KYBER

May 1, 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 KYBER

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

[ ]:
```
import sys
from sage.all import *

def BinomialDistribution(eta):
    """
        Determina um valor de uma distribuição polinomial, dado um limite.
    """
    r = 0
    for i in range(eta):
        r += randint(0, 1) - randint(0, 1)
    return r


class Kyber:

    def __init__(self):
        self.n = 256
        self.q = 7681
        self.eta = 4
        self.k = 3
        self.D = BinomialDistribution
        self.f = [1]+[0]*(self.n-1)+[1]
        self.ce = self.n

    def generate_keys(self, seed=None):
        """
```

```
            Gera um par de chaves (pública e privada).
        """
        n, q, eta, k, D = self.n, self.q, self.eta, self.k, self.D

        if seed is not None:
            set_random_seed(seed)

        R, x = PolynomialRing(ZZ, "x").objgen()
        Rq = PolynomialRing(GF(q), "x")
        f = R(self.f)

        A = matrix(Rq, k, k, [Rq.random_element(degree=n-1) for _ in␣
↪range(k*k)])
        s = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        t = (A*s + e)  % f

        return (A, t), s

    def encrypt(self, public_key, m=None, seed=None):
        """
            Cifragem IND-CPA sem compressão de dados.
            Parâmetros:
            ----------
            public_key
            m: mensagem opcional
            seed: usada para random sampling se necessário
        """

        n, q, eta, k, D = self.n, self.q, self.eta, self.k, self.D

        if seed is not None:
            set_random_seed(seed)

        A, t = public_key

        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R(self.f)

        r  = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e1 = vector(R, k, [R([(D(eta)) for _ in range(n)]) for _ in range(k)])
        e2 = R([(D(eta)) for _ in range(n)])

        if m is None:
            m = (0,)

        u = (r*A + e1) % f
```

```
        u.set_immutable()
        v = (r*t + e2 + q//2 * R(list(m))) % f
        return u, v

    def decrypt(self, secret_key, c, decode=True):
        """
            Decifragem IND-CPA.
            Parâmetros:
            -----------
            secret_key
            c: ciphertext
            decode: fazer decoding final
        """

        n, q = self.n, self.q

        s = secret_key
        u, v = c

        R, x = PolynomialRing(ZZ, "x").objgen()
        f = R(self.f)

        m = (v - s*u) % f
        m = list(m)
        while len(m) < n:
            m.append(0)

        m = self.balance(vector(m), q)

        if decode:
            return self.decode(m, q, n)
        else:
            return m

    @staticmethod
    def decode(m, q, n):
        """
            Decode vector `m` to `{0,1}^n` depending on distance to `q/2`.
            Parâmetros:
            -----------
            m: vetor de tamanho `leq n`
            q: módulo

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

```
    def encap(self, public_key, seed=None):
        """
            Encapsulamento IND-CCA.
            Parâmetros:
            -----------
            public_key
            seed: usada para random sampling
        """
        n = self.n

        if seed is not None:
            set_random_seed(seed)

        m = random_vector(GF(2), n)
        m.set_immutable()
        set_random_seed(hash(m))
        K_ = random_vector(GF(2), n)
        K_.set_immutable()
        r = ZZ.random_element(0, 2**n-1)

        c = self.encrypt(public_key, m, r)

        K = hash((K_, c))
        return c, K

    def decap(self, secret_key, public_key, c):
        """
            Decaps IND-CCA.
            Parâmetros:
            -----------
            secret_key
            public_key
            c: ciphertext
        """
        n = self.n

        m = self.decrypt(secret_key, c)
        m.set_immutable()
        set_random_seed(hash(m))

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

        c_ = self.encrypt(public_key, m, r)

        if c == c_:
```

```
            return hash((K_, c))
        else:
            return hash(c)

    def test_kyber_cpa(self, t=16):
        """
            Testar correção do encrypt/decrypt IND-CCA.
            Retorna erro se o texto decifrado não for igual ao original.
        """
        for i in range(t):
            # gerar chaves
            public_key, secret_key = self.generate_keys(seed=i)
            # gerar uma mensagem aleatória (random_vector)
            m0 = random_vector(GF(2), self.n)

            c = self.encrypt(public_key, m0, seed=i)
            m1 = self.decrypt(secret_key, c)

            assert(m0 == m1)


    def test_kyber_cca(self, t=16):
        """
            Testar correção do encaps/decaps IND-CCA.
            Retorna erro se as chaves original e final não forem iguais.
        """
        for i in range(t):
            # gerar chaves
            public_key, secret_key = self.generate_keys(seed=i)
            # encapsulamento
            c, K0 = self.encap(public_key, seed=i)
            # desencapsulamento
            K1 = self.decap(secret_key, public_key, c)
            # assert
            assert(K0 == K1)

    def test_kyber(self, t=16):

        print("Kyber IND-CPA:")
        self.test_kyber_cpa(t)
        print("correct")

        # testar IND-CCA
        print("Kyber IND-CCA ")
        self.test_kyber_cca(t)
        print("correct")
```

```
    def balance(self, e, q=None):
        """
            Calcula a representação de `e`, com elementos entre `-q/2` and `q/2`
            Parâmetros:
            ----------
            e: vetor (polinomial ou escalar)
            q: modulo opcional

            retorna um vetor
        """
        try:
            p = parent(e).change_ring(ZZ)
            return p([self.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)
```

[2]:
```
def main():
    kyber = Kyber()

    print("Testar com apenas um teste:")
    kyber.test_kyber(1)

    print("Testar com 20 testes:")
    kyber.test_kyber(20)

    print("Testar com 30 testes:")
    kyber.test_kyber(30)

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

```
Testar com apenas um teste:
Kyber IND-CPA:
correct
Kyber IND-CCA
correct
Testar com 20 testes:
Kyber IND-CPA:
correct
Kyber IND-CCA
correct
```

```
Testar com 30 testes:
Kyber IND-CPA:
correct
Kyber IND-CCA
correct
```