ntru

May 2, 2022

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

1 NTRU

[ ]:
```
#definição dos parametros que definem o algorito
n = 677
p = 3
q = next_prime(p*n)

iid_bits = 5408
fixed_type_bits = 25688


#Criação dos aneis identificados na documentação a serem usados
Z.<x> = ZZ[]
phi4 = x - 1
phi_n4 = (x^n - 1) / (x-1)
R = Z.quotient(phi4 * phi_n4)

S.<x> = PolynomialRing(GF(3))
S3 = QuotientRing(S, ((x^n - 1) / (x - 1)))


SS.<x> = PolynomialRing(GF(q))
Sq = QuotientRing(SS, (x^n - 1) / (x-1))

R.<x> = GF(q)[]
phi3 = x - 1
phi_n3 = (x^n - 1) / (x-1)
Rq = R.quotient(phi3 * phi_n3)

Zx.<x> = PolynomialRing(ZZ)
```

2 Funções auxiliares

[ ]:
```
#função que gera os polinomios f e g
# #definição detalhada no ponto 1.10.1 da documentação do NTRU

def sample_fg(seed):
    #f_bits tem o tamanho da amostra iid_bits
    f_bits = seed[:iid_bits]
    #g_bits tem o tamanho da amostra fixed_type_bits
    g_bits = seed[iid_bits: iid_bits + fixed_type_bits]

    #Geração dos polinomios
    f = ternary(f_bits)
    g = fixed_type(g_bits)
    return f, g

#definição detalhada no ponto 1.10.2 da documentação do NTRU
#função identica a sample_fg
def sample_rm(rm_bits):
    r_bits = seed[:iid_bits]
    m_bits = seed[iid_bits: iid_bits + fixed_type_bits]
    r = ternary(r_bits)
    m = fixed_type(m_bits)
    return r, m


#definição detalhada no ponto 1.10.3 da documentação do NTRU
#Retorna um polinomio ternário
def ternary(bits):
    v=0
    i=0
    while(i<n-1):
        somatorio=0
        for j in range (7):
            somatorio+=(2^j) *bits[8*i+j+1]
        v = v + somatorio * x^i
        i=i+1
    aa = S3(v)

    ss = aa.lift().map_coefficients(lambda c: c.lift_centered(), ZZ)

    v = ss
    return v

#definição detalhada no ponto 1.10.4 da documentação do NTRU
#Retorna um polinomio ternário com um numero de coeficientes
#de q/16-1 igual a 1 e outros tantos igual a -1
```

```
def fixed_type(bits):
    a = []
    for i in range(n - 1):
        a.append(0)
    v = 0
    i = 0
    somatorio = 0
    Zx.<x> = ZZ[]

    while i < (q//16)-1:
        somatorio = 0
        for j in range(29):
            somatorio += 2^(2+j) * bits[30*i+1+j]
        a[i] = 1 + somatorio
        i = i + 1

    while i < (q//8)-2:
        somatorio = 0
        for j in range(29):
            somatorio += 2^(2+j) * bits[30*i+1+j]
        a[i] = 2 + somatorio
        i = i + 1

    while i < n-1:
        somatorio = 0
        for j in range(29):
            somatorio += 2^(2+j) * bits[30*i+1+j]
        a[i] = 0 + somatorio
        i = i + 1

    a.sort()
    i = 0

    while i < n-1:
        v = v + (a[i] % 4)* x^i
        i = i + 1


    aa = S3(v)
    ss = aa.lift().map_coefficients(lambda c: c.lift_centered(), ZZ)
    v = ss
    return v


#função que recebe os polinomios f e g
#Retorna um tuplo de polinomios, sendo que o h satisfaz h*f=3*g
#e o hq satisfaz que h*hq=1
```

```
#Definição detalhada no ponto 1.11.2 da documentação NTRU
def DPKE_Public_Key(f,g):
    #p=3
    G = p * g
    v0=Sq(G*f)
    v1=(v0).inverse_of_unit()
    v1=v1.lift()
    h=Rq(v1*G*G)
    hq=Rq(v1*f*f)
    #aplica-se o arrendondamento do h por q para respeitar a condição
    return _round(h,q),hq




def _round(pol,r):              # análogo a _lift mas converte os coeficientes
                                # ao intervalo simétrico  com r elementos (r é␣
 ↪ímpar)
    u = map(lambda n: n%r, map(lift,pol.list()))
    rr = r//2
    return Zx([n if n <= rr else n - r  for n in u])
```

3 PKE

```
[ ]: #função que recebe como input uma string de bits com o tamanho
#sample_key_bits
#Retorna a chave pública e a chave privada
#Definição detalhada no ponto 1.11.1 da documentação NTRU
def DPKE_Key_Pair(seed):
    #geração dos polinomios f e g
    f,g=sample_fg(seed)

    #definição de fp como sendo o inverso de f com o ring S3
    fp=S3(f).inverse_of_unit()

    #geração dodos polinomios h e hq para a definição das chaves
    #publica e privada
    (h,hq)= DPKE_Public_Key(f,g)

    #chave privada sendo a concatenação de f,fp e hq
    #chave publica sendo o polinomio h
    priv = f, fp, hq
    pub = h
    return pub,priv

#função que recebe os polinomios r e m e a chave pública
```

```
#retorna uma array de bytes cifrado
#Definição detalhada no ponto 1.11.3 da documentação NTRU
def encrypt(r, h, m):
    rh = Rq(r) * h
    #definição do ciphertext com sendo Rq(r*h+m)
    c = rh + Rq(m)
    #realiza-se o arrendondamento do c por q
    b = _round(c,q)
    return b

#função que recebe a chave privada (f,fp,hq) e o array de bytes
#retorna uma array de bytes decifrado
#Definição detalhada no ponto 1.11.4 da documentação NTRU
def decrypt( f, fp, hq, c):
    Zx.<x>  = ZZ[]
    #definição de v1 como sendo Rq(c*f)
    v1 = _round((Rq(c) * Rq(f)),q)

    #definição de m como sendo S#(v1*fp)
    m = S3(v1 * fp)                                  # Caclular m em S/3
    m_s3 = _round(m,p)

    #definição de r como sendo Sq((c-m)*hq)
    r = (Sq(c) - Sq(m_s3)) * hq

    return _round(r,q), m_s3
```

4 TESTE PKE

[ ]:
```
#gera a seed
seed = os.urandom(fixed_type_bits + iid_bits )

#Gera a chave pública e privada
pub, priv = DPKE_Key_Pair(seed)

#Gerar outra seed diferente
rm_bits = os.urandom(fixed_type_bits + iid_bits)

#Gerar os polinómios r e m
r, m = sample_rm(rm_bits)

#cifragem
c = encrypt(r,pub, m)

#decifragem
f, fp, hq = priv
```

```
rr, pt = decrypt(f, fp, hq, c)

#Verificação
m == pt
```

[ ]: True

5 KEM

[ ]:
```
#Definição do Hash usando a cifra sha3_256
#retorna o hash dos polinomios r+m
def Hash1( r, m):
    r1 = r.list()
    m1 = m.list()
    m = hashlib.sha3_256()
    m.update(str(r1).encode())
    m.update(str(m1).encode())
    return m.hexdigest()
```

[ ]:
```
#Recebe como input uma string de bits
#Retorna a chave publica e privada
#Definição detalhada no ponto 1.12.1 da documentação NTRU
def generate_keys_kem(seed):
    #geração das chaves publica e privada usando
    #o algoritmo da PKE
    pub,priv=DPKE_Key_Pair(seed)

    #Concatenação de uma prf_key(gerada aleatoriamente com tamanho 256)
    #à chave privada
    prf_key=os.urandom(256)
    priv=priv+(prf_key,)

    return pub,priv

#Recebe como input a chave publica
#Retorna um array de bytes e a chave partilhada
#Definição detalhada no ponto 1.12.2 da documentação NTRU
def encapsulate(pub):
    #Geração de bytes para formar os polinomios r e m
    coins=os.urandom(256)
    (r,m)=sample_rm(coins)

    #aplicar o hash(sha3_256) a estes polinomios e obter a
    #chave partilhada
    shared_key=Hash1(r,m)
```

```
    #usar o encrypt da PKE para gerar o array de bytes
    ct=encrypt(r,pub,m)

    return ct,shared_key

#Recebe como input a chave privada
#Retorna a chave partilhada
#Definição detalhada no ponto 1.12.3 da documentação NTRU
def decapsulate( priv, c):
    #separação da chave privada nas suas componentes
    f, fp, hq, s = priv

    #aplicação do decrypt da PKE para obter os polinomios r e m
    r, m = decrypt(f, fp, hq, c)

    #aplicação da hash aos polinomios e uma vez que é simetrica
    #a chave gerada deve coincidir com a gerada na encapsulate
    shared_key = Hash1(r,m)

    return shared_key
```

6 TESTE KEM

[ ]:
```
seed = os.urandom(fixed_type_bits + iid_bits )

#Gerar a chave pública e privada
pub, priv = generate_keys_kem(seed)

#Encapsular
c, shared_key1 = encapsulate(pub)

#Desencapsular
shared_key= decapsulate(priv, c)

#Verificação
print(shared_key1 == shared_key)
```

```
True
```