Trabalho Prático 1

Grupo 17, constituído por:

-- Joana Castro e Sousa, PG47282

-- Tiago Taveira Gomes, PG47702

-- João Carlos Pereira Rodrigues, PG46534

Pergunta 2:

Use o SageMath para:

a) Construir uma classe Python que implemente um KEM- RSA. A classe deve

i. Inicializar cada instância recebendo o parâmetro de segurança (tamanho em bits do módulo RSA) e gere as chaves pública e privada.

ii. 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.

IMPLEMENTAÇÃO

Para gerar uma chave pública e a chave privada, inicialmente é necessário gerar certos parâmetros:

i) dois números primos ("q" e "p"), de modo a que o módulo "n" tenha como tamanho de parâmetro segurança;

ii) um ("phi") para calcular o expoente da chave pública.

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

# Secure hash FIPS 180
def hash_sha256(seed):
    digest = hashes.Hash(hashes.SHA256())
    digest.update(seed)
    digest.update(seed)
    return digest.finalize()

class KEM_RSA:
    def __init__(self, param):
        p = random_prime(2^(param/2) - 1, 2^(param/2-1))
        q = random_prime(2^(param/2) - 1, 2^(param/2-1))
        self.n = p * q

        # Função totiente de Euler
        phi = (p - 1)*(q - 1)
```

```
        e = self.gen_e(phi)
        d = self.gen_d(phi, e)

        self.param = param

        # Public Key: (n,e)
        self.public_key = (self.n, e)
        # Private Key: (n,d)
        self.private_key = (self.n, d)
```

Uma vez definidos os parâmetros, começa o trabalho de definir as chaves públicas e privadas. Assim, iremos definir duas funções que o façam:

i) **gen_e : Escolhe uma chave pública (n, e) tal que, 0 < e < phi(n) & co-primo com n e phi(n)**

ii) **gen_d : Escolhe uma chave privada (n, d) tal que, d*e mod phi(n) = 1**

In [ ]:
```
class KEM_RSA(KEM_RSA):
    def gen_e(self, phi):
        # 1 < e < phi
        e = ZZ.random_element(phi)
        # co-primo com phi e N
        while gcd(e, phi) != 1:
            e = ZZ.random_element(phi)
        return e

    def gen_d(self, phi, e):
        # Inversa modular
        d = inverse_mod(e, phi)
        return d
```

Neste momento, estamos capazes de implementar a técnica de KEM. Para tal, iremos apresentar dois algoritmos ("encaps": vocacionado para encapsular pequenas quantidades de informação ("chaves") que ele próprio gera; e "decaps": revela a chave a partir do encapsulamento desta).

Assim, para o algoritmo de encapsulamento seguimos os seguintes passos:

i) gerar um valor inteiro aleatório ("z"), entre 0 e n-1;

ii) cifrar "z" com a chave pública RSA, obtendo-se o encapsulamento;

iii) derivar a chave simétrica ("k") através de um kdf, em que k = kdf("z")

In [ ]:
```
class KEM_RSA(KEM_RSA):
    def encrypt_asym(self, pub_key, msg):
        n, e = pub_key
        return pow(msg, e, n)

    def decrypt_asym(self, priv_key, ct):
        n, d = priv_key
        return pow(ct, d, n)

    def encaps(self, pub_key):
        n, ex = pub_key
        # 1 < z < n
        z = ZZ.random_element(n)
        z_as_bytes = int(z).to_bytes(int(z).bit_length() + 7 // 8, 'big')
```

```
        salt = os.urandom(16)
        key = self.kdf(z_as_bytes, salt)

        e = self.encrypt_asym(pub_key, z)
        e_as_bytes = int(e).to_bytes(int(e).bit_length() + 7 // 8, 'big')

        return key, e_as_bytes, salt

    def decaps(self, e, salt):
        e_int = int.from_bytes(e, 'big')

        z = self.decrypt_asym(self.private_key, e_int)
        z_as_bytes = int(z).to_bytes(int(z).bit_length() + 7 // 8, 'big')

        key = self.kdf(z_as_bytes, salt)
        return key

    def kdf(self, password, salt):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=100000,)
        key = kdf.derive(password)
        return key
```

Verificação e teste dos algorimos:

In [ ]:
```
Bob_rsa = KEM_RSA(1024)
Alice_rsa = KEM_RSA(1024)

alice_pub_key = Alice_rsa.public_key

k, e, salt = Bob_rsa.encaps(alice_pub_key)
print("Chave partilhada: ", k)

key  = Alice_rsa.decaps(e, salt)
print("Chave partilhada: ",key)
```

```
Chave partilhada:  b'`\x16\xcbK\xb1\xfb\x02r\xba\x9c\xf6\x89[\x064\x1a\x9c5\x10\xfa\xcc\xa2\xc8\x94WC%d9\xc2\x95\xe7'
Chave partilhada:  b'`\x16\xcbK\xb1\xfb\x02r\xba\x9c\xf6\x89[\x064\x1a\x9c5\x10\xfa\xcc\xa2\xc8\x94WC%d9\xc2\x95\xe7'
```

Uma vez resolvidos os algoritmos, o nosso próximo passo passar por transformar um KEM em um PKE-IND-CCA usando uma transformação de Fujisaki-Okamoto (FOT)

In [ ]:
```
class PKE_RSA:
    def __init__(self, param, salt):
        p = random_prime(2^(param/2) - 1, 2^(param/2-1))
        q = random_prime(2^(param/2) - 1, 2^(param/2-1))
        self.n = p * q
        # Função totiente de Euler
        phi = (p - 1)*(q - 1)
        e = self.gen_e(phi)
        d = self.gen_d(phi, e)

        self.param = param
```

```
        # Public Key: (n,e)
        self.public_key = (self.n, e)
        # Private Key: (n,d)
        self.private_key = (self.n, d)

        self.salt = salt

    def gen_e(self, phi):
        # 1 < e < phi
        e = ZZ.random_element(phi)
        # co-primo com phi e N
        while gcd(e, phi) != 1:
            e = ZZ.random_element(phi)
        return e

    def gen_d(self, phi, e):
        # Inversa modular
        d = inverse_mod(e, phi)
        return d

    def otp_enc(self, key, msg):
        return bytes(a ^^ b for a, b in zip(msg, key))

    def otp_dec(self, key, ct):
        return bytes(a ^^ b for a, b in zip(ct, key))

    def encrypt_sym(self, pub_key, msg):
        k, e = self.encaps(pub_key)
        ct = self.otp_enc(k, msg)
        return e, ct

    def decrypt_sym(self, e, ct):
        k = self.decaps(e)
        pt = self.otp_dec(k, ct)
        return pt

    def hash_g(self, msg):
        h = hashes.Hash(hashes.SHA3_256())
        h.update(msg)
        digest = h.finalize()
        return digest

    def encrypt_asym(self, pub_key, msg):
        n, e = pub_key
        return pow(msg, e, n)

    def decrypt_asym(self, priv_key, ct):
        n, d = priv_key
        return pow(ct, d, n)

    def decaps(self, e):
        e_int = int.from_bytes(e, 'big')
        m = self.decrypt_asym(self.private_key, e_int)
        m_as_bytes = int(m).to_bytes((int(m).bit_length() + 7) // 8, 'big')
        key = self.kdf(m_as_bytes, 32)

        return key

    def encrypt(self, pub_key, msg):
```

```
        n, e = pub_key

        x = ZZ.random_element(n)

        # Gerar r
        r = os.urandom(32)
        r_as_bytes =  r
        r_as_int = int.from_bytes(r_as_bytes,"big")

        msg_as_bytes = msg.encode('utf-8')

        # Obter y ofuscando o plaintext x
        y = bytes(a ^^ b for a, b in zip( msg_as_bytes, self.kdf(r_as_bytes, len(msg_as_bytes))))

        # Obter yr
        yr = y + r_as_bytes
        yr_as_int = int.from_bytes(yr, "big")

        # (e,k) <- KEM
        k = self.kdf(yr,32)
        e = self.encrypt_asym(pub_key, yr_as_int)
        e_as_bytes = int(e).to_bytes((int(e).bit_length() + 7) // 8, 'big')

        # Obter tag c ofuscando r com chave k
        c = self.otp_enc(k, r_as_bytes)

        return y, e_as_bytes , c

    def decrypt(self, y, e, c):
        k = self.decaps(e)

        r = self.otp_dec(k, c)
        r_as_int = int.from_bytes(r,"big")
        yr = y + r
        yr_as_int = int.from_bytes(yr, "big")

        ee = self.encrypt_asym(self.public_key, yr_as_int)
        ee = int(ee).to_bytes((int(ee).bit_length() + 7) // 8, 'big')

        kk = self.kdf(yr,32)

        if ee != e or kk != kk:
            return "Error"
        else:
            pt = bytes(a ^^ b for a, b in zip(y, self.kdf(r,len(y))))
            return pt


    def kdf(self, password, len):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=len,
            salt=self.salt,
            iterations=100000,)
        key = kdf.derive(password)
        return key
```

Finalizando, apenas nos resta testar esta implementação do PKE apresentada:

In [ ]:
```
salt = os.urandom(16)

msg = "Uma mensagem para cifrar e decifrar em EC 2021/2022.\nCom participação da Alice e do Bob.\n:)"
msg_as_bytes = str.encode(msg)

Bob_rsa = PKE_RSA(1024,salt)
Alice_rsa = PKE_RSA(1024, salt)

alice_pub_key = Alice_rsa.public_key

y, e, c = Bob_rsa.encrypt(alice_pub_key, msg)


pt  = Alice_rsa.decrypt(y, e, c)
print(pt.decode())
```

```
Uma mensagem para cifrar e decifrar em EC 2021/2022.
Com participação da Alice e do Bob.
:)
```