TP2-Problema2-NTRU

May 2, 2022

1 TRABALHO PRÁTICO 2 - GRUPO 14

1.1 NTRU-PKE

Para a implementação do NTRU-PKE, o grupo decidiu seguir a submissão NTRU(https://ntru.org/f/ntru-20190330.pdf) que possuí um conjunto de passos para implementar as versões de NTRU em IND-CCA e IND-CPA seguro.

Primeiramente, são declarados os parâmetros definidos na versão ntruhps4096821, conforme o documento supracitado e são ainda criados os aneis necessários Zx, R, Rq, Sq e S3.

Imports

[1]:
```
import sys
import math
import random as rn
import hashlib
from pickle import dumps, loads
from cryptography.hazmat.primitives import hashes
```

[2]:
```
##valores que compõem NTRU versao ntruhps4096821 (https://ntru.org/f/
↪ntru-20190330.pdf)
N = 821 #N
p = 3
q = 4091 #Q utilização de um primo proximo de 4096

#criaçao dos aneis de polinómios (Pág.4)
Z.<w> = ZZ[]
phi = w-1
phi_n = (w^N -1) / (w-1)

PR = PolynomialRing(IntegerRing(),'x')
R = PR.quotient(phi * phi_n, 'x')

PRQ = PolynomialRing(GF(q),'x')
RQ = PRQ.quotient(phi * phi_n, 'x')

PS = PolynomialRing(IntegerRing(),'x')
```

```
S = PS.quotient(phi_n, 'x')

PS3 = PolynomialRing(GF(3),'x')
S3 = PS3.quotient(phi_n, 'x')

PSQ = PolynomialRing(GF(q),'x')
SQ = PSQ.quotient(phi_n, 'x')
```

De seguida, foram implementadas um conjunto de funções que permitem resolver o problema, das quais se destacam:

Gerar o par de chaves (função keypair())

1. A seed é uma *string* de bits aleatórios que serve para gerar os polinómios ternários f e g.
2. Os polinómios f e g são gerados com recurso à função `sample_fg`, cuja função é utilizar a seed e gerar dois polinómios ternários f e g.
3. Depois de gerados os polinómios f e g, pode-se então passar para a fase de cálculo dos elementos da chave privada *(f,fp,hq)* e da chave pública h. Nesta etapa, como são necessários alguns cálculos de inversas, utilizou-se um ciclo para que no caso de falha numa das inversas, o programa volte a iterar e gerar novos polinómios f e g válidos.

Função de cifragem (função encrypt())

1. A função `encrypt` recebe como *input* a chave pública h, um r e um m que será a mensagem a enviar. O r é um elemento aleatoriamente gerado.
2. Após isso, é calculado o criptograma através da expressão $c \leftarrow (r \cdot h + m0) mod(q, 1n)$.

Função de decifragem (função decrypt())

1. A função de decifra recebe como parâmetros a chave privada *(f,fp,hq)* e o criptograma c.
2. Depois, são efetuados um conjunto de cálculos que fazem uso da função `balancemod` para balancear os coeficientes das operações:
   - $a \leftarrow (c \cdot f) mod(q, 1n)$
   - $m \leftarrow (a \cdot fp) mod(3, n)$
   - $r \leftarrow ((c - m0) \cdot hq) mod(q, n)$
3. Por fim, retornam-se os valores do r, da mensagem m e da *flag* 0, em caso de sucesso.

[3]:
```
#funçao: gerar os polinomios f e g que serão utilizados
#        na geração do par de chaves, a partir de uma seed
def sample_fg(seed):
    assert len(seed) >= N*2
    f = Z(ternary(seed[:N]))
    g = Z(ternary(seed[N:]))
    return f,g

#funçao: gerar os polinomios r e m que serão utilizados
#        na fase de encapsulamento
def sample_rm(seed):
    assert len(seed) >= N*2
    r = Z(ternary(seed[:N]))
```

```
    m = Z(ternary(seed[N:]))
    return r,m

#funçao: gerar um polinómio ternário de grau N-2
def ternary(seed):
    assert len(seed) >= N
    t = [(x % 3) -1 for x in seed[:N-1]]
    return t

#funçao: mylift responsável por converter o polinomio para o anel de inteiros
def mylift(f):

    return Z([int(y) for y in f])

#funçao: permite balancear os coeficientes de um polinomio
#        neste caso é pretendido que se compreendam entre -q/2 e q/2
def balancemod(f,q):

    g = [((f[i] + q//2) % q) - q // 2 for i in range(N)]
    return Z(g)

#funçao: gerar o par de chaves publica e privada
def keypair(seed):
    while True:
        try:
            f, g = sample_fg(seed)
            #2º Passo :  fp <- (1/f) mod(3;n)
            fp = balancemod(mylift(1/S3(f)),3)
            #3º Passo: fq <- (1/f) mod (q;n)
            fq = balancemod(mylift(1/SQ(f)),q)
            #4º Passo: h <- (3.g.fq) mod(q;1n)
            h = balancemod(3 * mylift(R(g*fq)),q)
            #5º Passo: hq <- (1/h) mod (q;n)
            hq = balancemod(mylift(1/SQ(h)), q)
            break
        except:
            pass
    #retornar os tuplos de chave privada + publica
    return ((f,fp,hq),h)

#funçao: recebe como inputs a chave publica + o tuplo r,m
def encrypt(h, rm):
    r, m = rm
    # 2/3º passo: c <- (r.h + m') mod (q,1n) // return c
    return balancemod(R(r*h + m), q)

#funçao: recebe como parametros o tuplo de chave privada + o criptograma
```

```
def decrypt(sk,c):

    (f,fp,hq) = sk
    # 2º Passo: a <- (c.f) mod(q,1n)
    a = balancemod(mylift(RQ(c*f)),q)
    # 3º Passo:  m <- (a.fp) mod(3,n)
    m = balancemod(mylift(S3(a*fp)),3)
    # 4º Passo: r <- ((c-m').hq) mod(q,n)
    r = balancemod(mylift(SQ((c-m) * hq)),q)
    return (r, m , 0)
```

Cenário de teste De seguida apresentamos um cenário de teste para a implementação produzida, onde geramos o par de chaves, a mensagem a m ser cifrada e comparamos a decifra, bem como os valores de r.

[4]:
```
#Cenario de teste - NTRU PKE
#Gerar o par de chaves
seed_fg = os.urandom(N*2)
(sk,pk) = keypair(seed_fg)


seed_rm = os.urandom(N*2)
r,m = sample_rm(seed_rm)

c = encrypt(pk, (r,m))
(r1,plain,flag) = decrypt(sk,c)
#print('M: ',m)
#print('Plain: ',plain)


if r == r1 and plain == m and flag == 0:
    print(m == plain)
    print("As mensagens e os r's são identicos!!")
else:
    print("Algo correu mal!")
```

```
True
As mensagens e os r's são identicos!!
```

1.2 NTRU-KEM

Para esta segunda parte da implementação, era proposto o desenvolvimento de um KEM-IND-CPA seguro. Dentro da implementação do NTRU-KEM serão reaproveitadas funções definidas no exemplo anterior NTRU-PKE e os paramêtros da versão NTRU declarados inicialmente.

Gerar o par de chaves (função keygen())

1. Para a geração do par de chaves pública e privada, utilizou-se a função definida no exemplo acima, `keypair` que nos permite obter o tuplo secreto *(f,fq,hq)* e o parâmetro da chave pública *h*.
2. De seguida, fica a faltar a geração do parâmetro *s* que é gerado aleatoriamente.

Encapsulamento e geração de chave (função encapsulate())

1. A função `encapsulate` recebe como *input* a chave pública *h* e retorna o par *(c,k)*, onde *c* é o encapsulamento da chave e o *k* a chave em si.
2. Segue-se uma sequência de bits que é gerada aleatoriamente.
3. De seguida, procede-se à cifragem do *(r,m)* através da função `encrypt` declarada no exemplo anterior, sendo este o encapsulamento da chave.
4. Por fim, é feito o hash do *r* e *m* para obter a chave simétrica *k*.

Desencapsulamento da chave (função decapsulate())

1. A função `decapsulate` recebe como parâmetros a chave secreta e o encapsulamento da chave e retorna a chave simétrica *k*.
2. Primeiramente, é efetuada a decifragem de acordo com a função declarada no NTRU-PKE, `decrypt` do encapsulamento da chave *c* juntamente com os parâmetros *(f,fq,hq)* da chave secreta.
3. Procede-se ao cálculo do hash de *r* e *m* para obter a chave simétrica *k1*.
4. De seguida, o cálculo do hash de *s*, o elemento aleatório e *c* o encapsulamento da chave, para o caso do desencapsulamento falhar.
5. Se a operação falhar, retorna *k2* e em caso de sucesso é retornado *k1*.

[5]:
```
#funçao: funçao de hash utilizada no processo de
#        encapsulamento e desenc.
#        recorre ao shake256

def H(a,b):
    m = hashes.Hash(hashes.SHAKE256(int(256)))
    m.update(dumps(a) + dumps(b))
    return m.finalize()

#funçao: funçao de hash utilizada no processo de
#        desencapsulamento.
#        recorre ao shake256
def H1(a,b):
    m = hashes.Hash(hashes.SHAKE256(int(256)))
    m.update(a + dumps(b))
    return m.finalize()

#funçao: gera o par de chaves recorrendo a uma seed
#        reaproveita a funçao keypair definida no NTRU-PKE
#        retorna os tuplos de chave privada + publica
def keygen(seed):
    #1º Passo: ((f,fp),h) <- KeypairPKE()
```

```
    sk,pk = keypair(seed)
    (f,fp,hq) = sk
    # 2º Passo: s <-$ {0,1}^256
    s = os.urandom(32)
    return ((f,fp,hq,s),pk)

#funçao: gerar e encapsular a chave que for acordada a partir
#        de uma chave pública
#        Output: criptograma + chave
def encapsulate(h):
    seed = os.urandom(N*2)
    # 2º Passo: # (r,m) <- Sample_rm(seed)
    (r,m) = sample_rm(seed)
    # 3º Passo:  c <- Encrypt(h,(r,m))
    c = encrypt(h, (r,m))
    # 4º Passo: # k <- H1(r,m)
    k = H(r,m)
    #k = hash_shake256(r,m)
    return (c,k)

#funçao: desencapsular uma chave, a partir da chave privada
def decapsulate(sk,c):

    (f,fp,hq,s) = sk
    (r, m, fail) = decrypt((f,fp,hq),c)
    k1 = H(r,m)
    k2 = H(s,c)
    # if fail = 0 return k1 else return k2
    if fail == 0:
        return k1
    else:
        return k2
```

Cenário de teste De seguida apresentamos um cenário de teste para a implementação do KEM efetuada, onde geramos o par de chaves e fazemos o encapsulamento da chave que no final será comparada com o resultado obtido do desencapsulamento do criptograma.

[6]:
```
#Cenario de teste - NTRU KEM
#Gerar o par de chaves

seed = os.urandom(N*2)
sk,pk = keygen(seed)

c,k = encapsulate(pk)
```

```
k2 = decapsulate(sk,c)

if k == k2:
    print(k == k2)
    print("As chaves são iguais!")
else:
    print("Algo correu mal!")
```

```
True
As chaves são iguais!
```