Trabalho prático 2

Grupo 5:

- Duarte Oliveira \<pg47157>
- Melânia Pereira \<pg47520>

Post-Quantum Cryptography na categoria de criptosistemas PKE-KEM

Criação de protótipo em Sagemath de uma técnica representativa da família de criptosistemas pós-quânticos NTRU ("lattice based").
Pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.
Para o desenvolvimento destas soluções foram seguidas as especificações dos documentos oficiais, mais concretamente do mais recente: https://ntru.org/f/ntru-20190330.pdf.

PKE

Começamos por inicializar os parâmetros de acordo com o conjunto de parâmetros ntru-hrss recomendado **ntruhrss701** (de acordo com o submetido na 3ª ronda do PQC) em que n=701. Assumimos q=4096 e p=3 e criamos também os anéis necessários (Zx,R,Rq).

Seguimos, depois para a geração de chaves, depois a cifragem e, finalmente, a decifragem.

Geração de chaves

Desta função obtemos um par de chaves, a chave pública h e a chave privada composta por **(**f**,**f_q**,**h_q**)**

- Começamos por gerar os polinómios f e g, recorrendo a uma *seed* e à função **Sample_fg(seed)**, que funciona da seguinte forma:
  - a *seed* é uma *bit string* com o tamanho suficiente para ser dividida em metade e gerar os dois polinómios;
  - depois, percorrendo a seed bit a bit, a geração do polinómio é feita de acordo com o valor do bit, ou seja:
    - se o bit da seed for 0, o coeficiente do polinómio será 1
    - se o bit da seed for 1, o coeficiente será -1
    - para ficar com o tamanho certo, o resto dos coeficientes do polinómio são completados com 0
    - finalmente, é feito um shuffle dos coeficientes
- Passamos então para o cálculo das inversas, tendo sempre em conta que os

polinómios gerados podem não possuir inversa, para mitigar este erro, recorreu-se a um cicli que, no caso de falha do cálculo, faz uma nova geração de f e g, recalculando depois as inversas. As inversas calculadas são então as seguintes:

- f_q = $(1/f)$ mod q
- f_p = $(1/f)$ mod p
- h_q = $(1/h)$ mod q
- De notar que, como é necessário o cáluclo da inversa modulo q de h, é, também nesta fase, calculada a chave pública **h** da seguinte forma:
  - h = $(p.g.f_q)$ mod q Obtemos, assim, a chave pública e a chave privada.

Cifragem

Para cifrar, são necessários os seguintes parâmetros:

- conhecimento da chave pública do destinatário (h calculado na geração de chaves)
- um polinómio r gerado de forma aleatória através da função **Sample_rm(coins)** e com auxílio da função **randomBitString(size)**
- a mensagem m a cifrar (no nosso caso também gerada de forma aleatória em conjunto com o polinómio r) O criptograma é calculado da seguinte forma:
- c = $(r.h+m)$ mod q

Decifragem

Para decifrar, é necessário ter a chave privada (f,f_q,h_q) e o criptograma c. Com estes dados, são calculados os seguintes parâmetros:

- a = $(c.f)$ mod q
- m = $(a.f_p)$ mod p
- r = $((c-m).h_q)$ mod q Se os polinómios r e m não forem ternários, retorna (0,0,1), senão, retorna (r,m,0)

In [1]:

```
import random, hashlib



class NTRU_PKE(object):

    def __init__(self, N=701, Q=4096, D=495, timeout=None):

        # Todas as inicializações de parâmetros são baseadas na submissao
        self.n = N
        self.q = Q
        self.d = D

        # Definição dos aneis
        Zx.<x>  = ZZ[]
        self.Zx = Zx
        Qq = PolynomialRing(GF(self.q), 'x')
        x = Zx.gen()
        y = Qq.gen()
        R = Zx.quotient(x^self.n-1)
```

```
        self.R = R
        Rq = QuotientRing(Qq, y^self.n-1)
        self.Rq = Rq


    # Gera uma string de bits com tamanho size
    def randomBitString(self, size):
        # Gera uma sequencia de n bits aleatorios
        u = [random.choice([0,1]) for i in range(size)]
        # Mistura os valores da lista, so para aumentar a aleatoriedade
        random.shuffle(u)
        return u


    # Verifica se um polinomio e ternario
    def isTernary(self, f):
        res = True
        v = list(f)
        for i in v:
            if i > 1 or i < -1:
                res = False
                break
        return res


    # Produz o polinomio f modulo q. Mas, em vez de ser entre 0 e q-1, fic
    def arredonda_mod(self, f, q):
        g = list(((f[i] + q//2) % q) - q//2 for i in range(self.n))
        return self.Zx(g)


    # Produz a inversa de um polinomio f modulo x^n-1 modulo p, em que p é
    def inversa_modP(self, f, p):
        T = self.Zx.change_ring(Integers(p)).quotient(x^self.n-1)
        return self.Zx(lift(1 / T(f)))


    # Como a funcao de cima, mas o q aqui é uma potencia de 2
    def inversa_mod2(self, f, q):
        assert q.is_power_of(2)
        g = self.inversa_modP(f, 2)
        while True:
            r = self.arredonda_mod(self.R(g*f), q)
            if r == 1:
                return g
            g = self.arredonda_mod(self.R(g*(2 - r)), q)


    # Gera um polinomio ternario
    def Ternary(self, bit_string):
        # cria um array
        result = []
        # Itera d vezes
        for j in range(self.d):
            # Se o bit for 0, acrescenta 1, senao -1
            if bit_string[j] == 0:
                result += [1]
            elif bit_string[j] == 1:
                result += [-1]
        # Preenche com 0's o array restante
        result += [0]*(self.n-self.d)
```

```
        # Mistura os valores do array
        random.shuffle(result)

        return self.Zx(result)


    # Gera um polinomio f e um polinomio g
    def Sample_fg(self, seed):
        x = self.R.gen()

        # Parse de fg_bits em f_bits||g_bits
        f_bits = seed[:self.d]
        g_bits = seed[self.d:]

        # Definir f = Ternary_Plus(f_bits)
        f = self.Ternary(f_bits)
        # Definir g0 = Ternary_Plus(g_bits)
        g = self.Ternary(g_bits)

        return (f,g)


    # Gera um polinomio r e um polinomio
    def Sample_rm(self, coins):
        # sample_iid_bits = 8*n - 8
        sample_iid_bits = 8*self.n - 8

        # Parse de rm_bits em r_bits||m_bits
        r_bits = coins[:sample_iid_bits]
        m_bits = coins[sample_iid_bits:]

        # Set r = Ternary(r_bits)
        r = self.Ternary(r_bits)
        # Set m = Ternary(m_bits)
        m = self.Ternary(m_bits)

        return (r,m)


    # Funcao usada para gerar o par de chaves pública e privada
    def keyGen(self, seed):
        while True:
            try:
                (f,g) = self.Sample_fg(seed)

                fp = self.inversa_modP(f, 3)

                fq = self.inversa_mod2(f, self.q)

                # gq <- (1/f) mod (q;φn) (para garantir que h e invertivel
                gq = self.inversa_mod2(g, self.q)

                h = self.arredonda_mod(3*self.R(g*fq), self.q)

                hq = self.inversa_mod2(h, self.q)
                break
            except:
                seed = self.randomBitString(2*self.d)
                pass

        return {'sk' : (f,fp,hq) , 'pk' : h}
```

```
    # Recebe como parametros a chave publica e o tuplo (r,m)
    def encrypt(self, h, rm):
        r,m = rm[0],rm[1]

        c = self.arredonda_mod(self.R(h*r)+m, self.q)

        return c


    # Recebe como parametros a chave privada (f,fp,hq) e ainda o criptogra
    def decrypt(self, sk, c):
        a = self.arredonda_mod(self.R(c*sk[0]), self.q)

        m = self.arredonda_mod(self.R(a * sk[1]), 3)

        aux = (c-m) * sk[2]
        r = self.arredonda_mod(self.R(aux), self.q)

        # Se os polinomios nao forem ternarios, retorna erro
        if not self.isTernary(r) and not self.isTernary(m):
            (0,0,1)
        return (r,m,0)
```

In [2]:

```
# Parametros do NTRU (ntruhps4096821)
N=701
Q=4096
D=495

# Inicializacao da classe
ntru = NTRU_PKE(N,Q,D)

print("[Teste da cifragem e decifragem]")
keys = ntru.keyGen(ntru.randomBitString(2*D))
rm = ntru.Sample_rm(ntru.randomBitString(11200))

c = ntru.encrypt(keys['pk'], rm)

rmDec = ntru.decrypt(keys['sk'], c)

if rmDec[0] == rm[0] and rmDec[1] == rm[1] and rmDec[2] == 0:
    print("As mensagens e os polinómios r são iguais!")
else:
    print("A decifragem falhou.")
```

```
[Teste da cifragem e decifragem]
As mensagens e os polinómios r são iguais!
```

KEM

Para a inicialização da classe KEM, aproveitamos a classe PKE e as suas funções de geração de chave, cifra e decifra, definidas em cima. São usados também os mesmos parâmetros descritos anteriormente, recomendados na 3ª ronda do PQC (**ntruhrss701** em que n=701, q=4096 e p=3).

Definimos, então:

- uma função de geração de chave (que tira partido da já definida no PKE)
- uma função de encapsulamento (que tira partida da função de cifra definida no PKE)
- uma função de desencapsulamento (que tira partido da função de decifra definida no PKE)

Geração de chaves

Como já referido, tiramos partido da função **keyGen(seed)** do PKE, que devolve uma chave privada (f, f_q, h_q) e uma chave pública h.
Desta forma, apenas falta a geração do s, que é feita com a função **randomBitString(size)** que devolve uma sequência de bits aleatória.
Obtemos, assim, as chaves e o parâmetro s necessário para o desencapsulamento.

Encasulamento

O encapsulamento é feito com a chave pública do destinatário.
Os passos para o cálculo do encapsulamento da chave são os seguintes:

- começamos por gerar uma sequência aleatória de bits $coins$ com a função **randomBitString(size)**
- depois, geramos dois polinómios r e m, de forma aleatória, através da função **Sample_rm(coins)**
- usamos a função **encrypt(h)** do PKE para fazer a cifragem de (r, m), sendo que o resulatdo desta cifragem é o 'encapsulamento' da chave
- obtemos a chave simétrica k calculando o hash de (r, m)

Desencapsulamento

Para desencapsular, é necessário ter como parâmetros o encapsulamento da chave c e a chave privada (f, f_q, h_q). Esta função retorna a chave simétrica k (calculada anteriormente).

- Começamos, então, por fazer a decifragem do encapsulamento da chave c, através da função **decrypt(sk, c)**, definida no PKE, que devolve $(r, m, fail)$
- de seguida, obtemos a chave simétrica $k1$ calculando o hash de (r, m)
- é obtida ainda uma chave diferente $k2$ através do cálculo do hash de (s, c) para o caso de falha na decifra
- se fail for 0, então a chave $k1$ é devolvida, senão, é devolvida a chave $k2$

In [3]:

```
import random, hashlib
import numpy as np


class NTRU_KEM(object):

    def __init__(self, N=701, Q=4096, D=495, timeout=None):

        # Todas as inicializações de parâmetros são baseadas na submissao
        self.n = N
        self.q = Q
        self.d = D

        # inicializacao da instancia NTRU_PKE
        self.pke = NTRU_PKE(self.n, self.q, self.d)


    #função para calcular o hash (recebe dois polinomios)
    def Hash1(self, e0, e1):
        ee0 = reduce(lambda x,y: x + y.binary(), e0.list() , "")
        ee1 = reduce(lambda x,y: x + y.binary(), e1.list() , "")
        m = hashlib.sha3_256()
        m.update(ee0.encode())
        m.update(ee1.encode())
        return m.hexdigest()


    #função para calcular o hash (recebe uma string de bits e um polinomio
    def Hash2(self, e0, e1):

        ee1 = reduce(lambda x,y: x + y.binary(), e1.list() , "")
        m = hashlib.sha3_256()
        m.update(e0.encode())
        m.update(ee1.encode())
        return m.hexdigest()


    # Funcao usada para gerar o par de chaves pública e privada(acrescenta
    def keyGen(self, seed):
        keys = self.pke.keyGen(seed)

        s = ''.join([str(i) for i in self.pke.randomBitString(256)])

        return {'sk' : (keys['sk'][0],keys['sk'][1],keys['sk'][2],s) , 'pk


    # Funcao que serve para encapsular a chave que for acordada a partir d
    def encaps(self, h):
        coins = self.pke.randomBitString(256)

        (r,m) = self.pke.Sample_rm(self.pke.randomBitString(11200))

        c = self.pke.encrypt(h, (r,m))

        k = self.Hash1(r,m)

        return (c,k)

    # Funcao usada para desencapsular uma chave, a partir do seu "encapsul
    def desencaps(self, sk, c):
        (r,m,fail) = self.pke.decrypt((sk[0], sk[1], sk[2]), c)
```

```
        k1 = self.Hash1(r,m)

        k2 = self.Hash2(sk[3],c)

        if fail == 0:
            return k1
        else:
            return k2
```

In [4]:

```
# Parametros do NTRU (ntruhps4096821)
N=701
Q=4096
D=495

# Inicializacao da classe
ntru = NTRU_KEM(N,Q,D)

print("[Teste do encapsulamento e desencapsulamento]")
keys1 = ntru.keyGen(ntru.pke.randomBitString(2*D))

(c,k) = ntru.encaps(keys1['pk'])
print("Chave = " + k)

k1 = ntru.desencaps(keys1['sk'], c)
print("Chave = " + k1)

if k == k1:
    print("A chave desencapsulada é igual à resultante do encapsulamento!"
else:
    print("O desencapsulamento falhou.")
```

```
[Teste do encapsulamento e desencapsulamento]
Chave = 03bc15df06b652b7d51c11e6335f13f8354282ab1db59deacc719a0b3d7adcd5
Chave = 03bc15df06b652b7d51c11e6335f13f8354282ab1db59deacc719a0b3d7adcd5
A chave desencapsulada é igual à resultante do encapsulamento!
```