Trabalho prático 0 - Estruturas Criptográficas

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

Grupo 12

Problema 2

2.a) Começou-se por desenvolver um gerador pseudo-aleatório do tipo *XOF* através do *SHAKE-256* para gerar palavras de 64 bits. Para além disso, restringiu-se o tamanho das palavras a 2^n e armazenou-se as mesmas em *long integers*. Também se utilizou a *seed* do gerador como ***cypher_key*** e realizou-se uma autênticação do criptograma e dos dados associados atrvés do próprio *SHAKE256*. Para tal, utilizou-se a função ***generate*** que recebe uma seed e um parâmetro n e devolve uma lista de 2^n palavras aleatórias com 64 bytes.

In [12]:
```
import os
import timeit
import string
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
```

Começou-se por importar os modulos necessários para o desenvolvimento do exercício. Realizou-se uma importação de hashes criptográficas KDF2 para efetuarem uma derivação de uma ou mais chaves secretas através de uma função pseudo-aleatória e de uma *password*.

De seguida, realizou-se a implementação da função ***generate***.

In [2]:
```
def generator(seed, n):
    i = 0
    lista = []
    digest = hashes.Hash(hashes.SHAKE256((2**n) * 8)) # calcula uma string com tamanho de 2^n *8 bytes (64 bits)
    digest.update(seed) # bytes a ser hashed (seed)
    p = digest.finalize()
    while i < (2**n): # dividir a mensagem em blocos de 8 bytes
        lista.append(p[:8])
        p = p[8:]
        i += 1
    return lista


def kdf2(password, salt, n):
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length= (2 ** n) * 8,
        salt=salt,
        iterations=100000,
    )
    key = kdf.derive(password.encode('utf8'))
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length= (2**n) * 8,
        salt=salt,
        iterations=100000,
    )
    kdf.verify(password.encode('utf8'), key)
    return key
```

Começou-se a implementação da função com a utilização de uma função Hash para cálculo de uma string com tamanho de 2^n * 8 bytes (64 bits). Tamanho este que é definido na alínea i do exercício em questão. Dentro dessa função foi utilizado o SHAKE256 para gerar uma sequência de palavras que garante a autênticação do criptograma e dos dados associados ao mesmo. Finalmente, foi criado um ciclo com tamanho fixo (2^n), para armazenar os valores gerados numa lista vazia que irá ser o valor devolvido pela função.

De seguida, realizou-se a função kdf2 para gerar a *seed* do gerador. Para isso utilizou-se a função de derivação PBKDF2HMAC, que ao utilizar o algoritmo do SHAKE256, irá gerar uma sequência de palavras aleatórias de tamanho da password.

2.b) De modo a ser possível realizar a cifragem e decifragem das mensagens foram implementadas as funções *cypher* e *decypher*.

In [3]:
```
def xor_function(by1, by2):
    return bytes([_a ^ _b for _a, _b in zip(by1, by2)])

def cypher(message, n, password):
    m = message.encode('utf8')
    generated_list = generator(kdf2(password, salt, n), n)
    i = 0
```

```
    blocks = []
    while i < (2**n):
        blocks.append(m[:8])
        m = m[8:]
        i += 1
    cripto = ''.encode('utf8')
    j = 0
    while j < len(blocks):
        cripto += (xor_function(blocks[j], generated_list[j]))
        j += 1
    return cripto

def decypher(message, n, password):
    generated_list = generator(kdf2(password, salt, n), n)
    i = 0
    cripto_list = []
    while i < (2**n):
        cripto_list.append(message[:8])
        message = message[8:]
        i += 1
    original_text = ''
    j = 0
    while j < len(cripto_list):
        original_text += (xor_function(cripto_list[j], generated_list[j]).decode('utf8'))
        j += 1
    return original_text
```

A resolução passou pela criação de 2 funções: *cypher* e *decypher*.

A função *cypher*, começa por transfomar a mensagem para bytes através da função *encode*. De seguida, utiliza a função definida previamente como *generate* para criar aleatóriamente as 2^n palavras. Após isso, realiza uma divisão das mensagens em blocos de 8 bytes e realiza uma operação XOR entre os blocos de 8 bits e o resultado obtido pelo gerador pseudo-aleatório.

Já a função *decypher* possui um comportamento semelhante à função explicada anteriormente. Começa também por criar aleatóriamente as 2^n palavras e realizar a devida divisão das mensagens. Por fim, realiza a operção XOR entre a lista de criptogramas e de blocos gerados para obter o texto original.

Exemplos do funcionamento

In [ ]:

```
if __name__ == '__main__':
    salt = os.urandom(16)
    message = input("Insert message here:")
    n = int(input("Insert parameter here:"))
    password = input("Insert password here:")
    cyphered_text = cypher(message,n,password)
    print("Here's your encrypted message !")
    print(cyphered_text)
    print("Now here's your original message!")
    print(decypher(cyphered_text,n,password))
```

```
Here's your encrypted message !
b'\x11S\xceE\xfd\xe3:\x0f\xf9z\xdf.\x88:\xd8\x96\x14T\x9fQ'
Now here's your original message!
super secret message
```

Exemplo 1

Insert message here: super secret message
Insert parameter here: 2
Insert password here: password
Here's your encrypted message !
b'\x11S\xceE\xfd\xe3:\x0f\xf9z\xdf.\x88:\xd8\x96\x14T\x9fQ'
Now here's your original message!
super secret message

Exemplo 2

Insert message here: A super hyper mega secret message
Insert parameter here: 3
Insert password here: passwordofthatmessage
Here's your encrypted message !
b'z\xd5%ual\xb1\xa1\xc8\xde\xf1\xe1"\xc9\x85Pj/me\x0c\xa8\x07Y\xe6PE\xd7B\xaf?\xf1\x91'
Now here's your original message!
A super hyper mega secret message

Exemplo 3

Insert message here: Another super hyper mEga seCret meSsaGe 1234
Insert parameter here: 4
Insert password here: 12345
Here's your encrypted message !
b'\xa2,a\xb4\xb6}
6l\xec\x18\x84\xc7\x18\xf4\xa0\xce\x9a\xc5\x94\xae\x85\xc7\xcd7\x93U\xd4\x10o%\xbb\xac\xf9\x16\x19_Bw[\xcaD\xb5\x05'
Now here's your original message!
Another super hyper mEga seCret meSsaGe 1234

Problema 3

3. Para uma melhor análise da eficiência, deciciu-se utilizar o *timeit* que descreve o tempo de execução de cada método.

In [22]:
```
def main(message,n,password):
    cyphered_text = cypher(message,n,password)
    decypher(cyphered_text,n,password)

if __name__ == '__main__':
    salt = os.urandom(16)
    message = input("Insert message here:")
    n = int(input("Insert parameter here:"))
    password = input("Insert password here:")
    print(timeit.timeit(stmt = "main(message,n,password)",number=1, globals=globals()))
```

```
3.611001500000043
```

Ao utilizar a mesma mensagem ("super secret message") e a mesma password ("password") obteve-se a seguinte tabela para os diferentes n parâmetros:

n| |Tempo de execução(s)
1| |0.26620309999998426
2| |0.27461110000001554
3| |0.539764200000036
4| |1.0711420999999746
5| |3.611001500000043

Podemos concluir que, no algoritmo do exercício 2 o tempo de execução aumenta consideravelmente consoante o incremento do parâmetro n. Isto deve-se ao aumento exponencial do tamanho da string a ser calculada na função gerador. No entanto, este mostra-se bastante eficiênte para casos onde o parâmetro n apresenta valores relativamente baixos (n<=3), sendo o seu uso apropriado para estes casos.

Relativamente ao primeiro problema, apesar de não ter sido possível para a equipa de trabalho elaborar testes, teoricamente a técnica desenvolvida no segundo problema será preferível. O protocolo de Diffie-Hellman só pode ser utilizado com troca de chaves simétricas e é vulnerável a ataques do tipo *man-in-the-middle*. Para além disto, trata-se de um protocolo que exige mais do CPU e outros recursos; com isto em conta seria expectável tempos de execução piores em relação à técnica aplicada no problema 2.