TP2_Ex1

March 28, 2023

1 Trabalho 2 - Exercício 1 - Grupo 8

Para este exercício, deveremos desenvolver uma classe Python que implementasse o algoritmo KEM-**El Gamal**, utilizando **SageMath**.

O algoritmo **El Gamal** é um algoritmo de cifra assimétrica, que consiste na adaptação do protocolo Diffie-Hellman (DH) num KEM ("Key Encapsulation Mechanism").

A implementação deste algoritmo pode ser dividido em partes:

1. Inicialização da instância e gerar as chaves pública e privada.
2. Implementação de um mecanismo de ofuscação que permite encapsular a chave. Este mecanismo poderá ser utilizado na cifragem de mensagem.
3. Implementação de um método de revelação da chave. Este mecanismo pode ser usado na decifragem.

A implementação deste algoritmo deverá ser completada com a implementação de um PKE que seja IND-CCA seguro. Para tal, iremos recorrer à **transformação de Fujisaki-Okamoto**.

As partes da resolução do exercício serão apresentados de seguida.

[1]:
```
import os

from cryptography.hazmat.primitives import hashes
from base64 import b64encode, b64decode
```

1.1 Inicialização da instância e KeyGen:

Como indicamos anteriormente, o algoritmo El Gamal é uma adaptação do protocolo Diffie-Hellman. As técnicas da família Diffie-Hellman usam as propriedades de um grupo cíclico multiplicativo

$$\mathbb{Z}_p^*$$

, no qual p é um primo grande. p será tal que

$$\phi(p)$$

tem um divisor primo q grande.

Para garantir a segurança deste algoritmo é necessário que os valores de p e q cumpram as restrições mencionadas. Assim, para os gerar, iremos começar por gerar um primo q de, pelo menos, 160 bits.

De seguida, iremos gerar sucessivamente inteiros seguindo a fórmula *pi = q * 2^i + 1* até que *pi* seja um primo cujo o tamanho em bits seja maior ou igual ao parâmetro de segurança passado na inicialização.

Após termos obtido o valor de p e q, iremos obter o valor de g. Este g será um inteiro que pertença ao grupo multiplicativo

$$\mathbb{Z}_p^*$$

Com estes parâmetros comuns obtidos, a chave privada será:

$$a \neq 0 \in \mathbb{Z}_q$$

gerada aleatoriamente e a chave pública será:

$$\beta \equiv g^a \bmod p$$

Esta formalização foi implementada na função *key_gen* da classe ElGamal.

1.2 Implementação do mecanismo de encapsulamento da chave:

Após termos obtidos as chaves e os valores de p, q e g, iremos agora implementar o mecanismo de encapsulamento da chave gerada. Este mecanismo de encapsulamento ou ofuscação é implementado pelo KEM ou *"key encapsulation mechanism"*.

O KEM é uma técnica assimétrica que gera, comunica e ofusca a chave privada que é utilizada na cifragem de uma mensagem. Este processo de cifragem que permite ofuscar os dados chama-se DEM e corresponde a uma cifra simétrica. O resultado do mecanismo KEM será um elemento privado k, que será usado pelo DEM, e um elemento público e, que corresponde ao encapsultamento do elemento privado.

O KEM pode ser implementado seguindo as fórmulas apresentadas no **Capítulo 3a** da Unidade Curricular Estruturas Criptográficas. A fórmula apresentada é:

$$KEM(\beta) \equiv \vartheta\, r \leftarrow \mathbb{Z}_q \setminus 0 \centerdot \vartheta\, \mathsf{key} \leftarrow \beta^r \bmod p \centerdot \vartheta\, \mathsf{enc} \leftarrow g^r \bmod p \centerdot (\mathsf{key}\,,\, \mathsf{enc})$$

Como podemos ver através da análise da fórmula, o KEM utiliza os valores públicos do utilizador que poderá decifrar a mensagem.

Para complementar o KEM, optamos por implementar o DEM ou *"data encapsulation mechanism"*, apesar de este não ter sido pedido. Este mecanismo recorre aos dados obtidos do KEM para encapsular a mensagem do utilizador. A combinação do KEM e do DEM permite formar o mecanismo de cifragem de um PKE ou *"Public Key Encryption"*.

$$E(x) \equiv \vartheta\,(e, k) \leftarrow \mathsf{KEM} \centerdot (e\,,\, DEM(k, x))$$

O mecanismo de ofuscação/encapsulamento foi implementado na função *KEM* e como indicamos implementamos também o método *DEM* na classe ElGamal, apresentada posteriormente. Combinamos o KEM e o DEM numa função *encrypt*, que permite implementar o processo apresentado na última fórmula.

1.3 Implementação do mecanismo de revelação da chave:

Outra parte essencial do algoritmo El Gamal é a revelação da chave. Como podemos ver o mecanismo KEM devolve dois elementos um público e e um privado k. O elemento k foi utilizado no processo de ofuscação dos dados. Este processo de ofuscação utilizou uma cifra simétrica. Assim, será necessário que para revelar os dados ofuscados se utilize a mesma chave k usada na cifra. Contudo, para garantir a segurança da cifra o elemento k não pode ser revelado.

O mecanismo *KRev* é um mecanismo que permite obter o elemento privado k gerado pelo KEM a partir do seu encapsulamento e. A este mecanismo chama-se revelação da chave.

Este mecanismo KRev poderá ser implementado seguindo a fórmula apresentada no **Capítulo 3a** do UC Estruturas Criptográficas:

$$KRev(a, \mathsf{enc}) \;\equiv\; \mathsf{enc}^a \bmod p$$

Para revelar k utilizando o KRev irá ser utilizada a chave privada do utilizador gerada no processo da *key_gen*.

Adicionalmente, existe também um mecanismo DRev que corresponde ao oposto do DEM, i.e, se o DEM corresponde à cifra dos dados, o DRev corresponde à decifra. Como vimos, para que seja possível descodificar a mensagem, o DRev deve receber a mesma chave usado no DEM.

À semelhança do que foi introduzido na secção anterior, podemos definir o mecanismo de decifra de um PKE se combinarmos o KRev com o DRev.

$$D(e, c) \equiv \vartheta\, k \leftarrow \mathsf{KRev}(e) \,\centerdot\, DRev(k, c)$$

Na classe ElGamal que se segue, implementamos o mecanismo de revelação da chave na função *KRev*. Complementarmente, implementamos também o DRev e a função de decifra, apesar de estas últimas não terem sido pedidas.

Assim, de seguida, podemos ver a classe ElGamal implementada.

[2]:
```
class ElGamal:

    def __init__(self):
        self.private_key = 0 # privado
        self.public_key = 0  # publico
        self.p = 0           # publico
        self.g = 0           # publico
        self.q = 0           # publico

    # Esta função deve inicializar a instância, recebendo parâmetro de segurança
    # O parâmetro de segurança é o tamanho em bits da ordem do grupo ciclico
    # Deve gerar as chaves pública e privada

    # KeyGen_alpha -> (pk, sk)
    def key_gen(self, parametro_seguranca):

        q_bits = 160
```

```
        self.q = random_prime(2^q_bits-1, False, 2^(q_bits-1))

        i = 0
        size = 0
        while (not is_prime(self.p)) or (size < parametro_seguranca):
            self.p = self.q * pow(2, i) + 1
            size = len(self.p.binary())
            i = i + 1

        # print(f"P: {str(self.p)}. Type: {type(self.p)} Is prime: {is_prime(self.p)} ")
        # print(f"Q: {str(self.q)}. Type: {type(self.q)}  Is prime: {is_prime(self.q)} ")

        # pI # Ring of integers modulo p
        pI = Integers(self.p) # Integers == IntegerModRing:
        Zp = pI.unit_group() # Grupo multiplicativo : unit_group

        # print()
        # print(f" Is cyclic? {Zp.is_cyclic()}")
        # print(f" Order: {Zp.order()} ")
        # print(f" Is finite? {Zp.is_finite() }")
        # print(f" Multiplicative Generator: {pI.multiplicative_generator() }")
        # print()
        #
        # Dado que o valor p é primo, todos os valores de Zp, exceto 0, pertence ao Z*p. Portanto, geramos g de Zp de ordem q
        # que seja diferente de 0.
        while self.g == 0:
            self.g = Integer(pI.random_element(self.q))

        self.private_key = ZZ.random_element(self.q)
        # print(f" Private_key: {self.private_key}. Type {type(self.private_key)}")

        self.public_key = power_mod(self.g,(self.private_key), self.p)
        # print(f" Public_key: {self.public_key}. Type {type(self.public_key)}")
        return self.p, self.q, self.g, self.public_key

    def KEM(self, p, q, g, public_key):
        r = ZZ.random_element(q)
        key = power_mod(public_key, r, p)
        enc = power_mod(g, r, p)
        return key, enc


    def xor(self, message, key):
```

```python
        output = bytes([x ^^ y for(x,y) in zip(message, key)])
        return output

    # Função de cifra simétrica usada para codificação
    def DEM(self, key, plaintext):
        key_binary = str(key).encode('utf-8')
        message_binary = plaintext.encode('utf-8')
        ciphertext = self.xor(message_binary, key_binary)
        c = ciphertext.decode('utf-8', errors = 'replace')
        return c

    # E(x) = (e,k) <- KEM . c <- DEM(k, x) . (e, c)
    def encrypt(self, message, p, q, g, public_key):
        (key, enc) = self.KEM(p, q, g, public_key)
        ciphertext = self.DEM(key, message)
        return enc, ciphertext

    def KRev(self, enc):
        key = power_mod(enc, self.private_key, self.p)
        return key

    # Função de cifra simétrica usada para descodificar
    def DRev(self, key, ciphertext):
        c = ciphertext.encode('utf-8')
        key_binary = str(key).encode('utf-8')
        plaintext = self.xor(c, key_binary)
        plaintext = plaintext.decode('utf-8', errors = 'replace')
        return plaintext

    # D(e,c) = k <- KRev . p <- DRev(k, c) . p
    def decrypt(self, enc, ciphertext):
        key = self.KRev(enc)
        plaintext = self.DRev(key, ciphertext)
        return plaintext
```

1.4 Transformação Fujisaki-Okamoto:

Para este exercício foi nos também pedido para construir um PKE que seja IND-CCA seguro. Este PKE (public key encryption) deverá ser construído a partir do KEM definido anteriormente e da transformação de Fujisaki-Okamoto.

A transformação de Fujisaki-Okamoto permite converter um esquema PKE com segurança IND-CPA num esquema PKE com segurança IND-CCA.

Para que o KEM desenvolvido anteriormente seja IND-CPA seguro é suficiente que o algoritmo KEM seja suficientemente aleatório, i.e, é necessário que o número de pares (e,k) gerado pelo KEM seja de ordem igual ou superior a

$$2^{lambda}$$

onde lambda corresponde ao parâmetro de segurança.

Para aplicar a transformação FO, é necessário decompor o KEM num "hash" aleatório h e um "hash" seguro f de tal modo que

$$\mathsf{KEM} \;\equiv\; \vartheta\, r \leftarrow h \,\centerdot\, f(r)$$

$$\forall\, r \;\centerdot\; (e,k) = f(r) \quad \text{sse} \quad \mathsf{KRev}(e) \simeq k$$

Ou seja, para fazer a transformação Fujisaki-Okamoto, deveremos decompor a função KEM numa função que gera valores aleatórios, e uma função f que utiliza este valor gerado. Esta função f deverá implementar o resto do processo do KEM.

A função KEM do ElGamal possui um componente que gera um valor aleatório r, assim, visto que era necessário decompor a função, optamos por implementar uma função f que implementasse o resto do algoritmo de KEM exceto a obtenção de um valor aleatório, ou seja, implementasse:

$$f(y||r,r) \;\equiv\; \vartheta\,\mathsf{key} \leftarrow \beta^{r} \bmod p \,\centerdot\, \vartheta\,\mathsf{enc} \leftarrow g^{r} \bmod p \,\centerdot\, (\mathsf{key}\,,\,\mathsf{enc})$$

A partir desta transformação iremos obter o esquema assimétrico E', D' através de:

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

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

Estas fórmulas encontram-se no **Capítulo 2a** dos documentos da UC Estruturas Criptográficas. Na nossa implementação optamos por seguir a lógica presente nestas fórmulas.

No programa desenvolvido, o criptograma é formado pela ofuscação da mensagem y, pelo encapsulamento da chave e e por uma ofuscação da chave c. O g deverá corresponder a uma hash do valor aleatório r. Este g deverá devolver um valor de tamanho igual ao da mensagem x.

Através da transformação de Fujisaki-Okamoto, podemos obter uma PKE que seja IND-CCA e IND-CPA segura. A partir desta, obtemos também uma PKE que permita recuperar a mensagem x e, adicionalmente, verificar a autenticidade do criptograma, uma vez que, neste algoritmo o y devolvido pela cifra corresponde ao mensagem cifrada, e c corresponde a uma tag. A verificação da tag c, permite verificar se a mensagem recebida era a esperada.

Na implementação que se segue foram implementadas as funções f, *encrypt_* e *decrypt_*. A função *encrypt_* corresponde à implementação em Python da fórmula E' e a *decrypt_* corresponde à implementação de *D'*. Por último, a função f já foi introduzida anteriormente e corresponde à implementação do KEM com a exceção da geração do r.

Para esta implementação utilizamos as funções definidas anteriormente na classe ElGamal.

[3]:

```
# Instância utilizada para teste das funções utilizadas
elgamal_aux = ElGamal()
elgamal_aux.key_gen(256)

def f(message, r):
    key = power_mod(elgamal_aux.public_key, r, elgamal_aux.p)
    enc = power_mod(elgamal_aux.g,r, elgamal_aux.p)
    return enc, key

def encrypt_(message):
    # r = h
    r = ZZ.random_element(0, 2^10)
    r_s = str(r).encode('utf-8')
    # print(f" r_s: {r_s}. Type {type(r_s)}. Len: {len(r_s)}")

    # g = g(r)
    size = len(message)
    h = hashes.Hash(hashes.SHAKE256(size))
    h.update(r_s)
    g = h.finalize()
    # print(f" g: {g}. Type {type(g)}. Len: {len(g)}")

    # y = x xor g(r)
    message_binary = message.encode('utf-8')
    y = elgamal_aux.xor(message_binary, g)
    y = b64encode(y).decode('utf-8')
    # print(f" y: {y}. Type {type(y)}. Len: {len(str(y))}")

    # (e,k) = f(y||r)
    concate_message = y + str(r)
    (e,k) = f(concate_message, r)
    # print(f" K: {k}. Type {type(k)}. Len: {len(str(k))}")

    # c = k xor r
    key_binary = str(k).encode('utf-8')
    r_binary = int(r).to_bytes(len(key_binary), byteorder= "big")
    c = elgamal_aux.xor(key_binary, r_binary)
    c = b64encode(c).decode('utf-8')

    return y, e, c


def decrypt_(y, e, c):
    # k = KRev(e)
    k_i = elgamal_aux.KRev(e)
    k = str(k_i).encode('utf-8')
```

```
    #r = xor(c, k)
    C = b64decode(c)
    r_binary = elgamal_aux.xor(C, k)
    r = int.from_bytes(r_binary, byteorder= "big")

    #(e1, k1) = f(y||r)
    concate_message = y + str(r)
    (e1, k1) = f(concate_message, r)

    # if (e,k) != f(y||r)
    if e1 != e or k1 != k_i:
        print("Invalid")
        return
    else:
        # plain = y xor g(r)
        y = b64decode(y.encode('utf-8'))

        # g(r)
        size = len(y)
        r_s = str(r).encode('utf-8')
        h = hashes.Hash(hashes.SHAKE256(size))
        h.update(r_s)
        g = h.finalize()

        plaintext = elgamal_aux.xor(y, g)
        plaintext = plaintext.decode('utf-8')

    return plaintext
```

1.5 Testes:

Após termos implementado estes mecanismos, definimos alguns testes que permitem verificar o funcionamento do que foi implementado.

Primeiro, iremos testar o PKE que foi implementado para testar o funcionamento do El Gamal.

[4]:
```
elgamal1 = ElGamal()
p, q, g, public_key = elgamal1.key_gen(1024)

input = "Esta string sera utilizada como input para o algoritmo El Gamal."

print(f" Teste da Cifra El Gamal: input = {input}")

enc, ciphertext = elgamal1.encrypt(input, p, q, g, public_key)
print(f" Resultado da Cifra:\n  ENC {enc}\n  CIPHERTEXT {ciphertext}")

plaintext = elgamal1.decrypt(enc, ciphertext)
print(f" Resultado da Decifra: {plaintext}")
```

```python
print()
print()
```

```
 Teste da Cifra El Gamal: input = Esta string sera utilizada como input para o
algoritmo El Gamal.
 Resultado da Cifra:
  ENC 246706255576058115348995701972993644570176586522510502196319487950105635398
0395655329955744782919933072028761072145257159940836826775385725859110325151357
360168967905835653535015139773843707331714702081759551885240431231297898904829043
69116555530759191377379906959679254072216761154782709261858265794040298507936135
  CIPHERTEXT tKLRKAAY^SAWDRGGYZPOW]PQWTYXZBFECUKUZSXWZG^DZ[p_~R]UT
 Resultado da Decifra: Esta string sera utilizada como input para o algoritmo El
Gamal.
```

De seguida, podemos testar o funcionamento do PKE implementado seguindo a transformação Fujisaki-Okamoto.

[5]:
```python
input = "Input para verificar funcionamento do FOT."

print(f" Teste da Cifra KEM usando o Transformação de Fujisaki-Okamoto: input =␣
 ↪{input}")
y, e, c = encrypt_(input)
print(f"Resultado de cifrar usando o FOT :\n y:{y}.\n e:{e}.\n c:{c}.")

plaintext = decrypt_(y, e, c)
print(f"Resultado da Decifra: {plaintext}")
```

```
 Teste da Cifra KEM usando o Transformação de Fujisaki-Okamoto: input = Input
para verificar funcionamento do FOT.
Resultado de cifrar usando o FOT :
 y:8xftMW+bt/tiRb2iRESyU17zU8y7ZfPx6vwotBpZz7TLEuC6Lz/WNi3G.
 e:20249393774313330280033238139962669335631495511693482471648250771743353895278
744298342784905407.
 c:MzUxOTExODE1MzIyNjgwOTk1MjEzNTg3MTE5MjE3Mjk5NzEzMzYwMDkyMzcwNDIzOTY5OTI4MjY1N
DYyMzgwOTUyNDc4ODc1OTEyODExNTI5NjE3MTUwNDg0MzEzN6A=.
Resultado da Decifra: Input para verificar funcionamento do FOT.
```