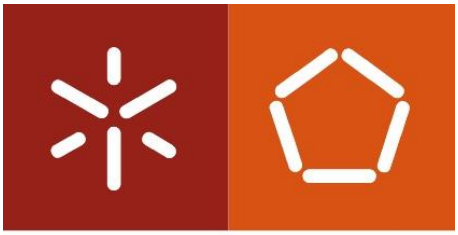

Universidade do Minho

Departamento de Informática

Mestrado em Engenharia Informática

Criptografia e Segurança de Informação

Engenharia de Segurança
Ficha Prática 1

Grupo Nº 3

Ariana Lousada (PG47034) Luís Carneiro (PG46541)
Rui Cardoso (PG42849)

7 de março de 2022

Capítulo 1

Parte I: Criptografia – conceitos básicos

1.1 Pergunta P1.1

Hoje é reconhecida a importância de eliminar o secretismo como factor na segurança dos sistemas criptográficos ("no securith through obscurity"). Existem autores que defendem que tal já está retratados nos Princípios de Kerckhoff de 1883. Comente (mas não exceda o equivalente a meia folha de papel A4).

A importância de eliminar o secretismo como fator na segurança dos sistemas criptográficos está, de facto, implícita na segunda lei de Kerckhoff - "o sistema criptográfico não deve requerer secretismo, e não deve ser um problema caso caia em mãos inimigas".

Esta afirmação baseia-se na premissa que é muito mais fácil manter chaves relativamente pequenas seguras, do que todo um sistema criptográfico.

Por exemplo, para descobrir o funcionamento detalhado de um sistema criptográfico qualquer, é possível subornar, chantangear ou até ameaçar quem tiver algum tipo de conhecimento do sistema; com acesso ao software, é exequível corrê-lo em modo debugging ou até fazer um *core dump*; se for utilizado algum hardware específico, este pode ser desmantelado e analisado a fundo. Para além disto, quantas mais pessoas souberem como este sistema funciona, maior é o risco da divulgação do seu funcionamento.

Por outro lado, a chave, à partida, não vai ser divulgada pelo utilizador, pelo que o atacante só a obtem caso a consiga decifrar em tempo útil, o que não deve acontecer num sistema criptográfico dito seguro. Mesmo que esta seja decifrada, é (relativamente) fácil gerar uma nova e resolver rapidamente o problema.

Adicionalmente, ao tornar o sistema público é mais provável e mais rápido detetar falhas no seu design e/ou implementação, pois este é mais analisado.

Por último, implementar o secretismo por cima de outras boas práticas de segurança não é, de todo, criticado: só quando é o único mecanismo usado para, neste caso, garantir a segurança de sistemas criptográficos.

Capítulo 2

Parte II: Exemplos de Cifras Clássicas

2.1 Pergunta P1.1

Uma maneira de dificultar o ataque por força bruta a uma mensagem cifrada com cifra de Cesar, é encadear várias cifras de Cesar.

Utilizando a técnica do exercício anterior, desenvolva um programa em python3 que:

- **cifre cleartext aplicando várias cifras de Cesar, sequencialmente.**
- **decifre ciphertext aplicando várias cifras de Cesar, sequencialmente.**
- **faça um ataque de força bruta ao espaço de todas as chaves de cifras sequenciais aplicadas ao ciphertext (por uma questão de simplificação, um dos parâmetros a passar a esta função é a quantidade de cifras sequenciais aplicadas ao ciphertext), indicando no final qual o cleartext que tem mais probabilidade de ser o texto original.**
- **No final faça a comparação dos tempos necessários para ataque de força bruta a um texto de 100 caracteres, para uma a 5 cifras sequenciais. Em cada um dos casos indique qual é o espaço total de chaves que estão a ser avaliadas.**

No seguinte código Python3 foi utilizada a libraria pycipher para cifrar determinado texto, neste caso "Engenharia de Segurança", sequencialmente 10000 vezes através de cifras de César com chaves aleatórias. Em seguida foi utilizada uma função de força bruta para tentar decifrar o texto.

```python
from pycipher import Caesar
import random
import os
from ngram_score import ngram_score

#Parametros
numeroEncandeamentos = 10000
textoInicial = 'Engenharia de Segurança'
quadgramfile = 'spanish_quadgrams.txt'


#Cifra
print("Started Ciphering text: " + textoInicial)
for x in range(numeroEncandeamentos):
    key = random.randint(1,26) #Chave para a cifra
    if x==0:
        finalCypher = Caesar(key).encipher(textoInicial)
    else:
        finalCypher = Caesar(key).encipher(finalCypher)

#Decifer
print("Started Decifering text: " + finalCypher)

#Abrir o ficheiro dos quadgram
dir_path = os.path.dirname(os.path.realpath(__file__))
quadgram = os.path.join( dir_path, quadgramfile)
fitness = ngram_score(quadgram) # load our quadgram statistics

#Função de bruteforce
def break_caesar(ctext):
    scores = []
    for j in range(numeroEncandeamentos):
        for i in range(26):
            scores.append((fitness.score(Caesar(i).decipher(ctext)),i))
        #print("Progress: "+ str(j)+"/" + str(numeroEncandeamentos))
    return max(scores)

max_key = break_caesar(finalCypher)
print (Caesar(max_key[1]).decipher(finalCypher))
```

```
Started Ciphering text: Engenharia de Segurança
Started Decifering text: TCVTCWPGXPSTHTVJGPCP
ENGENHARIADESEGURANA
[Finished in 11.9s]
```

Figura 2.1: Main.py `dcode`.

No seguinte código python3 classificaram-se as tentativas de força bruta em relação ao quadgram espanhol.

```python
from math import log10

class ngram_score(object):
    def __init__(self,ngramfile,sep=' '):
        ''' load a file containing ngrams and counts, calculate log probabilities '''
        file1 = open('ngramfile', 'r')
        self.ngrams = {}
        with open(ngramfile) as file:
            for line in file:
                key,count = line.split(sep)
                self.ngrams[key] = int(count)
            self.L = len(key)
            self.N = sum(self.ngrams.values())
            #calculate log probabilities
            for key in self.ngrams.keys():
                self.ngrams[key] = log10(float(self.ngrams[key])/self.N)
            self.floor = log10(0.01/self.N)

    def score(self,text):
        ''' compute the score of text '''
        score = 0
        ngrams = self.ngrams.__getitem__
        for i in range(len(text)-self.L+1):
            if text[i:i+self.L] in self.ngrams: score += ngrams(text[i:i+self.L])
            else: score += self.floor
        return score
```

Figura 2.2: ngram_score.py

No gráfico seguinte pode se ver o tempo que demora em milissegundos a decifrar o texto 'Engenharia de Segurança' por força bruta.

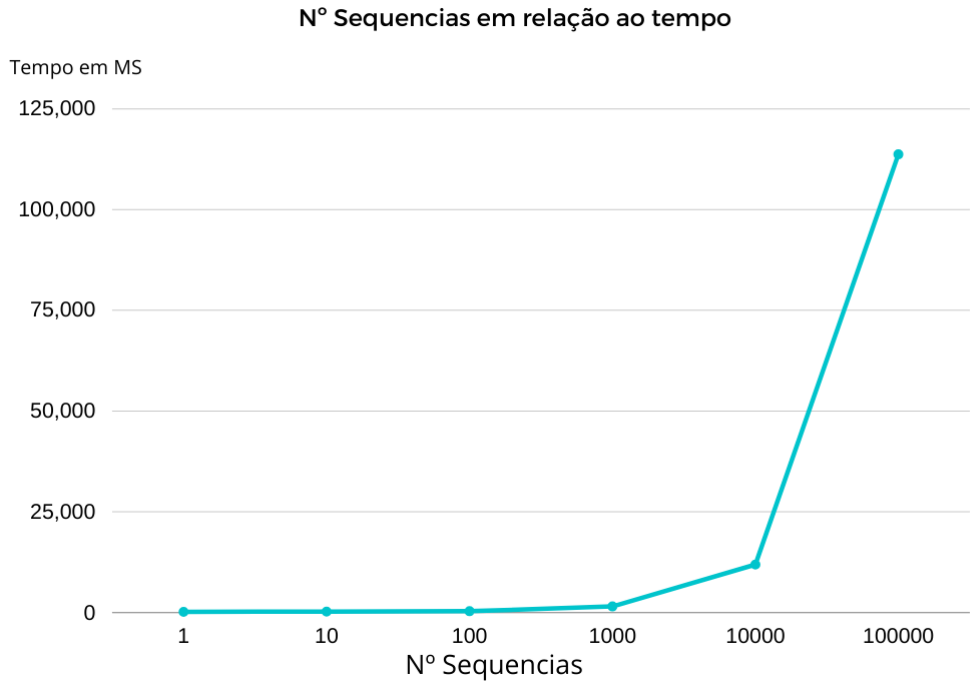

Figura 2.3: Tempo relação sequência

Por ultimo foi utilizado um texto com 100 caracteres "OjalamissueñossehicieranrealidadsehicieranrealidadporquetengounmontonDoraemonpuedehacerquesecumplant" para 1 a 5 cifras sequenciais.

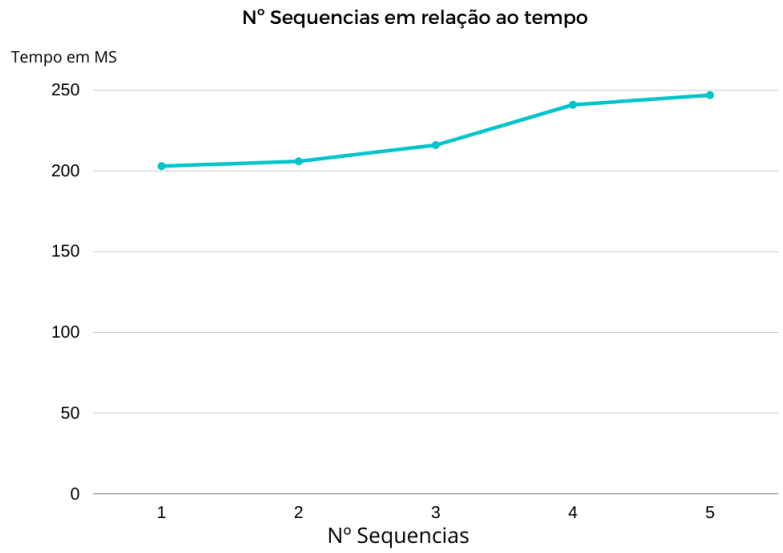

Figura 2.4: Tempo relação sequência

Todos os testes foram feitos num processador Intel I7-4510U @ 3.1GHz.
Para calcular o total de chaves para cada caso utilizou-se a seguinte formula:

$$26 * (total\ numero\ de\ caracteres)^{Numero\ de\ sequencias}$$

1 sequência = 26
2 sequências = 676
3 sequências = 17 576
4 sequências = 456 976
5 sequências = 11 881 376

2.2 Pergunta P2.1

Se for utilizada uma cifra por substituição mono-alfabética (com chave desconhecida), e interceptar o ciphertext `OXAO`, justifique se o plaintext correspondente pode ser `DATA`.

Após a utilização da ferramenta `dcode` na experiência 2.1. decidiu-se utilizar a opção `Attack with probable word/known plain text` com a *word* `DATA`.

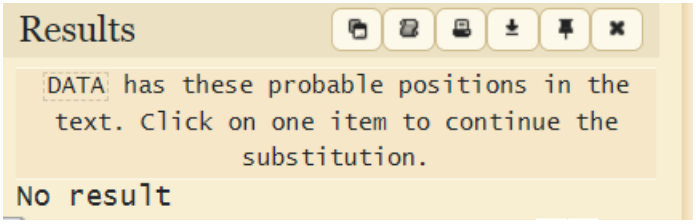

Figura 2.5: Resultados da ferramenta `dcode`.

Numa cifra mono-alfabética(definida por apenas um alfabeto de cifragem) cada caractere do `plain text` está mapeado a um outro caractere fixo do *cipher text*. Este mapeamento é único para cada par de caracteres.

Tendo isto em conta, o *cipher text* `OXAO` nunca poderá a corresponder ao *plain text* `DATA`. Para isto ser possível, a mesma letra do alfabeto de cifragem, como por exemplo o caractere `O`, teria que mapear dois caracteres diferentes, neste caso `D` e `A` simultaneamente. Isto iria originar ambiguidade, daí a este tipo de mapeamento não ser possível neste tipo de cifra.

2.3 Pergunta P3.1

A cifra de Hill é um exemplo de cifra por substituição poli-alfabética (para além do exemplo clássico da cifra de Vigenère que foi visto na aula teórica), baseada na álgebra linear.
Utilize código já desenvolvido (procure no github, por exemplo) para cifrar e decifrar utilizando a cifra de Hill.
Note que na resposta a esta pergunta tem que incluir a descrição de todos os passos que o código está a fazer, assim como identificar as componentes do código onde tal é feito.

A cifra de Hill trata-se de uma cifra de substituição poligráfica que tem como base conceitos de álgebra linear. Cada letra é normalmente representada por um valor resultante da divisão inteira por 26. De modo a cifrar uma mensagem, cada bloco de n letras é multiplicado por uma matriz de dimensão $n * n$ invertível, dividindo os seus valores posteriormente por 26 e substituindo-os pelo valor de resto resultante. Para decifrar uma mensagem, cada bloco é multiplicado pela matriz inversa utilizada para a sua cifra. Esta matriz corresponde à *key* de codificação e deve ser escolhida aleatoriamente do *set* de matrizes de dimensão $n * n$ invertíveis possíveis.

Após uma pesquisa por parte da equipa de trabalho encontrou-se o seguinte excerto de código python que implementa a cifra de Hill, utilizando o esquema de código ASCII para cada representar cada letra.

Para cifrar uma mensagem:

```
def encrypt(msg):
    # Substituir espacos em branco por "nothing" (limpeza de input)
    msg = msg.replace(" ", "")
    # Criacao da matriz key
    C = make_key()
    # Verificar se o codigo e divisivel por 2. Caso nao seja adicionar 0.
    len_check = len(msg) % 2 == 0
    if not len_check:
        msg += "0"
    # Criar matriz de input
    P = create_matrix_of_integers_from_string(msg)
    # Calculo do tamanho da mensagem
    msg_len = int(len(msg) / 2)
    # Calcular P * C
    encrypted_msg = ""
    for i in range(msg_len):
        # Calculo do produto escalar
        row_0 = P[0][i] * C[0][0] + P[1][i] * C[0][1]
        # Divisao inteira pelo valor 26. Adicao de 65 para retornar a codificacao A-Z de
     ascii
        integer = int(row_0 % 26 + 65)
        # Converter novamente para chr e adicionar ao texto
        encrypted_msg += chr(integer)
        # Repetir calculos para a segunda coluna
        row_1 = P[0][i] * C[1][0] + P[1][i] * C[1][1]
        integer = int(row_1 % 26 + 65)
        encrypted_msg += chr(integer)
    return encrypted_msg
```

Função auxiliar para criar a matriz chave:

```
def make_key():
     # Make sure cipher determinant is relatively prime to 26 and only a/A - z/Z are
    given
     # Verificar se o determinante da matriz e relativamente primo para 26. Verificar
    tambem se o input contem apenas letras (a-z e A-Z).
    determinant = 0
    C = None
    while True:
        cipher = input("Input 4 letter cipher: ")
        C = create_matrix_of_integers_from_string(cipher)
        determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
        determinant = determinant % 26
        inverse_element = find_multiplicative_inverse(determinant)
        if inverse_element == -1:
            print("Determinant is not relatively prime to 26, uninvertible key")
        elif np.amax(C) > 26 and np.amin(C) < 0:
            print("Only a-z characters are accepted")
            print(np.amax(C), np.amin(C))
        else:
            break
    return C
```

Para decifrar uma mensagem:

```
def decrypt(encrypted_msg):
    # Pedir pela keyword para obter a matriz chave
    C = make_key()
    # Calcular a matriz inversa da matriz chave
    determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
    determinant = determinant % 26
    multiplicative_inverse = find_multiplicative_inverse(determinant)
    C_inverse = C

    C_inverse[0][0], C_inverse[1][1] = C_inverse[1, 1], C_inverse[0, 0]
    # Substutuir valores
    C[0][1] *= -1
    C[1][0] *= -1
    for row in range(2):
        for column in range(2):
            C_inverse[row][column] *= multiplicative_inverse
            C_inverse[row][column] = C_inverse[row][column] % 26

    P = create_matrix_of_integers_from_string(encrypted_msg)
    msg_len = int(len(encrypted_msg) / 2)
    decrypted_msg = ""
    # Aplicar os mesmos calculos a mensagem usados na funcao encrypt(), utilizando a
    matriz inversa calculada anteriormente.
    for i in range(msg_len):

        column_0 = P[0][i] * C_inverse[0][0] + P[1][i] * C_inverse[0][1]

        integer = int(column_0 % 26 + 65)

        decrypted_msg += chr(integer)

        column_1 = P[0][i] * C_inverse[1][0] + P[1][i] * C_inverse[1][1]
        integer = int(column_1 % 26 + 65)
        decrypted_msg += chr(integer)
    if decrypted_msg[-1] == "0":
        decrypted_msg = decrypted_msg[:-1]
    return decrypted_msg

def find_multiplicative_inverse(determinant):
    multiplicative_inverse = -1
    for i in range(26):
        inverse = determinant * i
        if inverse % 26 == 1:
            multiplicative_inverse = i
            break
    return multiplicative_inverse
```

Função auxiliar que calcula a matriz correspondente à mensagem por cifrar (proveniente do input):

```
def create_matrix_of_integers_from_string(string):
    # Colocar em cada lugar da matriz o codigo ascii correspondente de cada letra
    integers = [chr_to_int(c) for c in string]
    length = len(integers)
    M = np.zeros((2, int(length / 2)), dtype=np.int32)
    iterator = 0
    for column in range(int(length / 2)):
        for row in range(2):
            M[row][column] = integers[iterator]
            iterator += 1
    return M
```

2.4 Pergunta P4.1

Explique porque é que, na cifra one-time pad, os problemas inerentes à geração e distribuição da chave, assim como a necessidade de utilizar um "verdadeiro" gerador de números aleatórios, tornam (na prática) a cifra inviável.

Na cifra one-time pad, os problemas inerentes à geração e distribuição da chave, assim como a necessidade de utilizar um "verdadeiro" gerador de números aleatórios, tornam (na prática) a cifra inviável.

Primeiramente, a chave gerada tem que ser verdadeiramente aleatória, o que não é trivial de alcançar; isto apenas é possível através de fenómenos quânticos que, por definição, são aleatórios e impossíveis de prever. São exemplos o ruído térmico produzido dentro de um circuito elétrico ou até o decaimento radioativo de uma fonte nuclear. Mesmo assim, pode ser necessário corrigir a tendência dos números gerados. Só este requisito implica que este protocolo não seja de todo escalável para uso em massa.

Adicionalmente, a chave tem de ser conhecida exclusivamente *à priori* pelas entidades que querem comunicar. Ora, este conhecimento prévio da chave torna esta cifra realmente impraticável para o uso no quotidiano e na Internet, visto que só nos permitiria comunicar com conhecidos (com todos os problemas que a geração da chave acarreta).

É de notar que esta cifra ainda pode ter alguns usos práticos, nomeadamente na espionagem e comunicação militar, pois este método permite cifrar e decifrar à mão, com apenas papel e caneta.

Outros requisitos não mencionados no enunciado desta pergunta que inviabilizam a sua adoção ainda mais incluem a não reutilização da chave utilizada para cifrar uma mensagem, o tamanho da chave ser pelo menos igual ao da mensagem a transmitir e a destruição da chave após o seu uso.

2.5 Pergunta P5.1

Desenvolva o código para a cifra/decifra por transposição dupla, documentando adequadamente o código desenvolvido. Na resposta a esta pergunta inclua um exemplo manual (i.e., exemplificando sem o uso do código desenvolvido) de como funciona a dupla transposição para o cleartext 'este exemplo mostra a transposicao dupla', com as chaves de transposição MINHOve ENGSEG.

O código para cifrar com transposição dupla de colunas em Python3 para o texto 'este exemplo mostra a transposicao dupla' com as chaves de transposição 'Minho' e 'ENGSEG' vai ser apresentado de seguida.

```python
import re
import numpy as np
import math

key = 'MINHO'
secondKey ='ENGSEG'
text = 'este exemplo mostra a transposicao dupla'

#Primeiro precisamos de saber quantas letras tem a primeira key
#Para sabermos o numero de columas da matriz
letersAmount = len([x for x in key if x.isalpha()])

#Agora removemos o espaços do texto
text = "".join(text.split())

#Agora precisamos de calcular o numero de linhas da matriz
#Fazemos isto ao dividir o comprimento do texto pelo numero
#de letras da primeira key e arredondamos para o numero inteiro
#mais alto
rowNumber = math.ceil((len([x for x in text if x.isalpha()]))/letersAmount)

#Agora criamos a matriz e inicializamos com todos os valores
#a '-'
column = letersAmount
row = rowNumber
firstMatrix = np.chararray((row,column))
firstMatrix[:] = '-'

```

Figura 2.6: Parte 1

```
#Agora precisamos de preencher a matriz com o texto
counterRow=0
counterColumn=0
counter=0
for letter in text:
    firstMatrix[counterRow,counterColumn] = letter
    counterColumn +=1
    if counterColumn==letersAmount:
        counterColumn=0
        counterRow+=1

#Esta função devolve o index da primeira letra do texto
#que aparece no alfabeto
def GetIndexFirstLetter(word):
    for i in range(len(word)):
        if word[i] == '-':
            continue
        flag = True
        for j in range(len(word)):
            if word[i]==word[j]:
                continue
            if word[i] == '-':
                continue
            if word[j] != '-':

                #print("Comparing: " + str(word[i]) + " with: " + word[j])
                if word[i] > word[j]:
                    flag=False
        if(flag):
            return i
    return "Não foi possivel encontrar a prmeira letra"

```

Figura 2.7: Parte 2

```
#Aqui vamos criar uma string com o texto cifrado com a
#primeira transposição
tempKey = key
fullString=''
for i in range(column):
    index = GetIndexFirstLetter(tempKey)

    for j in range(row):
        fullString +=str(firstMatrix[j,index])

    tempKey = tempKey[:index] + '-' + tempKey[index+1:]

#Aqui limpamos a string
fullString = "".join(fullString.split('-'))
fullString = "".join(fullString.split('\''))
fullString = "".join(fullString.split('b'))


#Agora vamos criar a matriz para a segunda transposição
#e inicializar com todos os valores igual a '-'
secondKeyLetersAmount = len([x for x in secondKey if x.isalpha()])
secondKeyRowNumber = math.ceil((len([x for x in fullString if x.isalpha()]))/secondKeyLetersAmount)

secondColumn = secondKeyLetersAmount
secondRow = secondKeyRowNumber
secondMatrix = np.chararray((secondRow,secondColumn))
secondMatrix[:] = '-'
```

Figura 2.8: Parte 3

```
90  #Preenchemos a segunda matriz com o texto da primeira
91  #transposição
92  counter=0
93  for j in range(secondRow):
94      for i in range(secondColumn):
95          if(counter<len([x for x in fullString if x.isalpha()])):
96              secondMatrix[j,i]=fullString[counter]
97          else:
98              secondMatrix[j,i]=''
99          counter+=1
100
101 #print(secondMatrix)
102 #-------------------------------------------------------------------------------#
103 #Cifra final
104 #Por ultimo repetimos o processo para retirar por ordem alfebetica as colunas
105 #da matriz para uma string
106 tempKey = secondKey
107 fullString=''
108 for i in range(secondColumn):
109     index = GetIndexFirstLetter(tempKey)
110
111     for j in range(secondRow):
112         fullString +=str(secondMatrix[j,index])
113
114     tempKey = tempKey[:index] + '-' + tempKey[index+1:]
115
116 #Limpamos a string
117 fullString = "".join(fullString.split('-'))
118 fullString = "".join(fullString.split('\''))
119 fullString = "".join(fullString.split('b'))
```

Figura 2.9: Parte 4

```
121 #Adicionamos um espaço ' ' entre cada 5 letras da string
122 for index in range(5, len([x for x in fullString if x.isalpha()])+7, 6):
123     fullString = fullString[ : index] + str(" ") + fullString[index : ]
124
125 #Texto cifrado com transposição dupla!
126 print(fullString)
127
```

Figura 2.10: Parte 5

Isto pode ser calculado à mão da seguinte forma:

Começa-se por criar uma tabela com a primeira chave escrita na primeira linha e preencher o resto com o texto que se pretende cifrar, escrevendo no cimo da primeira linha em cada coluna a ordem alfabética da chave.

Ordem:	3	2	4	1	5
Chave:	M	I	N	H	O
	E	S	T	E	E
	X	E	M	P	L
	O	M	O	S	T
	R	A	A	T	R
	A	N	S	P	O
	S	I	C	A	O
	D	U	P	L	A

Figura 2.11: Matriz Primeira Transposição

De seguida, cria-se uma segunda matriz com a segunda chave. Esta matriz vai ser preenchida pela ordem das colunas definidas em cima, de acordo com as cores estabelecidas/mostradas.

Ordem:	1	5	3	6	2	4
Chave:	E	N	G	S	E	G
	E	P	S	T	P	A
	L	S	E	M	A	N
	I	U	E	X	O	R
	A	S	D	T	M	O
	A	S	C	P	E	L
	T	R	O	O	A	

Figura 2.12: Matriz Segunda Transposição

Volta-se a escrever no cimo de cada coluna a ordem alfabética da chave.

Por fim, volta-se a repetir o processo e escreve-se a chave cifrada com transposição dupla, neste caso, 'ELIAA TPAOM EASEE DCOAN ROLPS USSRT MXTPO'.

Para decifrar é necessário conhecer ambas as chaves e que tipo de transposição sofreram(colunas ou linhas) e fazer o processo oposto.

Capítulo 3

Referências

3.1 Parte I - P1.1

- https://en.wikipedia.org/wiki/Kerckhoffs%27s_principle

3.2 Parte II - P1.1

- http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-caesar-cipher/

3.3 Parte II - P2.1

- https://dev.to/darshangawade/monoalphabetic-cipher-30g3

3.4 Parte II - P3.1

- https://gist.github.com/EppuHeilimo/0a901056f9e48a451e0c30a55537ad1b
- https://www.greatcipherchallenge.org/polyalphabetic-ciphers

3.5 Parte II - P4.1

- https://en.wikipedia.org/wiki/One-time_pad
- https://en.wikipedia.org/wiki/Hardware_random_number_generator
- https://www.dcode.fr/vernam-cipher

3.6 Parte II - P5.1

- https://www.pbs.org/wgbh/nova/decoding/doubtrans.html