KYBER

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.

1.2.1 Documento utilizado para realização do KYBER: https://pq-crystals.org/kyber/data/kyber-specification-round3.pdf

Kyber é um esquema criptográfico de pós-quântico baseado em reticulados, projetado para fornecer segurança contra ataques de computadores quânticos. Ele é construído em torno de operações em polinômios e utiliza criptografia de chave pública para garantir confidencialidade e autenticidade dos dados.

Aqui está um resumo do Kyber:

```
Algoritmo: Kyber utiliza operações em polinômios sobre um anel de coeficientes modulares para r

Par de Chaves: O Kyber gera um par de chaves composto por uma chave pública e uma chave privada

Cifragem: O esquema de cifragem do Kyber permite cifrar mensagens de forma segura. Ele usa a ch
```

Decifragem: A decifragem é o processo de recuperar a mensagem original a partir do texto cifra

Encapsulamento: O Kyber também oferece um esquema de encapsulamento de chaves. Ele permite enc

Desencapsulamento: O desencapsulamento é o processo de recuperar a chave de sessão original a

Segurança: O Kyber é projetado para resistir a ataques realizados por computadores quânticos,

Funcões auxiliares:

A primeira função é a BinomialDistribution(vD). Ela recebe um parâmetro eta que determina o nú

A segunda função é a balance(e, q=None). Ela recebe um elemento e (que pode ser um vetor, poli

[1]:
```
import sys
from sage.all import parent, ZZ, vector, PolynomialRing, GF
from sage.all import randint, set_random_seed, random_vector, matrix
```

[2]:
```
# Função auxiliar para determinar um valor de uma distribuição polinomial, dado␣
 ↪um limite
def BinomialDistribution(vD):
    r = 0
    for i in range(vD):
        r += randint(0, 1) - randint(0, 1)
    return r


# Calcular a representação de `e`, com elementos entre `-q/2` and `q/2`
def balance(e, q=None):
    # e: a vector, polynomial or scalar
    # q: optional modulus, if not present this function tries to recover it␣
 ↪from `e`
    # returns: a vector, polynomial or scalar over/in the integers
    try:
        p = parent(e).change_ring(ZZ)
        return p([balance(e_, q=q) for e_ in e])
    except (TypeError, AttributeError):
        if q is None:
            try:
                q = parent(e).order()
            except AttributeError:
                q = parent(e).base_ring().order()
        e = ZZ(e)
        e = e % q
        return ZZ(e-q) if e>q//2 else ZZ(e)
```

[3]:
```
class Kyber:
```

```
    tP = 256 # tamanho_polynomial: representa o tamanho dos polinômios usados␣
↪nas operações criptográficas
    m = 7681 # modulo: representa o módulo usado nas operações criptográficas)
    vD = 4 # valor_dispersao: determina a dispersão dos valores aleatórios␣
↪gerados durante
    # o processo criptográfico
    nB = 3 # numero_blocos: controla o número de blocos usados na matriz e nos␣
↪vetores de chave secreta
    dA = staticmethod(BinomialDistribution) # distribuicao_aleatoria: uma␣
↪função que representa uma
    # distribuição  binomial usada para gerar valores aleatórios)
    pB = [1]+[0]*(tP-1)+[1] # polinomio_base: uma lista de coeficientes que␣
↪define um polinômio usado nas
    # operações criptográficas

    @classmethod
    # Gerar um par de chaves (pública e privada)
    # Algoritmo baseado do Algoritmo 1 do documento especificado do Kyber
    def gerarChaves(cls, seed=None):

        tP, m, vD, nB, dA = cls.tP, cls.m, cls.vD, cls.nB, cls.dA

        if seed is not None:
            set_random_seed(seed)

        R, x = PolynomialRing(ZZ, "x").objgen()
        Rq = PolynomialRing(GF(m), "x")
        pB = R(cls.pB)

        A = matrix(Rq, nB, nB, [Rq.random_element(degree=tP-1) for _ in␣
↪range(nB*nB)])
        s = vector(R, nB, [R([(dA(vD)) for _ in range(tP)]) for _ in range(nB)])
        e = vector(R, nB, [R([(dA(vD)) for _ in range(tP)]) for _ in range(nB)])
        t = (A*s + e)  % pB  # NOTE ignoring compression

        return (A, t), s

    @classmethod
    # IND-CPA cifragem sem compressão de dados
    def cifrar(cls, pk, z=None, seed=None):
        # pk: chave publica
        # z: mensagem

        tP, m, vD, nB, dA = cls.tP, cls.m, cls.vD, cls.nB, cls.dA

        if seed is not None:
```

```
            set_random_seed(seed)

        A, t = pk

        R, x = PolynomialRing(ZZ, "x").objgen()
        pB = R(cls.pB)

        r  = vector(R, nB, [R([(dA(vD)) for _ in range(tP)]) for _ in␣
↪range(nB)])
        e1 = vector(R, nB, [R([(dA(vD)) for _ in range(tP)]) for _ in␣
↪range(nB)])
        e2 = R([(dA(vD)) for _ in range(tP)])

        if z is None:
            z = (0,)

        u = (r*A + e1) % pB
        u.set_immutable()
        v = (r*t + e2 + m//2 * R(list(z))) % pB
        return u, v

    @classmethod
    # IND-CPA decifragem
    def decifrar(cls, sk, c, decodificar=True):
        # sk: chave privada
        # c: ciphertext
        # decodificar: executar a decodificação final

        tP, m = cls.tP, cls.m

        s = sk
        u, v = c

        R, x = PolynomialRing(ZZ, "x").objgen()
        pB = R(cls.pB)

        z = (v - s*u) % pB
        z = list(z)
        while len(z) < tP:
            z.append(0)

        z = balance(vector(z), m)

        if decodificar:
            return cls.decodificar(z, m, tP)
        else:
            return m
```

```
    @staticmethod
    # Decodificar o vetor `m` para `{0,1}^n` dependendo da distância para `q/2`
    def decodificar(z, m, tP):
        # z: um vetor de comprimento `leq tP`

        return vector(GF(2), tP, [abs(e)>m/ZZ(4) for e in z] + [0 for _ in␣
↪range(tP-len(z))])

    @classmethod
    # IND-CCA encapsulamento sem compressão nem hash extra
    def encapsular(cls, pk, seed=None):
        # pk: chave publica

        tP = cls.tP

        if seed is not None:
            set_random_seed(seed)

        m = random_vector(GF(2), tP)
        m.set_immutable()
        set_random_seed(hash(m))

        nB = random_vector(GF(2), tP)
        nB.set_immutable()
        r = ZZ.random_element(0, 2**tP-1)

        c = cls.cifrar(pk, m, r)

        nB = hash((nB, c))
        return c, nB

    @classmethod
    # IND-CCA desencapsulamento
    def desencapsular(cls, sk, pk, c):
        # sk: chave privada
        # pk: chave publica
        # c: ciphertext

        tP = cls.tP

        m = cls.decifrar(sk, c)
        m.set_immutable()
        set_random_seed(hash(m))

        nB = random_vector(GF(2), tP)
        nB.set_immutable()
```

```
        r = ZZ.random_element(0, 2**tP-1)

        c_ = cls.cifrar(pk, m, r)

        if c == c_:
            return hash((nB, c))
        else:
            return hash(c)
```

[4]:
```
# Testar a implementação de IND-CPA
def testarKyberCpa(cls=Kyber, t=16):
    for i in range(t):
        # gerar chaves
        pk, sk = cls.gerarChaves(seed=i)
        # gerar uma mensagem aleatória (random_vector)
        m0 = random_vector(GF(2), cls.tP)
        # cifragem
        c = cls.cifrar(pk, m0, seed=i)
        # decifragem
        m1 = cls.decifrar(sk, c)
        # asserção
        assert(m0 == m1)


# Testar a implementação de IND-CCA
def testarKyberCca(cls=Kyber, t=16):
    for i in range(t):
        # gerar chaves
        pk, sk = cls.gerarChaves(seed=i)
        # encapsulamento
        c, K0 = cls.encapsular(pk, seed=i)
        # desencapsulamento
        K1 = cls.desencapsular(sk, pk, c)
        # asserção
        assert(K0 == K1)


# Testar ambas as implementações
def testarKyber(cls=Kyber, t=16):
    # testar IND-CPA
    print("Começou o IND-CPA", end=" ")
    testarKyberCpa(cls, t)
    print("Terminou o IND-CPA")

    # testar IND-CCA
    print("Começou o IND-CCA", end=" ")
    testarKyberCca(cls, t)
```

```
    print("Terminou o IND-CCA")
```

[6]:
```
print("Testar 1 vez:")
testarKyber(Kyber, 1)

print("Testar 2 vezes:")
testarKyber(Kyber, 2)

print("Testar 3 vezes:")
testarKyber(Kyber, 3)

print("Testar 10 vezes:")
testarKyber(Kyber, 10)
```

```
Testar 1 vez:
Começou o IND-CPA Terminou o IND-CPA
Começou o IND-CCA Terminou o IND-CCA
Testar 2 vezes:
Começou o IND-CPA Terminou o IND-CPA
Começou o IND-CCA Terminou o IND-CCA
Testar 3 vezes:
Começou o IND-CPA Terminou o IND-CPA
Começou o IND-CCA Terminou o IND-CCA
Testar 10 vezes:
Começou o IND-CPA Terminou o IND-CPA
Começou o IND-CCA Terminou o IND-CCA
```

[ ]:
```
```