NTRU-TP2

May 2, 2022

1 Trabalho prático 2 - Estruturas Criptográficas

Autores: Ariana Lousada (PG47034), Cláudio Moreira (PG47844)

Grupo 12

2 NTRU

Passou-se para a implementação do protótipo NTRU (KEM IND-CPA e PKE IND-CCA). De seguida, serão apresentadas os resultados das resoluções de ambos os algoritmos, tendo por base o documento *ntru.pdf*.

2.1 Passively secure DPKE

Começou-se por implementar um algoritmo que permitisse uma segurança IND-CPA, isto é, contra *Chosen PlainText Atacks*. Baseamo-nos para essa implementação no Passively secure DPKE do pdf fornecido pelo docente no pdf referido anteriormente. Sendo assim, implementaram-se as seguintes funções auxiliares:

- ***pol_ternario***: Ao ser executado cria uma lista com elementos entre o intervalo -1 a 1. Posteriormente realiza uma devolução (com base na lista calculdada) de um polinómio ternário.
- ***pol_ternario_16_valores***: Através da função pol_ternário, realiza a criação de uma lista com elementos entre -1 e 1. No entanto, ao realizar essa criação adiciona Q/16-1 elementos iguais a 1 e Q/16-1 elementos iguais a -1 (sendo Q uma das constantes do NTRU).
- ***pol_ternario_reunião***: Utiliza as funções descritas acima para cirar os polinómios necessários para obtenção das chaves.
- ***pack***: Recebe como argumento um polinómio e codifica-o em bytes.
- ***unpack***: Realiza o contrário da função ***pack*** devolvendo um polinómio do tipo Rq.
- ***unpackSq***:Realiza o contrário da função ***pack*** devolvendo um polinómio do tipo Sq.
- ***chave_publica***:Através dos polinómios criados retorna um h e um hq.

De seguida passou-se para a implementação do KEM através das funções principais gerador_chaves, cifrar, decifrar.

- ***gerador_chaves***: Realiza a geração das chaves públicas e privadas sendo a chave pública utilizada para cifrar a mensagem e a chave privada para decifrar o criptograma.

- ***cifrar***: Realiza a cifragem da mensagem através da chave pública e da mensagem.
- ***decifrar***: Realiza a decifragem do criptograma com recurso à chave privada e ao criptograma

[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]:
```
N = 509
Q = 2048
T = N//4


_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)))


metadados = os.urandom(16)
listanouce = []
def geradorNounce(tamanhoNounce):
    nounce = os.urandom(tamanhoNounce)
    if not (nounce in listanouce):
        listanouce.append(nounce)
        return nounce
    else:
        geradorNounce(tamanhoNounce)
```

```
def pol_ternario(n=N,t=T):
    valor = [rn.choice([-1,1]) for i in range(t)] + [0]*(8*(n-1)-t)
    rn.shuffle(valor)
    return Rq(valor)

def pol_ternario_16_valores(n=N,t=T):
    Q_tam = Q//16 -1
    h = (30*(n-1))-2*Q_tam
    valor1 = [rn.choice([1]) for i in range(t)] + [0]*(Q_tam-t)
    valor2 = [rn.choice([-1]) for i in range(t)] + [0]*(Q_tam-t)
    valor3 = [rn.choice([0]) for i in range(t)] + [0]*(h-t)
    valor = [*valor1,*valor2,*valor3]
    rn.shuffle(valor)
    return Rq(valor)

def pol_ternario_reunião(n=N, t=T):
    pol1 = pol_ternario(n,t)
    pol2 = pol_ternario_16_valores(n,t)
    return pol1,pol2

def tamanho(stringB, numberS):
    contador = 2
    auxContador = 1
    i = 0
    while i < len(stringB):
        if numberS == auxContador:
            i = i + 2
            while (i < len(stringB)) and (stringB[i] != 120 or stringB[i + 1] !
 ↪= 1):
                contador = contador + 1
                i = i + 1
            auxContador = auxContador + 1

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

def pack(polinomio):
    check_List=isinstance(polinomio, list)
    if(not check_List):
        polinomio=polinomio.list()
        polinomioBytes= bytes(_Z(polinomio))
        comprimir = zlib.compress(polinomioBytes,1)
```

```
    else:
        comprimir = zlib.compress(bytes(_Z(polinomio)),1)
    return comprimir

def unpack(pack):
    unpack = zlib.decompress(pack)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Rq(newUnpack)

def unpackSq(pack):
    unpack = zlib.decompress(pack)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Sq(newUnpack)

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)

def gerador_chaves():
    f, g = pol_ternario_reunião()
    fq = f.inverse_of_unit()
    h, hq = chave_publica(f,g)
    chave_privada_1= pack(f)
    chave_privada_2 =pack(fq)
    chave_privada_3 =pack(hq)
    chave_privada_packed = chave_privada_1+ chave_privada_2+chave_privada_3
    chave_publica_packed = pack(h)
    return (chave_privada_packed, chave_publica_packed)

def cifrar(packed_chave_publica, 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_chave_publica)
    c = Rq(r*h + m1)
    packed_texto_cifrado= pack(c)
    aesgcm = AESGCM(key)
```

```
    nonce = geradorNounce(12)
    m1_cifrado = aesgcm.encrypt(nonce, bytes(_Z(m1)), metadados)
    m1_cifrado += nonce
    return packed_texto_cifrado, m1_cifrado

def decifrar(packed_chave_privada, packed_texto_cifrado, key, m1_cifrado):
    tf = tamanho(packed_chave_privada,1)
    tfq = tamanho(packed_chave_privada,2)
    thq = tamanho(packed_chave_privada,3)

    packed_f =  packed_chave_privada[:tf]
    packed_chave_privada = packed_chave_privada[tf:]
    packed_fq = packed_chave_privada[:tfq]
    packed_hq = packed_chave_privada[tfq:]

    c = Rq(unpack(packed_texto_cifrado))
    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())]
    falha = 0
    for i in r.list():
        if i==1 or i==0 or i==-1:
            falha = 0
        else:
            falha = 1
            break
    for i in m1_novo.list():
        if i==1 or i==0 or i==-1:
            falha = 0
        else:
            falha = 1
            break
    resultado = packed_rm[:102] + packed_rm[-102:]
    return resultado, falha
```

3 Resultados obtidos

Utilizando 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, 1, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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, 1, 1, 0, 1, 1,␣
 ↪1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1,
           1, 0, 1, 1, 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, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1,␣
 ↪1, 1])

print("Mensagem calculada :", mensagem,"\n")

packed_chave_privada, packed_chave_publica = gerador_chaves()
print("Chave privada calculada :", packed_chave_privada, "\n")
print("Chave pública calculada :", packed_chave_publica, "\n")

texto_cifrado, m1_cifrado= cifrar(packed_chave_publica, mensagem.list(), key)
print("Texto cifrado : ", texto_cifrado,"\n")


texto_limpo,fail = decifrar(packed_chave_privada, texto_cifrado,key, m1_cifrado)
print("Texto Decifrado :", RT(texto_limpo),"\n")

print("Verificação do Texto Limpo = Resultado da decifragem ->", texto_limpo==␣
 ↪mensagem.list())
```

```
Mensagem calculada : w^203 + w^202 + w^201 + w^199 + w^198 + w^197 + w^196 +
w^195 + w^194 + w^193 + w^191 + w^190 + w^189 + w^188 + w^187 + w^186 + 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^138 + 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^107 + 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^62 + w^61 +
w^60 + w^59 + w^58 + w^57 + w^56 + w^55 + w^54 + w^53 + 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^15 + 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 calculada : b'x\x01]P\t\x0e\x800\x08c\xff\xff\xb4\xf4\x829M\xdczB\
xac\xaa:\xfd\xea9\xb9\x18C\xb3n\x93\x0ck\xdb\x1b#(h\xa7\x0ewN\x97\xc60\xf1\xc4`\
x11\xc5\xbc\x81M\xa1\xb4\x01\xbfV\xbcZ\xd2\x18<E\xda\\F\n\xac\xb9\x82\x1ab\x16<5
o>\x18\xddx\x9ag\xb9\x80\x97\xa5\x90 LV_\xd3$\xf1cd\x8a\x138\x0c\xcf\x00\xb6\x8d
6\r0\xaf\xe5\x9a\x8d\xc2[\xe0\x80a\xa8\xfc\xe4\xae>\x1fl\x7f\x00jx\x01M\x8e\x0b\
x16\xc4 \x08\x03\xf5\xfe\x97^f\x12\xec\xf6Y\xc4\xfc\xe0\x9e{9\xe7\xcc?\xdd\xf4\x
fe\\@\x10 \xdfU\xb5\x9eg\x19\r\xb2\xe85%J\xec\xf3\x93\xe5\x8b\xb2\xb0\xd1\x9dV\x
acS\x89w\t\x16H[\xbb\xef$\x19\x84a\xc7\xba s\xf8\x08G\x11\xd6\xaa\x81nY\xbdo!i\x
b2b{\xa2l\xb0A\r\xceT\x92\x00\xf0lXz\x99\xc7\x9a\x85\xa0\x0b\xfd\x19\xf0\x11\x02
#\x9c\x02\xd0\x83 \x8a\x0e\xe2\xf1\xc2"[dp(\xe5\x95p\xcd\xb1\x0c\xf3\x03\xfbz\x0
0\xf8x\x01M\x8e\x81\x11\xc4 \x0c\xc3`\xff\xa5\x1b\xc9N[\xfe\xaf\x84D\x16\x9csg\x
1dV\xb6\xa9\xef\xfc\xfcO\x87\xa1@\x9a|m\xa4\xbf`$h:FV\x93\xda\xc1\xd9\x89\x8a\xe
8|oI\xf0\x9b\xfd\xd3U\xfaB_\x99J\xbd\\\xc4\xab\xe8\x13&E?7p\xc8\xd5\x91\xb5\xbdO
\x11\r\xcc$\x869\x13af<\x8e:\xedKb\xdeSaH\x96Q7\x95\xc8*!\xb05\x85\x12>\x9b\xa9\
x14\xb36\x01\x94\x85\xa7Xxo\x91}s5\xea\xfcHX\xa2\xf1\x9c\xfb\x00\xf6<\x00\xf7'

Chave pública calculada : b'x\x01U\x8e\t\x0e\xc40\x08\x03\x93\xff\x7fz=c\xe8*\xa
d\x14\xc0\x17\x9c{\xef9\'oJ\xbb\x14F\x01\x10\xe0!\xdb\xbc\xa4\xfe\xaat\xc6\x8cl\
x02\xe8fL\x01\xa6\x14\xb4\xa8\xdb\xae\x97\xe0P\xf4\x1f\xb1l\x90\xa8SE$|_H\xbd\x9
5O\xbf\xf1[9\x8a\xbeg\xd4\xd2@\x99\xdd\x0f\x8dn>\rr<\x06|L3\xe6\n\x06l\xcf(` lIe
\xf6"\x12n\xe9R\xf6O\xc8(3\xdb\xf5\xf1\x84Y\xe4\xc2W\x15\xa6\x17\x04\xd6W\r\xa9\
xf9\x7f\x00u\x01\x01'

Texto cifrado :  b'x\x01M\x8f\x0b\x16\xc4 \x08\x03\xf1\xfe\x97^f\x12\xedj\x1f \x
f9@g\xe6\x9c3g\xf6lJ\xb9\x89\xd6K\x7f\xf0\x12\xfb)z:\xd5h\xb8u\xc1R\x02\x89\xea{
\xb5L_\xa8\x1c:w\x17*%\xe4V\x82\x1a\xcb\x14^\xc8\xf3~!\x16\x19_?\x9d\\\x0c\xa7\x
148\xe4\x15\x7f\xbaJ\x19\xa2\x8e\x10\x06\x08\xe42\xe5\xf9\xd4\r\x1f0s\xd7\x90\x9
bI\xe9\x7f\xf4\xcb\x8edc\xd4\x18xjVX\xb4\xbb]\x84A\xa13A\xbf\x06\x1f\xc1(\xf7\xa
2!\xe8\xd3\xf4\x03\xf0g\x00\xef'

Texto Decifrado : w^203 + w^202 + w^201 + w^199 + w^198 + w^197 + w^196 + w^195
+ w^194 + w^193 + w^191 + w^190 + w^189 + w^188 + w^187 + w^186 + 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^138 + 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^107
+ 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^62 + w^61 +
w^60 + w^59 + w^58 + w^57 + w^56 + w^55 + w^54 + w^53 + 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^15 + w^14 +
w^13 + w^12 + w^10 + w^9 + w^7 + w^6 + w^5 + w^4 + w^3 + w^2 + 1

Verificação do Texto Limpo = Resultado da decifragem -> True
```

3.1 Strongly secure KEM

De seguida passou-se para a implementação de um algoritmo que garanta uma segurança IND-CCA, isto é, contra *Chosen Ciphertext Attacks*.Baseamo-nos para essa implementação no Strongly secure KEM fornecido pelo docente no pdf referido anteriormente.

Tal como numa fase anterior, começou-se por definir funções auxiliares. No entanto estas são iguais às referidas anteriormente exceto na pack e unpack. Para além disso também se implementaram as seguintes funções novas:

- ***bytes_para_bits***: Realiza uma transformação de bytes para bits.
- ***bits_para_bytes***: Realiza uma transformação de bits para bytes.
- ***bits_aleat***: Gera bits aleatórios consoante um tamanho definido.

De seguida, implementaram-se as funções principais:

- ***gerar_chaves_2***: Realiza a geração de chaves publicas e privadas.
- ***encapsulate***: Realiza a cifragem do packed_rm, devolvendo a chave partilhada e o packed_texto_cifrado.
- ***decapsulate***: Realiza o oposto da função de encapsular devolvendo no final a chave partilhada.

[4]:
```
def tamanho(stringB, numberS):
    contador = 10
    auxContador = 1
    i = 0
    while i < len(stringB):
        if numberS == auxContador:
            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  ):
                contador = contador + 1
                i = i + 1
            auxContador = auxContador + 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 ):
            auxContador = auxContador + 1
        if auxContador > numberS:
            break
    return contador

def gerar_chaves():
    f, g = pol_ternario_reunião()
    fq = f.inverse_of_unit()
    h, hq = chave_publica(f,g)
    chave_privada_1= pack2(f)
```

```
    chave_privada_2 =pack2(fq)
    chave_privada_3 =pack2(hq)
    chave_privada_packed = chave_privada_1+ chave_privada_2+chave_privada_3
    chave_publica_packed = pack2(h)
    return (chave_privada_packed, chave_publica_packed)

def bits_aleat(p):
    chave1 = ""
    for i in range(p):
        temp = str(rn.randint(0, 1))
        chave1 += temp
    return(chave1)

def bytes_para_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

def bits_para_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')

def pack2(polinomio):
    list_checker=isinstance(polinomio, list)
    if(not list_checker):
        polinomio=polinomio.list()
        polinomioB= bytes(_Z(polinomio))
        compress = gzip.compress(polinomioB)
    else:
        polinomioB= bytes(_Z(polinomio))
        compress = gzip.compress(polinomioB)
    return compress

def unpack(compress):
    unpack = gzip.decompress(compress)
    newUnpack=[]
    for i in unpack:
```

```
        newUnpack.append(i)
    return Rq(newUnpack)

def unpackSq(compress):
    unpack = gzip.decompress(compress)
    newUnpack=[]
    for i in unpack:
        newUnpack.append(i)
    return Sq(newUnpack)

def cifragem2(packed_chave_publica, packed_rm, chave):
    lista_in = []
    for i in packed_rm:
        lista_in.append(i)
    packed_r = packed_rm[:(tamanho(lista_in,1))]
    packed_m = packed_rm[-(tamanho(lista_in,2)):]
    r = unpack(packed_r)
    m0 = unpack(packed_m)
    m1 = m0.lift()
    h = unpack(packed_chave_publica)
    c = Rq(r*h + m1)
    packed_texto_cifrado = pack2(c)
    aesgcm = AESGCM(chave)
    nonce = geradorNounce(12)
    m1_cifrado = aesgcm.encrypt(nonce, bytes(_Z(m1)), metadados)
    m1_cifrado += nonce
    return packed_texto_cifrado, m1_cifrado

def decifragem2(packed_private_key, packed_ciphertext, key, m1_cifrado):
    t_chave1 = tamanho(packed_private_key,1)
    t_chave2 = tamanho(packed_private_key,2)

    packed_f =  packed_private_key[:t_chave1]
    packed_private_key = packed_private_key[t_chave1:]
    packed_fq = packed_private_key[:t_chave2]
    packed_hq = packed_private_key[t_chave2:]

    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

def gerar_chaves_2():
    fg_bits = bits_aleat(19304)
    prf_chave = bits_aleat(256)
    packed_dpke_chave_privada, packed_chave_publica= gerar_chaves()
    packed_chave_privada = packed_dpke_chave_privada +␣
 ↪bits_para_bytes(prf_chave)
    return packed_chave_privada, packed_chave_publica

def encapsulate(packed_chave_publica, chave):
    coins = bits_aleat(19304)
    r, m = pol_ternario_reunião()
    packed_rm = pack2(r) + pack2(m)
    chave_publica = hash(bytes_para_bits(packed_rm))
    packed_texto_cifrado, m1_cifrado = cifragem2(packed_chave_publica,␣
 ↪packed_rm,chave)
    return chave_publica, packed_texto_cifrado, m1_cifrado

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_para_bits(packed_rm))
    concat = bytes_para_bits(prf_key) + bytes_para_bits(packed_ciphertext)
    random_key = hash(concat)
    if fail == 0:
        return shared_key
    else:
        return random_key
```

4 Resultados obtidos

[5]:
```
chave = os.urandom(32)
packed_chave_privada, packed_chave_publica = gerar_chaves_2()
print("Chave pública=", packed_chave_publica ,"\n")
print("Chave privada=", packed_chave_privada, "\n")
chave_publica, packed_texto_cifrado, m1_cifrado =␣
 ↪encapsulate(packed_chave_publica, chave)
print("Chave partilhada cifragem =", chave_publica, "\n")
chave_publica_2 = decapsulate(packed_chave_privada, packed_texto_cifrado,chave,␣
 ↪m1_cifrado)
print("Chave partilhada decifragem=", chave_publica_2, "\n")
print("Verificação : Shared key cifragem = Shared key decifragem ->",␣
 ↪chave_publica==chave_publica_2)
```

```
Chave pública= b'\x1f\x8b\x08\x00\xaf@pb\x02\xffeP\x01\x0e\x00!\x08\x82\xff\x7f\
xfa*\x01\xads\xad\xa6\x81( Np\xc5y\xf6\xb53UX5\xa70\xfa\x1c\x11Q\xb8\x82\xa8K\x1
5\xc9\x81/RS\x80\x80\x15)4J-h\xcdV\x1ac\t\xc6\xe6\x02V\xee\xaf\xf0\x06ufx\xf5\x9
1U\x04\x8e\r\xbf\xa9\x18\x0f\xb3"m\x92\xcc\xbc\x8ao\xc3\xd8i\xe5^(\xa4\xcc\x87\x
96\xbd\xba\xe4\\\xa3\xa4M\x0c\xf7\xb4\x1fE?7\xbb\xfc\x01\x00\x00'

Chave privada= b'\x1f\x8b\x08\x00\xaf@pb\x02\xffm\x91\x81\x0e\x00!\x08B}\xff\xff
\xd3\xb7\xd6IH\xb9\xd5\xd4\x10\xd1\x8a\xa2\xb6\xd1\xce\x8eV^\x19\xcc[61\xcdA\x87
O\xa3/\x83Xo\xd4\x85&\x12F\x1doE\xbbr\x1cK\x97hI\xda\xba\x84\xe2\x13\x9dw"~V\x87
.\xf2\x01\xdbaJ\xc8}%\xc5\x0c@\xeb\x128\xdb1\xbe\xef\xc7\x8d\x19\xf8\x00T\xae\x0
7\xef\xf9\x01\x00\x00\x1f\x8b\x08\x00\xaf@pb\x02\xffUP\t\x0e\xc0
\x0c\x82\xff\x7f\xdah\x81b\x16\xd7C@Z\x00 1\xdf\xcdnP/\xd5+^\xe4\xc3\xdeT\xb8\xa
1\xd0\x1a\xd5\x1b\x90n\xcd\xa0\xf4\xaaAF\x1b\x92\xaf\x0eJ]u\x1c\xaeE\nP\xc6\xf5\
xac-\x19\xba\xbf\x9d{(\x96`_XT\xc3q\xfd\x7f\xb3\xc1\\\xb6\x04w\x07y\xc4^\xc6\x87
\xed#\x87\xfc\xe7\x8a\x89\x102n\x16\xce\xda7\x16\xc0Eo\xaa\xf6\x84\x03[\xff\xb0\
xf8\xfd\x01\x00\x00\x1f\x8b\x08\x00\xaf@pb\x02\xffU\x90\x81\x12\x80 \x0cB\xe1\xf
f\x7f:\xd3\x07\xd3\xba\xd36\x1a\x03d\xad\xd7:\xcf\xbem:\xe6{\x9f\x07v\xae\xbf\xb
3\x01\xd3,\x01\x98\x19g\xd6TpP\xc2\xd5f\xf6X\xd1\xa0\xa1\x8f\xa6Q\x08}\xce\x08\x
8a\xa8\x81f+f:\x86\xd6z\xe9\x9a\xba\x86\xab\xa5Bh\x8d\x9f\xda~R\xcc\xf0\xed^W\xb
6\t\x8c\x99\xe0\r*?\xa5\xa5g\xf7\x98u\x03\x8a\xa6G\xedx\xb8\x13\xf8\x00\xad\xfb\
n\r\xfa\x01\x00\x00\x07`r\x93\xb5J\x93\xac\xb9\xa1\\k\x86\x85P\xd3\x98\xe2\xac\x
```

```
1f\x96\x97 \xb6\x0c9\xfb\t|\x19F\xe9'

Chave partilhada cifragem = -1483479879066453196

Chave partilhada decifragem= -4874812219208855167

Verificação : Shared key cifragem = Shared key decifragem -> False
```