7 - i):
            # O bit em questão tem que ser 1
            b.append(1)
        else:
            # O bit em questão tem que ser 0
            b.append(0)

        # Obtem o resto da representação do número
        c = c % 2 ** (7 - i)

    # Retornar o array de bits
    return b


# Função que transforma uma string numa lista de {-1, 1}, equivalente a bits
def string_to_bits(m):
    # Lista de bits
    bit_list = []

    for char in m:
        # Adiciona à lista de bits a representação binária do char
        bit_list.append(chr_to_byte(char))

    # Faz-se uma concatenação às listas todas, resultando numa só lista
    bit_list = list(itertools.chain.from_iterable(bit_list))  # Concatenação

    # Lista que irá ter a representação {-1, 1} em vez de binária
    new_bit_list = []
    for bit in bit_list:
        # Se bit é 0, então passa para -1
        if bit == 0:
            new_bit_list.append(-1)
        # Mante-se igual 1
        else:
            new_bit_list.append(1)
```

```
    # Retorna a lista obtida
    return new_bit_list


# Função que converte uma string para um polinómio
def string_to_poly(m, index_0s):
    # Obtem lista com a representação em {-1, 1} de m
    m_bits = string_to_bits(m)

    # Obtem o número de bits de m
    lenght_bits_m = len(m_bits)
    # Obtem o número de 0s necessários para completar a lista
    num_0s = len(index_0s)

    # Lista do polinómio com representação {-1, 0, 1}
    poly_with_0s = []
    i = 0
    j = 0
    while i < lenght_bits_m:
        if j < num_0s and i == index_0s[j]:
            # Coloca um 0 na lista do polinómio e passa para a próxima posição da lista
index_0s
            poly_with_0s.append(0)
            j += 1
        else:
            # Coloca o valor que estava na representação da string ({-1, 1})
            poly_with_0s.append(m_bits[i])
            i += 1

    # Retorna o polinómio correspondente
    return Rr(poly_with_0s)


# Função que converte bits para string
def bits_to_string(bit_string):
    # Inicializa a string
    string = ""

    # Percorre a lista de bits, de 8 em 8
    for i in range(0, len(bit_string), 8):

        byte = 0
        for j in range(0, min(8, len(bit_string) - i)):
            # Calcula o byte (byte_i = b_i * 2^7-i => 7-i resulta em [7, 6, 5, 4, 3, 2, 1
, 0])
            byte += bit_string[j + i] * 2 ** (7 - j)

        # Obtem o carater correspondente ao byte e concatena à string
        string += chr(byte)

    # Retorna a string
    return string


# Função que obtém o hash da concatenação de duas strings
def hashs(arg1, arg2=""):
    return sha256((str(arg1) + str(arg2)).encode()).digest()


# Função que efetua o xor entre duas strings binárias
def xoring(key, text):
    # Se o text for maior que a key, então esta última é multiplicada as vezes que forem
precisas
    if len(text) > len(key):
        t1 = len(text) / len(key)
        key *= ceil(t1)

    # retorna o xor
    return bytes(a ^^ b for a, b in zip(key, text))
```

In [58]:

```
# Matrizes circulantes de tamanho r com r primo
R = PolynomialRing(K, name='w')
w = R.gen()
Rr = QuotientRing(R, R.ideal(w ^ r + 1))
```

In [59]:

```
# Funções auxiliares para o algoritmo Bit-Flip
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:])
```

In [0]:

```
# Função que implementa o algoritmo Bit-Flip
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
```

In [63]:

```
# Função que obtém um polinómio disperso
def sparse_pol(sparse=3):
    coeffs = [1] * sparse + [0] * (r - 2 - sparse)
    rn.shuffle(coeffs)
    return Rr([1] + coeffs + [1])
```

```
# Função que produz um par de polinómios dispersos de tamanho "r", com um número total de erros "t"
def noise(t):
    el = [um] * t + [zero] * (n - t)
    rn.shuffle(el)

    return Rr(el[:r]), Rr(el[r:])
```

KEM IND-CPA seguro

Nesta parte implementamos o esquema KEM IND-CPA seguro. A classe possui vários métodos, incluindo a geração de chaves (keygen), cifragem (encrypt), decifragem (decrypt), encapsulamento (encaps), desencapsulamento (decaps), cifragem com KEM (encrypt_kem) e decifragem com KEM (decrypt_kem). Esta implementação segue a documentação fornecida pelo site oficial do algortimo BIKE.

In [67]:

```
# Classe que implementa o KEM-IND-CPA
class BIKE:
    def __init__(self):
        self.pk = None
        self.sk = None

    # Função que gera a chave, conforme a documentação
    def keygen(self):
        while True:
            h0 = sparse_pol()
            h1 = sparse_pol()

            if h0 != h1 and h0.is_unit() and h1.is_unit():
                break

        self.sk = (h0, h1)
        g = 1 / h1

        self.pk = g * h1, g * h0

        return self.sk, self.pk

    # Função para cifrar, conforme a documentação
    @staticmethod
    def encrypt(pk, m, e):
        e0, e1 = e

        f0, f1 = pk

        c = (m * f0 + e0, m * f1 + e1)

        return c

    # Função para decifrar, conforme a documentação
    def decrypt(self, c):
        h0, h1 = self.sk

        code = expand2(c)
        H = block_matrix(2, 1, [Rot(h0), Rot(h1)])
        synd = code * H
        cw = BF(H, code, synd)
        (m, cw) = unexpand2(cw)

        assert cw * h1 == m * h0

        f0, f1 = self.pk

        c0, c1 = c
        # Obter os erros
        e0_, e1_ = c0 - m * f0, c1 - m * f1
```

```
        return m, (e0_, e1_)

    # Função de encapsulamento, conforme a documentação
    def encapsulate(self, pk):
        m = Rr.random_element()

        e = noise(t)

        c = self.encrypt(pk, m, e)

        e0, e1 = e

        k = hashs(e0, e1)

        return c, k

    # Função de desencapsulamento, conforme a documentação
    def decapsulate(self, c):
        r, e = self.decrypt(c)

        e0_, e1_ = e

        k = hashs(e0_, e1_)

        return k

    # Função de cifragem com KEM, usando a chave partilhada
    def encrypt_kem(self, pk, m):
        e, k = self.encapsulate(pk)

        c = xoring(k, m.encode('utf-8'))

        return e, c

    # Função de decifragem com KEM, usando a chave partilhada
    def decrypt_kem(self, e, c):
        k = self.decapsulate(e)

        m = xoring(k, c).decode('utf-8')

        return m
```

Exemplo de execução do KEM

In [68]:

```
# Cria uma instância da classe BIKE
bike = BIKE()

# Gera um par de chaves
sk, pk = bike.keygen()

# Encapsulamento: gera o criptograma e a chave partilhada
e1, k_encaps = bike.encapsulate(pk)

# Cifra a mensagem
e, c = bike.encrypt_kem(pk, "Unidade Curricular de Estruturas Criptográficas")

# Desencapsulamento: obtem a chave partilhada do criptograma
k_decaps = bike.decapsulate(e1)

#Verifica se as chaves obtidas são iguais
if k_encaps == k_decaps:
    print("As chaves partilhadas são iguais.")
else:
    print("erro")

# Decifra a mensagem
m = bike.decrypt_kem(e, c)
```

```
if m == "Unidade Curricular de Estruturas Criptográficas":
    print("Cifragem e decifragem bem sucedida!!")
else:
    print("erro")

print("Mensagem original:", "Unidade Curricular de Estruturas Criptográficas")
print("Mensagem decifrada:", m)
print()
#print(e1)
```

```
As chaves partilhadas são iguais.
Cifragem e decifragem bem sucedida!!
Mensagem original: Unidade Curricular de Estruturas Criptográficas
Mensagem decifrada: Unidade Curricular de Estruturas Criptográficas
```

PKE IND-CCA seguro

Esta parte implementa uma classe "BIKE_CCA", que realiza o esquema PKE IND-CCA seguro. Esta classe utiliza a classe "BIKE" anteriormente implementada e adiciona métodos para cifragem e decifragem usando a transformação Fujisaki-Okamoto como um método de desofuscação(reveal).

In [69]:

```
# Classe que implementa o PKE-IND-CCA
class BIKE_CCA:
    def __init__(self):
        self.sk = None
        self.pk = None
        self.bike = BIKE()

    # Função que gera a chave, usando a função keygen da classe BIKE
    def keygen(self):
        self.sk, self.pk = self.bike.keygen()

        return self.sk, self.pk

    # Função para cifrar com a transformação Fujisaki-Okamoto
    def encrypt_fo(self, m, pk):
        r = sha256((os.urandom(32))).digest()

        g = hashs(r, "")

        y = xoring(g, bytes(m, encoding='utf-8'))

        e, k = self.encapsulate(r, y, pk, noise(t))

        c = xoring(k, r)

        return y, e, c

    # Função de encapsulamento com a transformação Fujisaki-Okamoto
    def encapsulate(self, r, y, pk, noised):
        yr = bits_to_string(r + y)
        yr = string_to_poly(yr, [])

        e = self.bike.encrypt(pk, yr, noised)

        k = hashs(yr, "")

        return e, k

    # Função para decifrar com a transformação Fujisaki-Okamoto
    def decrypt_fo(self, y, e, c):
        k, noised = self.reveal(e)

        r = xoring(k, c)
```

```
        _e, _k = self.encapsulate(r, y, self.pk, noised)

        if (_e, _k) != (e, k):
            raise Exception("A mensagem não pode ser decifrada")

        g = hashs(r, "")

        m = xoring(g, y)

        return m.decode('utf-8')

    # Função de desofuscação com a transformação Fujisaki-Okamoto
    def reveal(self, e):
        yr, noised = self.bike.decrypt(e)

        k = hashs(yr, "")

        return k, noised
```

Exemplo de execução do PKE

In [70]:

```
# Cria uma instância da classe BIKE_CCA
bike = BIKE_CCA()

# Gera um par de chaves
sk, pk = bike.keygen()

# Cifra a mensagem
y, e, c = bike.encrypt_fo("Trabalho prático número 3", pk)

# Decifra a mensagem
m = bike.decrypt_fo(y, e, c)

#Verifica se as mensagens são iguais
if m == "Trabalho prático número 3":
    print("Cifragem e decifragem bem sucedida!!")
else:
    print("erro")

print("Mensagem original:", "Trabalho prático número 3")
print("Mensagem decifrada:", m)
```

```
Cifragem e decifragem bem sucedida!!
Mensagem original: Trabalho prático número 3
Mensagem decifrada: Trabalho prático número 3
```

In [ ]:

```
```