ex1

March 28, 2023

0.1 TP2 | Exercício 1 | Grupo 15

0.1.1 Pedro Araújo - pg50684

0.1.2 Nuno Dias Mata - Pg44420

0.1.3 Questão proposta , `parte 1`:

1. Construir uma classe Python que implemente um KEM - ElGamal

a. Inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits da ordem do grupo cíclico) e gere as chaves pública e privada.

b. Conter funções para encapsulamento e revelação da chave gerada.

- A fim de comprir estes passos da parte 1 do primeiro exercício foi realizado os seguintes passos na criação da classe ElGamalKEM:

1. Criar função de inicialização , onde irá inicializar os parametros: salt (para chave simétrica) , key_length (tamanho da chave) , prime (número primo grande aleatório) , g (gerador do grupo cíclico finito) , privk (chave privada que será numero aleatório da grandeza do número primo) , publick (chave pública que é dada por realizar a operação de `y = g^x (mod p)` utilizando o grupo ciclico , chave privada e o nº primo)
2. Após a incialização de todas variáveis necessárias, é utilizado uma função auxiliar `generate_rand` que irá gerar um número aleatório que servirá por enquanto para efeitos de teste como mensagem a ser encapsulada e seed para a geração da chave simétrica. Para parte 2 irá ser utilizado uma mensagem real.
3. Agora é possível realizar o Encapsulamento , com a função `encapsulate` que recebe como parâmetro a seed para a chave simétrica e para mensagem de encapsulamento. Para a cifragem é criado 2 operações : c1 (parte aleatória que é gerada a cada vez que é encapsulada utilizando o grupo cíclico e um numero aleatório) e c2 (que irá utilizar a chave pública para encapsular a mensagem) , e após essa cifragem também é gerado uma chave simétrica utilizando o pbkdf2.
4. Para a parte do desencapsulamento , na função `decapsulate` , é utilizado as 2 cifragens produzidas (c1 e c2) onde será extraída a mensagem/seed utilizando a chave privada. Após extraída, para verificar se a chave produzida é simétrica é criado mais uma vez uma chave utilizando o pbkdf2.

[4]:
```
import os
import hashlib
```

```python
from sage.all import *

class ElGamalKEM:
    def __init__(self, key_bits):
        # salt utilizado para criar a chave simétrica
        self.salt = os.urandom(16)
        # tamanho da chave
        self.key_length = 2**key_bits
        # geração do primo da grandeza da chave
        self.prime = random_prime(self.key_length, lbound=2**(key_bits - 1))
        # grupo cíclico finito
        G = IntegerModRing(self.prime)
        self.g = G.random_element()
        #Chave privada
        self.privk = randrange(1, self.prime - 2)
        # Chave pública
        self.publick = self.g**self.privk % self.prime


    def encapsulate(self, seedKey):
        k = randrange(1, self.prime - 1)
        #chave cifradas
        c1 = self.g**k % self.prime
        c2 = (seedKey * self.publick**k) % self.prime

        keyBytes = str(seedKey).encode()
        # chave simétrica
        chave = hashlib.pbkdf2_hmac('sha256', keyBytes, self.salt, 100000)
        return chave , c1 , c2

    def decapsulate(self, c1, c2):
        s = c1**self.privk % self.prime
        if s == 0 :
            return 0
        key = c2 // s % self.prime

        keyBytes = str(key).encode()
        chave = hashlib.pbkdf2_hmac('sha256', keyBytes, self.salt, 100000)

        return chave

    def generate_rand(self):
        self.key = randrange(1, self.prime - 1)

    def set_key(self, key):
        self.key = key
```

```
    def get_public_key(self):
        return self.publick

    def get_private_key(self):
        return self.privk

    def get_key(self):
        return self.key
```

- Parte de testes da classe

[2]:
```
# Criando uma instância da classe utilizando
key_Blength = 1024
kem = ElGamalKEM(key_Blength)

# Gerando a chave simétrica
kem.generate_rand()
# Encapsulando a chave simétrica com a chave pública do destinatário
chave,c1, c2 = kem.encapsulate(kem.get_key())

# Decapsulando a chave simétrica usando a chave privada do destinatário
key = kem.decapsulate(c1, c2)
if key == 0:
    print("A decapsulação falhou , o destinatário não tem a chave privada␣
 ↪correspondente correta para decapsular")
# Verificando se a chave decapsulada é igual à chave simétrica original
print(chave)
print(key)
if key == chave:
    print("As chaves são iguais!")
else:
    print("Falha na revelação")
```

```
b'\x1bz\xfe)\xf6\xff\x9d\x14\xad\xfe\xcb\xca+\x86$\x84R&\xa2\x80\xd2\xb4\x84\x1f
H\x9f\xa1\xec\xbeC\x8c*'
b'\x1bz\xfe)\xf6\xff\x9d\x14\xad\xfe\xcb\xca+\x86$\x84R&\xa2\x80\xd2\xb4\x84\x1f
H\x9f\xa1\xec\xbeC\x8c*'
As chaves são iguais!
```

0.1.4 Parte 2 :

c) Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.

- A fim usar a classe de KEM na transformação de FuFujisaki-Okamoto para formar um PKE IND-CCA seguro , foi realizado os seguintes passos:

1. `inicalização` : é inciado a classe com o parâmetro necessário para geração da classe ElGamalKEM(tamanho da chave) , de forma que gere um variável kem que possua todos os

atributo desta classe.

2. `cifra` : na função de cifra da mensagem 'encrypt' é realizado os seguintes passos:
   - `r aleatório` : é gerado um inteiro r aleatório utilizando a função da classe kem generate_rand() , onde irá gerar 1 < r < (grandeza da chave)
   - `função de hash no r` : é colocado na variável g , a operação de realizar um função hash SHA-256 no inteiro r
   - `XOR na mensagem` : é efetuado o xor na mensagem a cifrar e o g gerado anteriormente, e guarda na variável maskxor, onde depois é colocado na variável new_r o valor de maskxor adicionado ao valor de r em bytes.
   - `encapsulamento de KEM` : é usado o encapsulamento de kem , onde coloca como parametro o new_r que contém a mensagem cifrada para ser encapsulada, e a função devolve as cifras e também chave simétrica ´chave_pub´
   - `ofuscação da mensagem` : para ofuscar a mensagem é realizado um outro XOR entre a chave simétrica e o r inicial .
3. `decifra` : na função de decifrar o ciphertext "decrypt":
   - `revelação da mensagem` : é chamado a função de KEM de desencapsular a mensagem , onde retorna a chave simétrica
   - `obter o r` : para obter o r é realizado o xor entre a ofuscação gerada e a chave simétrica
   - `gerar outra chave` : a fim de saber se a chave realmente é simétrica é gerado uma nova chave realizando a função encapsulate com a seed gerada
   - `obter a mensagem` : para se obter a mensagem e assim ser decifrado , é gerado a função hash sobre o mesmo r inicial , e então o xor entre a função hash e o mcx que foi resultado da mascara xor na cifra.

[3]:
```
import os,hashlib

class PKEFO:
    def __init__(self,key_length):
        self.kem = ElGamalKEM(key_length)
        self.n = key_length


    def encrypt(self, message):
        # função hash g utilizando uma seed aleatória r , sendo 1 < r <␣
 ↪(tamanho da chave)
        r = self.kem.generate_rand()
        g = hashlib.sha256(str(r).encode()).digest()
        #Aplicar XOR à mensagem
        maskxor = bytes([a ^ b for a, b in zip(message, g)])
        new_r = maskxor + str(r).encode()
        new_r_int = int.from_bytes(new_r, "big")

        #Cifrar utilizando o KEM da alínea anterior
        chave_pub, c1 , c2 = self.kem.encapsulate(new_r_int)
```

```
        #Com a chave simétrica, aplicando XOR a r
        lastXor = bytes([a ^ b for a, b in zip(chave_pub, str(r).encode())])
        return maskxor, c1, c2, lastXor

    def decrypt(self, mcx, c1, c2, lastXor):
        #Obtemos a chave com o KEM definido antes
        chave_pub = self.kem.decapsulate(c1,c2)

        #Aplicamos o XOR com a chave simetrica de ambos para decifrar
        r = bytes([a ^ b for a, b in zip(lastXor, chave_pub)])
        #y = Integer('0x' + hashlib.sha256(mcx).hexdigest())
        new_r = mcx + r
        new_r_int = int.from_bytes(new_r, "big")

        #Encapsular utilizando o exercício anterior
        nova_chave_pub , newc1, newc2 = self.kem.encapsulate(new_r_int)

        if chave_pub != nova_chave_pub:
            print("A chave não é simétrica")
            raise IOError
        else:
            g = hashlib.sha256(r).digest()
            message = bytes([a ^ b for a, b in zip(mcx, g)])

        return message.decode("utf-8")
```

- Testes na classe

[5]:
```
pke = PKEFO(1024)

#message = os.urandom(32)
message = "Este e o tp2 ex1 do grupo 15"

print("Mensagem a ser cifrada: " + message)

# Cifragem da mensagem
maskxor, c1, c2, lastXor = pke.encrypt(message.encode('utf-8'))

# Decifragem do criptograma
try:
    message1 = pke.decrypt(maskxor, c1, c2, lastXor)
    print("Texto decifrado: " + message1)
    if message == message1:
        print("A mensagem foi decifrada com sucesso!")
    else:
        print("A mensagem é diferente da original, ocorreu erros...")
except IOError as e:
```

```
    print("Erro ao decifrar a mensagem!!!!")
```

```
Mensagem a ser cifrada: Este e o tp2 ex1 do grupo 15
Texto decifrado: Este e o tp2 ex1 do grupo 15
A mensagem foi decifrada com sucesso!
```

```
[ ]:
```