NTRU

May 10, 2021

1 Exercício 1 - grupo 6 - Ana Margarida Campos (A85166) , Nuno Pereira (PG42846)

Neste primeiro exercício foi proposta a implementação de duas versões, IND-CPA e IND-CCA, do protótipo **NTRU** que foi candidato ao concurso *NIST-PQC*, neste caso da terceira ronda. De seguida, são apresentadas em duas seções (Passively secure DPKE e Strongly secure KEM) os resultados da resolução deste exercício acompanhado de uma explicação detalhada de cada função implementada. Este, foi desenvolvido tendo por base o documento *ntru.pdf*.

1.1 Passively secure DPKE

A primeira implementação consistiu no desenvolvimento de funções que permitissem uma segurança IND-CPA, ou seja, segurança contra ataques *Chosen Plaintext Attacks*. Numa primeira fase foi necessário implementar algumas funções auxiliares, sendo estas:

- ***__ternary***: cria uma lista com elementos entre -1 a 1 e, posteriormente, com base nesta lista, retorna um polinómio ternário (ou seja com coeficientes -1,0 e 1);
- ***__fixedType***: cria uma lista com elementos entre -1 a 1 mas com a particularidade de existir Q/16-1 elementos iguais a 1 e Q/16-1 elementos iguais a -1. Esta lista é depois convertida em polinómio ternário;
- ***sample__fg***: faz recurso de ambas funções anteriores (***__ternary*** e ***__fixedType***), obtendo, deste modo, os polinómios necessários para a obtenção das chaves;
- ***pack***: recebe como argumento um polinómio e codifica-o em *bytes*;
- ***unpack***: contrário da função ***pack***, ou seja, recebe um conjunto de *bytes* e retorna um polinómio do tipo *Rq*;
- ***unpackSq***: recebe um conjunto de *bytes* e retorna um polinómio do tipo *Sq*;
- ***chave__publica***: utiliza os polinómios criados com recurso à função ***sample__fg*** e retorna um *h* e *hq* que vão ser fundamentais para a criação das chaves pública e privada.

As funções principais centram-se na geração de chaves, cifragem da mensagem passada como parâmetro e posterior decifragem do texto cifrado, obtendo deste modo, a mensagem original:

- ***gerar__chaves***: com recurso às funções anteriormente apresentadas, ocorre a geração das chaves pública e privada. A primeira é essencial para a cifragem da mensagem e a segunda para a decifragem do criptograma;

- ***cifragem***: esta função tem como objetivo principal cifrar uma mensagem. Desta forma, recebe como parâmetros a chave pública e a mensagem e, com recurso às funções anteriores, cria o texto cifrado;
- ***decifragem***: tem como objetivo decifrar um criptograma, obtendo como resultado o texto limpo correspondente. Recebe como argumentos a chave privada e o criptograma.

Uma dificuldade encontrada centrou-se na obtenção do texto limpo correto, pelo que, a maneira utilizada para ultrapassar este obstáculo foi a da cifragem do parâmetro *m1* utilizando o modo *GCM*, e a posterior decifragem, permitindo, deste modo, obter o resultado esperado.

[1]:
```
# imports necessários para a resolução
import random as rn
import numpy as np
from sympy import Symbol, Poly
import os
import math
import zlib
import gzip
from cryptography.hazmat.primitives.ciphers.aead import AESGCM
```

[2]:
```
# constantes do NTRU
N = 509
Q = 2048
T = N//4

# criação dos anéis necessários
_Z.<w>  = ZZ[]
R.<w>   = QuotientRing(_Z ,_Z.ideal(w^N - 1))

_Q.<w>  = GF(Q)[]
Rq.<w>  = QuotientRing(_Q , _Q.ideal(w^N - 1))

_Q.<w>  = GF(Q)[]
RT.<w>  = QuotientRing(_Q , _Q.ideal(w^204))

_E.<w>  = ZZ[]
S.<w>   = QuotientRing(_E,_E.ideal(w^N - 1))

_Q.<w>  = GF(Q)[]
Sq.<w>  = QuotientRing(_Q , _Q.ideal((w^N - 1)/(w-1)))

_Q3.<w>  = GF(3)[]
S3.<w>  = QuotientRing(_Q3 , _Q3.ideal((w^N - 1)/(w-1)))


# funções auxiliares:
metadados = os.urandom(16)
```

```
listanouce = []
# gerador de nounce
def geraNounce(tamNounce):
    nounce = os.urandom(tamNounce)
    if not (nounce in listanouce):
        listanouce.append(nounce)
        return nounce
    else:
        geraNounce(tamNounce)

# criação de um polinómio ternário
def _ternary(n=N,t=T):
    u = [rn.choice([-1,1]) for i in range(t)] + [0]*(8*(n-1)-t)
    rn.shuffle(u)
    return Rq(u)

# criação de um polinómio ternário com igual número de elementos iguais a -1 e 1
def _fixedType(n=N,t=T):
    q = Q//16 -1
    h = (30*(n-1))-2*q
    u1 = [rn.choice([1]) for i in range(t)] + [0]*(q-t)
    u2 = [rn.choice([-1]) for i in range(t)] + [0]*(q-t)
    u3 = [rn.choice([0]) for i in range(t)] + [0]*(h-t)
    u = [*u1,*u2,*u3]
    rn.shuffle(u)
    return Rq(u)

# criação dos polinómios com recurso às funções anteriores
def sample_fg(n=N, t=T):
    f = _ternary(n,t)
    g = _fixedType(n,t)
    return f,g

# obtenção do tamanho necessário para o unpack
def tamanho(stringB, numberS):
    count = 2
    auxCount = 1
    i = 0
    while i < len(stringB):
        if numberS == auxCount:
            i = i + 2
            while (i < len(stringB)) and (stringB[i] != 120 or stringB[i + 1] !
 ↪= 1):
                count = count + 1
                i = i + 1
            auxCount = auxCount + 1
```

```
        i = i + 1
        if (i + 2) < len(stringB) and (stringB[i] == 120 and stringB[i + 1] ==␣
↪1):
            auxCount = auxCount + 1
        if auxCount > numberS:
            break
    return count

# passagem do polinómio para bytes
def pack(polinomio):
    check_List=isinstance(polinomio, list)
    if(not check_List):
        polinomio=polinomio.list()
        polinomioB= bytes(_Z(polinomio))
        compress = zlib.compress(polinomioB,1)
    else:
        compress = zlib.compress(bytes(_Z(polinomio)),1)
    return compress

# passagem de um conjunto de bytes para um polinómio do tipo Rq
def unpack(pack):
    unpack = zlib.decompress(pack)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Rq(newUnpack)

# passagem de um conjunto de bytes para um polinómio do tipo Sq
def unpackSq(pack):
    unpack = zlib.decompress(pack)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Sq(newUnpack)

# geraçãp de um h e hq que vão ser necessários na criação das chaves públicas e␣
↪privadas
def chave_publica(f, g):
    G = g * 3
    v0 = Sq(G*f)
    v1 = v0.inverse_of_unit()
    h = Rq(v1*G*G)
    hq = Rq(v1*f*f)
    return(h,hq)

# funções principais:
```

```
# geração das chaves pública e privada
def gerar_chaves():
    f, g = sample_fg()
    fq = f.inverse_of_unit()
    h, hq = chave_publica(f,g)
    pf= pack(f)
    pfq =pack(fq)
    phq =pack(hq)
    packed_private_key = pf+ pfq+phq
    packed_public_key = pack(h)
    return (packed_private_key, packed_public_key)

# cifragem de uma mensagem
def cifragem(packed_public_key, packed_rm, key):
    packed_r = pack(packed_rm[:102])
    packed_m = pack(packed_rm[-102:])
    r = unpack(packed_r)
    m0 = unpack(packed_m)
    m1 = m0.lift()
    h = unpack(packed_public_key)
    c = Rq(r*h + m1)
    packed_ciphertext = pack(c)
    aesgcm = AESGCM(key)
    nonce = geraNounce(12)
    m1_cifrado = aesgcm.encrypt(nonce, bytes(_Z(m1)), metadados)
    m1_cifrado += nonce
    return packed_ciphertext, m1_cifrado

# decifragem de um criptograma obtendo a mensagem original
def decifragem(packed_private_key, packed_ciphertext, key, m1_cifrado):
    tf = tamanho(packed_private_key,1)
    tfq = tamanho(packed_private_key,2)
    thq = tamanho(packed_private_key,3)
    packed_f =  packed_private_key[:tf]
    packed_private_key = packed_private_key[tf:]
    packed_fq = packed_private_key[:tfq]
    packed_hq = packed_private_key[tfq:]
    c = Rq(unpack(packed_ciphertext))
    f = Rq(unpack(packed_f))
    fq = unpack(packed_fq)
    hq = unpackSq(packed_hq)
    aesgcm = AESGCM(key)
    nonce = m1_cifrado[-12:]
    m1_cifrado = m1_cifrado[:-12]
    m1 = aesgcm.decrypt(nonce, m1_cifrado, metadados)
    y = []
    for i in m1:
```

```
        y.append(i)
    m1_novo = Rq(y).lift()
    r = Rq((c-m1_novo)*hq)
    packed_rm = [*r, *(m1_novo.list())]
    fail = 0
    for i in r.list():
        if i==1 or i==0 or i==-1:
            fail = 0
        else:
            fail = 1
            break
    for i in m1_novo.list():
        if i==1 or i==0 or i==-1:
            fail = 0
        else:
            fail = 1
            break
    result = packed_rm[:102] + packed_rm[-102:]
    return result, fail
```

De seguida são apresentados os resultados obtidos com recurso às funções anteriores. Neste caso, estamos a considerar uma mensagem fixa.

[3]:
```
key = os.urandom(32)

mensagem = RT([1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1,␣
 ↪0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1,
           0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,␣
 ↪1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1,
           1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1,␣
 ↪1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
           1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,␣
 ↪0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1,
           1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1,␣
 ↪1, 1])

print("Mensagem=", mensagem,"\n")

packed_private_key, packed_public_key = gerar_chaves()
print("Chave privada=", packed_private_key, "\n")
print("Chave pública=", packed_public_key, "\n")

texto_cifrado, m1_cifrado= cifragem(packed_public_key, mensagem.list(), key)
print("texto cifrado=", texto_cifrado,"\n")
```

```
texto_limpo,fail = decifragem(packed_private_key, texto_cifrado,key, m1_cifrado)
print("decifragem polinomio==", RT(texto_limpo),"\n")

print("Texto Limpo = Resultado da decifragem:", texto_limpo== mensagem.list())
```

```
Mensagem= w^203 + w^202 + w^201 + w^199 + w^198 + w^197 + w^195 + w^193 + w^191
+ w^190 + w^189 + w^188 + w^187 + w^184 + w^183 + w^182 + w^181 + w^180 + w^179
+ w^178 + w^177 + w^176 + w^175 + w^174 + w^173 + w^172 + w^170 + w^169 + w^167
+ w^166 + w^164 + w^163 + w^160 + w^156 + w^154 + w^153 + w^152 + w^151 + w^149
+ w^148 + w^147 + w^146 + w^145 + w^144 + w^143 + w^141 + w^139 + w^137 + w^135
+ w^134 + w^132 + w^131 + w^130 + w^129 + w^128 + w^127 + w^126 + w^125 + w^124
+ w^123 + w^122 + w^120 + w^119 + w^118 + w^117 + w^116 + w^115 + w^114 + w^113
+ w^112 + w^111 + w^110 + w^108 + w^106 + w^105 + w^103 + w^102 + w^100 + w^99 +
w^97 + w^96 + w^95 + w^94 + w^93 + w^92 + w^90 + w^89 + w^88 + w^86 + w^84 +
w^82 + w^81 + w^79 + w^77 + w^75 + w^74 + w^73 + w^72 + w^71 + w^70 + w^67 +
w^64 + w^63 + w^61 + w^60 + w^59 + w^58 + w^57 + w^56 + w^55 + w^54 + w^52 +
w^49 + w^47 + w^46 + w^44 + w^43 + w^42 + w^41 + w^40 + w^38 + w^35 + w^34 +
w^33 + w^32 + w^31 + w^30 + w^29 + w^26 + w^25 + w^23 + w^22 + w^20 + w^19 +
w^14 + w^13 + w^12 + w^10 + w^9 + w^7 + w^6 + w^5 + w^4 + w^3 + w^2 + 1
```


```
Chave privada= b"x\x01e\x8e\x01\x12\x800\x08\xc3\xdc\xff?-\xa1\x14\x86\xee<\xa4%
e{\x1e\xce\xe1;Q\xf9\xa5\xacR\n\x8f)g\xcd\xef\x81\xc6\x85HD\xc8\xb1h\x8e%\x9e}\x
11\xa3\xbc\xe5b\xa2\xbd\x14Y\xe3\x86\x01d\xb2.\xe1\x91\x05)\x13\xb5^\xd1:\xe7\xf
8\x19lz\xc9\xb9\x08H\xd1\xf6\x92,\xdc[\xf3\xfa\x0f\xe7\xc0\xda \xd1\xa4\x9azb\x0
4X\x0b\x82\xd3\x90\x16a\xd3\xfdm\xcdg\xb4\x88X\xf4\x02k\xa4\x00fx\x01U\x8e\t\x0e
\xc40\x08\x03\xcb\xff?\xbd\xcc\xd8\xb4\xdaH\t\x87\xcd\x90\x99\x99g\x9e=I\xf6Meg\
xbb\x91\x88\xf8\x90\xdfH\xefnS\x82\x96\xd8\x1dN\xfa\x01\x02\xd2y\xf0\xdbY\x1c{\n
p\xdd\x16\xd8\xbdZ\xf7\xe9/\xb0\xe5_$\x1e\xf7;W\xa1\x8a\xf3\xf7\x00\xfc&\xf8\xb5
\x0b\xd2\x95\xd0a\xa8\x15o%_\xc0\xb2G\x0c[\xab\x85\xa2^^\xa6YU\xff\x1b\xbb\x1e8=
\x18!\x85veEe\xbcb\xf0\x9bm\xc8`\xab?J\xad\xed1\x13(\xfd\x1f\xf8\xbc\x00\xf8x\x0
1U\x8e\t\x12\x800\x08\x03\xdb\xff\x7f\xda\xec\x86VeF$\xe4\xa0k\xa5\xf6\xde\xf6N\
xce\xd9u\xb5W\x86\x08\xf2o\x13\x8f\xf2\xda$\xb1('3\x85\xa5\x83:9B,\x12TT3\x1dW\n
\xafSaS^\xa6\x18\x15\x19\x95\xbb\xa3\x89\xd3\xb9\x9c\xf9W\xd9\x90|\x1ev\xdd\xa6\
xb8\x85\x8f\x05\xbf\xc2\x0b\xbb\x03J\xd8'N\xf1\xdc\xa9^\xc0\xdaG\xf0\x8e~\xe7\xb
4\x0c,\xa9i_\x8c\xc6=DA\xa1HC\t\xfd\xcdx\xf5\xe7\x19n
\xafw.\xd5\xfb\x00\xff\x8c\x01\n"
```

```
Chave pública= b'x\x01M\x8e\x01\x12\x83@\x0c\x02\xcf\xff\x7f\xba\xec\x92dj\xd5r\
xb0$\xbe/\xd7{\xcf\xe7{\xd1U\xd8\x04\xcd\x94\xd1\x9a\xf5=`h\nN \x14\x836\xe3\xe6
\x85T\x9f\xc8Z7\xdf\xc0)\x85\xaaE\xb9\xf5v]V\xa8]c>1&\xf7\x01x\xaesB\x81r\x04\x1
d\xeb\x96\x051\x8bE\x91\xdbG\xef\x89\xe9\x1e0N\xda*2Sw\xb1\x94$s\xc3a\xcc\xcd\x1
fg\x7f\xb3\xaa\x11A\x14\xb1\xc8U\xe1\xff\x83~\x81\x18\xb8\x95}\x0f\xeb\x08\xbc\x
99\xb4\xebpr\xfd\x00\x0f\xe6\x01\x0c'
```

```
texto cifrado= b"x\x01M\x8e\x0b\x16\x800\x08\xc3\xe4\xfe\x97\x96$\x9bO?\xac\xa5\
xa5lf\x9e\xe9\x9fg\x11d\xc1\xbf'\xd1\xb3h\xdf\n]\x15:\xf4\x1a\xf2\xbc&\xb3\xb4\x
e9\xfe6\xa9\xc3\x0c(s\x89\xf4\xd6#c\x02r\x90\x9d\xe7\x9a\x8cUU\xd3\x01\xbd\x17\x
```

```
e2R\xc97\x83I\x9f\x12\x8d\xc5\x7f\xf4\x00\x06V&@N\xc3\xbd+}2P\x8d\x00\x80\x04\xe
c[\xc8\x9a\n\xab\xcb|>\x8f\x8a)\xce\xaft\x16\x14w\xbc\\\x06\x89\xaf\x8b\x9d\x9a\
xbd\x9e\x1a\x90\x04\xbd\x9a\xe5u\x8a\xd0=\xf3\x02\x12\n\x01\r"

decifragem polinomio== w^203 + w^202 + w^201 + w^199 + w^198 + w^197 + w^195 +
w^193 + w^191 + w^190 + w^189 + w^188 + w^187 + w^184 + w^183 + w^182 + w^181 +
w^180 + w^179 + w^178 + w^177 + w^176 + w^175 + w^174 + w^173 + w^172 + w^170 +
w^169 + w^167 + w^166 + w^164 + w^163 + w^160 + w^156 + w^154 + w^153 + w^152 +
w^151 + w^149 + w^148 + w^147 + w^146 + w^145 + w^144 + w^143 + w^141 + w^139 +
w^137 + w^135 + w^134 + w^132 + w^131 + w^130 + w^129 + w^128 + w^127 + w^126 +
w^125 + w^124 + w^123 + w^122 + w^120 + w^119 + w^118 + w^117 + w^116 + w^115 +
w^114 + w^113 + w^112 + w^111 + w^110 + w^108 + w^106 + w^105 + w^103 + w^102 +
w^100 + w^99 + w^97 + w^96 + w^95 + w^94 + w^93 + w^92 + w^90 + w^89 + w^88 +
w^86 + w^84 + w^82 + w^81 + w^79 + w^77 + w^75 + w^74 + w^73 + w^72 + w^71 +
w^70 + w^67 + w^64 + w^63 + w^61 + w^60 + w^59 + w^58 + w^57 + w^56 + w^55 +
w^54 + w^52 + w^49 + w^47 + w^46 + w^44 + w^43 + w^42 + w^41 + w^40 + w^38 +
w^35 + w^34 + w^33 + w^32 + w^31 + w^30 + w^29 + w^26 + w^25 + w^23 + w^22 +
w^20 + w^19 + w^14 + w^13 + w^12 + w^10 + w^9 + w^7 + w^6 + w^5 + w^4 + w^3 +
w^2 + 1

Texto Limpo = Resultado da decifragem: True
```

1.2 Strongly secure KEM

A segunda implementação consistiu no desenvolvimento de funções que permitissem uma segurança IND-CCA, ou seja, segurança contra ataques *Chosen Ciphertext Attacks*. Tal como anteriormente, tornou-se necessário, numa primeira fase, a implementação de funções auxiliares. Estas são iguais às desenvolvidas previamente, com algumas diferenças mínimas, nomeadamente nas funções de ***pack*** e ***unpack***. Foram também implementadas duas funções auxiliares novas sendo estas:

- ***bytes_to_bits***: tranforma um conjunto de *bytes* em *bits*;
- ***bits_to_bytes***: transforma um conjunto de *bits* em *bytes*;
- ***rand_bits***: gera palavras de *bits* com tamanho passado como argumento.

As funções principais são agora a geração de chaves, o encapsulamento e o desencapsulamento:

- ***geracao_chaves***: com recurso às funções auxiliares, ocorre a geração das chaves pública e privada;
- ***encapsulate***: recebe como argumento a chave pública e retorna a chave partilhada bem como o *packed_ciphertext* que corresponde à cifragem do *packed_rm*;
- ***decapsulate***: tem como objetivo a obtenção da chave partilhada. Para tal recebe como argumentos a chave privada e o criptograma e, através de um conjunto de operações, devolve o esperado, a chave partilhada.

[4]:
```
# funções auxiliares:
```

```python
# obtenção do tamanho necessário para o unpack
def tamanho(stringB, numberS):
    count = 10
    auxCount = 1
    i = 0
    while i < len(stringB):
        if numberS == auxCount:
            i = i + 10
            while (i < len(stringB)) and (stringB[i] != 31 or stringB[i + 1] !=␣
 ↪139 or stringB[i + 2] != 8 or stringB[i + 3] != 0  ):
                count = count + 1
                i = i + 1
            auxCount = auxCount + 1

        i = i + 1
        if (i + 10) < len(stringB) and (stringB[i] == 31 and stringB[i + 1] ==␣
 ↪139 and stringB[i + 2] == 8 and stringB[i + 3] == 0 ):
            auxCount = auxCount + 1
        if auxCount > numberS:
            break
    return count

# geração das chaves pública e privada
def gerar_chaves():
    f, g = sample_fg()
    fq = f.inverse_of_unit()
    h, hq = chave_publica(f,g)
    pf= pack2(f)
    pfq =pack2(fq)
    phq =pack2(hq)
    packed_private_key = pf+ pfq+phq
    packed_public_key = pack2(h)
    return (packed_private_key, packed_public_key)

# gera palavras de *bits* com tamanho passado como argumento
def rand_bits(p):
    key1 = ""
    for i in range(p):
        temp = str(rn.randint(0, 1))
        key1 += temp
    return(key1)

# conversão de um conjunto de bytes para bits
def bytes_to_bits(s):
    s = s.decode('ISO-8859-1')
    result = []
    for c in s:
```

```
        bits = bin(ord(c))[2:]
        bits = '00000000'[len(bits):] + bits
        result.extend([int(b) for b in bits])
    u=""
    for i in result:
        u = u + str(i)
    return u

# conversão de um conjunto de bits para bytes
def bits_to_bytes(bits):
    chars = []
    for b in range(int(len(bits) / 8)):
        byte = bits[b*8:(b+1)*8]
        chars.append(chr(int(''.join([str(bit) for bit in byte]), 2)))
    return ''.join(chars).encode('ISO-8859-1')

# passagem de um polinomio para um conjunto de bytes
def pack2(polinomio):
    check_List=isinstance(polinomio, list)
    if(not check_List):
        polinomio=polinomio.list()
        polinomioB= bytes(_Z(polinomio))
        compress = gzip.compress(polinomioB)
    else:
        polinomioB= bytes(_Z(polinomio))
        compress = gzip.compress(polinomioB)
    return compress

# passagem de bytes para polinómio Rq
def unpack(compress):
    unpack = gzip.decompress(compress)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Rq(newUnpack)

# passagem de bytes para polinómio Sq
def unpackSq(compress):
    unpack = gzip.decompress(compress)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Sq(newUnpack)

# cifragem igual à anterior com modificação do pack
def cifragem2(packed_public_key, packed_rm, key):
    lista = []
```

```
    for i in packed_rm:
        lista.append(i)
    packed_r = packed_rm[:(tamanho(lista,1))]
    packed_m = packed_rm[-(tamanho(lista,2)):]
    r = unpack(packed_r)
    m0 = unpack(packed_m)
    m1 = m0.lift()
    h = unpack(packed_public_key)
    c = Rq(r*h + m1)
    packed_ciphertext = pack2(c)
    aesgcm = AESGCM(key)
    nonce = geraNounce(12)
    m1_cifrado = aesgcm.encrypt(nonce, bytes(_Z(m1)), metadados)
    m1_cifrado += nonce
    return packed_ciphertext, m1_cifrado

# decifragem igual à anterior com modificação do pack
def decifragem2(packed_private_key, packed_ciphertext, key, m1_cifrado):
    tf = tamanho(packed_private_key,1)
    tfq = tamanho(packed_private_key,2)
    thq = tamanho(packed_private_key,3)
    packed_f =  packed_private_key[:tf]
    packed_private_key = packed_private_key[tf:]
    packed_fq = packed_private_key[:tfq]
    packed_hq = packed_private_key[tfq:]
    c = Rq(unpack(packed_ciphertext))
    f = Rq(unpack(packed_f))
    fq = unpack(packed_fq)
    hq = unpackSq(packed_hq)
    aesgcm = AESGCM(key)
    nonce = m1_cifrado[-12:]
    m1_cifrado = m1_cifrado[:-12]
    m1 = aesgcm.decrypt(nonce, m1_cifrado, metadados)
    y = []
    for i in m1:
        y.append(i)
    m1_novo = Rq(y).lift()
    r = Rq((c-m1_novo)*hq)
    for i in r.list():
        if i==1 or i==0 or i==-1:
            fail = 0
        else:
            fail = 1
            break
    for i in m1_novo.list():
        if i==1 or i==0 or i==-1:
            fail = 0
```

```
        else:
            fail = 1
            break
    packed_rm = pack2(r) + pack2(m1_novo.list())
    return packed_rm,fail

# funções principais:


# gera as chaves pública e privada
def geracao_chaves():
    fg_bits = rand_bits(19304)
    prf_key = rand_bits(256)
    packed_dpke_private_key, packed_public_key = gerar_chaves()
    packed_private_key = packed_dpke_private_key + bits_to_bytes(prf_key)
    return packed_private_key, packed_public_key

# encapsulamento do packed_rm
def encapsulate(packed_public_key, key):
    coins = rand_bits(19304)
    r, m = sample_fg()
    packed_rm = pack2(r) + pack2(m)
    shared_key = hash(bytes_to_bits(packed_rm))
    packed_ciphertext, m1_cifrado = cifragem2(packed_public_key, packed_rm,key)
    return shared_key, packed_ciphertext, m1_cifrado

# desancapsulamento obtendo a chave partilhada
def decapsulate(packed_private_key, packed_ciphertext, key, m1_cifrado):
    prf_key = packed_private_key[-32:]
    packed_dpke_private_key = packed_private_key[:
 ↪(len(packed_private_key)-len(prf_key))]
    tf = tamanho(packed_dpke_private_key,1)
    tfq = tamanho(packed_dpke_private_key,2)
    thq = tamanho(packed_dpke_private_key,3)
    packed_f =  packed_dpke_private_key[:tf]
    packed_dp = packed_dpke_private_key[tf:]
    packed_fq = packed_dp[:tfq]
    packed_hq = packed_dp[tfq:]
    packed_rm, fail =␣
 ↪decifragem2(packed_dpke_private_key,packed_ciphertext,key, m1_cifrado)
    shared_key = hash(bytes_to_bits(packed_rm))
    concat = bytes_to_bits(prf_key) + bytes_to_bits(packed_ciphertext)
    random_key = hash(concat)
    if fail == 0:
        return shared_key
    else:
        return random_key
```

De seguida são apresentados os resultados obtidos com recurso às funções anteriores.

[6]:
```
key = os.urandom(32)
packed_private_key, packed_public_key = geracao_chaves()
print("Chave pública=", packed_public_key, "\n")
print("Chave privada=", packed_private_key, "\n")
shared_key, packed_ciphertext, m1_cifrado = encapsulate(packed_public_key, key)
print("Shared key cifragem =", shared_key, "\n")
shared_key2 = decapsulate(packed_private_key, packed_ciphertext,key, m1_cifrado)
print("Shared key decifragem=", shared_key2, "\n")
print("Shared key cifragem = Shared key decifragem", shared_key==shared_key2)
```

```
Chave pública= b'\x1f\x8b\x08\x00\xb8I\x99`\x02\xffUP\t\x02\xc0 \x08\x82\xff\x7f
z-\x01\xa9\xad\x99\xe98$I\x00\x9c\x85{\xc4\r>\xdd\xeb\xb3\xdd\xf1\x1f\xf58\x99*\
xb4\xd2p\x01(\xac\xc9\x99\xca~\xcc\x11\x12\x14|~\xe1\xe0\x16\x91\xfb\xc2>\xbdz[\
xbb\xeeP\xa5\xd9S@Y\x95d\xc1\xee \x94<\x05\xb3#\xba\xc1\xc7}\x99a\xa8l\xf4\x99\x
1e6x\x0c\x91\xc1r\xcb\x1d\xbb\x84Fp\x8f\xb9\x8c\xdbZ\x10D{\xc2\x07u5o\x93\xfc\x0
1\x00\x00'

Chave privada= b"\x1f\x8b\x08\x00\xb8I\x99`\x02\xffm\x90\x01\x0e\xc0 \x0c\x02{\x
ff\xff\xf4\xb2\xd8M\xa05\xd1()\x08TU\xd1\xbb\x17\xe3<W\xca!\xecn\x02\xf6T\x12\x0
b]u\x0f\xc8T\x80\x17\x84;\xb3\xb0E\x17\xd1'\x12|\x89i\x88\xe9s\x98\x0f\x93\x12(:
!\x89\xe2U\x1cx\x8fV\x08\x7f\t\xeb\xdc\xfd='\xd82 \xa6#\xd6\x03\x8c^Cq\xfc\x01\x
00\x00\x1f\x8b\x08\x00\xb8I\x99`\x02\xffM\x91\t\x0e\x00!\x0c\x02\xe1\xff\x9f^W\x
87Rc\x8c\xf6\xe0\xa8\xb2\xe4\xb3\xcey\xf7]\x16\x91$\xfe\x9aIL\x8e\xb8\xdf\xed\x0
f5\xfc2\xe4\xb9\xa8u \xa7\x01\x14\xd3\xc9\xe32-6\x0f\xc3\xa3$\xa6:\x08\xd9\xe0\x
07s\xb4\x04\xba\nG\x19\x930\x1d^\xf5XX\xe8\xf5\x8e\xb6\xbc@t\xe5\xd1\xbb\xa7;\xa
aT\xd5\xf1\xe8`.S\xe5\xda\xbcR\xf4\xbb_8S\xd4\xf8\x07\xba\x96\xd0\xf1\x01%at\xa7
\xfc\x01\x00\x00\x1f\x8b\x08\x00\xb8I\x99`\x02\xffUQ\x8b\x12\x800\x08b\xff\xff\x
d3-y\xb9\xbav:\x11\xd0pp\xdf\xff\xc1\r\xf0\x1f\x93\xdf\x80\xdf\x9cg\x02'S\x82p\x
ecvrBE\x0c)D.\x1e\xe3\xc4.Y\xf6U\xd5-tT\x84\xb3e\xce\xdc\xee\x94\xa2\xf5\xc9\x8b
\x9a\x00\xe24&\xec8^y\xfb\x18P \xc1\xae\xa3\xc58\xec\xc2\xf6b\x03\x01\x96VY,\xf9
\x0e\xe7\x9f\x91\xbbN\x85L\xd0:\xde\xc6\x1a\xdf\xbe\x06\xf1\x01\xf9\x87\x8d{\xfc
\x01\x00\x00\xa9\x85T\xd8\x06\xaa \xe3K\xf4\x96`\xae\xaf\x9d\xcf\xa58\x9a{\x9c\x
9b\x98\xda\x99\x00\xbc\x98\x88\xf3\xcf^"

Shared key cifragem = -4300721993484117417

Shared key decifragem= -4300721993484117417

Shared key cifragem = Shared key decifragem True
```