TP2_EX1

March 28, 2023

1 TP2

1.1 Grupo 17:

PG50315 - David Alexandre Ferreira Duarte

PG51247 - João Rafael Cerqueira Monteiro

1.2 Exercício 1.

1. Construir uma classe Python que implemente um KEM - ElGamal. A classe deve
    1. Inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits da ordem do grupo cíclico) e gere as chaves pública e privada.
    2. Conter funções para encapsulamento e revelação da chave gerada.
    3. Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.

[1]:
```
# Imports

from sage.all import *
from hashlib import sha256
import random
import math
import secrets

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import padding
```

A classe **KEMElGamal** foi implementada para gerar chaves pública e privada. A classe recebe o parâmetro de segurança que é o tamanho em *bits* da ordem do grupo cíclico. A chave privada é gerada como um número aleatório entre 1 e $p-2$, onde p é um número primo gerado aleatoriamente. A chave pública é gerada como o par *(p, g, h)*, onde g é um elemento primitivo do grupo cíclico e $h = g^x mod p$, onde x é a chave privada. A classe também possui funções para encapsulamento e revelação da chave gerada.

A função `init`: é uma função de inicialização recebe como entrada o parâmetro de segurança *sec_param* e inicializa as variáveis necessárias para o esquema **ElGamal**, incluindo a escolha de um primo aleatório p, a escolha de um gerador primitivo g e um valor x aleatório, que representa a chave privada. Ela também calcula a chave pública h, que é $g^x mod p$

A função `encapsulate` recebe como entrada a chave pública *pub_key* e gera uma chave de sessão aleatória k e um valor s, que é a potência de g elevada a um valor y aleatório, que é a chave de sessão criptografada. A função retorna k e s.

A função `decapsulate` recebe como entrada a chave privada *priv_key* e o texto cifrado *ciphertext*. Ela usa a chave privada para calcular a chave de sessão k, que é $s^x mod p$. A função retorna k.

[2]:
```
class KEMElGamal:

    def __init__(self, sec_param):
        """
        Inicializa as chaves pública e privada para o algoritmo de troca de chaves de ElGamal

        :param sec_param: Tamanho da chave
        """

        # Gerar um número primo aleatório de tamanho 2^sec_param
        self.p = random_prime(2 ** sec_param)
        # Criar um campo finito sobre p
        self.F = GF(self.p)
        # Encontrar um elemento primitivo do campo finito
        self.g = self.F.primitive_element()

        # Gerar uma chave privada aleatória x
        self.x = randint(1, self.p-2)
        # Calcular a chave pública h = g^x mod p
        self.h = pow(self.g, self.x, self.p)

        # Definir as chaves pública e privada
        self.priv_key = self.x
        self.pub_key = (self.p, self.g, self.h)

    def encapsulate(self, pub_key):
        """
        Encapsula a chave de sessão usando a chave pública do algoritmo de troca de chaves de ElGamal

        :param pub_key: Chave pública
        :return: k - Chave de sessão encapsulada
                 s - Valor intermediário usado na decapsulação da chave de sessão
        """

        p, g, h = pub_key

        # Gerar um valor aleatório y
        y = random.randint(1, p-1)
```

```
        # Calcular k = h^y mod p
        k = pow(h,y,p)
        # Calcular s = g^y mod p
        s = pow(g,y,p)

        return k, s

    def decapsulate(self, priv_key, ciphertext):
        """
        Decapsula a chave de sessão usando a chave privada do algoritmo de␣
↪troca de chaves de ElGamal

        :param priv_key: Chave privada
        :param ciphertext: Texto cifrado contendo s e iv
        :return: k - Chave de sessão decapsulada
        """

        s, iv = ciphertext

        # Calcular k = s^x mod p
        k = pow(s, priv_key, self.p)

        return k
```

A classe **PKE_FujisakiOkamoto** foi implementada para usar a transformação de **Fujisaki-Okamoto** para construir um **PKE IND-CCA** seguro. A classe recebe o parâmetro de segurança e a instância **KEMElGamal**. A função de criptografia usa a função de encapsulamento da instância **KEMElGamal** para gerar uma chave de criptografia e, em seguida, usa essa chave para cifrar a mensagem usando **AES** em modo **CBC**. A função de decodificação usa a função de decapsulamento da instância **KEMElGamal** para recuperar a chave de criptografia e, em seguida, usa essa chave para decifrar a mensagem usando **AES** em modo **CBC**.

A função `init`: é uma função de inicialização recebe como entrada o parâmetro de segurança *sec_param* e uma instância da classe **KEMElGamal** *kem*. Ela inicializa as variáveis necessárias para o esquema *FO*, incluindo a instância da classe **KEMElGamal** *kem* e o tamanho da chave usada pelo algoritmo **AES**.

A função `encrypt` recebe como entrada a chave pública *pub_key* e a mensagem *msg*. Ela gera uma chave de sessão aleatória utilizando o esquema **KEM** de **ElGamal** e usa a chave de sessão para criptografar a mensagem usando o algoritmo **AES** no modo **CBC** com preenchimento `PKCS#7`. A função retorna o texto cifrado, que é um tuplo *(s, ct, iv)*, onde *s* é o valor criptografado da chave de sessão, *ct* é o texto cifrado e *iv* é o vetor de inicialização usado pelo algoritmo **AES**.

A função `decrypt` recebe como entrada a chave privada *priv_key* e o texto cifrado *ct*. Ela utiliza a chave privada para decifrar a chave de sessão *s* utilizando o esquema **KEM** de **ElGamal** e usa a chave de sessão para decifrar o texto cifrado utilizando o algoritmo **AES** no modo **CBC** com preenchimento `PKCS#7`. A função retorna a mensagem decifrada.

[3]:
```python
class PKE_FujisakiOkamoto:

    def __init__(self, sec_param, kem):
        """
        Inicializa o esquema de criptografia Fujisaki-Okamoto

        :param sec_param: Tamanho da chave
        :param kem: Objeto da classe KEMElGamal contendo as chaves públicas e privadas
        """

        self.kem = kem
        self.key_size = sec_param

    def encrypt(self, pub_key, msg):
        """
        Criptografa a mensagem usando a chave pública

        :param pub_key: Chave pública
        :param msg: Mensagem a ser criptografada
        :return: c - Texto cifrado contendo s, ct e iv
        """

        # Encapsular a chave de sessão
        k, s = self.kem.encapsulate(pub_key)
        # converte a chave de sessão encapsulada em uma chave AES de tamanho key_size
        aes_key = int(k).to_bytes(self.key_size, byteorder='big')
        # gera um vetor de inicialização aleatório de tamanho key_size
        iv = secrets.token_bytes(self.key_size)
        # cria um objeto de cifração AES com modo CBC
        cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv))

        # adiciona padding à mensagem original
        padder = padding.PKCS7(self.key_size * 8).padder()
        padded_msg = padder.update(msg) + padder.finalize()

        # criptografa a mensagem com a chave AES
        encryptor = cipher.encryptor()
        ct = encryptor.update(padded_msg) + encryptor.finalize()

        # retorna o texto cifrado junto com a chave de sessão encapsulada e o vetor de inicialização
        c = (s, ct, iv)

        return c
```

```
    def decrypt(self, priv_key, ct):
        """
        Descriptografa o texto cifrado usando a chave privada

        :param priv_key: Chave privada
        :param ct: Texto cifrado contendo s, ct e iv
        :return: msg - Mensagem descriptografada
        """

        # Extrai os valores s, ct e iv do texto cifrado
        s, ct, iv = ct
        # Decapsula a chave de sessão usando a chave privada
        k = self.kem.decapsulate(priv_key, (s, iv))
         # converte a chave de sessão encapsulada em uma chave AES de tamanho␣
 ↪key_size
        aes_key = int(k).to_bytes(self.key_size, byteorder='big')
        # cria um objeto de decifração AES com modo CBC
        cipher = Cipher(algorithms.AES(aes_key), modes.CBC(iv))

        # decifra o texto cifrado com a chave AES
        decryptor = cipher.decryptor()
        unpadded_msg = decryptor.update(ct) + decryptor.finalize()

        # remove o padding da mensagem descriptografada
        unpadder = padding.PKCS7(self.key_size * 8).unpadder()
        msg = unpadder.update(unpadded_msg) + unpadder.finalize()

        return msg
```

[4]:
```
# Cria um objeto KEMElGamal com um tamanho de chave de 128 bits
kem = KEMElGamal(128)

# Cria um objeto PKE_FujisakiOkamoto com um tamanho de chave de 16 bytes e o␣
 ↪objeto KEM criado anteriormente
pke = PKE_FujisakiOkamoto(16, kem)

# Obtém a chave pública e privada do objeto KEM
pub_key = pke.kem.pub_key
priv_key = pke.kem.priv_key

# Define uma mensagem de exemplo e a codifica em bytes
msg = "Estruturas Criptograficas - Grupo 17!"
msg_encoded = msg.encode()

# Criptografa a mensagem usando a chave pública do objeto KEM e retorna o texto␣
 ↪cifrado
ciphertext = pke.encrypt(pub_key, msg_encoded)
```

```
# Descriptografa o texto cifrado usando a chave privada do objeto KEM e retorna␣
 ↪a mensagem original
decrypted_msg = pke.decrypt(priv_key, ciphertext)

# Decodifica a mensagem original de volta para texto
msg_decoded = decrypted_msg.decode()


# Imprime a mensagem original, a mensagem decifrada e se são iguais ou não
print("Mensagem Original: ", msg)
# print("Ciphertext: ", ciphertext)
print("Mensagem Decifrada: ", msg_decoded)
print("A mensagem original e a mensagem decifrada são iguais (True), diferentes␣
 ↪(False)")
print(msg == msg_decoded)
```

```
Mensagem Original:  Estruturas Criptograficas - Grupo 17!
Mensagem Decifrada:  Estruturas Criptograficas - Grupo 17!
A mensagem original e a mensagem decifrada são iguais (True), diferentes (False)
True
```