Trabalho Prático 2 - NTRU - Grupo 18

Objetivos do trabalho Prático:

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

De modo a fazer a melhor implementação possível desta técnica, a equipa decidiu seguir e guiar-se pela submissão do **NITRU** especificada no seu documento técnico, dado que esta já apresenta um **PKE-IND-CCA**.

Para iniciar a nossa classe, utilizamos valores obtidos através do NTRU-HPS (no nosso caso o default será **ntruhps4096821**), sendo ainda criados oa anéis que serão utilizados em todo o processo (Zx, R e Rq), não sendo necessários outros, dado que iremos sempre trabalhar com valores arredondados.

Posto isto chega então a altura de colocar as mãos na massa e desenvolver o PKE.

Gerar Chaves

Neste momento iremos gerar duas chaves através da função **geraChaves()**:

- **Chave pública :** h
- **Chave Privada :** (f, fq, hq)

Como argumento da função iremos possuir uma **seed**, que será uma string de bits e que será utilizada para gerar polinómios ternários f e g. É de notar que, uma vez que esta seed será dividida em dois para gerar os dois polinómios diferentes, ela deverá ser então suficientemente grande para tal.

Geramos estes polinómios através da função Sample_fg(seed), mediante **(f,g)** $<-$ **Sample_fg(seed)** da seguinte forma:

- Quando o bit é 0, passa a 1 como coeficiente, caso o bit seja 1 fica a -1 como coeficiente. Finalmente acrescenta-se ao resto do polinómio bits com o valor 0 e fazemos shuffle ao polinómio.

Ao fim de gerarmos os polinómios, iremos então determinar os elementos da chave privada (f, fq, hq) e os da chave pública (h). Neste processo iremos então realizar as operações determinadas pela submissão. É de relembrar que, por vezes, podemos não conseguir efetuar algumas inversas e por isso podem ter de ser criados novos polinómios f e g.

As operações efetuadas são as seguintes:

- fp $<-$ (1/f) mod(3,$\Phi(n)$)

- fq $< -$ (1/f) mod(q,$\Phi(n)$)
- h $< -$ (3.g.fq) mod(q,$\Phi(1)\Phi(n)$)
- hq $< -$ (1/f) mod(q,$\Phi(n)$)

Cifragem

A função de cifragem **cifra()** irá receber 3 argumentos:

- **h :** Valor da chave pública
- **r :** Parâmetro gerado de forma aleatória
- **m :** Mensagem a enviar

Neste momento necessitamos apenas de calcular o criptograma gerado através da expressão **c** $< -$ **(r.h + m) mod(q,$\Phi(n)$)** (É m e não m', dado que o NTRU-HPS diz que **Lift(m) = m' = m**)

Decifragem

A função de decifragem **decifra()** irá receber 2 argumentos:

- **(f,fp,hq) :** Chave privada
- **c :** Criptograma a decifrar

Seguidamenteteremos de efetuar as seguintes operações:

- a $< -$ (c.f) mod(q,$\Phi(1)\Phi(n)$);
- m $< -$ (a.fp) mod(3,$\Phi(n)$);
- r $< -$ ((c-m).hq) mod(q,$\Phi(n)$)
- Se os polinómios (r,m) nao forem ternários, retorna **(0,0,1)**, senão retorna **(r,m,0)**.

No treceito passo é m e não m', dado que, tal como dito anteriormente o NTRU-HPS diz que **Lift(m) = m' = m**);

In [ ]:

```
import random, hashlib
import numpy as np

# Baseado no esquema da Figura 9 na página 25 do documento: https://www.dropbox.com/

class NTRU_PKE(object):

    # Iniciação do objeto NTRU_PKE
    def __init__(self, N=821, Q=4096, D=495, timeout=None):

        # Inicialização de parâmetros baseada nas recomendações do ntruhps4096821 (P
        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 (BitString) aleatória com tamanho size
    def randomBitString(self, size):

        # Gera uma sequencia de n bits aleatorios
        u = [random.choice([0,1]) for i in range(size)]

        # Shuffle dos valores da lista, para aumentar ainda mais a aleatoriedade
        random.shuffle(u)

        return u



    # Verifica se um polinomio f é ternário (coeficientes são todos 1, -1 ou 0)
    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.
    # Neste caso, em vez de ser entre 0 e q-1, fica entre -q/2 e q/2-1
    # Por exemplo, com um q de 256, os valores ficariam entre -128 e 127
    def middle_mod(self, f, q):

        g = list (( (f[i] + q//2) % q) - q//2 for i in range(self.n))

        return self.Zx(g)



    # Produz o módulo da inversa de um polinomio f com o módulo x^n-1 modulo p, em q
    def invers_mod(self, f, p):

        T = self.Zx.change_ring(Integers(p)).quotient(x^self.n-1)

        return self.Zx(lift(1 / T(f)))



    #Produz o módulo da inversa de um polinomio f com o módulo x^n-1 modulo q, em qu
    def invers_mod2(self, f, q):
```

```
        assert q.is_power_of(2)

        g = self.invers_mod(f, 2)

        while True:
            r = self.middle_mod(self.R(g*f), q)

            if r == 1:
                return g

            g = self.middle_mod(self.R(g*(2 - r)), q)



    # Gera um polinómio ternário (coeficientes são todos 1, -1 ou 0)
    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 em Lf (no nosso caso, em T+) e um polinomio g em Lg (no no
    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 em Lr (no nosso caso, em T) e um polinomio m em Lm (no nos
    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)



    def geraChaves(self, seed):

        while True:
            try:
                # (f, g) <- Sample_fg(seed)
                (f,g) = self.Sample_fg(seed)

                # fq <- (1/f) mod (q;φn)
                fq = self.invers_mod2(f, self.q)

                # h <- (3.g.fq) mod(q;φ1 φn)
                h = self.middle_mod(3*self.R(g*fq), self.q)

                # hq <- (1/h) mod (q;φn)
                hq = self.invers_mod2(h, self.q)

                # fp <- (1/f) mod(3;φn)
                fp = self.invers_mod(f, 3)
                break

            except:
                # Para que a nova iteracao tenha uma nova seed
                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 cifra(self, h, rm):

        r = rm[0]
        m = rm[1]

        # c <- (r.h + m') mod (q, φ1 φn)
        c = self.middle_mod(self.R(h*r) + m, self.q)

        return c



    # Recebe como parametros a chave privada (f,fp,hq) e ainda o criptograma
    def decifra(self, sk, c):

        # a <- (c.f) mod(q,φ1φn)
        a = self.middle_mod(self.R(c*sk[0]), self.q)

        # m <- (a.fp) mod(3,φn)
        m = self.middle_mod(self.R(a * sk[1]), 3)
```

```
        # r <- ((c-m').hq) mod(q,φn)
        aux = (c-m) * sk[2]
        r = self.middle_mod(self.R(aux), self.q)

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

        return (r,m,0)
```

Testes PKE

In [ ]:

```
# Parametros do NTRU (ntruhps4096821)
N=821
Q=4096
D=495

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

print("Teste de cifragem e decifragem \033[1m[PKE]\033[0m\n")
keys = ntru.geraChaves(ntru.randomBitString(2*D))
rm = ntru.Sample_rm(ntru.randomBitString(11200))

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

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

if rmDec[0] == rm[0] and rmDec[1] == rm[1] and rmDec[2] == 0:
    print("\033[1mAs mensagens e os r's 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]

As mensagens e os r's são iguais!
 └> Processo completado com sucesso. Cifragem e Decifragem bem efetuadas.
```

NTRU - KEM

De modo a fazer a melhor implementação possível desta técnica, a equipa decidiu seguir e guiar-se pela submissão do **NITRU** especificada no seu documento técnico, dado que esta já apresenta um **KEM-IND-CPA**.

Para iniciar a nossa classe, utilizamos valores obtidos através do NTRU-HPS (no nosso caso o default será **ntruhps4096821**), sendo ainda criados oa anéis que serão utilizados em todo o processo (Zx, R e Rq), não sendo necessários outros, dado que iremos sempre trabalhar com valores arredondados.

Podemos perceber que iremos recorrer a funções usadas na inicialização do **PKE**, bem como das funções **geraChaves()**, **cifra()** e **decifra()** ao longo deste processo.

Posto isto chega então a altura de colocar as mãos na massa e desenvolver o KEM.

Gerar Chaves

Neste momento iremos gerar duas chaves através da função **geraChaves()**:

- **Chave pública :** h
- **Chave Privada :** (f, fq, hq, s)

Iremos utilizar a função de geração de chaves da classe NTRU_PKE para obter os parametros (f, fq, hq).

Posteriomente teremos ainda de determinar um **s**, que será feito seguindo a seguinte forma **s** $<-^{\$}$ **{0,1}**^{256} que basicamente fará a geração de bits aleatórios.

Encapsulamento e geração de Hash da Chave

A função de encapsulamento **encapsula()** irá receber 1 argumento:

- **h :** Valor da chave pública

Esta função irá retornar o **c** que é o encapsulamento da chave, e o **k** que é a chave em si;

Neste momento geramos ainda uma sequência aleatória de bits, denomidadas **coins** através da forma: **coins** $<-^{\$}$ **{0,1}**^{256}

Geramos agora dois polinómios ternários **r** e **m**, através da função Sample_rm(coins) da seguinte forma: **(r,m)** $<-$ **Sample_rm(coins)**

Passaremos agora á cifragem do (r,m) através da forma: **c** $<-$ **Encrypt(h,(r,m))**, sendo o **c** o encapsulamento.

Finalmente fazemos o Hash do **r** e do **m** de forma a obter a chae simétrica da seguinte forma: **k** $<-$ **H1(r,m)**

Desencapsulamento da chave

A função de desencapsulamento **desencapsula()** irá receber 2 argumentos:

- **(f,fp,hq) :** Chave privada
- **c :** Criptograma a decifrar

Iremos primeiramente fazer a decifragem do encapsulamento **c** atraves da função **decifra()** do NTRU_PKE através da seguinte forma: **(r,m,fail)** $<-$ **Decrypt((f,fp,hq),c)**. Não esquecer que esta função não recebe a chave privada contendo o **s**.

Fazemos agora o Hash do **r** e do **m** de forma a obter a chae simétrica da seguinte forma: **k1** $<-$ **H1(r,m)**

Seguindamente fazemos o Hash do **s** e do **c** de forma a obter uma segunda via caso a chave falhe da seguinte forma: **k2** $<-$ **H2(s,c)**

Finalmente, se fail = 0, então retornamos k1, senão retornamos k2.

In [ ]:

```
import random, hashlib
import numpy as np

# Baseado no esquema da Figura 10 na página 25 do documento: https://www.dropbox.com

class NTRU_KEM(object):

    def __init__(self, N=821, Q=4096, D=495, timeout=None):

        # Todas as inicializações de parâmetros são baseadas na submissao com os par
        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 HashPols(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 HashPolBs(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 ao geraCh
    def geraChaves(self, seed):

        # ((f,fp,hq),h) <- KeyGen'(seed)
        keys = self.pke.geraChaves(seed)

        # s <-$ {0,1}^256
        s = ''.join([str(i) for i in self.pke.randomBitString(256)])

        # return ((f,fp,hq,s),h)
        return {'sk' : (keys['sk'][0],keys['sk'][1],keys['sk'][2],s) , 'pk' : keys['


    # Funcao que serve para encapsular a chave que for acordada a partir de uma chav
    def encapsula(self, h):

        # coins <-$ {0,1}^256
        coins = self.pke.randomBitString(256)
```

```
        # (r,m) <- Sample_rm(coins)
        (r,m) = self.pke.Sample_rm(self.pke.randomBitString(11200))

        # c <- Encrypt(h,(r,m))
        c = self.pke.cifra(h, (r,m))

        # k <- H1(r,m)
        k = self.HashPols(r,m)

        return (c,k)



    # Funcao usada para desencapsular uma chave, a partir do seu "encapsulamento" e
    def desencapsula(self, sk, c):

        # (r,m,fail) <- Decrypt((f,fp,hq),c)
        (r,m,fail) = self.pke.decifra((sk[0], sk[1], sk[2]), c)

        # k1 <- H1(r,m)
        k1 = self.HashPols(r,m)

        # k2 <- H2(s, c)
        k2 = self.HashPolBs(sk[3],c)

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

Testes KEM

In [ ]:

```
# Parametros do NTRU (ntruhps4096821)
N=821
Q=4096
D=495

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

print("Teste do encapsulamento e desencapsulamento \033[1m[KEM]\033[0m\n")
keys1 = ntru.geraChaves(ntru.pke.randomBitString(2*D))

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

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

if k == k1:
    print("\n\033[1mAs chaves encapsulada e desencapsulada são iguais!\033[0m")
    print(" └> Processo completado com sucesso. Capsulamento e Desencapsulamento bem
else:
    print("O desencapsulamento falhou!!!")
```

```
Teste do encapsulamento e desencapsulamento [KEM]
```

```
Chave encapsulada    = b85618d367759e2e5ce349f4457a455697356bfd1e482abeb33712353596e
697
Chave desencapsulada = b85618d367759e2e5ce349f4457a455697356bfd1e482abeb33712353596e
697
```

As chaves encapsulada e desencapsulada são iguais!

```
 └> Processo completado com sucesso. Capsulamento e Desencapsulamento bem efetuados.
```