Trabalho Prático 1 - Grupo 15

João Gonçalves - pg46535

Sara Queirós - pg47661

Exercício 2

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

In [ ]:
```
import os
import random
import hashlib
from sage.arith.power import generic_power
```

In [ ]:
```
class KEMRSA():
    def __init__(self, param, random):
        #parametro de segurança
        self.p = 0
        self.q = 0
        self.n = 0
        self.e = 0
        self.d = 0
        self.to_key = None
        self.size = param
        self.chave_cifrada = None
        self.chave_decifrada = None
        self.chave_gerada_e = None
        self.chave_gerada_d = None
        self.salt = random
        self.private_key = None
        self.public_key  = None

    def generate_keys(self):

        #temos de encontrar p e q primos com size/2 bits
        while not (self.p in Primes()):
            self.p = next_prime(ZZ.random_element(pow(2,self.size/2-1)+1, po

        while not (self.q in Primes()):
            self.q = next_prime(ZZ.random_element(pow(2,self.size/2-1)+1, po

        #calcular o n
        self.n = self.p * self.q

        #calcular o phi
        phi = (self.p -1) * (self.q-1)

        #e is a pseudo-random integer
```

```
        self.e = ZZ.random_element(phi)

        while (gcd(e,phi)!=1):
            self.e = ZZ.random_element(phi)

        #we use the extended Euclidean algorithm to calculate d
        bezout = xgcd(self.e,phi)
        self.d = Integer(mod(bezout[1], phi))

        if not (mod(self.d*self.e, phi) == 1):
            self.generate_keys()

        #Assim temos todos os parâmetros para as chaves privadas e públicas
        self.private_key = (self.n, self.d)
        self.public_key  = (self.n, self.e)


    #Encapsular a chave
    def encapsulamento(self, chave, second_public_key):
        self.to_key = int(chave)
        #Gera o salt
        key_to_bytes = self.to_key.to_bytes(int(self.size/8), "big")
        #Aplicar a função de hash de sha256
        self.chave_gerada_e = hashlib.pbkdf2_hmac('sha256', key_to_bytes, se
        #cifrar a chave publica do outro interveniente
        self.cifrar(second_public_key)
        return self.chave_cifrada, self.chave_gerada_e

    def cifrar(self, second_public_key):
        n , e = second_public_key
        self.chave_cifrada = power_mod(self.to_key, e, n)

    def decifrar(self, msg_to_decifrar):
        self.chave_cifrada = msg_to_decifrar
        key_to_dec = int(self.chave_cifrada)
        self.chave_decifrada = power_mod(key_to_dec, self.d, self.n)

    #Decifrar a mensagem recebida
    def revelar(self, msg_to_decifrar):
        self.decifrar(msg_to_decifrar)
        #Converter a chave recebida para bytes
        key_to_bytes = int(self.chave_decifrada).to_bytes(int(self.size/8),
        #aplicar a função de hash para decifrar
        self.chave_gerada_d = hashlib.pbkdf2_hmac('sha256', key_to_bytes, se
        return self.chave_decifrada, self.chave_gerada_d

    #Verificar se a chave do receiver e do sender coincidem
    def verificar(self, receiver, sender):
        self.chave_gerada_e = receiver
        self.chave_gerada_d = sender
        if (self.chave_gerada_e == self.chave_gerada_d):
            print("As chaves coincidem! Mensagem Inicial intacta")
        else:
            print("As chaves não coincidem!")
```

Exercício a)

A classe KEMRSA inicializa-se com 1024 bits.

A partir daí gera uma chave privada e respetivamente a pública.

Dado um elemento aleatório, o "receiver" encapsula com esse elemento a chave pública do "sender".

Após encapsular, devolve a chave que deve ser comum e a mensagem cifrada.

O outro interveniente recebe e decifra a mensagem.

No fim, verifica-se se as chaves são iguais, caso a mensagem se verifique intacta.

As funções de encapsulamento utilizam o SHA256 para cifrar a chave que é enviada.

In [ ]:
```
random = os.urandom(16)
kem_first = KEMRSA(1024, random)
kem_second = KEMRSA(1024, random)

kem_first.generate_keys()
kem_second.generate_keys()

elem = ZZ.random_element(1024)

msg_enc, sender_key = kem_first.encapsulamento(elem, kem_second.public_key)
dec, receiver_key = kem_second.revelar(msg_enc)

kem_first.verificar(sender_key, receiver_key)

print("Chave de a: ", sender_key)
print("Chave de b: ", receiver_key)
```

```
As chaves coincidem! Mensagem Inicial intacta
Chave de a:  b'Jy6\x06T\xfc4u\xadm\x18w\xbel\x948eG\x14\xc4\xc4I\xd5\xb3[\xa
d\xb0\xb1\x80x\xfd\xb2'
Chave de b:  b'Jy6\x06T\xfc4u\xadm\x18w\xbel\x948eG\x14\xc4\xc4I\xd5\xb3[\xa
d\xb0\xb1\x80x\xfd\xb2'
```

Exercício b)

Fujisaki-Okamoto é uma transformação utilizada com o intuito de aumentar a segurança de qualquer esquema de chave-pública em algo mais seguro.

Um PKE que seja IND-CCA seguro implica que mesmo que um atacante tenha acesso à chave de decifrar, que o esquema seja indistinguível para ele.

Para isso adicionaram-se as funções **encrypt** e **decrypt** que cifram e decifram várias componentes encadeadas até obter a mensagem final e todas as verificações necessárias.

In [ ]:
```
class PKE:
    def __init__(self,kem):
        self.kem = kem

    #Funções de hash privadas
    def __hash_function_h(self, message):
        digest = hashlib.sha256()
        digest.update(message)
        return digest.digest()

    def __hash_function_g(self, message):
```

```
        digest = hashlib.sha256()
        digest.update(message)
        return digest.digest()


# Metodos Publicos
    def encrypt(self, message, public_key_d):
        # Cifrar a mensagem com uma função hash
        r = self.__hash_function_h(message)
        #Aplicar XOR à mensagem
        mcx = bytes([a ^^ b for a, b in zip(message, self.__hash_function_g(
        # Concatenamos y e r
        new_r = mcx + r
        new_r_int = int.from_bytes(new_r, "big")

        #Cifrar utilizando o KEM da alínea anterior
        cifrada_kem, chave_ambos = self.kem.encapsulamento(new_r_int, public

        #Com a chave simétrica, aplicando XOR a r
        simXor = bytes([a ^^ b for a, b in zip(chave_ambos, r)])
        return mcx, cifrada_kem, simXor

    def decrypt(self, mcx, cifrada_kem, simXor):
        #Obtemos a chave com o KEM definido antes
        _ , chave_ambos = self.kem.revelar(cifrada_kem)

        #Aplicamos o XOR com a chave simetrica de ambos para decifrar
        r = bytes([a ^^ b for a, b in zip(simXor, chave_ambos)])
        new_r = mcx + r
        new_r_int = int.from_bytes(new_r, "big")

        #Encapsular utilizando o exercício anterior
        nova_cifra_kem, nova_chave_ambos = self.kem.encapsulamento(new_r_int

        if chave_ambos != nova_chave_ambos:
            print("A chave não é simétrica")
        else:
            if cifrada_kem != nova_cifra_kem:
                print("Mensagem não coincide com a inicial!")
            else:
                message = bytes([a ^^ b for a, b in zip(mcx, self.__hash_fun
                print("Mensagem recebida: ", message)
```

In [ ]:
```
salt = os.urandom(12)

first_RSA = KEMRSA(1024, salt)
first_RSA.generate_keys()

second_RSA = KEMRSA(1024, salt)
second_RSA.generate_keys()


first_PKE = PKE(first_RSA)
mcx, cifrada_kem, simXor = first_PKE.encrypt(b"Mensagem para teste", second_

second_PKE = PKE(second_RSA)
second_PKE.decrypt(mcx, cifrada_kem, simXor)
```

```
Mensagem recebida:  b'Mensagem para teste'
```