BIKE - Bit Flipping Key Encapsulation

Descrição do Problema

Este notebook tem como objetivo a implementação do algoritmo BIKE de um KEM (Key Encapsulation Mechanism) que seja IND-CPA seguro, e um PKE (Public Key Encryption) que seja IND-CCA seguro. Para tal, será utilizado o algoritmo BIKE (Bit Flipping Key Encapsulation), que é um algoritmo de criptografia pós-quântica, que utiliza um código de correção de erros como base. Foi utilizada a especificação mais recente do BIKE que pode ser encontrada [aqui].

Objetivos

De forma resumida, os objetivos deste trabalho prático são:

- **Criação de um protótipo em Sagemath para o algoritmo BIKE.**
- **Pretende-se implementar um KEM, que seja IND-CPA seguro, e um PKE que seja IND-CCA seguro.**

Resolução do Problema

In [38]:

```
from sage.all import *
# noinspection PyUnresolvedReferences
from sage.modules.vector_mod2_dense import Vector_mod2_dense
```

Parâmetros

Parâmetros para o nível de segurança 1

In [5]:

```
r = 257  # 12323  # Comprimento do bloco (block length)
n = r * 2  # Comprimento do código (code length)
w = 142  # Peso da linha (row weight)
t = 134  # Peso do erro (error weight)
l = 256  # Comprimento do segredo partilhado (shared secret size) | NOTA: Este parametro
é fixo para todos os níveis de segurança

# BGF decoder parameters - nível de segurança 1
NbIter = 5  # Número de iterações do decoder
tau = 3  # Threshold Gap | TODO: Confirmar se este comentário está correto
threshold = lambda S, _i: max(floor(0.0069722 * S + 13.530), 36)  # Threshold function
```

Também poderiam ter sido utilizados outros parâmetros, conforme o nível de segurança pretendido, como se pode observar na tabela abaixo:

Nível de Segurança	r	w	t	DFR
1	12,323	142	134	2^−128
2	24,659	206	199	2^−192
3	40,973	274	264	2^−256

In [6]:

```
F = GF(2)
```

```
M = F ** l  # Message space

R = PolynomialRing(F, 'x')
x = R.gen()
Rr = QuotientRing(R, R.ideal(x ** r - 1))  # Polynomial ring R / (x^r - 1)

KK = F ** l  # Private key space

print("Message space M:   ", M)
print("Shared key space K:", KK)
print("Polynomial ring R: ", R)
print("Quotient ring Rr:  ", Rr)

MElement = type(M.random_element())  # Basicamente binário
RElement = type(Rr.random_element())  # Elemento de Rr
KElement = type(KK.random_element())  # Basicamente binário

print("MElement:", MElement)
print("RElement:", RElement)
print("KElement:", KElement)
```

```
Message space M:    Vector space of dimension 256 over Finite Field of size 2
Shared key space K: Vector space of dimension 256 over Finite Field of size 2
Polynomial ring R:  Univariate Polynomial Ring in x over Finite Field of size 2 (using GF
2X)
Quotient ring Rr:   Univariate Quotient Polynomial Ring in xbar over Finite Field of size
2 with modulus x^257 + 1
MElement: <class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
RElement: <class 'sage.rings.polynomial.polynomial_quotient_ring.PolynomialQuotientRing_g
eneric_with_category.element_class'>
KElement: <class 'sage.modules.vector_mod2_dense.Vector_mod2_dense'>
```

Funções auxiliares

In [37]:

```
def generate_sparse(weight: int, size: int) -> RElement:
    """
    Gera um sparse vector.
    Entrada: weight - número de elementos não nulos (Hamming weight)
             size - tamanho do vector
    Saída: elemento de Rr
    """
    while True:
        # Generate a random list of size 'size' with 'weight' non-zero elements
        sparse_rep = [0] * size
        for _ in range(weight):
            rand_index = randint(0, size - 1)
            while sparse_rep[rand_index] != 0:
                rand_index = randint(0, size - 1)

            sparse_rep[rand_index] = 1

        assert sum(sparse_rep) == weight
        return Rr(sparse_rep)
```

In [15]:

```
def bytes_to_bits(b: bytes) -> list:
    assert type(b) == bytes

    return [int(bit) for byte in b for bit in bin(byte)[2:].zfill(8)]
```

In [16]:

```
def expand(lis: list, size: int) -> list:
    assert type(lis) == list

    return lis + [0] * (l - len(lis))
```

In [17]:

```python
# noinspection PyPep8Naming
def R_to_bytes(r: RElement) -> bytes:
    assert type(r) == RElement

    return bytes(r.list())


# noinspection PyPep8Naming
def bytes_to_R(b: bytes) -> RElement:
    assert type(b) == bytes

    return Rr(list(b))


assert bytes_to_R(R_to_bytes(Rr([1, 0, 1]))) == Rr([1, 0, 1])


# noinspection PyPep8Naming
def M_to_bytes(m: MElement) -> bytes:
    assert type(m) == MElement

    bits = m.list()
    bit_string = ''.join(str(bit) for bit in bits)  # convert the list of bits to a string

    return int(bit_string, 2).to_bytes(len(bits) // 8, byteorder='big')


# noinspection PyPep8Naming
def bytes_to_M(b: bytes) -> MElement:
    assert type(b) == bytes

    bytess = expand(bytes_to_bits(b), l)

    assert len(bytess) == l

    return M(bytess)


assert bytes_to_M(M_to_bytes(M([1, 0] * (l // 2)))) == M([1, 0] * (l // 2))
```

In [18]:

```python
def getHammingWeight(m: MElement) -> int:
    acc = 0
    for i in m:
        if i == 1:
            acc += 1

    return acc


assert getHammingWeight(M([1, 0] * (l // 2))) == l // 2
```

In [19]:

```python
def xor(a: MElement, b: MElement) -> MElement:
    assert len(a) == len(b)
    return M([a[i] ^ b[i] for i in range(len(a))])
```

Funções de Hash necessárias

Função H

In [20]:

```python
# noinspection PyPep8Naming
def H(m: MElement) -> (RElement, RElement):
    assert type(m) == MElement
    # TODO: Migrate this to use AES256-CTR PRNG if needed

    e0 = generate_sparse(t, r)
    e1 = generate_sparse(t, r)

    return e0, e1


H(M([1, 0] * (l // 2)))
```

Out[20]:

```
(xbar^256 + xbar^255 + xbar^252 + xbar^251 + xbar^250 + xbar^249 + xbar^245 + xbar^241 +
xbar^240 + xbar^234 + xbar^231 + xbar^230 + xbar^229 + xbar^228 + xbar^226 + xbar^225 + x
bar^224 + xbar^220 + xbar^217 + xbar^216 + xbar^213 + xbar^210 + xbar^205 + xbar^202 + xb
ar^201 + xbar^200 + xbar^197 + xbar^196 + xbar^195 + xbar^194 + xbar^193 + xbar^192 + xba
r^191 + xbar^187 + xbar^186 + xbar^184 + xbar^183 + xbar^182 + xbar^181 + xbar^180 + xbar
^179 + xbar^178 + xbar^177 + xbar^175 + xbar^174 + xbar^173 + xbar^170 + xbar^168 + xbar^
165 + xbar^162 + xbar^160 + xbar^157 + xbar^156 + xbar^155 + xbar^154 + xbar^152 + xbar^1
51 + xbar^149 + xbar^148 + xbar^146 + xbar^144 + xbar^143 + xbar^142 + xbar^140 + xbar^13
7 + xbar^136 + xbar^134 + xbar^133 + xbar^130 + xbar^128 + xbar^125 + xbar^124 + xbar^123
+ xbar^122 + xbar^120 + xbar^119 + xbar^118 + xbar^109 + xbar^107 + xbar^106 + xbar^105 +
xbar^104 + xbar^99 + xbar^98 + xbar^97 + xbar^95 + xbar^94 + xbar^92 + xbar^90 + xbar^88
+ xbar^87 + xbar^86 + xbar^79 + xbar^77 + xbar^73 + xbar^72 + xbar^70 + xbar^69 + xbar^68
+ xbar^67 + xbar^66 + xbar^65 + xbar^63 + xbar^61 + xbar^59 + xbar^58 + xbar^57 + xbar^56
+ xbar^54 + xbar^52 + xbar^48 + xbar^47 + xbar^46 + xbar^45 + xbar^44 + xbar^43 + xbar^42
+ xbar^39 + xbar^38 + xbar^36 + xbar^34 + xbar^33 + xbar^26 + xbar^21 + xbar^20 + xbar^19
+ xbar^17 + xbar^16 + xbar^14 + xbar^13 + xbar^11 + xbar^8 + xbar^6 + xbar^5 + xbar^4 + 1
,
 xbar^256 + xbar^254 + xbar^253 + xbar^252 + xbar^251 + xbar^250 + xbar^246 + xbar^245 +
xbar^242 + xbar^241 + xbar^240 + xbar^239 + xbar^237 + xbar^235 + xbar^233 + xbar^230 + x
bar^224 + xbar^222 + xbar^221 + xbar^220 + xbar^219 + xbar^218 + xbar^216 + xbar^213 + xb
ar^212 + xbar^211 + xbar^210 + xbar^208 + xbar^207 + xbar^206 + xbar^204 + xbar^203 + xba
r^202 + xbar^200 + xbar^198 + xbar^197 + xbar^196 + xbar^195 + xbar^194 + xbar^192 + xbar
^189 + xbar^187 + xbar^186 + xbar^182 + xbar^179 + xbar^177 + xbar^176 + xbar^175 + xbar^
173 + xbar^171 + xbar^169 + xbar^168 + xbar^167 + xbar^166 + xbar^164 + xbar^163 + xbar^1
61 + xbar^159 + xbar^157 + xbar^156 + xbar^154 + xbar^151 + xbar^146 + xbar^143 + xbar^14
1 + xbar^137 + xbar^134 + xbar^133 + xbar^132 + xbar^131 + xbar^129 + xbar^128 + xbar^127
+ xbar^125 + xbar^124 + xbar^123 + xbar^120 + xbar^119 + xbar^117 + xbar^116 + xbar^115 +
xbar^110 + xbar^109 + xbar^107 + xbar^106 + xbar^105 + xbar^104 + xbar^103 + xbar^102 + x
bar^101 + xbar^97 + xbar^96 + xbar^94 + xbar^93 + xbar^91 + xbar^88 + xbar^85 + xbar^83 +
xbar^82 + xbar^75 + xbar^74 + xbar^72 + xbar^71 + xbar^70 + xbar^68 + xbar^66 + xbar^62 +
xbar^61 + xbar^59 + xbar^57 + xbar^55 + xbar^54 + xbar^51 + xbar^49 + xbar^46 + xbar^43 +
xbar^41 + xbar^39 + xbar^35 + xbar^34 + xbar^32 + xbar^28 + xbar^27 + xbar^25 + xbar^23 +
xbar^19 + xbar^18 + xbar^14 + xbar^13 + xbar^12 + xbar^10 + xbar^9 + xbar^6 + xbar^4 + xb
ar + 1)
```

Função L

In [21]:

```python
# noinspection PyPep8Naming
def L(e0: RElement, e1: RElement) -> MElement:
    assert type(e0) == RElement
    assert type(e1) == RElement

    # Apply the SHA384 hash function to the concatenation of e0 and e1
    from hashlib import sha384

    m = sha384()

    m.update(R_to_bytes(e0))

    m.update(R_to_bytes(e1))

    digest = m.digest()

    # Concat all the bits of the digest into a list of bits
```

```
    digest = bytes_to_bits(digest[-l // 8:])  # We only need l bits (l / 8 bytes)

    return M(digest)  # Returns the MElement corresponding to the digest


L(Rr([1, 0, 1]), Rr([1, 0, 1]))
```

Out[21]:

```
(0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1,
0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0,
0, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1,
1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0,
0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0,
0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0,
1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0,
0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1)
```

Função K

In [22]:

```
# noinspection PyPep8Naming
def K(m: MElement, c0: RElement, c1: MElement) -> KElement:
    assert type(m) == MElement
    assert type(c0) == RElement
    assert type(c1) == MElement

    # Apply the SHA384 hash function to the concatenation of m, c0 and c1
    from hashlib import sha384

    digest = sha384(M_to_bytes(m) + R_to_bytes(c0) + M_to_bytes(c1)).digest()

    digest = bytes_to_bits(digest[:l // 8])  # We only need l bits (l / 8 bytes)

    return KK(digest)  # Returns the KElement corresponding to the digest


K(M([1, 0] * 128), Rr([1, 0, 1]), M([1, 0] * 128))
```

Out[22]:

```
(1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1,
0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0,
1, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1,
1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1,
0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1,
0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1,
1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0,
0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0,
0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0)
```

Função de computação do sindrome (syndrome computation)

In [23]:

```
def compute_syndrome(c0: RElement, h0: RElement) -> RElement:
    assert type(c0) == RElement
    assert type(h0) == RElement

    return c0 * h0
```

Geração de chaves

In [24]:

```
def keygen() -> ((RElement, RElement), MElement, RElement):
```

```
    """
    Geração de chaves
    Entrada: Nenhum
    Saída: (pk, sk)
    """
    h0 = generate_sparse(w // 2, l)
    h1 = generate_sparse(w // 2, l)

    sigma = M.random_element()

    h0_inv = 1 / h0
    h = h1 * h0_inv

    return (h0, h1), sigma, h
```

In [25]:

```
# Teste da geração de chaves

(priv_key, sigma, public_key) = keygen()
print("public_key: ", public_key.lift())
```

```
public_key:  x^256 + x^255 + x^253 + x^251 + x^250 + x^249 + x^247 + x^245 + x^244 + x^2
43 + x^241 + x^240 + x^237 + x^232 + x^231 + x^230 + x^229 + x^228 + x^224 + x^223 + x^2
20 + x^219 + x^216 + x^211 + x^209 + x^207 + x^200 + x^199 + x^196 + x^194 + x^192 + x^1
87 + x^186 + x^185 + x^183 + x^179 + x^177 + x^175 + x^174 + x^173 + x^172 + x^171 + x^1
69 + x^166 + x^165 + x^163 + x^160 + x^159 + x^158 + x^157 + x^156 + x^155 + x^154 + x^1
53 + x^151 + x^148 + x^147 + x^146 + x^145 + x^143 + x^140 + x^133 + x^129 + x^128 + x^1
25 + x^123 + x^122 + x^119 + x^118 + x^116 + x^115 + x^113 + x^111 + x^109 + x^107 + x^1
03 + x^100 + x^99 + x^98 + x^97 + x^96 + x^92 + x^90 + x^89 + x^88 + x^86 + x^85 + x^83
+ x^80 + x^73 + x^72 + x^69 + x^66 + x^63 + x^61 + x^60 + x^59 + x^56 + x^55 + x^53 + x^
52 + x^51 + x^50 + x^49 + x^45 + x^42 + x^41 + x^40 + x^38 + x^37 + x^36 + x^35 + x^32 +
x^30 + x^28 + x^27 + x^26 + x^23 + x^20 + x^16 + x^14 + x^12 + x^8 + x^6 + x^4 + x^3 + x^
2 + x + 1
```

Encapsulamento

In [26]:

```
def calculate_c(e0: RElement, e1: RElement, h: RElement, seed: MElement) -> (RElement, M
Element):
    assert type(e0) == RElement
    assert type(e1) == RElement
    assert type(h) == RElement
    assert type(seed) == MElement

    return e0 + e1 * h, seed + L(e0, e1)
```

In [27]:

```
def encapsulate(h: RElement) -> (KElement, (RElement, MElement)):
    """
    Encapsulamento de uma chave.
    :param h: Chave pública
    :return: (chave partilhada e ciphertext)
    """
    assert type(h) == RElement

    seed: MElement = M.random_element()
    (e0, e1) = H(seed)

    c = calculate_c(e0, e1, h, seed)
    c0, c1 = c

    k = K(seed, c0, c1)

    return k, c
```

In [28]:

```
# Teste do encapsulamento

(priv_key, sigma, public_key) = keygen()

(k, c) = encapsulate(public_key)

print("k: ", k.lift())
print("c: ", c)
```

```
k:  (1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0
, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0
0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1,
1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,
1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0,
1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0,
0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0,
0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1,
0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1)
c:  (xbar^256 + xbar^255 + xbar^254 + xbar^253 + xbar^251 + xbar^250 + xbar^248 + xbar^24
4 + xbar^243 + xbar^242 + xbar^239 + xbar^238 + xbar^237 + xbar^236 + xbar^235 + xbar^234
+ xbar^230 + xbar^228 + xbar^226 + xbar^225 + xbar^224 + xbar^222 + xbar^221 + xbar^220 +
xbar^217 + xbar^216 + xbar^214 + xbar^213 + xbar^207 + xbar^205 + xbar^199 + xbar^197 + x
bar^195 + xbar^193 + xbar^190 + xbar^189 + xbar^186 + xbar^183 + xbar^182 + xbar^181 + xb
ar^180 + xbar^177 + xbar^176 + xbar^175 + xbar^173 + xbar^169 + xbar^168 + xbar^167 + xba
r^165 + xbar^163 + xbar^162 + xbar^160 + xbar^156 + xbar^155 + xbar^154 + xbar^153 + xbar
^151 + xbar^149 + xbar^148 + xbar^147 + xbar^143 + xbar^136 + xbar^122 + xbar^119 + xbar^
115 + xbar^113 + xbar^112 + xbar^109 + xbar^108 + xbar^107 + xbar^96 + xbar^95 + xbar^94
+ xbar^92 + xbar^90 + xbar^87 + xbar^86 + xbar^85 + xbar^83 + xbar^82 + xbar^81 + xbar^80
+ xbar^78 + xbar^77 + xbar^73 + xbar^72 + xbar^69 + xbar^68 + xbar^67 + xbar^65 + xbar^61
+ xbar^60 + xbar^54 + xbar^52 + xbar^47 + xbar^45 + xbar^42 + xbar^40 + xbar^39 + xbar^38
+ xbar^31 + xbar^30 + xbar^29 + xbar^28 + xbar^27 + xbar^25 + xbar^22 + xbar^20 + xbar^19
+ xbar^16 + xbar^15 + xbar^13 + xbar^7 + xbar^6 + xbar^5 + xbar^3 + xbar + 1, (1, 0, 0, 1
, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1
1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0,
1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1,
1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,
0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0,
1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0,
1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0,
1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0))
```

Desencapsulamento

In [29]:

```
# noinspection PyUnresolvedReferences
from sage.matrix.matrix_mod2_dense import Matrix_mod2_dense
# noinspection PyUnresolvedReferences
from sage.matrix.matrix_integer_dense import Matrix_integer_dense


def decoder(x: RElement, h0: RElement, h1: RElement) -> (RElement, RElement):
    """
    Decodificador do ciphertext. Retorna o erro.
    :param x: produto do c0 com h0.
    :param h0: chave privada componente 0.
    :param h1: chave privada componente 1.
    :return: erro do ciphertext.
    """
    assert type(x) == RElement
    assert type(h0) == RElement
    assert type(h1) == RElement

    # Convert x to a vectorSpace element
    x = RElement_to_VectorSpace(x)

    H_mat = get_H_matrix(h0, h1)

    return BGF(x, H_mat)
```

```
def BGF(s: Vector_mod2_dense, H: Matrix_mod2_dense) -> (RElement, RElement):
    """
    Função BGF (Black Gray Flip) usada no decodificador.
    :param s: Vetor de bits.
    :param H: Matriz derivada dos blocos circulantes h0 e h1.
    :return:
    """
    assert type(s) == Vector_mod2_dense
    assert type(H) == Matrix_mod2_dense

    print("BGF function")
    e: Vector_mod2_dense = copy(VectorSpace(GF(2), n).zero())
    d = w // 2

    HTranspose = H.transpose()

    for i in range(1, NbIter + 1):
        T = threshold(getHammingWeight(s + e * HTranspose), i)
        e, black, gray = BFIter(s + e * HTranspose, e, T, H)
        if i == 1:
            e = BFMaskedIter(s + e * HTranspose, e, black, ((d + 1) // 2) + 1, H)
            e = BFMaskedIter(s + e * HTranspose, e, gray, ((d + 1) // 2) + 1, H)

    if s == e * HTranspose:
        (e0, e1) = e[:r], e[r:]
        return e0, e1
    else:
        return Rr(0), Rr(0)


def BFIter(s: Vector_mod2_dense, e: Vector_mod2_dense, T: int, H: Matrix_mod2_dense) ->
(RElement, RElement, RElement):
    """
    Black-Gray-Flip (BGF) BFIter function.
    :param s: the syndrome vector
    :param e: the error vector
    :param T: the threshold
    :param H: the parity-check matrix
    :return: a tuple containing the updated error vector, the set of black bits, and the
set of gray bits
    """
    assert type(s) == Vector_mod2_dense
    assert type(e) == Vector_mod2_dense
    assert type(T) == int
    assert type(H) == Matrix_mod2_dense

    n = H.ncols()
    black = copy(VectorSpace(GF(2), n).zero())
    gray = copy(VectorSpace(GF(2), n).zero())

    for j in range(n):
        if ctr(H, s, j) >= T:
            e[j] += 1
            black[j] = 1
        elif ctr(H, s, j) >= T - tau:
            gray[j] = 1

    return e, black, gray


def ctr(H: Matrix_mod2_dense, s: Vector_mod2_dense, j: int) -> int:
    """
    ctr(H; s; j). This function computes a quantity referred to as the counter (aka the n
umber of unsatisfied parity-checks) of j.
    It is the number of '1' (set bits) that appear in the same position in the syndrome s
and in the j-th column of the matrix H.
    """
    assert type(H) == Matrix_mod2_dense
    assert type(s) == Vector_mod2_dense
    assert type(j) == int
```

```
    return getHammingWeight(s.pairwise_product(H.column(j)))


def BFMaskedIter(s: Vector_mod2_dense, e: Vector_mod2_dense, mask: Vector_mod2_dense, T:
int,
                 H: Matrix_mod2_dense) -> RElement:
    """
    Black-Gray-Flip (BGF) BFMaskedIter function.
    :param s: the syndrome vector
    :param e: the error vector
    :param mask: the mask vector
    :param T: the threshold
    :param H: the parity-check matrix
    :return: the updated error vector
    """
    assert type(s) == Vector_mod2_dense
    assert type(e) == Vector_mod2_dense
    assert type(mask) == Vector_mod2_dense
    assert type(T) == int
    assert type(H) == Matrix_mod2_dense

    n = H.ncols()

    for j in range(n):
        if ctr(H, s, j) >= T:
            e[j] = e[j] + mask[j]

    return e
```

In [30]:

```
def RElement_to_VectorSpace(element: RElement) -> Vector_mod2_dense:
    assert type(element) == RElement

    elem_coefs = element.lift().list()

    v = vector(GF(2), elem_coefs + [0] * (r - len(elem_coefs)))

    return v
```

In [31]:

```
def get_H_matrix(h0: RElement, h1: RElement) -> Matrix_integer_dense:
    assert type(h0) == RElement

    print("get_H_matrix function")

    H = block_matrix(1, 2, [get_circulant_matrix(h0), get_circulant_matrix(h1)])

    assert H.dimensions() == (r, n)

    return H


def get_circulant_matrix(element: RElement) -> Matrix_mod2_dense:
    assert type(element) == RElement

    print("get_circulant_matrix function")
    vec = element.lift().list()
    # Fill the rest of the vector with zeros
    vec = vec + [0] * (r - len(vec))

    circ = matrix.circulant(vec)

    return circ
```

In [32]:

```
def decapsulate(h0: RElement, h1: RElement, sigma: MElement, c0: RElement, c1: MElement)
-> KElement:
```

```
    """
    Decapsulate function.
    :param h0: componente 0 da chave privada
    :param h1: componente 1 da chave privada
    :param sigma: sigma
    :param c0: componente 0 do ciphertext
    :param c1: componente 1 do ciphertext
    :return: segredo partilhado
    """
    assert type(h0) == RElement
    assert type(h1) == RElement
    assert type(sigma) == MElement
    assert type(c0) == RElement
    assert type(c1) == MElement

    e_ = decoder(c0 * h0, h0, h1)

    m_ = c1 + L(e_[0], e_[1])

    if e_ == H(m_):
        return K(m_, c0, c1)
    else:
        return K(sigma, c0, c1)
```

In [33]:

```
# Teste do decapsulate

(priv_key, sigma, public_key) = keygen()

(k, (c0, c1)) = encapsulate(public_key)

k_ = decapsulate(*priv_key, sigma, c0, c1)

print("k: ", k)
print("k_: ", k_)

print("Total bits: ", l)
print("Different bits: ", getHammingWeight(k - k_))  # FIXME: Não está a funcionar como
esperado :(
```

```
decoder function
get_H_matrix function
get_circulant_matrix function
get_circulant_matrix function
BGF function
k:  (1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, (
, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, (
1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0,
1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 0,
0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0,
1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0,
0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1)
k_:  (0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1,
1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0,
1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1,
1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0,
0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,
0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 1)
Total bits:  256
Different bits:  131
```