NTRU

May 10, 2021

1 TP 2 - Implementação NTRU

Este notebook tem a implementação do nosso NTRU-DPKE OW-CPA, NTRU-KEM IND-CCA2 e NTRU-PKE-CCA usando a TFO (NTRU-HPS). #### Variáveis de Instância Abaixo temos as variáveis de instância e constantes globais para os algoritmos implementados. Quanto aos parâmetros para o NTRU-HPS nós temos apenas uma versão recomendada pelas especificões mas iremos listar as suas condições. É aconselhado que para a leitura deste documento também se tenha a especificação do NTRU de 30-03-2019 à mão.

[1]:
```
# n é um primo e tanto 2 como 3 são de ordem n - 1 em (ZZ/n)^x
n = 509
# p é 3
p = 3
# q é uma potência de dois
q = 2048

# Lf e L3 correspondem a T ou ao conjunto _S3 (representação canónica de S3,␣
 ↪não confundir com o nome da variável _S3)
# Lg e Lm é o subconjunto de T com d/2 coeficientes iguais a -1 e d/2␣
 ↪coeficientes iguais a 1

# Lift é uma função, no caso do HPS o Lift de p é igual a _S3(p)

# Aneis de polinómios
_QR.<w> = GF(q)[]
_Rq.<w> = QuotientRing(_QR, _QR.ideal(w^n - 1))

_QR3.<w> = GF(3)[]
_R3.<w> = QuotientRing(_QR3, _QR3.ideal(w^n - 1))

_QS.<w> = GF(q)[]
_Sq.<w> = QuotientRing(_QS, _QS.ideal((w^n - 1)/(w - 1)))

_QS2.<w> = GF(2)[]
_S2.<w> = QuotientRing(_QS2, _QS2.ideal((w^n - 1)/(w - 1)))

_QS3.<w> = GF(3)[]
_S3.<w> = QuotientRing(_QS3, _QS3.ideal((w^n - 1)/(w - 1)))
```

```
# Aqui temos as constantes globais e as suas condições
sample_iid_bits = 8 * (n - 1)
sample_fixed_type_bits = 30 * (n - 1)
sample_key_bits = sample_iid_bits + sample_fixed_type_bits
sample_plaintext_bits = sample_iid_bits + sample_fixed_type_bits
prf_key_bits = 256

packed_s3_bytes = math.ceil((n - 1)/ 5)
packed_sq_bytes = math.ceil((n - 1) * (log(q,2))/8)
packed_rq0_bytes = math.ceil((n - 1) * (log(q,2))/8)

dpke_public_key_bytes = packed_rq0_bytes
dpke_private_key_bytes = (2 * packed_s3_bytes) + packed_sq_bytes
dpke_plaintext_bytes = 2 * packed_s3_bytes
dpke_ciphertext_bytes = packed_rq0_bytes

kem_public_key_bytes = dpke_public_key_bytes
kem_private_key_bytes = dpke_private_key_bytes + math.ceil(prf_key_bits / 8)
kem_ciphertext_bytes = dpke_ciphertext_bytes
kem_shared_key_bits = 256
```

1.1 KEM IND-CCA2

Abaixo temos a implementação documentada do KEM "fortemente seguro" que está descrito nas especificações.

[2]:
```
# 1.12 Strongly Secure KEM
# 1.12.1 Key_Pair
# b seed -> (B packed_private_key, B packed_public_key)
def key_pair(seed):
    # Passo 1 seed = fg_bits || prf_key
    fg_bits = seed[:sample_key_bits]
    prf_key = seed[-prf_key_bits:]

    # Passo 2 (packed_dpke_private_key, packed_public_key) =␣
 ↪DPKE_Key_Pair(fg_bits)
    (packed_dpke_private_key, packed_public_key) = dpke_key_pair(fg_bits)

    # Passo 3 packed_private_key = packed_dpke_private_key ||␣
 ↪bits_to_bytes(prf_key)
    packed_private_key = packed_dpke_private_key + bits_to_bytes(prf_key)

    # Passo 4 Output (packed_private_key, packed_public_key)
    return (packed_private_key, packed_public_key)

# 1.12.2.1 Encapsulate
```

```
# B packed_public_key -> (b shared_key, B packed_ciphertext)
def encapsulate(packed_public_key):
    # Passo 1 let coins be uniform random bits (obviamente deveria ser um␣
 ↪gerador aleatório melhor)
    coins = random_bits(sample_plaintext_bits)

    # Passo 2 set (r, m) = sample_rm(coins)
    (r, m) = sample_rm(coins)

    # Passo 3 packed_rm = pack_S3(r) || pack_S3(m)
    packed_rm = pack_S3(r) + pack_S3(m)

    # Passo 4 shared_key = Hash(packed_rm) (o nosso algoritmo de hash recebe um␣
 ↪array de bytes e não um array de bits)
    shared_key = Hash(packed_rm)

    # Passo 5 packed_ciphertext = dpke_Encrypt(packed_public_key, packed_rm)
    packed_ciphertext = dpke_Encrypt(packed_public_key, packed_rm)

    # Passo 6 Output (shared_key, packed_ciphertext)
    return (shared_key, packed_ciphertext)

# 1.12.2.2 KEM Encapsulate para ser usado no PKE-CCA (recebe as coins)
def kem_encapsulate(packed_public_key, coins):
    (r, m) = sample_rm(coins)

    packed_rm = pack_S3(r) + pack_S3(m)

    shared_key = Hash(packed_rm)

    packed_ciphertext = dpke_Encrypt(packed_public_key, packed_rm)
    return (shared_key, packed_ciphertext)

# 1.12.3 Decapsulate
# B packed_private_key -> B packed_ciphertext -> b shared_key
def decapsulate(packed_private_key, packed_ciphertext):
    # Passo 1 packed_private_key = packed_f || packed_fp || packed_hq || prf_key
    packed_f = packed_private_key[0:packed_s3_bytes]
    packed_fp = packed_private_key[packed_s3_bytes:(2 * packed_s3_bytes)]
    packed_hq = packed_private_key[(2 * packed_s3_bytes):((2 * packed_s3_bytes)␣
 ↪+ packed_sq_bytes)]
    prf_key = packed_private_key[((2 * packed_s3_bytes) + packed_sq_bytes):]

    # Passo 2 (packed_rm, fail) = dpke_decrypt(packed_private_key,␣
 ↪packed_ciphertext)
    (packed_rm, fail) = dpke_decrypt(packed_private_key, packed_ciphertext)
```

```
    # Passo 3 shared_key = Hash(packed_rm
    shared_key = Hash(packed_rm)

    # Passo 4 random_key = Hash(prf_key + packed_ciphertext)
    random_key = Hash(prf_key + packed_ciphertext)

    # Passo 5 se fail = 0 output shared_key, senão output random_key
    if fail == 0:
        return shared_key
    else:
        return random_key
```

1.2 PKE OW-CPA

Abaixo temos a implementação do PKE descrito nas especificações. Assim como a secção anterior o código está devidamente documentado. No entanto temos um problema que impede a progressão do trabalho no NTRU, o primeiro passo da geração da chave pública, G = g * 3 é sempre igual a zero e então o terceiro passo obviamente não funciona.

[9]:
```
# 1.11 Passively secure DPKE
# 1.11.1 DPKE_Key_Pair
# b coins -> (B packed_private_key, B packed_public_key)
def dpke_key_pair(coins):
    # Passo 1 (f,g) = Sample_fg(coins)
    (f, g) = sample_fg(coins)

    # Passo 2 fp = S3_inverse(f)
    fp = f ^ (-1)

    # Passo 3 (h,hq) = DPKE_Public_Key(f,g)
    (h, hq) = dpke_public_key(f, g)

    # Passo 4 packed_private_key = pack_S3(f) || pack_S3(fp) || pack_Sq(hq)
    packed_private_key = pacl_s3(f) + pack_s3(fp) + pack_sq(hq)

    # Passo 5 packed_public_key = pack_Rq0(h)
    packed_public_key = pack_rq0(h)

    # Passo 6 output (packed_private_key,packed_public_key)
    return (packed_private_key, packed_public_key)

# 1.11.2 DPKE_Public_Key
# poly f -> poly g -> (poly h, poly hq)
def dpke_public_key(f, g):
    # Passo 1 G = 3*g # TODO isto dá 0, por isso o passo 2 dá 0 e então o passo␣
 ↪3 não funciona
    G = g * 3
```

```
    # print(G) # output é 0

    # Passo 2 v0 = Sq(G*f)
    v0 = _Sq(G * f).lift()

    # Passo 3 v1 = sq_inverse(v0)
    v1 = v0 ^ (-1)

    # Passo 4 h = Rq(v1 * G * G)
    h = _Rq(v1 * G * G).lift()

    # Passo 5 hq = Rq(v1 * f * f)
    hq = _Rq(v1 * f * f).lift()

    # Passo 6 output (h, hq)
    return (h, hq)

# 1.11.3 DPKE_Encrypt
# B packed_public_key -> B packed_rm -> B packed_ciphertext
def dpke_encrypt(packed_public_key, packed_rm):
    # Passo 1 packed_rm = packed_r || packed_m
    packed_r = packed_rm[:packed_s3_bytes]
    packed_m = packed_rm[-packed_s3_bytes:]

    # Passo 2 r = _S3(unpack_S3(packed_r))
    r = canonS3(_S3(unpack_S3(packed_r)).lift().list(), 1)

    # Passo 3 m0 = unpack_S3(packed_m)
    m0 = unpack_S3(packed_m)

    # Passo 4 m1 = Lift(m0)
    m1 = Lift(m0)

    # Passo 5 h = unpack_Rq0(packed_public_key)
    h = unpack_Rq0(packed_public_key)

    # Passo 6 c = Rq(r * h + m1)
    c = _Rq((r * h) + m1).lift()

    # Passo 7 packed_ciphertext = pack_Rq0(c)
    packed_ciphertext = pack_Rq0(c)

    # Passo 8 output packed_ciphertext
    return packed_ciphertext

# 1.11.4 DPKE_Decrypt
```

```
# B packed_private_key -> B packed_ciphertext -> (B packed_rm, bit fail)
def dpke_decrypt(packed_private_key, packed_ciphertext):
    # Passo 1 packed_private_key = packed_f  || packed_fp || packed_hq
    packed_f = packed_private_key[0:packed_s3_bytes]
    packed_fp = packed_private_key[packed_s3_bytes:(2 * packed_s3_bytes)]
    packed_hq = packed_private_key[(2 * packed_s3_bytes):((2 * packed_s3_bytes)
 ↪+ packed_sq_bytes)]

    # Passo 2 c = unpack_Rq0(packed_ciphertext)
    c = unpack_Rq0(packed_ciphertext)

    # Passo 3 f = _S3(unpack_S3(packed_f))
    f = canonS3(_S3(unpack_S3(packed_f)).lift().list(), 1)

    # Passo 4 fp = unpack_S3(packed_fp)
    fp = unpack_S3(packed_fp)

    # Passo 5 hq = unpack_Rq0(packed_hq)
    hq = unpack_Rq0(packed_hq)

    # Passo 6 v1 = _Rq(c*f)
    v1 = canonRq(_Rq(c * f).lift().list(), q//2)

    # Passo 7 m0 = _S3(v1 * fp)
    m0 = canonS3(_S3(v1 * fp).lift().list(), 1)

    # Passo 8 m1 = Lift(m0)
    m1 = Lift(m0)

    # Passo 9 r = _Sq((c - m1) * hq)
    r = canonSq(_Sq((c - m1) * hq).lift().list(), q//2)

    # Passo 10 packed_rm = pack_S3(r) || pack_S3(m0)
    packed_rm = pack_S3(r) + pack_S3(m0)

    # Passo 11 se r pertencer a Lr e m0 pertencer a Lm fail = 0
    # Passo 12 senão fail = 1
    fail = 1
    if check_fail_r(r) & check_fail_m0(m0, (q//8) - 2):
        fail = 0
    # Passo 13 Output (packed_rm, fail)
    return (packed_rm, fail)

# [Coeffs] v -> bool r
# Check_fail_r verifica se r pertence a Lr, ou seja, se os coeficientes estão
 ↪todos entre -1 e 1
def check_fail_r(v):
```

```
    r = 1

    for i in range(len(v)):
        if v[i] > 1 | v[i] < -1:
            r = 0
            break
    return r


# [Coeffs] v -> int d -> bool r
# Check_fail_m0 verifica se m0 pertence a Lm, ou seja,
# se os coeficientes estão todos entre -1 e 1 e há d/2 coeficientes iguais a -1
 ↪e d/2 coeficiente iguais a 1
def check_fail_m0(v, d):
    one = 0
    minus_one = 0
    r = 1

    for i in range(len(v)):
        if v[i] == 1:
            one += 1
        if v[i] == -1:
            minus_one += 1
        if v[i] > 1 | v[i] < -1:
            r = 0
            break
    return (one == d//2) & (minus_one == d//2) & r
```

1.3 Sampling

Nesta secção temos as funções responsáveis por converter as strings de bits em polinómios de acordo com as nossas necessidades.

[4]:
```
# 1.10 Sampling
# 1.10.1 Sample_fg
# b fg_bits -> (poly f, poly g)
# Implementação NTRU HPS
def sample_fg(fg_bits):
    # Passo 1 fg_bits = f_bits || g_bits
    f_bits = fg_bits[:sample_iid_bits]
    g_bits = fg_bits[-sample_fixed_type_bits:]

    # Passo 2 f = Ternary(f_bits)
    f = ternary(f_bits)

    # Passo 3 g = Fixed_Type(g_bits)
    g = fixed_type(g_bits)
```

```
    # Passo 4 output (f, g)
    return (f, g)

# 1.10.2 Sample_rm
# b rm_bits -> (poly r, poly m)
# Implementação NTRU HPS
def sample_rm(rm_bits):
    # Passo 1 rm_bits = r_bits || m_bits
    r_bits = rm_bits[:sample_iid_bits]
    m_bits = rm_bits[-sample_fixed_type_bits:]

    # Passo 2 r = Ternary(r_bits)
    r = ternary(r_bits)

    # Passo 3 m = Fixed_Type(m_bits)
    m = fixed_type(m_bits)

    # Passo 4 output (r, m)
    return (r, m)

# 1.10.3 Ternary
# b string -> ternpoly p
def ternary(b):
    # Passo 1 v = 0
    v = []
    # Passo 2 i = 0
    # Passo 3, 5 e 6 for
    for i in range (n - 1):
        # Passo 4
        tmp = 0
        for j in range (8):
            tmp += (2 ** j) * b[(8 * i) + j]
        v.append(tmp)

    # Passo 7 Output _S3(v)
    return canonS3(_S3(v).lift().list(), 1)

# 1.10.4 Ternary_Plus
# b string -> ternpoly p
def ternary_plus(b):
    # Passo 1 v = Ternary b
    v = ternary(b)
    # Passo 2
    t = 0
    for i in range (n - 1):
        t += v[i] * v[i + 1]
```

```
    # Passo 3 se t < 0 s = -1 senão s = 1
    s = 1
    if t < 0:
        s = -1

    # Passo 4, 5, 7, 8
    for i in range (0, n - 1, 2):
        # Passo 6 vi = s * vi
        v[i] = v[i] * s

    # Passo 9 output _S3(v)
    return canonS3(_S3(v).lift().list(), 1)


# 1.10.5 Fixed_Tpye
# b string -> ternpoly p
def fixed_type(b):
    # Passo 1 A = 0
    A = []
    for i in range (n - 1):
        A.append(0)

    # Passo 2 v = 0
    v = []
    # Passo 3, 4, 6, 7
    for i1 in range ((q // 16) - 1):
        tmp = 0
        # Passo 5
        for j in range (30):
            tmp += (2**(j + 2)) * b[(30*i1) + j]
        A[i1] = 1 + tmp

    # Passo 8, 10, 11
    for i2 in range (i1 + 1, (q // 8) - 2):
        tmp = 0
        # Passo 9
        for j in range (30):
            tmp += (2**(j + 2)) * b[(30*i2) + j]
        A[i2] = 2 + tmp

    # Passo 12, 14, 15
    for i3 in range (i2 + 1, n - 1):
        tmp = 0
        # Passo 13
        for j in range (30):
            tmp += (2**(j + 2)) * b[(30*i3) + j]
```

```
        A[i3] = tmp

    # Passo 16 Sort A
    A = sorted(A)

    # Passo 17, 18, 20, 21
    for i4 in range (n - 1):
        # Passo v = v + (Ai mod 4)xi
        v.append(A[i4] % 4)

    # Passo 22 output _S3(v)
    poly = _S3(v).lift().list()
    poly = canonS3(poly, 1)
    return poly
```

1.4 Arithmetic

Nesta secção apenas temos a Lift porque o SageMath trata das inversões de polinómios por nós.

[ ]:
```
# 1.9 Arithmetic
# 1.9.1 e 2, o SageMath já faz a inversa
# 1.9.3 Lift
# poly m -> poly p
# NTRU HPS
def lift(m):
    # Passo 1 p = output _S3(m)
    return canonS3(_S3(m).lift().list(), 1)
```

1.5 Encodings

Nesta secção temos as funções responsáveis por transformar polinómios em arrays de bytes e vice-versa. Na especificação da função 1.8.4 no documento temos o passo 6: v = v + c$x\hat{}i$ - c$x\hat{}$(n-1), não temos a certeza se o passo 6 está a ser feito como deve de ser.

[5]:
```
# 1.8 Encodings
# 1.8.3 pack_Rq0
# poly a -> B B
def pack_rq0(a):
    # Passo 1 v = _Rq(a)
    v = canonRq(_Rq(a).lift().list(), q//2)

    # Passo 2 b = (0)
    b = []
    for i in range ((n - 1) * log(q, 2)):
        b[i] = 0

    # Passo 3 4 6 7
```

```
    for i in range (n -1):
        # Passo 5
        tmp = v[i] % q
        for j in range (log(q, 2)):
            b[(i * log(q, 2)) + j] = tmp % 2
            tmp = tmp // 2

    # Passo 8 output bits_to_bytes(b)
    return bits_to_bytes(b)

# 1.8.4 unpack_Rq0
# B B -> poly a
def unpack_rq0(B):
    # Passo 1 b = bytes_to_bits(B, (n-1) * logq)
    b = bytes_to_bits(B, (n - 1) * log(q, 2))

    # Passo 2 3 4 7 8
    v = []
    aux = 0
    for i in range (n -1):
        # Passo 5
        tmp = 0
        for j in range (log(q, 2)):
            tmp += b[(i * log(q, 2)) + j] * (2**j)
        # Passo 6 será que é assim?
        v[i] = tmp
        aux -= tmp
    v[n - 1] = aux

    # Passo 9 Output Rq(v)
    return _Rq(v).lift

# 1.8.5 pack_Sq
# poly a -> B B
def pack_sq(a):
    # Passo 1 v = _Sq(a)
    v = canonSq(_Sq(a).lift().list(), q//2)

    # Passo 2 b = (0)
    b = []
    for i in range ((n - 1) * log(q, 2)):
        b[i] = 0


    # Passo 3 4 6 7
    for i in range (n -1):
        # Passo 5
```

```
        tmp = v[i] % q
        for j in range (log(q, 2)):
            b[(i * log(q, 2)) + j] = tmp % 2
            tmp = tmp // 2

    # Passo 8 output bits_to_bytes(b)
    return bits_to_bytes(b)

# 1.8.6 unpack_Sq
# B B -> poly a
def unpack_sq(B):
    # Passo 1 b = bytes_to_bits(B, (n-1) * logq)
    b = bytes_to_bits(B, (n - 1) * log(q, 2))

    # Passo 2 3 4 7 8
    v = []
    for i in range (n -1):
        # Passo 5
        tmp = 0
        for j in range (log(q, 2)):
            tmp += b[(i * log(q, 2)) + j] * (2**j)
        # Passo 6
        v[i] = tmp

    # Passo 9 Output Sq(v)
    return _Sq(v).lift()


# 1.8.7 pack_S3
# poly a -> B B
def pack_s3(a):
    # Passo 1 v = _S3(a)
    v = canonS3(_S3(a).lift().list(), 1)

    # Passo 2
    b = []
    for i in range (8 * math.ceil((n - 1) / 5)):
        b[i] = 0

    # Passo 3 4 7 8
    for i in range (math.ceil((n - 1) / 5)):
        # Passo 5
        c = []
        for j in range (5):
            c[j] = v[(5 * i) + j] % 3
        tmp = 0
        for k in range (5):
```

```
            tmp += (3**k) * c[k]
        # Passo 6
        for l in range (8):
            b[(8*i) + l] = tmp % 2
            tmp = tmp // 2

    # Passo 9 Output bits_to_bytes(b)
    return bits_to_bytes(b)

# 1.8.4 unpack_S3
# B B -> poly a
def unpack_s3(B):
    # Passo 1 b = bytes_to_bits(B, 8 * ceil((n - 1) / 5))
    b = bytes_to_bits(B, 8 * math.ceil((n - 1) / 5))

    # Passo 2 3 4 7 8
    v = []
    for i in range (8 * math.ceil((n - 1) / 5)):
        # Passo 5
        c = []
        tmp = 0
        for j in range (8):
            tmp += (2**j) * b[(8*1) + j]
        for k in range (5):
            c[k] = tmp % 3
            tmp = tmp // 3
        # Passo 6
        for l in range (5):
            v[(5 * i) + l] = c[l]

    # Passo 9 Output S3(v)
    return _S3(v).lift()
```

1.6 Código Alheio

Nesta secção temos funções responsáveis por transformar um polinómio na sua representação canónica, substituindo q/2 ou 1 a todos os seus membros.

[6]:
```
# Canonificação dos coeficientes (subtrair n (q / 2 ou 1) a todos)
def canonRq(v, n):
    for i in range (len(v)):
        v[i] = Integer(v[i]) - n

    return _Rq(v).lift()

def canonSq(v, n):
    for i in range (len(v)):
```

```
        v[i] = Integer(v[i]) - n

    return _Sq(v).lift()

def canonS3(v, n):
    for i in range (len(v)):
        v[i] = Integer(v[i]) - n

    return _S3(v).lift()
```

Continuando a secção anterior aqui temos uma conversão de bits para bytes e de bytes para bits que não segue exatamente o que é dito no início da secção 1.8 da especificação. Além disso temos uma função para o SHA3-256.

[7]:
```
# Conversões de B para b e b para B
def bytes_to_bits(B1):
    B = []
    for i in range(len(B1)):
        B.append(Integer(B1[i]))
    b = []
    size = len(B)
    for i in range (size):
        tmp = B[i]
        for j in range (8):
            b.append(tmp % 2)
            tmp = tmp // 2
    stop = 0
    i = len(b) - 1
    while stop == 0:
        if b[i] == 1:
            stop = 1
        else:
            del(b[i])
        i -= 1
    return b

def bits_to_bytes(b):
    s = len(b)
    smod = s % 8
    size = math.ceil(s/8)

    B = []
    for i in range(size - 1):
        tmp = 0
        for j in range (8):
            tmp += (2 ** j) * b[(8* i) + j]
        B.append(tmp)
```

```
    if(smod > 0):
        tmp = 0
        for i in range(smod):
            tmp += (2 ** i) * b[(8 * (size - 1)) + i]
        B.append(tmp)
    else:
        tmp = 0
        for j in range (8):
            tmp += (2 ** j) * b[(8* i) + j]
        B.append(tmp)

    return bytearray(B)

# Hash de um array de bytes
def Hash(message):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(message)
    return digest.finalize()
```

2 Testes

2.1 DPKE OW-CPA

Aqui temos o código para fazer teste do DPKE.

[ ]:
```
# Teste NTRU_DPKE
coins = []
for i in range(sample_key_bits + prf_key_bits):
    coins.append(randint(0,1))

(chave_privada, chave_publica) = dpke_key_pair(coins)

texto_limpo = os.urandom(dpke_plaintext_bytes)

criptograma = dpke_encrypt(chave_publica, texto_limpo)

(texto_limpo2, falhanco) = dpke_decrypt(chave_privada, criptograma)

if falhanco == 1:
    print("falhou")
else:
    print(texto_limpo == texto_limpo2)
```

2.2 KEM IND-CCA2

Aqui temos o código para fazer teste do KEM.

[ ]:
```
# Teste NTRU_KEM
coins = []
for i in range(sample_key_bits + prf_key_bits):
    coins.append(randint(0,1))

(chave_privada, chave_publica) = key_pair(coins)

(chave_partilhada, criptograma) = encapsulate(chave_publica)

chave_partilhada2 = decapsulate(chave_privada, criptograma)

print(chave_partilhada == chave_partilhada2)
```

2.3 Transformação FO

Aqui temos o código do TP1 ligeiramente alterado tendo em conta que o NTRU não foi implementado numa classe. É de salientar para não haver confusão que o output da encapsulate está "trocado" em vez de devolver (e,k) devolve (k,e).

[ ]:
```
class PKE:
    # construtor da classe
    def __init__(self, x, priv, pub):
        self.x = x
        self.priv = priv
        self.pub = pub

    # hash h
    def hashh(self, message):
        digest = hashes.Hash(hashes.SHA256())
        digest.update(message)
        return digest.finalize()

    # hash g
    def hashg(self, message):
        digest = hashes.Hash(hashes.BLAKE2s(32))
        digest.update(message)
        return digest.finalize()

    # xor de um byte a com um byte b (o sagemath faz interferência com o␣
 ↪operador '^')
    def mini_xor(self, a, b):
        tmpa = a
        tmpb = b
        r0 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r1 = tmpa % 2 + tmpb % 2
```

```
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r2 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r3 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r4 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r5 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r6 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r7 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)

        soma = 0
        if r0 == 1:
            soma += 1
        if r1 == 1:
            soma += 2
        if r2 == 1:
            soma += 4
        if r3 == 1:
            soma += 8
        if r4 == 1:
            soma += 16
        if r5 == 1:
            soma += 32
        if r6 == 1:
            soma += 64
        if r7 == 1:
            soma += 128

        return soma

    # xor de dois arrays de bytes
    def xor(self, a, b):
        size = len(b)
        if len(a) < len(b):
            size = len(a)
```

```
        xored = bytearray(size)
        for i in range(size):
            xored[i] = self.mini_xor(a[i], b[i])
        return xored

    # E'
    def cifrar(self):
        # primeiro passo, r <- h
        self.r = self.hashh(self.x)
        # segundo passo, y <- x XOR g(r)
        self.y = self.xor(self.x, self.hashg(self.r))
        # terceiro passo, r' <- y || r
        self.rl = self. y + self.r
        # quarto passo, KEM(r')
        (self.k, self.e) = kem_encapsulate(self.pub, self.rl)
        self.k = bits_to_bytes(self.k)
        # finalmente c = k XOR r
        self.c = self.xor(self.k, self.r)

    # D'
    def decifrar(self):
        # k <- KREv(e)
        self.k = decapsulate(self.priv, e)
        # r <- c XOR k
        self.r = self.xor(self.c, self.k)
        # r' = y || r
        self.rl = self.y + self.r
        # (e, k) = f(rl)
        (self.k2, self.e2) = kem_encapsulate(self.pub, self.rl)
        self.k2 = bits_to_bytes(self.k2)
        # verificação f(rl) == (e, k)
        if (self.k2 == self.k) & (self.e2 == self.e):
            # x == y XOR g(r)
            self.x = self.xor(self.y, self.hashg(self.r))
            print("True")
        else:
            print("False")
```

2.4 PKE IND-CCA

Aqui temos o código para fazer teste do PKE criado com a transformação FO.

[ ]:
```
# Teste NTRU_FOT_PKE
coins = []
for i in range(sample_key_bits + prf_key_bits):
    coins.append(randint(0,1))
```

```
(priv, pub) = key_pair(coins)
# inicializamos a classe PKE
b = PKE(b'teste', priv, pub)
# fazemos E'(x)
b.cifrar()
# fazemos D' (yec)
b.decifrar()
```