Trabalho Prático 2 - KYBER - Grupo 18

Objetivos do trabalho Prático:

- Criar um protótipo em Sagemath da técnica de criptografia pós-quântica KYBER, implementando um KEM IND-CPA seguro e um PKE IND-CCA seguro.

Imports

Seção onde constam todos os imports necessários

In [ ]:
```
import os
from hashlib import shake_128, shake_256, sha256, sha512
from bitstring import BitArray
from random import choice
import ast
```

Classe NTT (Number Theoretic Transform)

Neste projeto, iremos ainda utilizar a classe NTT (Number Theoretic Transform) fornecida pelo docente, acrescentando algumas alterações face a necessidades encontradas ao longo do projeto.

In [ ]:
```
# Classe que implementa o NTT (Number Theoretic Transform)

class NTT(object):
#
    def __init__(self, n=128, q=None, base_inverse=False):
        if not  n in [4,8,16,32,64,128,256,512,1024,2048,4096]:
            raise ValueError("improper argument ",n)
        self.n = n
        if not q:
            self.q = 1 + 2*n
            while True:
                if (self.q).is_prime():
                    break
                self.q += 2*n
        else:
            if q % (2*n) != 1:
                raise ValueError("Valor de 'q' não verifica a condição NTT")
            self.q = q

        self.F = GF(self.q) ;  self.R = PolynomialRing(self.F, name="w")
        w = (self.R).gen()

        g = (w^n + 1)
        x = g.roots(multiplicities=False)[-1]
        self.x = x
        if  base_inverse:
            rs = [x^(2*i+1)  for i in range(n)]
            self.base = crt_basis([(w - r) for r in rs])
        else:
            self.base = None
```

```
    def ntt(self,f,inv=False):
        def _expand_(f):
            if isinstance(f, list):
                u = f
            else :
                u = f.list()

            return u + [0]*(self.n-len(u))

        def _ntt_(x,N,f,inv=inv):
            if N==1:
                return f
            N_ = N//2 ; z =  x^2
            f0 = [f[2*i]   for i in range(N_)] ; f1 = [f[2*i+1] for i in range(N_)]
            ff0 = _ntt_(z,N_,f0,inv=inv) ; ff1 = _ntt_(z,N_,f1,inv=inv)

            s  = self.F(1) if inv else x
            ff = [self.F(0) for i in range(N)]
            for i in range(N_):
                a = ff0[i] ; b = s*ff1[i]
                ff[i] = a + b ; ff[i + N_] = a - b
                s = s * z
            return ff


        vec = _expand_(f)
        if  not inv:
            return self.R(_ntt_(self.x,self.n, vec, inv=inv))
        elif self.base != None:
            return sum([vec[i]*self.base[i] for i in range(self.n)])
        else:
            n_ = (self.F(self.n))^-1
            x_ = (self.x)^-1
            u = _ntt_(x_,self.n,vec, inv=inv)

            return self.R([n_ * x_^i * u[i] for i in range(self.n)])


    def random_pol(self,args=None):
        return (self.R).random_element(args)
```

Declaração e Inicialização de parâmetros

In [ ]:

```
# Declaração dos pârametros 'n' e 'q'

n = 256

q = next_prime(3*n)
while q % (2*n) != 1:
    q = next_prime(q+1)

#######################################################

# Declaração de anéis

_Z.<w> = ZZ[]
R.<w> = QuotientRing(_Z, _Z.ideal(w^n + 1))

_q.<w> = GF(q)[]
_Rq.<w> = QuotientRing(_q, _q.ideal(w^n + 1))
```

```
Rq = lambda x : _Rq(R(x))

#######################################################

# Inicialização do objeto NTT

T = NTT(n=n, q=q)
```

Funções auxiliares NTT

Devido à classe fornecida não realizar todas as tarefas que nos pretendemos, temos ainda de acrescentar métodos novos, capazes de dar resposta a todas as nossas necessidades. Posto isto, foram ainda criadas várias funções auxiliares para esse propósito.

In [ ]:

```
# Função que executa o ntt inverso para todos os elementos de uma matriz/array
def my_ntt_inv(f):
    if type(f[0]) is list:
        res = []
        for i in range(len(f)):

            if type(f[i][0]) is list:
                res.append([])
                for j in range(len(f[i])):
                    res[i].append(T.ntt(f[i][j], inv=True))

            else:
                res.append(T.ntt(f[i], inv=True))
    else:
        res = T.ntt(f, inv=True)

    return res



# Função que executa o ntt para todos os elementos de uma matriz/array
def my_ntt(f):

    if type(f) is list:
        res = []
        for i in range(len(f)):

            if type(f[i]) is list:
                res.append([])
                for j in range(len(f[i])):
                    res[i].append(T.ntt(f[i][j]))

            else:
                res.append(T.ntt(f[i]))
    else:
        res = T.ntt(f)

    return res



# Função que executa a multiplicação entre dois objetos ntt
def my_mult(ff1, ff2, N=n, Q=q):
    res = []

    for i in range(N):
        res.append((ff1[i] * ff2[i]) % Q)
```

```
    return res


# Função que executa a soma entre dois objetos ntt
def my_add(ff1, ff2, N=n, Q=q):
    res = []

    for i in range(N):
        res.append((ff1[i] + ff2[i]) % Q)

    return res


# Função que executa a subtração entre dois objetos ntt
def my_sub(ff1, ff2, N=n, Q=q):
    res = []

    for i in range(N):
        res.append((ff1[i] - ff2[i]) % Q)

    return res
```

Funções auxiliares Vectores

In [ ]:

```
# Função que executa a multiplicação entre uma matriz e um vetor (objetos ntt)
def mult_mat_vec(mat, vec, k=2, n=n):
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            mat[i][j] = my_mult(mat[i][j], vec[j])

    tmp = [[0] * n] * k
    for i in range(len(mat)):
        for j in range(len(mat[i])):
            tmp[i] = my_add(tmp[i], mat[i][j])

    return tmp


# Função que executa a multiplicação entre dois vetores (objetos ntt)
def mult_vec(vec1, vec2, n=n):
    for i in range(len(vec1)):
        vec1[i] = my_mult(vec1[i], vec2[i])

    tmp = [0] * n
    for i in range(len(vec1)):
        tmp = my_add(tmp, vec1[i])

    return tmp


# Função que executa a soma entre dois vetores (objetos ntt)
def sum_vec(vec1, vec2):
    for i in range(len(vec1)):
        vec1[i] = my_add(vec1[i], vec2[i])

    return vec1


# Função que executa a subtração entre dois vetores (objetos ntt)
```

```
def sub_vec(vec1, vec2):
    for i in range(len(vec1)):
        vec1[i] = my_sub(vec1[i], vec2[i])

    return vec1
```

Função Auxiliar de Compress e Compress Recursivo (Página 5 - Documento KYBER)

In [ ]:

```
# Função de compress, de acordo com o algoritmo da pg.5
def compress(x, d, q):
    coefs = x.list()

    new_coefs = []
    _2power = int(2 ** d)

    for coef in coefs:
        new_coef = round(_2power / q * int(coef)) % _2power
        new_coefs.append(new_coef)

    return Rq(new_coefs)



# Função de compress aplicada a todos os elementos de uma matriz/array
def compress_rec(f, d, q):
    if type(f) is list:
        res = []
        for i in range(len(f)):
            if type(f[i]) is list:
                res.append([])
                for j in range(len(f[i])):
                    res[i].append(compress(f[i][j], d, q))

            else:
                res.append(compress(f[i], d, q))
    else:
        res = compress(f, d, q)

    return res
```

Função Auxiliar de Decompress e Decompress Recursivo (Página 5 - Documento KYBER)

In [ ]:

```
# Função de decompress, de acordo com o algoritmo da pg.5
def decompress(x, d, q):
    coefs = x.list()

    new_coefs = []
    _2power = 2 ** d

    for coef in coefs:
        new_coef = round(q / _2power * int(coef))
        new_coefs.append(new_coef)

    return Rq(new_coefs)



# Função de decompress aplicada a todos os elementos de uma matriz/array
```

```
def decompress_rec(f, d, q):
    if type(f) is list:
        res = []
        for i in range(len(f)):
            if type(f[i]) is list:
                res.append([])
                for j in range(len(f[i])):
                    res[i].append(decompress(f[i][j], d, q))

            else:
                res.append(decompress(f[i], d, q))
    else:
        res = decompress(f, d, q)

    return res
```

Instanciação de funções (Página 11 - Documento KYBER)

In [ ]:

```
# Instanciação de funções, de acordo com a pg.11

def PRF(s,b):
    return shake_256(str(s).encode() + str(b).encode()).digest(int(2000))

def XOF(p,i,j):
    return shake_128(str(i).encode() + str(j).encode() + str(p).encode()).digest(int

def H(s):
    return sha256(str(s).encode()).digest()

def G(a,b=""):
    digest = sha512(str(a).encode() + str(b).encode() ).digest()
    return digest[:32], digest[32:]

def KDF(a,b=""):
    return shake_256(str(a).encode() + str(b).encode()).digest(int(2000))
```

Funções adicionais (Paginas 6 e 7 - Documento KYBER)

In [ ]:

```
# Função que efetua o xor entre duas strings binárias
def xoring(key, text):

    if len(text) > len(key):
        t1 = len(text) / len(key)
        key *= ceil(t1)

    return bytes(a ^^ b for a, b in zip(key, text))



# Função de parse, de acordo com o algoritmo da pg.6
def parse(b, q, n):
    i = 0
    j = 0
    a = []

    while j < n and i + 2 < len(b):
        d1 = b[i] + 256 * b[i + 1] % 16
        d2 = b[i+1]//16 + 16 * b[i + 2]

        if d1 < q:
```

```
            a.append(d1)
            j += 1

        elif d2 < q and j < n:
            a.append(d2)
            j += 1

        i += 3

    return Rq(a)



# Função de Centered Binomial Distribution, de acordo com o algoritmo da pg.7
def CBD(byte_array, base):
    f = []

    bit_array = BitArray(bytes=byte_array).bin[2:]
    for i in range(256):
        a = 0
        b = 0

        for j in range(base):
            a += 2**j if int(bit_array[2*i * base + j]) else 0
            b += 2**j if int(bit_array[2*i * base + base + j]) else 0

        f.append(a-b)

    return R(f)
```

KYBER - KEM

De modo a fazer a melhor implementação possível desta técnica, a equipa decidiu seguir e guiar-se pelo documento do **KYBER** especificada no seu documento técnico disponibilizado na drive, dado que possui todos os passos necessários.

Iremos inicialmente construir uma **PKE IND-CPA**, tal como apresentado no documento e transformar a mesma numa **KEM IND-CPA**.

In [ ]:

```
# Classe que implementa a versão PKE-IND-CPA do Kyber
class Kyber:

    # Inicializar parâmetros
    def __init__(self, n, k, q, n1, n2, du, dv):
        self.n = n
        self.k = k
        self.q = q
        self.n1 = n1
        self.n2 = n2
        self.du = du
        self.dv = dv



    # Função de geração da chave, de acordo com o algoritmo da pg.8
    def keygen(self):
        d = _Rq.random_element()
        p, o = G(d)
```

```
        N = 0

        A = [0, 0] # Inicializar matriz
        # Gerar matriz A
        for i in range(self.k):
            A[i] = []
            for j in range(self.k):
                A[i].append(T.ntt(parse(XOF(p, j, i), self.q, self.n)))

        # Gerar array "s" e "e"
        s = [0] * self.k
        for i in range(self.k):
            s[i] = T.ntt(CBD(PRF(o, N), self.n1))
            N += 1

        e = [0] * self.k
        for i in range(self.k):
            e[i] = T.ntt(CBD(PRF(o, N), self.n1))
            N += 1

        mult = mult_mat_vec(A, s)
        t = sum_vec(mult, e)

        self.pk = t, p
        self.sk = s

        return self.sk, self.pk



    # Função de cifragem, de acordo com o algoritmo da pg.9
    def encrypt(self, pk, m, coins):
        N = 0
        t, p = pk

        A = [0, 0] # Inicializar matriz
        # Gerar matriz A
        for i in range(self.k):
            A[i] = []
            for j in range(self.k):
                A[i].append(T.ntt(parse(XOF(p, i, j), self.q, self.n)))

        # Gerar "r" e "e1"
        r = [0] * self.k
        for i in range(self.k):
            r[i] = T.ntt(CBD(PRF(coins, N), self.n1))
            N += 1

        e1 = [0] * self.k
        for i in range(self.k):
            e1[i] = T.ntt(CBD(PRF(coins, N), self.n2))
            N += 1

        e2 = T.ntt(CBD(PRF(coins, N), self.n2))

        mult = mult_mat_vec(A, r)
        u = sum_vec(mult, e1)

        t = [] + t
        mult = mult_vec(t, r)
        v = my_add(mult, e2)
        v = my_add(v, T.ntt(m))

        u = my_ntt_inv(u)
```

```
    v = my_ntt_inv(v)

    c1 = compress_rec(u, self.du, self.q)
    c2 = compress_rec(v, self.dv, self.q)

    return (c1, c2)



# Função de decifragem, de acordo com o algoritmo da pg.9
def decrypt(self, c):
    u, v = c
    u = decompress_rec(u, self.du, q)
    v = decompress_rec(v, self.dv, q)

    u = my_ntt(u)
    v = my_ntt(v)

    s = [] + self.sk

    mult = mult_vec(s, u)
    m = my_sub(v, mult)

    return compress(T.ntt(m,inv=True), 1, q)



# PKE IND-CPA -----> KEM IND-CPA
#  |
#  |
#  |  Tranformação de PKE IND-CPA para KEM IND-CPA segundo o Capítulo 2
#  |
#  |
#  V



# Função de encapsulamento (necessária para o KEM)
def encaps(self, pk):

    # Gerar polinómio para o encapsulamento
    m1 = Rq([choice([0, 1]) for i in range(n)])
    coins = os.urandom(256)

    # Obter o criptograma
    e = self.encrypt(pk, decompress(m1, 1, q), coins)

    # Obter a chave partilhada
    k = H(m1)

    return e, k



# Função de desencapsulamento (necessária para o KEM)
def decaps(self, c):

    # Obter polinómio gerado no encapsulamento
    m = self.decrypt(c)

    # Obter a chave partilhada
    k = H(m)

    return k
```

```
    # Função de cifragem com KEM
    def encrypt_kem(self, pk, m):

        # Obter criptograma da chave partilhada e a chave partilhada
        e, k = self.encaps(pk)

        # Obter criptograma
        c = xoring(k, m.encode('latin1'))

        return e, c


    # Função de decifragem com KEM
    def decrypt_kem(self, e, c):

        # Obter chave partilhada
        k = self.decaps(e)

        # Obter mensagem
        m = xoring(k, c).decode('latin1')

        return m
```

Testes KEM

In [ ]:

```
kyber = Kyber(n, 2, q, 3, 2, 10, 4)
sk, pk = kyber.keygen()

print("Teste de cifragem e decifragem \033[1m[KEM]\033[0m\n")

text = "Teste de KEM (KYBER 1)"

# Cifra
e, c = kyber.encrypt_kem(pk, text)

# Decifra
m = kyber.decrypt_kem(e, c)

print("\033[1mMensagem inicial = \033[0m", text)
print("\033[1mMensagem final   = \033[0m", m)

if text == m:
    print("\n\033[1mAs mensagens iniciais e finais são iguais!\033[0m")
    print(" └> Processo completado com sucesso. Cifragem e Decifragem bem efetuadas.
else:
    print("A decifragem não teve sucesso!")
```

```
Teste de cifragem e decifragem [KEM]

Mensagem inicial =  Teste de KEM (KYBER 1)
Mensagem final   =  Teste de KEM (KYBER 1)

As mensagens iniciais e finais são iguais!
 └> Processo completado com sucesso. Cifragem e Decifragem bem efetuadas.
```

KYBER - PKE

De modo a fazer a melhor implementação possível desta técnica, a equipa decidiu seguir e guiar-se pelo documento do **KYBER** especificada no seu documento técnico disponibilizado na drive, dado que possui todos os passos necessários.

Iremos inicialmente construir uma **PKE IND-CPA**, tal como apresentado no documento e transformar a mesma numa **PKE IND-CCA**.

In [ ]:
```
# Classe que implementa a versão PKE-IND-CCA do Kyber (a partir da classe anterior P
class Kyber_CCA:

    # Inicializar parâmetros
    def __init__(self, n, k, q, n1, n2, du, dv):
        self.n = n
        self.k = k
        self.q = q
        self.n1 = n1
        self.n2 = n2
        self.du = du
        self.dv = dv

        self.kyber = Kyber(n, k, q, n1, n2, du, dv)


    # Função de geração da chave, recorrendo à função keygen da classe anterior
    def keygen(self):

        self.sk, self.pk = self.kyber.keygen()

        return self.sk, self.pk


    # Função de cifragem, recorrendo à função encrypt da classe anterior
    def encrypt(self, pk, r, y):

        # Obter hash r||y
        ry = H(bytes(r) + y)

        # Cifrar r e hash r||y
        c = self.kyber.encrypt(pk, decompress(r, 1, self.q), ry)

        return c


    # Função de decifragem, recorrendo à função decrypt da classe anterior
    def decrypt(self, c):

        r = self.kyber.decrypt(c)

        return r



    # PKE IND-CPA -----> PKE IND-CCA
    #  |
    #  |
    #  |  Tranformação de PKE IND-CPA para PKE IND-CCA segundo o Capítulo 2
    #  |
    #  |
    #  V
```

```
    # Função de cifragem FO (cap.2)
    def encrypt_fo(self, m, pk):
        r = Rq([choice([0, 1]) for i in range(n)])

        g = H(r)

        y = xoring(g, bytes(m, encoding='latin1'))

        c = self.encrypt(pk, r, y)

        return y, c


    # Função de decifragem FO (cap.2)
    def decrypt_fo(self, y, c):
        r = self.decrypt(c)

        _c = self.encrypt(pk, r, y)

        if c != _c:
            raise Exception("Mensagem não pode ser decifrada")

        g = H(r)

        m = xoring(g, y)

        return m.decode('latin1')
```

Testes PKE

In [ ]:

```
kyber = Kyber_CCA(n, 2, q, 3, 2, 10, 4)
sk, pk = kyber.keygen()

print("Teste de cifragem e decifragem \033[1m[PKE]\033[0m\n")

text = "Teste de PKE (KYBER 2)"

# Cifrar
y, c = kyber.encrypt_fo(text, pk)

# Decifrar
m = kyber.decrypt_fo(y, c)

print("\033[1mMensagem inicial = \033[0m", text)
print("\033[1mMensagem final   = \033[0m", m)

if text == m:
    print("\n\033[1mAs mensagens iniciais e finais são iguais!\033[0m")
    print(" └> Processo completado com sucesso. Cifragem e Decifragem bem efetuadas.
else:
    print("A decifragem não teve sucesso!")
```

```
Teste de cifragem e decifragem [PKE]

Mensagem inicial =  Teste de PKE (KYBER 2)
Mensagem final   =  Teste de PKE (KYBER 2)

As mensagens iniciais e finais são iguais!
 └> Processo completado com sucesso. Cifragem e Decifragem bem efetuadas.
```