Pedro Gonçalves (A82313) & Roberto Cachada (A81012)

Os algoritmos/notação implementados/utilizados neste *notebook* são baseados no documento (https://ntruprime.cr.yp.to/nist/ntruprime-20190330.pdf) da segunda ronda da candidatura ao concurso da NIST.

PKE-IND-CCA

A camada do **Streamlined NTRU Prime** que implementa um PKE determinista é a camada interna, denominada **Streamlined NTRU Prime Core**.

A implementação abaixo, é baseada no software disponível no site (https://ntruprime.cr.yp.to/ntruprime.sage) do NTRU.

In [ ]:

```
import sys
import hashlib

def sha512(s):
    h = hashlib.sha512()
    h.update(s.encode('utf-8'))
    return h.digest()

#Parámetros sntrup761
(p,q,w) = (Integer(761) ,Integer(4591) ,Integer(286))
#Verificação Parámetros
assert p.is_prime()
assert q.is_prime()
assert w > Integer(0)
assert Integer(2)*p >= Integer(3)*w
assert q >= Integer(16)*w+Integer(1)

#-------

F3 = GF( Integer(3) )
def ZZ_fromF3(c):
    assert c in F3
    return ZZ(c+Integer(1))-Integer(1)

Fq = GF(q)
q12 = ZZ((q-Integer(1))/Integer(2) )
def ZZ_fromFq(c):
    assert c in Fq
    return ZZ(c+q12)-q12

#------- Anel polinomial Z

Zx = ZZ['x']; (x,) = Zx._first_ngens(1)
R = Zx.quotient(x**p-x-Integer(1) , names=('xp',)); (xp,) = R._first_ngens(1)

def Weightw_is(r):
    assert r in R
    return w == len([i for i in range(p) if r[i] != Integer(0) ])

def Small_is(r):
    assert r in R
    return all(abs(r[i]) <= Integer(1)  for i in range(p))

def Short_is(r):
    return Small_is(r) and Weightw_is(r)

#------- Anel polinomial Z mod 3

F3x = F3['x3']; (x3,) = F3x._first_ngens(1)
R3 = F3x.quotient(x**p-x-Integer(1) , names=('x3p',)); (x3p,) = R3._first_ngens(
1)

def R_fromR3(r):
    assert r in R3
    return R([ZZ_fromF3(r[i]) for i in range(p)])

def R3_fromR(r):
    assert r in R
    return R3([r[i] for i in range(p)])
```

```
# ----- Anel polinomial Z mod q

Fqx = Fq['xq']; (xq,) = Fqx._first_ngens(1)
Rq = Fqx.quotient(x**p-x-Integer(1) , names=('xqp',)); (xqp,) = Rq._first_ngens(
1)
assert (xq**p-xq-Integer(1) ).is_irreducible() #A verificação final dos parâmetr
os (Verifica se o polinómio é irredutível em Z/q)

def R_fromRq(r):
    assert r in Rq
    return R([ZZ_fromFq(r[i]) for i in range(p)])

def Rq_fromR(r):
    assert r in R
    return Rq([r[i] for i in range(p)])

# ----- Arredondamento de polonomiais mod q

def Rounded_is(r):
    assert r in R
    return (all(r[i]%Integer(3)  == Integer(0)  for i in range(p))
        and all(r[i] >= -q12 for i in range(p))
        and all(r[i] <= q12 for i in range(p)))

def Round(a):
    assert a in Rq
    c = R_fromRq(a)
    r = [Integer(3) *round(c[i]/Integer(3) ) for i in range(p)]
    assert all(abs(r[i]-c[i]) <= Integer(1)  for i in range(p))
    r = R(r)
    assert Rounded_is(r)
    return r

# ----- Geração de Short

def Short_fromlist(L): # L is list of p uint32
    L = [L[i]&-Integer(2)  for i in range(w)] + [(L[i]&-Integer(3) )|Integer(1)
for i in range(w,p)]
    assert all(L[i]%Integer(2)  == Integer(0)  for i in range(w))
    assert all(L[i]%Integer(4)  == Integer(1)  for i in range(w,p))
    L.sort()
    L = [(L[i]%Integer(4) )-Integer(1)  for i in range(p)]
    assert all(abs(L[i]) <= Integer(1)  for i in range(p))
    assert sum(abs(L[i]) for i in range(p)) == w
    r = R(L)
    assert Short_is(r)
    return r

def random8():
    return randrange(Integer(256) )

def urandom32():
    c0 = random8()
    c1 = random8()
    c2 = random8()
    c3 = random8()
    return c0 +Integer(256) *c1 + Integer(65536) *c2 + Integer(16777216) *c3

def Short_random():
    L = [urandom32() for i in range(p)]
```

```
    return Short_fromlist(L)

def randomrange3():
    return ((urandom32() & Integer(0x3fffffff) ) * Integer(3) ) >> Integer(30)

def Small_random():
    r = R([randomrange3()-Integer(1)  for i in range(p)])
    assert Small_is(r)
    return r
```

In [ ]:

```
def KeyGen():
    while (True):
        g = Small_random() #gera um elemento g pertencente a R
        if R3_fromR(g).is_unit():
            break #Verifica se g é invertível em R/3, se não for é escolhido out
ro g
    f = Short_random() #Gerado um elemento f de Short(set of small weight-w elem
ents of R)
    h = Rq_fromR(g)/Rq_fromR(Integer(3) *f)
    return h,(f,Integer(1) /R3_fromR(g)) #Chave pública/Chave Privada

#Recebe um input e a chave pública e retorna um ciphertext (Input × PublicKey ->
Ciphertext)
def Encrypt(r,h):
    assert Short_is(r) #Verifica se r pertence ao set Short
    assert h in Rq #Verifica se h pertence ao anel R/q
    return Round(h*Rq_fromR(r))

#Recebe um ciphertext e a chave privada e retorna o input original (Ciphertext ×
SecretKey -> Input)
def Decrypt(c,k):
    f,v = k
    assert Rounded_is(c) #Verifica se c pertence ao Ciphertext/Rounded
    assert Short_is(f) #Verifica se f pertence ao Short
    assert v in R3 #Verifica se v pertence a R/3
    e = R3_fromR(R_fromRq(Integer(3) *Rq_fromR(f)*Rq_fromR(c))) #Calcula 3fc ∈
 R/3
    r = R_fromR3(e*v) #Multiplica e por v em R/3
    if Weightw_is(r):
        return r
    return R([Integer(1) ]*w+[Integer(0) ]*(p-w))
```

In [ ]:

```
h,j = KeyGen()
r = Short_random()
ct = Encrypt(r,h)
rr = Decrypt(ct,j)
print(r==rr)
```

KEM-IND-CPA

Já a camada do **Streamlined NTRU Prime** que implementa um KEM é a camada externa, denominada **Streamlined NTRU Prime**.

A implementação abaixo, é baseada no software disponível no site (https://ntruprime.cr.yp.to/sntrup4591761.sage) do NTRU.

In [ ]:

```
p = 761; q61 = 765; q = 6*q61+1; t = 143
Zx.<x> = ZZ[]; R.<xp> = Zx.quotient(x^p-x-1)
Fq = GF(q); Fqx.<xq> = Fq[]; Rq.<xqp> = Fqx.quotient(x^p-x-1)
F3 = GF(3); F3x.<x3> = F3[]; R3.<x3p> = F3x.quotient(x^p-x-1)

def random32():
    return randrange(-2^31,2^31)
def random32even():
    return random32() & (-2)
def random321mod4():
    return (random32() & (-3)) | 1
def randomrange3():
    return ((random32() & 0x3fffffff) * 3) >> 30

import itertools

def concat(lists):
    return list(itertools.chain.from_iterable(lists))

def nicelift(u):
    return lift(u + q//2) - q//2

def nicemod3(u): # r in {0,1,-1} with u-r in {...,-3,0,3,...}
    return u - 3*round(u/3)

def int2str(u,bytes):
    return ''.join(chr((u//256^i)%256) for i in range(bytes))

def str2int(s):
    return sum(ord(s[i])*256^i for i in range(len(s)))

def seq2str(u,radix,batch,bytes): # radix^batch <= 256^bytes
    return ''.join(int2str(sum(u[i+t]*radix^t for t in range(batch)),bytes)
         for i in range(0,len(u),batch))

def str2seq(s,radix,batch,bytes):
    u = [str2int(s[i:i+bytes]) for i in range(0,len(s),bytes)]
    return concat([(u[i]//radix^j)%radix for j in range(batch)] for i in range(l
en(u)))

def encodeZx(m): # assumes coefficients in range {-1,0,1}
    m = [m[i]+1 for i in range(p)] + [0]*(-p % 4)
    return seq2str(m,4,4,1)

def decodeZx(mstr):
    m = str2seq(mstr,4,4,1)
    return Zx([m[i]-1 for i in range(p)])

def encodeRq(h):
    h = [q//2 + nicelift(h[i]) for i in range(p)] + [0]*(-p % 5)
    return seq2str(h,6144,5,8)[:1218]

def decodeRq(hstr):
    h = str2seq(hstr,6144,5,8)
    if max(h) >= q: raise Exception("pk out of range")
    return Rq([h[i]-q//2 for i in range(p)])

def encoderoundedRq(c):
    c = [q61 + nicelift(c[i]/3) for i in range(p)] + [0]*(-p % 6)
```

```
    return seq2str(c,1536,3,4)[:1015]

def decoderoundedRq(cstr):
    c = str2seq(cstr,1536,3,4)
    if max(c) > q61*2: raise Exception("c out of range")
    return 3*Rq([c[i]-q61 for i in range(p)])

def randomR(): # R element with 2t coeffs +-1
    L = [random32even() for i in range(2*t)]
    L += [random321mod4() for i in range(p-2*t)]
    L.sort()
    L = [(L[i]%4)-1 for i in range(p)]
    return Zx(L)
```

In [ ]:

```
# Função keygen equivalente ao keygen do Streamlined NTRU Prime Core
def keygen():
    while True:
        g = Zx([randomrange3()-1 for i in range(p)])
        if R3(g).is_unit(): break
    grecip = [nicemod3(lift(gri)) for gri in list(1/R3(g))]
    f = randomR()
    h = Rq(g)/(3*Rq(f))
    pk = encodeRq(h)
    return pk,encodeZx(f) + encodeZx(grecip) + pk

#Utiliza o keygen do Streamlined NTRU Prime Core para gerar as chaves
def KEM_KeyGen():
    pk,sk = keygen()
    sk += pk
    return pk,sk

def encapsulate(pk):
    h = decodeRq(pk) #Faz decode à chave pública
    r = randomR() # Gera valor aleatório perntencente a R
    hr = h * Rq(r) # Calcula h*r em R
    m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)])
    c = Rq(m) + hr
    fullkey = hash(encodeZx(r))
    return fullkey[:32] + encoderoundedRq(c),fullkey[32:]

def decapsulate(cstr,sk):
    #Decode cstr e sk
    f,ginv,h = decodeZx(sk[:191]),decodeZx(sk[191:382]),decodeRq(sk[382:])
    confirm,c = cstr[:32],decoderoundedRq(cstr[32:])
    f3mgr = Rq(3*f) * c
    f3mgr = [nicelift(f3mgr[i]) for i in range(p)]
    r = R3(ginv) * R3(f3mgr)
    r = Zx([nicemod3(lift(r[i])) for i in range(p)])
    hr = h * Rq(r)
    m = Zx([-nicemod3(nicelift(hr[i])) for i in range(p)])
    checkc = Rq(m) + hr
    fullkey = hash(encodeZx(r))
    if sum(r[i]==0 for i in range(p)) != p-2*t:
        return False
    if checkc != c:
        return False
    if fullkey[:32] != confirm:
        return False
    return fullkey[32:]
```

In [ ]:

```
for keys in range(5):
    pk,sk = KEM_KeyGen()
    for ciphertexts in range(5):
        c,k = encapsulate(pk)
        assert decapsulate(c,sk) == k
```