Trabalho prático 1 - Estruturas Criptográficas

Autores: Ariana Lousada (PG47034), Cláudio Moreira (PG47844)

Grupo 12

Problema 2

2.a)

No exercício 2, era-nos pedido o desenvolvimento de um KEM-RSA através do SageMath. Numa fase inicial, deve receber um parâmetro de segurança para occorrer uma geração de chaves públicas e privadas. De seguida, atravém deste KEM deve ser desenvolvido um PKE que seja IND-CCA seguro através de uma transformação Fujisaki-Okamoto. Começamos por realizar uma importação dos módulos necessários para o desenvolvimento do exercício.

De seguida, foi realizada uma função key_generator que começa por gerar dois números primos aleatórios, p e q, com base no algoritmo RSA. Para isso foi utilizada a função random_prime do Sagemat. De seguida, existe um armazenamento na variável n do produto dos números primos p e q e um cálculo da função totiente de Euler. Posteriormente à criação da variável euler, é encontrado um ineiro e tal que este se encontra entre 1 e a variável euler e que e e euler sejam primos entre si. Finalmente, realizou-se uma cifragem e uma decifragem através das funções cipher e decypher com base no algoritmo RSA.

De seguida passou-se para o encapsulamento dos dados através da combinação do DEM com o KEM. O primeiro tem como função realizar uma ofusação dos dados enquanto que o segundo tem como objetivo a criação, comunicação e ofuscação da chave privada. Para isso, foram criadas as funções KEM e DEM.

A função KEM funciona de forma semelhante a um gerador aleatório que devolve um par de resultados. Já a função DEM realiza o encapsulamento da mensagem a partir de uma função XOR.

De seguida, foram criados algoritmos de revelação das chaves previamente escondidas. Para isso utilizaram-se as funções Reverse_KEM e Reverse_DEM que realizam a decifragem da chave previamente cifrada.

In [16]:

```
import hashlib
import binascii
from binascii import unhexlify, hexlify
class KEM_RSA:
    def __init__(self,s):
        self.s=s
    def key_generator(self):
        p = random_prime(2^self.s-1,True,2^(self.s-1))
        q = random_prime(2^self.s-1,True,2^(self.s-1))
        n = p*q
        euler = (p-1)*(q-1)
        e = ZZ.random_element(euler)
        while gcd(euler, e) != 1:
            e = ZZ.random_element(euler)
        d = inverse_mod(e,euler)
        return (d,p,q), (e,n)

    def cipher(self,message,e,n):
        cipher = pow(message,e,n)
        return cipher

    def decipher(self, message,e,n):
        decipher = pow(message,e,n)
        return decipher

    def xor_function(self, a, b):
        return bytes([ x^^y for (x,y) in zip(a, b)])

    def KEM(self,key):
        e, n = key
        r = ZZ.random_element(0,n - 1)
        cifra = kem_rsa.cipher(r,e,n)
        chave = hash(r)
        return (cifra,chave)

    def DEM(self, mensagem, key):
        m = binascii.hexlify(mensagem.encode('utf-8'))
        k = binascii.hexlify(str(key).encode('utf-8'))
        cript = kem_rsa.xor_function(m,k)
        return cript

    def Reverse_KEM(self, cipher, key1, key2):
        d,p,q = key1
        e, n = key2
        r = kem_rsa.decipher(cipher,d,n)
        key = hash(r)
        return key

    def DRev(self, cript, cipher, key1, key2):
        key = self.Reverse_KEM(cipher, key1, key2)
        k = binascii.hexlify(str(key).encode('utf-8'))
        original_text = kem_rsa.xor_function(cript,k)
        original_text = binascii.unhexlify(original_text.decode('utf-8'))
        original_text = original_text.decode('utf-8')
        return original_text
```

Exemplo da testagem.

In [56]:

```
kem_rsa = KEM_RSA(512)
key1, key2 = kem_rsa.key_generator()

cipher, keyEnc = kem_rsa.KEM(key2)
cript = kem_rsa.DEM("secret message ", keyEnc)
print("Criptogram: ",cript)


original_text = kem_rsa.DRev(cript,cipher, key1, key2)
print("Original Text: ", original_text)
```

```
Criptogram:  b'\x04\x01\x05\x05\x05\x03\x04\x00\x05\x03\x04\x01\x01\x03\x05Q\x05\x01\x04\x00\x04\x00
\x05\x08\x05\x01\x05\x00\x01\x08'
Original Text:  secret message
```

2.b)

Passou-se para a resolução da alínea b onde nos era pedido a construção de um PKE que seja IND-CAA seguro , a partir do KEM construido anteriormente e através da transformação *Fujisaki-Okamoto*. Para isso utilizou-se uma função de cifragem (cipher_FO) e uma função de decifragem (decipher_FO).

Sendo assim, começou-se com a definição da função cipher_FO. Esta, como o nome indica, tem o objetivo cifrar uma mensagem sendo que começa por aplicar uma função hash às variáveis h1 e h2,sendo que a aplicação na h1 é feita através de um número pseudo-aleatórtio. Já a aplicação na variável h2 é feita através do valor calculado em h1. De seguida, é realizado um XOR da mensagem a variável calculada em h2 para "esconder" a mensagem. Posteriormente, é realizado uma concatenação da varíavel y com h1 para obter o encapsulamento da chave. Finalmente, é realizado uma operação XOR entre a key e a variável h1.

De seguida, realizou-se a função decypher_FO. Esta começa por realizar uma decifragem da chave através do algoritmo definido anteriormente. Como é possível observar, esta função possui um comportamento semelhante à cipher_FO realizando novamente cifra com o RSA. No final é realizado apenas um *if/else* para garantir que o resultado da cifra não é o mesmo que o valor anterior.

In [57]:

```
def cipher_FO(key1, message):
    h1,h2 = key1
    r = hash(ZZ.random_element(0,h2 - 1))
    g = hash(str(r))
    m = binascii.hexlify(message.encode('utf-8'))
    g = binascii.hexlify(str(g).encode('utf-8'))
    y = kem_rsa.xor_function(m,g)
    y2 = int.from_bytes(y, "big")
    concatencao_y_r = str(y2) + str(r)
    cipher_var = kem_rsa.cipher(int(concatencao_y_r),h1,h2)
    key2 = hash(concatencao_y_r)
    k = binascii.hexlify(str(key2).encode('utf-8'))
    r_bytes = binascii.hexlify(str(r).encode('utf-8'))
    c = kem_rsa.xor_function(r_bytes,k)
    return y, cipher_var, c

def decipher_FO(key1,key2,y, cipher_var, c):
    h1, h2 = key1
    d,p,q = key2
    decipher = kem_rsa.decipher(cipher_var,d,h2)
    key = hash(str(decipher))
    k = binascii.hexlify(str(key).encode('utf-8'))
    r = kem_rsa.xor_function(c,k)
    r = binascii.unhexlify(r.decode('utf-8'))
    g = hash(r)
    g = binascii.hexlify(str(g).encode('utf-8'))
    cipher_var_2 = kem_rsa.cipher(decipher,h1,h2)
    if cipher_var!=cipher_var_2:
        print("ERRO")
        return
    else:
        original_text = kem_rsa.xor_function(y,g)
        original_text = binascii.unhexlify(original_text.decode('utf-8'))
        original_text = original_text.decode('utf-8')
        return original_text
```

Exemplo da testagem:

In [59]:

```
kem_rsa = KEM_RSA(512)
key1, key2 = kem_rsa.key_generator()
mensagem = "secret message"
y, cipher, c = cipher_FO(key2, mensagem)
print("Message:", mensagem, "\n" )
print("Hidden Message:", y, "\n")
print("Encapsulated key:", cipher, "\n")
print("Hidden key:", c, "\n")
original_text = decipher_FO(key2, key1, y, cipher, c)
print("Original Text:",original_text)
```

```
Message: secret message

Hidden Message: b'\x05W\x05\x00\x05\x00\x04\x02\x05\x03\x04\x0c\x01\x03\x05Q\x05\x01\x04\x00\x04\x07
\x05\x08\x05\x01\x05\x06'

Encapsulated key: 551829337806024976359784506540112678521075267757997797550001688383003754058203405324313700812080645590495956005733567072730124167765683729874369966032053424498711618944649932155891105611802887665778093187443860673849309322311500854957012876940434166471315703215829374467635764573789120233794183546589053003340
Hidden key: b'\x01U\x00\x0b\x00\x07\x00\x03\x00\t\x00\x00\x00\x00\x00\x02\x00\x02\x00\n\x00\x07\x00\
x05\x00\x05\x00\x02\x00\x02\x00\x01\x00\x00\x00\x0c\x00\r'

Original Text: secret message
```

In [ ]:

```
```