NTRUPrime_KEM

May 23, 2020

1 TP2 : Implementação do esquema KEM - NTRU-Prime

1.0.1 Gerar Parâmetros

Foi necessário gerar os parâmetros p, q e w sendo p e q primos e w é um inteiro positivo. A nossa classe recebe este w e gera os outros a partir deste. Os parâmetros têm de satisfazer: * 2p > 3w * q > 16w + 1 * x^p -x -1 é irredutível no anel dos polinómios Z/q[x]

1.0.2 Key Gen

- Foi necessário gerar um elemento pequeno do anel R - g
  - Repetiu-se o processo que gera g até ser invertível em R/3
- Foi gerado um elemento aleatório f pequeno de peso w em R
- Calculamos o quociente de g por 3* f em R/q, obtendo a Public Key
- Como sk, criamos o tuplo, f em R e a inversa de g em R/3

1.0.3 Encapsulation

- Geramos um elemento aleatório r pequeno com peso w em R e depois foi calculado o Hash deste r
- Com este Hash obtemos a confirmation C e a sessionkey K
- Calculamos h*r, sendo h a public key e fazemos o round_3 do obtido
- Retornamos por fim a nossa confirmation C e o nosso ciph

1.0.4 Desencapsulation

- Recebemos o cstr e reparamos este no tuplo respetivo C, ciph
- Através da sk é possível ter acesso a f e à inversa de g
- Posto isto, é necessário multiplicar 3f in R/q e por 1/g in R/3
- Depois, arrendondamos os coeficientes como inteiros entre -(q-1)/2 e (q-1)/2 e reduzir módulo 3, obtemos o polinómio em R/3
- Calculamos a Hash deste t obtido
- Com a operação seguinte ficamos com R3_to_small deste t
- Convertemos ainda aquilo que obtemos no passo acima para um polinómio pequeno r' em R.
- Fizemos o compute como no encapsulation
- Por fim, verificamos que os peso e o obtido é igual.
- Retornamos K1, caso contrário retornamos Falso.

[50]:
```
import hashlib
import random as rn
from random import choice, randint

class NTRU_Prime():

    def __init__(self,w):
        #único parâmetro que inicializa a classe e tem de ser um inteiro␣
↪positivo
        self.w=w

        #q > 16w + 1;
        q=17*self.w
        while True:
            if (1+q).is_prime():
                break
            else:
                q += 1
        q=q+1

        Zx.<x>  = ZZ[]
        Zq.<z>  = PolynomialRing(GF(q))

        # 2p >= 3w
        p = next_prime(2*self.w)
        while True:
            if  Zq(x^p-x-1).is_irreducible():
                break
            else:
                p = next_prime(p+1)
        self.p=p
        self.q=q

    def small_poly(self,p,t=None):
        """
        polinómios cujos coeficientes são -1, 0, 1
        """
        Zx.<x>  = ZZ[]
        if not t:
            return Zx([choice([-1,0,1]) for k in range(p)])

        u = [rn.choice([-1,1]) for i in range(t)] + [0]*(p-t)
        rn.shuffle(u)
        return Zx(u)

    def Hash(self,m): #função para calcular o hash de um objecto
        ww = reduce(lambda x,y: x + y.binary(), m.list() , "")
```

```
        return hashlib.sha512(ww.encode('utf-8')).hexdigest()

    def round_3(self,t):
        Zx.<x>  = ZZ[]
        def f(x):
            return ((x/3).round())*3
        r = self.q//2
        t1 = list(map(lambda x: f(lift(x+r) - r) , t.list()))
        return  Zx(t1)

    def round_(self,t,n=-112321):
        if n==-112321:
            n=self.q
        Zx.<x>  = ZZ[]
        """
            input:  polinómio em Gqr ou Z3r
            output: transpõe os coeficientes para o intervalo -n//2..+n//2
        """
        r = n//2
        t1 = list(map(lambda x: lift(x + r) - r , t.list()))
        return Zx(t1)



    def keygen(self):
        Zx.<x>  = ZZ[]
        Z3.<y>  = PolynomialRing(GF(3))
        Zq.<z>  = PolynomialRing(GF(self.q))
        R.<x> = Zx.quotient(x^self.p-x-1)
        R3.<y> = Z3.quotient(y^self.p-y-1)
        Rq.<z> = Zq.quotient(z^self.p-z-1)

        g = self.small_poly(self.p)
        # enquanto R3(g) não for invertível, geramos novo g.
        while not R3(g).is_unit():
            g = self.small_poly(self.p)

        inv_g = R3(g)^(-1)
        f = self.small_poly(self.p,self.w)

        self.sk = (f , inv_g)
        self.pk = Rq(g)/Rq(3*f)

        return self.pk, self.sk


    def weights(self, vec):
```

```
        c = 0
        for i in range(len(vec)):
            if (vec[i] != 0):
                c = c+1
        return c

    def R3_to_small(self,inp):
        def f(u):
            return u if u < 2 else -1
        return [f(u) for u in inp]


    def encapsulation(self):
        Zx.<x>  = ZZ[]
        Z3.<y>  = PolynomialRing(GF(3))
        Zq.<z>  = PolynomialRing(GF(self.q))
        R.<x> = Zx.quotient(x^self.p-x-1)
        R3.<y> = Z3.quotient(y^self.p-y-1)
        Rq.<z> = Zq.quotient(z^self.p-z-1)
        r = self.small_poly(self.p,self.w)
        key = self.Hash(r)
        r_ = self.R3_to_small(r)
        print(r_)
        # confirmation
        C = key[:32]
        # session
        K = key[32:]
        ciph   = self.round_3(Rq(r)*self.pk)
        return C,ciph

    def desencapsulation(self,cstr):
        Zx.<x>  = ZZ[]
        Z3.<y>  = PolynomialRing(GF(3))
        Zq.<z>  = PolynomialRing(GF(self.q))
        R.<x> = Zx.quotient(x^self.p-x-1)
        R3.<y> = Z3.quotient(y^self.p-y-1)
        Rq.<z> = Zq.quotient(z^self.p-z-1)
        C,ciph = cstr
        (f , inv_g) = self.sk

        #Multiplicar 3f in R/q e por 1/g in R/3
        e = inv_g * R3(self.round_(Rq(3*f) * Rq(ciph))) ;
        t = self.round_(e,n=3);

        key = self.Hash(t)
        r_ = self.R3_to_small(t)
        print(r_)
```

```
        # confirmation
        C1 = key[:32]
        # session
        K1 = key[32:]
        ciph1 = self.round_3(Rq(t)*self.pk)

        if self.weights(r_) == self.w or not all([a in [-1,0,1] for a in r_]):
            print("True")
        else: return False
        if C1 == C :
            print("True")
        else: return False
        if ciph1 == ciph :
            print("True")
        else: return False

        return K1
```

[51]:
```
A=NTRU_Prime(8)
```

[52]:
```
A.p,A.q,A.w
```

[52]:
```
(137, 137, 8)
```

[53]:
```
pk, sk = A.keygen()
```

[54]:
```
test = A.encapsulation()
```

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
-1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1]
```

[55]:
```
A.desencapsulation(test)
```

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
-1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, -1]
True
True
True
```

[55]:
```
'e754da312c4c6f680920ea7379c4074abe72b56cce82f51e9e9952f48f1c40639f82d5d0ee23ffb
5eed8a8904256a467'
```

[ ]:

[ ]: