TP2-Problema1-BIKE

May 2, 2022

1 TRABALHO PRÁTICO 2 - GRUPO 14

1.1 Problema 1 - BIKE

Este problema consistia em implementar o algoritmo KEM que seja IND-CPA seguro e num algoritmo PKE que seja IND-CCA seguro para a técnica pós-quântica baseada em códigos, **BIKE**. No caso da implementação do KEM foi aplicado os passos associados à técnica BIKE-1 presentes no paper BIKE e nos apontamentos. No caso do PKE foram aplicados as etapas necesárias para implementar as transformações Fujisaki-Okamoto que consegue converter um esquema PKE com segurança IND-CPA num esquema PKE com segurança IND-CCA, tal como esta presente nos apontamentos.

IMPORTS

[61]:
```
import random
import numpy as np
from cryptography.hazmat.primitives import hashes
```

1.1.1 Resolução do Problema - KEM (IND-CPA)

Na classe abaixo é implementado o algoritmo KEM que irá ofuscar (encapsular) pequenas quantidades de informação, "chaves", que este próprio gera. Assim sendo, será necessário implementar 3 funcionalidades principais:

Geração do par de chaves: A função `keyGen` tem como objetivo gerar um par de chaves que será utilizado para fazer o encapsulamento e desencapsulamento de uma chave gerada pelo algoritmo. Esta função começa por gerar a chave privada do problema a partir da função `coeffGen` que irá gerar dois parâmetros pertencentes a R de tamanho r cada um com um peso igual a *sparse*, isto é, com o *sparse* coeficientes iguais a 1. De seguida gera-se a chave privada segundo a expressão $(1, h0/h1)$, sendo h0 e h1 os parâmteros da chave privada.

Encapsulamento: É necessário implementar a função `encaps` para assim ofuscar a chave gerada pelo algoritmo. Esta função começa por gerar pequenos erros a partir da função `errorGen` que irá gerar dois polinómios de tamanho r de tal forma que a soma dos pesos dos dois erros seja igual a t. De seguida gera-se a hash da informação a ser encapsulada k utilizando os pequenos erros gerados anteriormente, através da função `hashGen`. Para gerar o encapsulamento da informação é necessário gerar um r aleatório denso utilizando o anel cíclico polinomial R que será utilizando juntamento com os valores da chave pública e os erros para gerar o encapsulamento da informação k, tal como está na seguinte expressão $(y0, y1) \leftarrow (r * f0 + e0, r * f1 + e1)$, com $(f0, f1)$ - chave publica e $(e0, e1)$ - pequenos erros.

Desencapsulamento Finalmente, será também necessário implementar a função `decaps` que faça o desencapsulamento da chave gerada pelo algoritmo. Nesta função é necessário calcular a matriz dispersa H e ao syndrome s de forma a serem utilizados pelo algoritmo *bit Flip* que vai descodificar os erros gerados na função *encaps*. Esses erros serão depois utilizados para gerar a hash da informação k gerada na função *encaps*. De seguida mostra-se os passos necessários para a implementação do desencapsulamento: * **Geração da matriz dispersa H:** Utilizando a chave privada, $(h0, h1)$, a matriz é criada a partir do par de matrizes cíclicas $\textbf{rot}(h0)\,, \textbf{rot}(h1)$, que são calculadas utilizando a função `Rot`. Esta função tem como objetivo gerar a matriz de rotação partir de um vetor utilizando as funções `toVectorR` e `rot`. Desta forma conseguimos obter a matriz dispensa H $= (\textbf{rot}(h0)\,|\textbf{rot}(h1))$. * **Geração do syndrome s:** Para calcular o s começamos por transformar o encapsulamento, isto é, o criptograma $(y0, y1)$ calculado no *encaps*, num vetor de tamanho n utilizando a função `toVectorN`. De seguida utilizamos a expressão $\; s \equiv h0 * y0 + h1 * y1 \;$ para determinar o valor do syndrome s. * **Algoritmo Bit Flip:** Este algoritmo iterativo foi implementado na função `bitFlip` e permite alterar os bits y (com y a corresponder ao vetor de tamanho n que representa o criptograma $(y0, y1)$), atualizando o valor do syndrome $\; s \;$ emk cada iteração até que no final a única solução possível da equação $\; s \; = \; \sum_{y_j \neq 0} \; s \cap h_j$ que corresponde à definição do s é $s = 0\,$. Utilizando como o input a matriz H, o vetor y e o syndrome s conseguimos implementar o algoritmo através dos seguintes passos: * Geramos o novo vetor x igual a y e o novo syndrome z igual a s, que serão alterados ao longo das iterações. * Definimos o número de interações do ciclo que serão iguais ao parâmetro n. Caso o s não tenha convergido para 0 ao fim de n interações então ocorreu um problema no desencapsulamento. * Enquanto não tivermos atingido o limite de iterações e enquanto o peso de s for diferente de 0: * Calculamos o peso dos vários elementos de $z \cap hj$ com $j \in \{1..n\}$, utilizando a função `hammingWeight`. * Calculamos qual o peso máximo desses elementos. * Caso o peso de um elemento seja o máximo então vamos alterar os bits da variavél x e atualizar o valor da syndrome z utilizando respetivamente $x_j \leftarrow \neg \; x_j \;$ e $z \; \leftarrow z + h_j$ * No final, caso o algoritmo convirja então é retornado o valor do $x = (x0, x1)$, caso contrário é retornado o valor NONE pois as iterações atigiram o limite sem o algoritmo ter convergido. * **Desencapsulamento da chave:** Para desencapsular a informação é necessário calcular os valores reais de e0 e e1 a partir dos seguintes calculos: * Sabendo que $x0 = r * f0 $ e $x1 = r * f1$ então temos:

$$y0 = r * f0 + e0 \equiv e0 = y0 - r * f0 \equiv e0 = y0 - x0 y1 = r * f1 + e1 \equiv e1 = y1 - r * f1 \equiv e1 = y1 - x1$$

* Com estas equações conseguimos obter os valores e0 e e1 que serão usados de forma a verificar a condição $|e0 + e1| = t$, com $|e0 + e1|$ igual à soma dos pessos de e0 e e1. Caso contrário ocorreu um error no processo de desencapsulamento. * Finalmente, os valores e0 e e1 serão utilizados para calcular a hash da informação encapsulada e assim desencapsular essa informação.

[62]:
```
class BIKE_KEM:

    #Função de inicialização das variaveis a usar nos métodos
    def __init__(self):
        self.r, self.t, self.n, self.F2, self.R = self.setup()

    #Parâmetros da técnica BIKE
    def setup(self):

        # comprimento do bloco - número primo alto
        r = 257
```

```
        # peso do erro - número positivo
        t = 16
        n = 2*r

        F2 = GF(2)

        F = PolynomialRing(F2, name='w')
        w = F.gen()

        # Um anel cíclico polinomial F2[X]/<X^r + 1>
        R= QuotientRing(F, F.ideal(w^r + 1))

        return r,t,n,F2,R

    #Gera os coeficentes de um polinómio com tamanho r - utilizado na geração da chave privada e pública
    def coeffGen(self, sparse=3):

        coeffs = [1]*sparse + [0]*(self.r-2-sparse)
        random.shuffle(coeffs)

        return self.R([1]+coeffs+[1])

    #Gera um dois polinomios aleatórios de tamanho r - utilizado para a geração dos erros
    def errorGen(self, t):

        res = [1]*t + [0]*(self.n-t)
        random.shuffle(res)

        return self.R(res[:self.r]), self.R(res[self.r:])

    #Geração do hash - chave encapsulada
    def hashGen(self,e0,e1):

        digest = hashes.Hash(hashes.SHA256())
        digest.update(e0.encode())
        digest.update(e1.encode())

        return digest.finalize()

    #Transformação de um polinómio num vetor de tamanho r
    def toVectorR(self,h):

        V = VectorSpace(self.F2, self.r)

        return V(h.list() + [0]*(self.r - len(h.list())))
```

```
#Transformação de um par num vetor de tamanho n
def toVectorN(self, c):

    V = VectorSpace(self.F2,self.n)

    f = self.toVectorR(c[0]).list() + self.toVectorR(c[1]).list()

    return V(f)

#Rotação de uma unidade num vetor
def rot(self,m):

    V = VectorSpace(self.F2,self.r)
    v = V()
    v[0] = m[-1]

    for i in range(self.r-1):
        v[i+1] = m[i]

    return v

#Gera matriz de rotação partir de um vetor
def Rot(self,h):

    M = Matrix(self.F2, self.r, self.r)

    M[0] = self.toVectorR(h)

    for i in range(1,self.r):
        M[i] = self.rot(M[i-1])

    return M

#Gera o peso de hamming de um polinómio binário x
def hammingWeight(self,x):

    return sum([1 if a == 1 else 0 for a in x])

#Implementação do algoritmo de Bit Flip
def bitFlip(self, H, y, s):

    x = y
    z = s
    nIter = self.r

    while self.hammingWeight(z) > 0 and nIter > 0:
```

```python
            nIter = nIter - 1

            #todos os pessos de hamming
            weights = [self.hammingWeight(z.pairwise_product(H[i])) for i in␣
↪range(self.n)]
            maximum = max(weights)

            for j in range(self.n):

                if weights[j] == maximum:

                    x[j] = 1-x[j]
                    z += H[j]

        if nIter == 0:
            return None

        return x

    #Função: Gera um par de chaves - pública e privada
    def keyGen(self):

        #Obtenção da chave privada
        h0 = self.coeffGen()
        h1 = self.coeffGen()

        #Obtenção da chave pública
        f = (1, h0/h1)

        return (h0,h1), f

    #Função: Encapsula uma chave - abordagem McEliece para um KEM-CPA
    def encaps(self, public):

        #Gera pequenos erros
        e0,e1 = self.errorGen(self.t)

        #Chave encapsulada
        key = self.hashGen(str(e0),str(e1))

        #Gerar aleatoriamente um r <- R denso
        r = self.R.random_element()

        #Encapsulamento da chave
        (y0,y1) = (r * public[0] + e0, r * public[1] + e1)
```

```python
        return key, (y0,y1)

    #Função: Desencapsula a chave - recebe a chave privada e o encapsulamento
    def decaps(self, private, c):

        #Calcula matriz H = rot(h0)|rot(h1)
        h0Rot = self.Rot(private[0])
        h1Rot = self.Rot(private[1])
        H = block_matrix(2,1,[h0Rot,h1Rot])

        #Transforma o criptograma c num vetor de tamanho n
        vectorC = self.toVectorN(c)

        #Computa syndrome
        s = vectorC * H

        #Descodifica s para recuperar o par de erros (e0',e1') utilizando o algoritmo de bitFlip
        error = self.bitFlip(H, vectorC, s)

        if error == None:
            print("Iterações atingiram o limite")
            return None
        else:

            listError = error.list()

            #Erros como par de polinómios
            error0 = self.R(listError[:self.r])
            error1 = self.R(listError[self.r:])

            #Como forma de recuperar os erros e0 e e1 originais
            e0 = c[0] - error0
            e1 = c[1] - error1

            #Verifica se houve erro no encoding
            if self.hammingWeight(self.toVectorR(e0)) + self.hammingWeight(self.toVectorR(e1)) != self.t:

                print("Erro no decoding")
                return None
            else:

                #Desencapsula chave
                key = self.hashGen(str(e0),str(e1))

        return key
```

Cenário de Teste

[63]:
```python
bike = BIKE_KEM()

private, public = bike.keyGen()

toEncap, c = bike.encaps(public)
print("Original Key: ", toEncap)

toDecap = bike.decaps(private,c)
print("Key: ", toDecap)

if toDecap != None and toDecap == toEncap:

    print("A chave desencapsulada é igual à original")
```

```
Original Key:  b'\xd8[9\xb7\x88\x98\x87kBB\xd9:\x11{\xce\xc9v\x8d-p\xb8\xe8\xcc\
x90Q\xd4m\xbc\x7fK&\xfe'
Key:  b'\xd8[9\xb7\x88\x98\x87kBB\xd9:\x11{\xce\xc9v\x8d-p\xb8\xe8\xcc\x90Q\xd4m
\xbc\x7fK&\xfe'
A chave desencapsulada é igual à original
```

1.1.2 Resolução do Problema - PKE (IND-CCA)

Na classe abaixo é implementado o algoritmo PKE que irá cifrar e posteriormente decifrar uma mensagem utilizando o par de chaves gerados. Normalmente a construção de um esquema de PKE que seja IND-CCA seguro é algo mais complicado, no entanto através da transformação de Fujisaki-Okamoto (FOT) consegue-se converter um esquema PKE com segurança IND-CPA num esquema PKE com segurança IND-CCA. Assim sendo, será necessário implementar 3 funcionalidades principais:

Geração do par de chaves: A função `keyGen` tem como objetivo gerar um par de chaves que será utilizado para fazer o encapsulamento e desencapsulamento de uma chave gerada pelo algoritmo. Para o problema PKE utilizamos a função keyGen implementada na classe BIKE_KEM.

Cifragem: É necessário implementar a função `encrypt` para assim cifrar uma mensagem. Esta função é implementada segundo o algoritmo de cifra FOT apresentado.

$$E(x) \;\equiv\; \vartheta\, r \leftarrow h \centerdot \vartheta\, y \leftarrow x \oplus g(r) \centerdot (e,k) \leftarrow f(y\|r) \centerdot \vartheta\, c \leftarrow k \oplus r \centerdot (y,e,c)$$

Começamos por gerar os erros pequenos utilizando a função *errorGen* da classe BIKE_KEM. De seguida vamos gerar um valor r aleatorio denso utilizando o anel ciclíco R que será utilizado na função de hash g gerando desta forma o valor $g(r)$. Este valor vai servir de máscara na operação de XOR juntamente com a mensagem m a ser cifrada. A função `xor` tem como objetivo implementar a operação XOR tal que caso a máscara seja menor que a mensagem então a máscara será repetida para fazer o processo de XOR para os restantes bytes da mensagem. O valor da operação de XOR, y, juntamente com a chave pública e os erros gerados serão utilizados na função `f` que utiliza o mesmo algoritmo que a função *encaps* da classe BIKE_KEM para assim gerar a chave que será utilizada para obter o ciphertext, k e o encapsulmento dos erros, e. A chave k e o valor de r serão utilizados no processo de XOR para assim gerar o ciphertext c.

Desencapsulamento Finalmente, será também necessário implementar a função `decrypt` para desencapsular a mensagem. Esta função é implementada segundo o algoritmo de cifra FOT apresentado.

$$D(y,e,c) \;\equiv\; \vartheta\, k \leftarrow \mathsf{KREv}(e) \centerdot \vartheta\, r \leftarrow c \oplus k \centerdot \text{ if } \;(e,k) \neq f(y\|r)\; \text{ then } \;\perp\; \text{ else } \; y \oplus g(r)$$

Começamos por fazer um processo de desencapsulamento semelhante ao apresentado na função *decaps* da classe BIKE_KEM. Para isso utiliza-se as funções `decapsError` e `decapsKey` que vão desencapsular os erros $(e0, e1)$ e a chave k, respetivamente. É de notar que estas funções implementam os mesmos algoritmos aresentados nas funções explicadas na classe BIKE_KEM. De seguida vamos obter o valor de r através da operação XOR utilizando o ciphertext c e a chave k que será utilizado na função f de forma a verificar se houve erros no processo de decifra. Caso os valores gerados pela função f sejam iguais aos valores k e e então vamos gerar a hash do valor de r obtendo o valor $g(r)$ que será utilizada na operação de XOR juntamneto com o valor y que representa o encapsulamento da mensagem. O resultado desta operação será a mensagem cifrada.

[69]:
```python
class BIKE_PKE:

    #Função de inicialização das variaveis a usar nos métodos
    def __init__(self):
        self.kem, self.r, self.t, self.n, self.F2, self.R = self.setup()

    #Parâmetros da técnica BIKE-PKE
    def setup(self):

        #Inicializa as variaveis iguais ao KEM
        kem = BIKE_KEM()

        r = kem.r
        t = kem.t
        n = kem.n

        F2 = kem.F2
        R = kem.R

        return kem,r,t,n,F2,R

    #Implemetação da função de hash - função g
    def g(self, r):

        digest = hashes.Hash(hashes.SHA256())
        digest.update(str(r).encode())
        g = digest.finalize()

        return g

    #Implementação da operação de XOR.
    def xor(self, data, mask):
```

```
        result = b''
        lengthData = len(data)
        lengthMask = len(mask)

        i=0

        while i < lengthData:

            for j in range(lengthMask):

                if i<lengthData:

                    result += (data[i]^^mask[j]).to_bytes(1, byteorder='big')
                    i += 1

                else:
                    break

        return result

    #Implementação do núcleo deterministico f - semelhante ao realizado em KEM
    def f(self, public, m, e0, e1):

        w = (m * public[0] + e0, m * public[1] + e1)

        key = self.kem.hashGen(str(e0),str(e1))

        return (key,w)

    #Desencapsula a chave gerada pelo o algoritmo - semelhante ao realizado em KEM
    def decapsKey(self,e0,e1):

        if self.kem.hammingWeight(self.kem.toVectorR(e0)) + self.kem.hammingWeight(self.kem.toVectorR(e1)) != self.t:

            print("Erro no decoding")
            return None

        else:

            key = self.kem.hashGen(str(e0),str(e1))

        return key

    #Desencapsula os erros - semelhante ao realizado em KEM
    def decapsError(self,private, e):
```

```
        #Calcula matriz H = rot(h0)|rot(h1)
        h0Rot = self.kem.Rot(private[0])
        h1Rot = self.kem.Rot(private[1])
        H = block_matrix(2,1,[h0Rot,h1Rot])

        #Transforma o criptograma num vetor de tamanho n
        vectorE = self.kem.toVectorN(e)

        #compute syndrome
        s = vectorE * H

        #Descodifica s para recuperar o vetor (e0,e1)
        error = self.kem.bitFlip(H, vectorE, s)

        if error == None:
            print("Iterações atingiram o limite")
            return None
        else:

            listError = error.list()

            error0 = self.R(listError[:self.r])
            error1 = self.R(listError[self.r:])

            e0 = e[0] - error0
            e1 = e[1] - error1

        return e0,e1

    #Função: Gera o par de chaves - utiliza o método keyGen do algoritmo KEM
    def keyGen(self):

        self.private, self.public = self.kem.keyGen()

        return  self.private, self.public


    #Função: Cifra uma mensagem utilizando o FOT -  r←h .  y←x g(r) .
 ↪(e,k)←f(y r) .  c←k r .  (y,e,c)
    def encrypt(self, msg, public):

        #Gerar erros pequenos
        e0,e1 = self.kem.errorGen(self.t)

        #Gerar aleatoriamente r <- R denso -  r ← h
        r = self.R.random_element()
```

```
        #Gerar g(r)
        g = self.g(r)

        #Aplicar  y ← x   g(r)
        y = self.xor(msg.encode(),g)

        # Transformar a string y num número para ser utilizada pelo o anel R
        yBinary = bin(int.from_bytes(y, byteorder=sys.byteorder))
        ryBinary = self.R(yBinary)

        #Aplicar (e,k) ← f(y r)
        (key, e) = self.f(public, ryBinary + r, e0, e1)

        #Aplicar  c ← k   r
        c = self.xor(str(r).encode(),key)

        return y, e, c

    #Função: Decifra uma mensagem utilizando o FOT - k←KREv(e)   r←c k
↪if(e,k) f(y r) then    else y g(r)
    def decrypt(self, private, y, e, c):

        #Aplicar  k ← KREv(e)
        e0, e1 = self.decapsError(private,e)
        k = self.decapsKey(e0,e1)

        #Aplicar  r ← c   k
        rXOR = self.xor(c,k)
        r = self.R(rXOR.decode())

        #Aplicar as mesmas transformações associadas ao processor de cifra
        yBinary = bin(int.from_bytes(y, byteorder=sys.byteorder))
        ryBinary = self.R(yBinary)

        #Aplicar if(e,k) f(y r) then    else y g(r)
        if (k,e) != self.f(self.public, ryBinary + r, e0, e1):

            print("Erro no decoding")
            return None

        else:

            #Gerar g(r)
            g = self.g(r)

            #Aplicar y   g(r)
```

```python
            plaintext = self.xor(y,g)

        return plaintext
```

Cenário de Teste

[70]:
```python
bikePKE = BIKE_PKE()
msg = "Mensagem a ser cifrada"
print("Mensagem original: " + msg)

private, public = bikePKE.keyGen()

msgEncaps, keyEncaps, ciphertext = bikePKE.encrypt(msg, public)
print("Ciphertext: ")
print(ciphertext)

plaintext = bikePKE.decrypt(private, msgEncaps, keyEncaps, ciphertext)
print("Plaintext: " + plaintext.decode())
```

```
Mensagem original: Mensagem a ser cifrada
Ciphertext:
b'\xf8\xc1\xaf\xc8h\x0c\xbc{\xda\x8b\x06\xb2\r\xf6\xd6\x1cax\xcdk\x8e\x99\x88\xa
4\x80\xc5\x93q\x92\xc0X\xdb\xaf\xd4\xac\xdbD`\xbby\xc3\x80\r\xe5\x18\xf5\xc50\r\
x7f\xccs\x85\x92\xdf\xb1\x83\xd6\xbf\x1d\x95\xc4L\xd0\xa4\x83\xb9\xd8WL\xd7\x7f\
xce\x93\x06\xee0\xe0\xc6#!\x13\xcax\x93\x99\xd4\xe6\x96\xd5\xac1\xf9\xc2K\xc4\xa
f\x88\xee\xcdT_\xfb\x13\xc8\x93\x16\xe5D\xb7\xd3 2?\xa6y\x97\x80\xdf\xed\xc1\xc0
\xaf"\xd5\xaeJ\xc2\xba\x83\xe5\x9aA\\\xe8?\xa4\x92\x14\xf10\xbc\x8451,\x8a\x15\x
97\x8b\xcc\xe6\xca\x97\xba!\xc6\x82&\xc2\xbd\x93\xee\x91\x16I\xeb,\x88\xfe\x14\x
f4W\xb7\x8fb$/\x999\xfb\x8b\xce\xf3\xc1\x9c\xed4\xc5\x91\n\xae\xbd\x92\xfa\x9a\x
1d\x1e\xfe/\x9b\xd2x\xf7_\xa1\x84is:\x9a*\xd7\xe7\xcd\xf6\xd5\x97\xe6c\xd0\x92\x
19\x82\xd1\x91\xfe\x89\x16\x15\xa9:\x98\xc1T\x9b]\xa7\x96bxm\x8f)\xc4\xcb\xa1\xf
4\xd1\x87\xedh\x87\x87\x1a\x91\xfd\xfd\xff\x83\x0f\x1e\xa2m\x8d\xc2G\xb71\xa6\x9
dzsf\xd8<\xc7\xd8\x8d\x98\xd0\x8e\xfec\x8c\xd0\x0f\x92\xee\xd1\x90\x8b\x0f\x0c\x
a9f\xda\xd7D\xa4\x1d\xc9\x95{bm\xd3k\xd2\xdb\x9e\xb4\xbf\x86\xf5z\x87\xdbX\x87\x
ed\xc2\xbc\xe4\x07\x06\xb1m\xd1\x80Q\xa7\x0e\xe5\xfaskz\xd8`\x85\xce\x9d\xa7\x93
\xe9\xfc{\x95\xd0S\xd0\xf8\xc1\xaf\xc8h\x0f\xb1}\xda\x8b\x06\xb2\r\xf6\xd6\x1cbz
\xc1k\x8e\x99\x88\xa4\x80\xc5\x93r\x90\xc7X\xdb\xaf\xd4\xac\xdbD`\xb8z\xc8\x80\r
\xe5\x18\xf5\xc50\r|\xcfz\x85\x92\xdf\xb1\x83\xd6\xbf\x1d\x96\xc6A\xd0\xa4\x83\x
b9\xd8WL\xd7|\xcc\x98\x06\xee0\xe0\xc6#!\x13\xc9}\x92\x99\xd4\xe6\x96\xd5\xac1\x
f9\xc1N\xc6\xaf\x88\xee\xcdT_\xfb\x13\xcb\x96\x13\xe5D\xb7\xd3 2?\xa6z\x93\x8d\x
df\xed\xc1\xc0\xaf"\xd5\xaeI\xc6\xbd\x83\xe5\x9aA\\\xe8?\xa4\x91\x13\xfd0\xbc\x8
451,\x8a\x15\x94\x8c\xcb\xe6\xca\x97\xba!\xc6\x82&\xc1\xba\x90\xee\x91\x16I\xeb,
\x88\xfe\x17\xf0]\xb7\x8fb$/\x999\xfb\x88\xca\xf6\xc1\x9c\xed4\xc5\x91\n\xae\xbe
\x97\xf6\x9a\x1d\x1e\xfe/\x9b\xd2x\xf4[\xa1\x84is:\x9a*\xd7\xe7\xce\xf2\xd4\x97\
xe6c\xd0\x92\x19\x82\xd1\x92\xfa\x89\x16\x15\xa9:\x98\xc1T\x9b^\xa3\x96bxm\x8f)\
xc4\xcb\xa1\xf7\xd5\x87\xedh\x87\x87\x1a\x91\xfd\xfd\xff\x89\x0e\x1e\xa2m\x8d\xc
2G\xb71\xa6\x97wsf\xd8<\xc7\xd8\x8d\x98\xd0\x84\xf9c\x8c\xd0\x0f\x92\xee\xd1\x90
\x8b\x05\r\xa9f\xda\xd7D\xa4\x1d\xc9\x95qbm\xd3k\xd2\xdb\x9e\xb4\xbf\x86\xfes\x8
7\xdbX\x87\xed\xc2\xbc\xe4\x07\x0c\xb1m\xd1\x80Q\xa7\x0e\xe5\xfasa\x7f\xd8`\x85\
```

```
xce\x9d\xa7\x93\xe9\xfcq\x97\xd0S\xd0\xf8\xc1\xaf\xc8h\x0f\xb8t\xda\x8b\x06\xb2\
r\xf6\xd6\x1cb|\xc0k\x8e\x99\x88\xa4\x80\xc5\x93r\x96\xc7X\xdb\xaf\xd4\xac\xdbD`
\xb8|\xcf\x80\r\xe5\x18\xf5\xc50\r|\xc9z\x85\x92\xdf\xb1\x83\xd6\xbf\x1d\x96\xc1
H\xd0\xa4\x83\xb9\xd8WL\xd7|\xca\x97\x06\xeeO\xe0\xc6#!\x13\xc9{\x93\x99\xd4\xe6
\x96\xd5\xac1\xf9\xc1H\xc5\xaf\x88\xee\xcdT_\xfb\x13\xcb\x90\x12\xe5D\xb7\xd3 2?
\xa6z\x95\x8a\xdf\xed\xc1\xc0\xaf"\xd5\xaeI\xc0\xbd\x83\xe5\x9aA\\\xe8?\xa4\x99\
x14\xe5D\xb7\xd3 2?\xa6r\x94\x99\xd4\xe6\x96\xd5\xac1\xf9\xc8O\xd0\xa4\x83\xb9\x
d8WL\xd7u\xce\x80\r\xe5\x18\xf5\xc50\ru\xcbk\x8e\x99\x88\xa4\x80\xc5\x93{\x96\xd
0S\xd0\xf8\xc1\xaf\xc8h\t\xb0m\xd1\x80Q\xa7\x0e\xe5\xfaukm\xd3k\xd2\xdb\x9e\xb4\
xbf\x80\xfac\x8c\xd0\x0f\x92\xee\xd1\x90\x8d\x00\x1e\xa2m\x8d\xc2G\xb71\xa0\x97b
xm\x8f)\xc4\xcb\xa1\xf1\xd3\x97\xe6c\xd0\x92\x19\x82\xd1\x94\xfe\x9a\x1d\x1e\xfe
/\x9b\xd2x\xf3V\xb7\x8fb$/\x999\xfb\x8f\xcb\xe6\xca\x97\xba!\xc6\x82&\xc6\xbc\x8
3\xe5\x9aA\\\xe8?\xa4\x96\x16\xe5D\xb7\xd3 2?\xa6~\x93\x99\xd4\xe6\x96\xd5\xac1\
xf9\xc5I\xd0\xa4\x83\xb9\xd8WL\xd7y\xcd\x80\r\xe5\x18\xf5\xc50\ry\xcek\x8e\x99\x
88\xa4\x80\xc5\x93w\x92\xd0S\xd0\xf8\xc1\xaf\xc8h\n\xb9m\xd1\x80Q\xa7\x0e\xe5\xf
aqkm\xd3k\xd2\xdb\x9e\xb4\xbf\x84\xfac\x8c\xd0\x0f\x92\xee\xd1\x90\x89\x05\x1e\x
a2m\x8d\xc2G\xb71\xa4\x96bxm\x8f)\xc4\xcb\xa1\xf5\xd1\x97\xe6c\xd0\x92\x19\x82\x
d1\x91\xf9\x9a\x1d\x1e\xfe/\x9b\xd2x\xf7Y\xb7\x8fb$/\x999\xfb\x8b\xcc\xe6\xca\x9
7\xba!\xc6\x82&\xc1\xb8\x83\xe5\x9aA\\\xe8?\xa4\x91\x13\xe5D\xb7\xd3 2?\xa6z\x91
\x99\xd4\xe6\x96\xd5\xac1\xf9\xc1K\xd0\xa4\x83\xb9\xd8WL\xd7|\xc8\x80\r\xe5\x18\
xf5\xc50\r|\xc9k\x8e\x99\x88\xa4\x80\xc5\x93z\x87\xdbX\x87\xed\xc2\xbc\xe4\x02\x
1e\xa2m\x8d\xc2G\xb71\xa4\x84is:\x9a*\xd7\xe7\xcd'
Plaintext: Mensagem a ser cifrada
```