Estruturas Criptográficas 2022/23 — TP1. Problema 2

Grupo 7. Leonardo Berteotti e Paulo R. Pereira

May 2, 2023

[1]:
```
import random
import numpy as np
from cryptography.hazmat.primitives import hashes
```

1 Implementação do KEM IND-CPA-secure

Na resolução do problema será criada uma classe que implementerá o algoritmo KEM, com o objetivo de ofuscar chaves, que a própria classe gera. Para cumprir esse objetivo, é necessário implementar 3 funcionalidades:

Encapsulamento

Esta função tem o objetivo de ofuscar a chave gerada pelo algoritmo. Para isso, inicialmente, a função gera pequenos erros a partir da função *errorGen* que gera dois polinómios de tamanho r, sendo que a soma dos pesos dos dois erros seja igual a t. Após este primeiro passo, gera-se a hash da informação a ser encapsulada utilizando os erros gerados. Para encapsular a informação, é necessário gerar um r aleatório denso, utilizando o anel cíclico polinomial R, que será utilizando juntamente com os valores da chave pública e os erros para gerar o encapsulamento da informação k, tal como está na seguinte expressão $(y0, y1)_{¸}(r * f0 + e0, r * f1 + e1)$, com $(f0, f1)$ - chave publica e $(e0, e1)$ - pequenos erros.

Desencapsulamento

Para além do encapsulamento, é necessário o desencapsulamento da chave gerada pela classe. Nesta função é calculada a matriz dispersa H e ao syndrome s, sendo estes utilziados no algoritimo *bifFlip*, para descodificar os erros gerados no encapsulamento. Esses erros são necessários para gerar a hash da informação. Em baixo mostram-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 $rot(h0), rot(h1)$, que são calculadas utilizando a função *Rot*. Esta função tem como objetivo gerar a matriz de rotação a partir de um vetor utilizando as funções *toVectorR* e *rot*. Desta forma conseguimos obter a matriz dispensa H = $(rot(h0)|rot(h1))$.
- **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.
  - 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 $z \cap h_j$ 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 $x_0 = r * f_0 $ e $x_1 = r * f_1$ então temos:

    $y_0 = r * f_0 + e_0 \equiv e_0 = y_0 - r * f_0 \equiv e_0 = y_0 - x_0 y_1 = r * f_1 + e_1 \equiv e_1 = y_1 - r * f_1 \equiv e_1 = y_1 - x_1$
  - Com estas equações conseguimos obter os valores e_0 e e_1 que serão usados de forma a verificar a condição $|e_0 + e_1| = t$, com $|e_0 + e_1|$ igual à soma dos pessos de e_0 e e_1. Caso contrário ocorreu um error no processo de desencapsulamento.
  - Finalmente, os valores e_0 e e_1 serão utilizados para calcular a hash da informação encapsulada e assim desencapsular essa informação.

- **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 iteraçõa 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:

keyGen

Esta função tem o objetivo de gerar um par de chaves que será utilizado no encapsulamento de desencapsulamento da chave gerada pela classe. A função começa por a chave privada do problema a partir da função *coeffGen* que gera 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, sendo h0 e h1 os parâmteros da chave privada.

[2]:
```
class BIKE:
    def __init__(self):
        # (block length): a prime number such that 2 is primitive modulo r
        r = 257
        # (error weight): a positive integer
        t = 16
        n = 2*r

        # Grupo Finito com 2 elementos
        F2 = GF(2)

        # Anel de polinomíos
        F = PolynomialRing(F2, name='w')
        w = F.gen()
```

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

        self.r = r
        self.t = t
        self.n = n
        self.F2 = F2
        self.R = 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):
```

```python
            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:
            nIter = nIter - 1
            # Todos os pesos 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

    def keyGen(self):
        # private key
        h0 = self.coeffGen()
        h1 = self.coeffGen()
        # public key
        f = (1, h0/h1)
        return (h0, h1) , f
```

[3]:

```python
class BIKE_KEM(BIKE):
    def __init__(self):
       BIKE.__init__(self)

    # 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)

        return key, (y0,y1)

    # 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
```

[4]:
```
bike = BIKE_KEM()

private, public = bike.keyGen()

toEncap, cipheredKey = bike.encaps(public)
print("Key:\n", toEncap)

toDecap = bike.decaps(private,cipheredKey)
print("\nKey:\n", toDecap)

if toDecap != None and toDecap == toEncap:
    print("\nKeys match!")
else:
    print("\nKeys don't match!")
```

```
Key:
 b"r\xd0;'\x8b9ui\xc9\x99\x15p}\xde\t\x99\x07\xa2Rt7\xae\t\xd2\xe0\x04V\xf5\x0c\
xcd0a"

Key:
 b"r\xd0;'\x8b9ui\xc9\x99\x15p}\xde\t\x99\x07\xa2Rt7\xae\t\xd2\xe0\x04V\xf5\x0c\
xcd0a"

Keys match!
```

2 Implementação do PKE IND-CCA-secure

A implementação segue a transformação de Fujisaki-Okamoto (FO) de forma a obter, a partir de um KEM-IND-CPA, um PKE-IND-CCA seguro.

keyGen

É utilizado o método *keyGen* implementado na classe BIKE_KEM. O objetivo é gerar a chave privada e a chave pública.

Encrypt

O método segue o algoritmo de cifra da transformação FO:

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

Começa-se por gerar os pequenos erros com recurso à função *errorGen* da classe BIKE_KEM De seguida, calcula-se o hash de um valor r aleatorio denso gerado com recurso ao anel ciclíco R. É efetuado o XOR deste valor (o hash) com a mensagem m a ser cifrada. A função f corresponde à função *encaps* já implementada na classe BIKE_KEM, e é usada para gerar a chave k e o encapsulmento dos erros e. Por fim, o *ciphertext* é calculado a partir do XOR da chave k com o valor aleatório r.

Decrypt

Este método é implementado segundo o algoritmo

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

Começa-se por usar as funções *decapsError* e *decapsKey* para desencapsular os erros $(e0, e1)$ e a chave k, respetivamente. De seguida, obtém-se o valor de r através do XOR entre o *ciphertext* c e a chave k. Se os valores calculados a partir de f (tal como no processo de *encrypt*) forem iguais ao e recebido e à revelação da chave, confirma-se a autenticidade da chave e gera-se o hash de r que será utilizado na operação de XOR com y, obtendo a mensagem original.

[5]:
```
class BIKE_PKE(BIKE):

    def __init__(self):
       BIKE.__init__(self)

    # Hash
    def g(self, r):
        digest = hashes.Hash(hashes.SHA256())
        digest.update(str(r).encode())
        g = digest.finalize()
        return g

    # 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
```

```python
    # 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.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.hammingWeight(self.toVectorR(e0)) + self.hammingWeight(self.
↪toVectorR(e1)) != self.t:
            print("Erro no decoding")
            return None
        else:
            key = self.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.Rot(private[0])
        h1Rot = self.Rot(private[1])
        H = block_matrix(2,1,[h0Rot,h1Rot])

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

        # Computa o syndrome
        s = vectorE * H

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

    def encrypt(self, msg, public):
        e0, e1 = self.errorGen(self.t)
        r = self.R.random_element()
        g = self.g(r)
        y = self.xor(msg.encode(),g)

        yBinary = bin(int.from_bytes(y, byteorder=sys.byteorder))
        ryBinary = self.R(yBinary)

        key, e = self.f(public, ryBinary + r, e0, e1)
        c = self.xor(str(r).encode(),key)

        return y, e, c

    def decrypt(self, private,public, y, e, c):
        e0, e1 = self.decapsError(private,e)
        k = self.decapsKey(e0,e1)

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

        yBinary = bin(int.from_bytes(y, byteorder=sys.byteorder))
        ryBinary = self.R(yBinary)

        if (k,e) != self.f(public, ryBinary + r, e0, e1):

            print("Erro no decoding")
            return None

        else:
            g = self.g(r)
            plaintext = self.xor(y,g)

        return plaintext
```

[6]:
```
bike = BIKE_PKE()
msg = "Hello there!"
print("Original message:")
print(msg)

sk, pk = bike.keyGen()

msg_encapsulation, key_encapsulation, ciphertext = bike.encrypt(msg, pk)
print("\nCiphertext: ")
print(ciphertext)
```

```python
plaintext = bike.decrypt(sk, pk, msg_encapsulation, key_encapsulation,␣
 ↪ciphertext)
print("\nDecrypted message:")
print(plaintext.decode())
```

```
Original message:
Hello there!

Ciphertext:
b"\x97y\xc7uQ\xbd\xac\xa1\xbf\x0e\xc9\xed\xf2\x83\x17\xcd\xc3\xc0\xfc\xc2\xa7\xf
3y\x86w\xad\xee=_\xd8\x8c\xc6\xc0l\xc4f}\xd1\xab\xa3\xa7\x05\xc2\xba\xe7\x80\x04
\xe1\xaf\xc7\xfb\xd5\xac\xf8.\x93t\xbe\xc2QY\xd5\x9a\xcd\xcb;\xd1en\xfd\xc7\xa5\
xab\x10\xc9\xb1\xb0\x95\x07\xf2\x83\xab\xfd\xd6\xb8\xf3%\xc4a\xbd\xd1}5\xd3\x98\
xde\xc00\x86pm\xee\xeb\xc9\xad\x11\xdb\xba\xbb\xc2\x12\xf1\x90\x87\x91\xd0\xb8\x
e2.\xcf6\xa8\xd2n\x19\xbf\x9e\xde\xd8;\x8d'x\xed\xf8\xe5\xc1\x17\xda\xad\xb0\xc9
E\xe4\x93\x94\xbd\xbc\xbe\xe0;\xc4=\xff\xc7m\n\x93\xf2\xdf\xd3/\x86,/\xf8\xfb\xf
6\xed{\xdb\xa9\xa3\xc2N\xb3\x86\x97\xae\x90\xd2\xe1=\xd46\xf4\x90x\t\x80\xde\xb3
\xd2)\x9f'$\xaf\xee\xf5\xfeW\xb7\xa8\xa2\xdaE\xb8\xd1\x82\xad\x83\xfe\x8d<\xd6!\
xff\x9b/\x1c\x83\xcd\x9f\xbe)\x942/\xa4\xb9\xe0\xfdD\x9b\xc4\xa2\xd0Q\xb3\xda\xd
5\xb8\x80\xed\xa1P\xd6$\xec\x90$K\x96\xce\x8c\x92E\x945=\xaf\xb2\xb7\xe8G\x88\xe
8\xce\xd0W\xa2\xd1\xde\xef\x95\xee\xb2|\xba$\xee\x89/@\xc1\xdb\x8f\x81i\xf85>\xb
7\xb9\xbc\xbfR\x8b\xfb\xe2\xbcW\xa2\xc7\xd5\xe4\xc2\xfb\xb1o\x96H\xed\x81:K\xca\
x8c\x9a\x82z\xd4Y=\xbe\xad\xb7\xb4\x05\x9e\xf8\xf1\x90;\xa1\xc0\xc7\xef\xc9\xac\
xa4l\x85d\x81\x82>[\xc1\x87\xcd\x97y\xc7uQ\xbd\xa9\xae\xbf\x0e\xc9\xed\xf2\x83\x
17\xcd\xc3\xc5\xf8\xc2\xa7\xf3y\x86w\xad\xee=[\xd4\x8c\xc6\xc0l\xc4f}\xd1\xab\xa
7\xab\x05\xc2\xba\xe7\x80\x04\xe1\xaf\xc7\xff\xd3\xac\xf8.\x93t\xbe\xc2QY\xd1\x9
c\xcd\xcb;\xd1en\xfd\xc7\xa6\xa6\x1c\xc9\xb1\xb0\x95\x07\xf2\x83\xab\xfe\xdb\xb9
\xf3%\xc4a\xbd\xd1}5\xd0\x95\xde\xc00\x86pm\xee\xeb\xc9\xae\x1c\xdb\xba\xbb\xc2\
x12\xf1\x90\x87\x91\xd3\xb5\xe2.\xcf6\xa8\xd2n\x19\xbf\x9d\xd5\xd8;\x8d'x\xed\xf
8\xe5\xc1\x14\xd1\xae\xb0\xc9E\xe4\x93\x94\xbd\xbc\xbd\xe47\xc4=\xff\xc7m\n\x93\
xf2\xdc\xd7#\x86,/\xf8\xfb\xf6\xed{\xd8\xad\xa7\xc2N\xb3\x86\x97\xae\x90\xd2\xe2
9\xd06\xf4\x90x\t\x80\xde\xb3\xd1,\x94'$\xaf\xee\xf5\xfeW\xb7\xab\xa6\xdbE\xb8\x
d1\x82\xad\x83\xfe\x8d?\xd2.\xff\x9b/\x1c\x83\xcd\x9f\xbe*\x900/\xa4\xb9\xe0\xfd
D\x9b\xc4\xa1\xd4V\xb3\xda\xd5\xb8\x80\xed\xa1P\xd5 \xed\x90$K\x96\xce\x8c\x92E\
x971>\xaf\xb2\xb7\xe8G\x88\xe8\xce\xd3S\xa3\xd1\xde\xef\x95\xee\xb2|\xba'\xea\x8
9/@\xc1\xdb\x8f\x81i\xf86:\xb8\xb9\xbc\xbfR\x8b\xfb\xe2\xbcT\xa6\xc7\xd5\xe4\xc2
\xfb\xb1o\x96H\xee\x85;K\xca\x8c\x9a\x82z\xd4Y>\xba\xa8\xb7\xb4\x05\x9e\xf8\xf1\
x90;\xa2\xc5\xcd\xef\xc9\xac\xa4l\x85d\x81\x81;X\xc1\x87\xcd\x97y\xc7uQ\xbe\xad\
xa6\xbf\x0e\xc9\xed\xf2\x83\x17\xcd\xc0\xc1\xff\xc2\xa7\xf3y\x86w\xad\xee>X\xd7\
x8c\xc6\xc0l\xc4f}\xd1\xa8\xa4\xaa\x05\xc2\xba\xe7\x80\x04\xe1\xaf\xc4\xfc\xd6\x
ac\xf8.\x93t\xbe\xc2QZ\xd2\x9f\xcd\xcb;\xd1en\xfd\xc7\xa6\xac\x17\xc9\xb1\xb0\x9
5\x07\xf2\x83\xab\xfe\xd1\xbd\xf3%\xc4a\xbd\xd1}5\xd0\x9f\xdd\xc00\x86pm\xee\xeb
\xc9\xae\x17\xde\xba\xbb\xc2\x12\xf1\x90\x87\x91\xd3\xbe\xe6.\xcf6\xa8\xd2n\x19\
xbf\x9d\xdf\xd4;\x8d'x\xed\xf8\xe5\xc1\x14\xdb\xab\xb0\xc9E\xe4\x93\x94\xbd\xbc\
xbd\xe27\xc4=\xff\xc7m\n\x93\xf2\xdc\xd1-\x86,/\xf8\xfb\xf6\xed{\xd8\xab\xa5\xc2
N\xb3\x86\x97\xae\x90\xd2\xe2?\xd06\xf4\x90x\t\x80\xde\xb3\xd1*\x94'$\xaf\xee\xf
```

```
5\xfeW\xb7\xab\xa0\xdbE\xb8\xd1\x82\xad\x83\xfe\x8d?\xd4.\xff\x9b/\x1c\x83\xcd\x
9f\xbe*\x960/\xa4\xb9\xe0\xfdD\x9b\xc4\xa1\xd2S\xb3\xda\xd5\xb8\x80\xed\xa1P\xd5
&\xea\x90$K\x96\xce\x8c\x92E\x977;\xaf\xb2\xb7\xe8G\x88\xe8\xce\xd3U\xa0\xd1\xde
\xef\x95\xee\xb2|\xba'\xef\x81/@\xc1\xdb\x8f\x81i\xf86?\xbf\xb9\xbc\xbfR\x8b\xfb
\xe2\xbc\\\xaa\xd1\xde\xef\x95\xee\xb2|\xba/\xed\x90$K\x96\xce\x8c\x92E\x9e>/\xa
4\xb9\xe0\xfdD\x9b\xc4\xa8\xdaE\xb8\xd1\x82\xad\x83\xfe\x8d6\xd26\xf4\x90x\t\x80
\xde\xb3\xd8.\x86,/\xf8\xfb\xf6\xed{\xd1\xa9\xb0\xc9E\xe4\x93\x94\xbd\xbc\xbb\xe
b.\xcf6\xa8\xd2n\x19\xbf\x9b\xd8\xc00\x86pm\xee\xeb\xc9\xa8\x16\xc9\xb1\xb0\x95\
x07\xf2\x83\xab\xf8\xd0\xac\xf8.\x93t\xbe\xc2Q\\\xd0\x8c\xc6\xc0l\xc4f}\xd1\xaf\
xa0\xbf\x0e\xc9\xed\xf2\x83\x17\xcd\xc7\xc3\xef\xc9\xac\xa4l\x85d\x81\x86;K\xca\
x8c\x9a\x82z\xd4Y9\xbc\xb9\xbc\xbfR\x8b\xfb\xe2\xbcS\xa2\xd1\xde\xef\x95\xee\xb2
|\xba \xef\x90$K\x96\xce\x8c\x92E\x93?/\xa4\xb9\xe0\xfdD\x9b\xc4\xa5\xd1E\xb8\xd
1\x82\xad\x83\xfe\x8d;\xd56\xf4\x90x\t\x80\xde\xb3\xd4,\x86,/\xf8\xfb\xf6\xed{\x
dd\xac\xb0\xc9E\xe4\x93\x94\xbd\xbc\xb8\xe0.\xcf6\xa8\xd2n\x19\xbf\x98\xdf\xc00\
x86pm\xee\xeb\xc9\xab\x15\xc9\xb1\xb0\x95\x07\xf2\x83\xab\xfc\xda\xac\xf8.\x93t\
xbe\xc2QX\xd5\x8c\xc6\xc0l\xc4f}\xd1\xaa\xa6\xbf\x0e\xc9\xed\xf2\x83\x17\xcd\xc2
\xc5\xef\xc9\xac\xa4l\x85d\x81\x826K\xca\x8c\x9a\x82z\xd4Y=\xb7\xb9\xbc\xbfR\x8b
\xfb\xe2\xbcW\xa4\xd1\xde\xef\x95\xee\xb2|\xba$\xe9\x90$K\x96\xce\x8c\x92E\x972/
\xa4\xb9\xe0\xfdD\x9b\xc4\xa1\xd6E\xb8\xd1\x82\xad\x83\xfe\x8d?\xd56\xf4\x90x\t\
x80\xde\xb3\xd4"

Decrypted message:
Hello there!
```