NTRU

May 10, 2021

1 TP2 - Estruturas Criptográficas

1.1 Elementos do grupo 4:

- André Morais, A83899
- Tiago Magalhães, A84485

1.2 NTRU-PKE

[1]:
```
import os
import math
import random as rn
from sage.all import *
import sys
import hashlib



class NTRU:
    def __init__(self):                                      #Criação dos parâmetros
        self.n = 677
        self.p = 3
        self.q = next_prime(self.p*self.n)

        self.iid_bits = 5408
        self.fixed_type_bits = 25688


        Z.<w> = ZZ[]                                         #Criação dos anéis a␣
 ↪utilizar
        phi4 = w - 1
        phi_n4 = (w^self.n - 1) / (w-1)
        self.R = Z.quotient(phi4 * phi_n4)

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

```
        SS.<y> = PolynomialRing(GF(self.q))
        phi_n2 = (y^self.n - 1) / (y-1)
        self.Sq = QuotientRing(SS, phi_n2)

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


    def round_(self,t, n=3):                        #Função para arredondar␣
↪os polinómios entre -1 e 1
        if n==-1:
            n=self.q

        Zx.<x>  = ZZ[]

        r = n//2
        res_list = []
        pol_list = t.list()
        for p in pol_list:
            res_list.append(lift(p+r) - r)

        return Zx(res_list)


    def ternary(self, bits):                        #Função que retorna um␣
↪polinómio ternário com os
                                                    #coeficientes -1, 0 e 1
        v = 0
        i = 0
        while i < self.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 = self.S3(v)
        ss = aa.lift().map_coefficients(lambda c: c.lift_centered(), ZZ)

        v = ss
        return v

    def fixed_type(self, bits):                     #Função que retorna um␣
↪polinómio ternário com exatemente
```

```
        a = []                                          #q/16-1 coeficientes 1 e -1
        for i in range(self.n - 1):
            a.append(0)
        v = 0
        i = 0
        somatorio = 0
        Zx.<x> = ZZ[]

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


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

        count = 0
        c2 = 0
        for el in list(v):
            if el==-1:
                count+=1
```

```
            if el == 1:
                c2+=1

        return v

    def sample_fg(self, seed):                                  #Função que retorna os polinómio f e g
        f_bits = seed[:self.iid_bits]
        g_bits = seed[self.iid_bits: self.iid_bits + self.fixed_type_bits]
        f = self.ternary(f_bits)
        g = self.fixed_type(g_bits)
        return f, g



    def key_gen(self,seed):


        f, g = self.sample_fg(seed)                             # Gerar 2 polinómios ternários

        fq = self.Sq(f).inverse_of_unit()                       # Calcular 1/f em S/q

        h = 3 * self.Rq(g) * self.Rq(fq.lift())                 # Calcular 3 * g * f em R/q

        hq = self.Sq(h.lift())^-1                               # Cacular hq em S/q

        fp = self.S3(f).inverse_of_unit()                       # Calcular fp em S/3

        priv = f, fp, hq
        pub = h
        return pub, priv


    def sample_rm(self,rm_bits):                                #Função que retorna os polinómio r e m
        r_bits = seed[:self.iid_bits]
        r = self.ternary(r_bits)
        m_bits = seed[self.iid_bits: self.iid_bits + self.fixed_type_bits]
        m = self.fixed_type(m_bits)
        return r, m
```

```python
    def encrypt(self, r, h, m):                            # Calcular c em R/q␣
↪
        rh = self.Rq(r) * h
        c = rh + self.Rq(m)
        b = self.round_(c,n=-1)
        return b


    def decrypt(self, f, fp, hq, c):                        # Calcular a em R/q
        Zx.<x>  = ZZ[]
        a = self.round_(self.Rq(c) * self.Rq(f), n=-1)
        m = self.S3(a) * fp                                 # Caclular m em S/3
        m_s3 = self.round_(m)

        r = (self.Sq(c) - self.Sq(m.lift())) * hq

        return r, m_s3
```

1.3 TESTE PKE

[2]:
```python
ntru = NTRU()
#Gerar a seed
seed = os.urandom(ntru.fixed_type_bits + ntru.iid_bits )

#Gerar a chave pública e privada
pub, priv = ntru.key_gen(seed)

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

#Gerar polinómio r e m
r, m = ntru.sample_rm(rm_bits)

#Encrypt
c = ntru.encrypt(r,pub, m)
f, fp, hq = priv

#Decrypt
rr, pt = ntru.decrypt(f, fp, hq, c)

#Verificação
m == pt
```

[2]: True

1.4 NTRU-KEM

[3]:
```
class NTRU(NTRU):

    def hash_sha256(self, M):                                   #Função de Hash
        m = hashlib.sha256()
        m.update(M)
        return m.digest()

    def _toZ(self,f,p=None):                                    #Função que␣
↪arredonda
        ff = list(f)
        if p == None:
            return ff
        else:
            fp = map(lift,[Mod(a,p) for a in ff])
        return [u if u <= p//2 else u-p for u in fp ]

    def key_gen_kem(self, seed):                                #Gera chave␣
↪pública e privada
        pub, priv = self.key_gen(seed)
        s = os.urandom(256)
        priv = priv + (s,)
        return pub, priv

    def encaps(self, h):                                        #Encapsula Chave
        coins = os.urandom(256)
        r, m = self.sample_rm(coins)
        c = self.encrypt(r, h, m)
        r_bytes = str(self._toZ(c)).encode('utf-8')
        m_bytes = str(self._toZ(m)).encode('utf-8')
        rm_packed = r_bytes + m_bytes
        k = self.hash_sha256(rm_packed )
        return c, k

    def decaps(self, priv, c):                                 #Desencapsula␣
↪Chave
        f, fp, hq, s = priv
        r, m = self.decrypt(f, fp, hq, c)
        r_bytes = str(self._toZ(c)).encode('utf-8')
        m_bytes = str(self._toZ(m)).encode('utf-8')
        rm_packed = r_bytes + m_bytes
        k1 = self.hash_sha256(rm_packed)
        return k1
```

1.5 TESTE KEM

[4]:
```python
ntru = NTRU()
#Gerar a seed
seed = os.urandom(ntru.fixed_type_bits + ntru.iid_bits )
#Gerar a chave pública e privada
pub, priv = ntru.key_gen_kem(seed)
#Encapsular
c, k = ntru.encaps(pub)
#Desencapsular
kk = ntru.decaps(priv, c)
#Verificação
k == kk
```

[4]: True

[ ]:
```

```

[ ]:
```

```

[ ]:
```

```