TP1-Problema2

March 28, 2022

1 TRABALHO PRÁTICO 1 - GRUPO 14

1.1 Problema 2

1.1.1 Resolução do Problema - Parte 1

Imports

[1]:
```
import hashlib
import binascii
from binascii import hexlify, unhexlify
```

1.1.2 Classe KEM_RSA

Neste problema era proposta a implementação da classe KEM-RSA. Numa primeira fase, ocorre a geração das chaves pública e privada a partir de um parâmetro de segurança, sendo ele o tamanho em bits inserido na inicialização da classe. Para tal, é implementada a função `key_gen` que com base no algoritmo *RSA* gera dois números primos aleatórios, **p** e **q**. Posteriormente, a partir da multiplicação destes dois primos é obtido **n**, o módulo para as chaves pública e privada. De seguida, obtém-se o valor do **phi** através da função totiente e é encontrado um número aleatório que seja relativamente primo com o phi(n). No fim, é retornado o tuplo **(d,p,q)** e **(e,n)** que correspondem à chave privada e pública.

De seguida, foram implementadas as funções de cifragem e decifragem de acordo com o algoritmo RSA, sendo elas a `encrypt` ($C = M^e mod N$) e `decrypt` ($M = C^d mod N$). De forma a encapsular os dados e retratar o algoritmo corretamente, foi necessária a combinação de dois mecanismos: o DEM (*Data Encapsulation Mechanism*) que actua sobre os dados a ofuscar e o KEM (*Key Encapsulation Mechanism*) que comunica e ofusca a chave privada requerida pelo DEM. Posto isto, foram criados os seguintes métodos:

1. **KEM:** função responsável por gerar a chave a ser utilizada pelo DEM e fazer o encapsulamento da mesma. Para a parte da geração da chave, foi cifrado com recurso à função `encrypt` do algoritmo RSA um número pseudo-aleatório **random_generated**. De seguida, é feito o encapsulamento desse número pseudo-aleatório a partir de uma função de hash;
2. **KRev:** função responsável por revelar a chave que foi encapsulada com o método anterior, KEM. Para tal, é ainda utilizada a função de decifra do RSA, `decrypt`;
3. **DEM:** função que permite o encapsulamento da mensagem a partir da operação XOR(`xor`) entre a chave e a mensagem;
4. **DRev:** função responsável pela decifragem de forma a obter a mensagem original. Esta recorre à função KRev para obter a chave, e posteriormente é efetuado o XOR entre o criptograma e a chave obtida.

[10]:
```
class KEM_RSA:
    def __init__(self,s):
        self.s = s

    #Funçao: Gerar chaves publicas e privada
    def key_gen(self):
        #gerar os parametros p e q , primos
        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
        #print('p generated: ',p)
        #print('q generated: ',q)
        #funçao totiente de phi
        phi = (p-1)*(q-1)
        #print('Phi:',phi)
        #numero inteiro que seja relativamente primo com o phi de n
        e = ZZ.random_element(phi)

        # descobrir "e" que seja primo
        while gcd(phi,e) != 1:
            #obtemos a nossa chave de cifragem "e"
            e = ZZ.random_element(phi)
        #obtençao da nossa chave de decifra "d"
        d = inverse_mod(e,phi)

        return(d,p,q), (e,n)

    #Funçao: cifrar com base no algoritmo RSA
    def encrypt(self,message,e,n):
        #C = M^e mod N
        cipher = pow(message,e,n)
        return cipher

    #Funçao: decifrar com base no RSA
    def decrypt(self,message,d,n):
        #M = C^d mod N
        plaintext = pow(message,d,n)
        return plaintext

    #Funçao: Operaçao XOR
    def xor(self,a,b):
        return bytes([ x^^y for (x,y) in zip(a,b)])

    #Funçao: Geraçao de chave e encapsulamento da chave a ser usada no DEM
    def KEM(self,pubk):
        #Parametros da public_key
        e, n = pubk
```

```
        #print('E:',e)
        #print('N:',n)
        random_generated = ZZ.random_element(0, n - 1)
        #print('Random: ',random_generated)
        cipher = self.encrypt(random_generated, e, n)
        #print('Cipher: ', cipher)
        key = hash(random_generated)
        print('Key KEM: ', key)
        return (cipher,key)

    #Funçao: Associado ao KRev revela a chave de encapsulamento
    def KRev(self,cipher,pk,pubk):
        #parametros da privatekey
        d,p,q = pk
        #parametros da publickey
        e,n = pubk
        random = self.decrypt(cipher,d,n)
        key = hash(random)
        return key

    #Funçao: encapsulamento da mensagem a partir do XOR'ing
    def DEM(self,message,key_krev):
        msg = binascii.hexlify(message.encode('utf-8'))
        key = binascii.hexlify(str(key_krev).encode('utf-8'))
        print('msg:',msg)
        print('Key DEM:', key)
        criptogram = self.xor(msg,key)
        return criptogram

    #Funçao: revelaçao do texto original através do XOR
    def DRev(self,criptogram,cipher,pk,pubk):
        key = self.KRev(cipher,pk,pubk)
        k = binascii.hexlify(str(key).encode('utf-8'))
        plaintext = self.xor(criptogram,k)
        #print('Plain: ',plaintext)
        plaintext = binascii.unhexlify(plaintext.decode('utf-8')).
↪decode('utf-8')
        return plaintext
```

Exemplo de Teste

[11]:
```
kem_rsa = KEM_RSA(1024)
privk, pk = kem_rsa.key_gen()
#print('Public: ',pk)
#print('Private: ',privk)
```

```
c,k = kem_rsa.KEM(pk)
#print('C: ',c)
#print('K: ',k)

criptogram = kem_rsa.DEM("secret message",k)
print('Criptograma: ', criptogram)


plaintext = kem_rsa.DRev(criptogram,c,privk,pk)
print('Plaintext: ',plaintext)
```

```
Key KEM:  504580612096035251
msg: b'736563726574206d657373616765'
Key DEM: b'3530343538303631323039363033335323531'
Criptograma:  b'\x04\x06\x05\x05\x05\x07\x04\x07\x05\r\x04\x04\x01\x06\x05U\x05\
x07\x04\x03\x04\n\x05\x07\x05\x07\x05\x06'
Plaintext:  secret message
```

1.1.3 Resolução do Problema - Parte 2

Ainda parte deste problema, numa outra alínea era pedida a construção a partir da classe KEM-RSA definida e utilizando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro. Para tal, foram desenvolvidas duas funções responsáveis pela cifra e decifra de acordo com a transformação de Fujisaki-Okamoto.

fot_encrypt: função responsável por cifrar a mensagem de acordo com a seguinte fórmula matemática descrita abaixo,

$E\ (x)\ r \leftarrow h\ y \leftarrow x\ g(r)\ (e,k) \leftarrow f(y\ r)\ c \leftarrow k\ r\ (y,e,c)$

1. gerar um **random_generated** (r) que é resultado do hash a um número pseudo-aleatório;
2. calcular **g** a partir da hash (random_generated);
3. gerar **xored** (y), que corresponde ao encapsulamento da mensagem, a partir do XOR entre o *plaintext* e o **g** do ponto 2.;
4. fazer a concatenação de **y** (xored) com **r** (random_generated) e fazer a cifra desta operação através do algoritmo RSA implementado na alínea anterior `encrypt`, obtendo a ofuscação da chave;
5. fazer o cálculo da **key** a partir da operação de hash ao resultado da concatenação do ponto 4.;
6. por fim, o encapsulamento da chave que é resultado do XOR entre a **key** e o **r** (random_generated).

[4]:
```
#Instancia da classe KEM_RSA
kem = KEM_RSA(512)
#gerar chaves publicas e privadas
privk, pubk = kem_rsa.key_gen()

#Funçao: Cifragem de acordo com a transformaçao de Fujisaki-Okamoto
#E (x) r←h y←x g(r) (e,k)←f(y r) c←k r (y,e,c)
```

```
def fot_encrypt(pubk,message):
    #parametros da publickey
    e, n = pubk
    # gerar um numero aleatorio "r"
    random_generated = hash(ZZ.random_element(0, n-1))
    #calcular hash(r)
    g = hash(str(random_generated))
    # XOR'ing da mensagem com o "g"=hash(r)
    msg = binascii.hexlify(message.encode('utf-8'))
    key = binascii.hexlify(str(g).encode('utf-8'))
    xored = kem_rsa.xor(msg,key)
    #print('Xored: ', xored)
    xored_int = int.from_bytes(xored,"big")
    #concatenaçao do output do xor com a hash gerada "random_generated-r"
    concatenate = str(xored_int) + str(random_generated)

    #cifragem através da instancia KEM_RSA algoritmo encrypt(encapsular chave)
    cipher = kem_rsa.encrypt(int(concatenate),e,n)
    #hash da concatenaçao que será a nossa key
    hash_key = hash(concatenate)
    k = binascii.hexlify(str(hash_key).encode('utf-8'))
    #XOR da key com o "r"-random_generated
    r = binascii.hexlify(str(random_generated).encode('utf-8'))
    #utilizaçao da funçao de xor do Kem_RSA - ofuscação da chave
    key_encaps = kem_rsa.xor(r,k)

    return xored, cipher, key_encaps
```

fot_decrypt: função responsável pela decifra da mensagem, obtendo a mensagem original de acordo com a fórmula matemática descrita abaixo,

$D\ (y, e, c)\ k \leftarrow KREv(e)\ r \leftarrow c\ k\ if(e, k)\ f(y\ r) then\ else y\ g(r)$

1. ocorre a revelação da chave a partir da decifragem, utilizando a instância da classe KEM-RSA função `decrypt` e posterior cálculo da hash;
2. obtenção do **r** (random_generated) através do XOR entre o **c** e a **key** e cálculo do **g** através da hash do **r** (random_generated);
3. fazer a concatenação de **y** (xored) com **r** (random_generated) e fazer a cifra desta operação através do algoritmo RSA implementado na alínea anterior `encrypt`;
4. verificação se o valor da **cipher** obtido é igual ao obtido no ponto 2;
5. por fim, é feito o XOR entre o **y** (xored) e o **g** para obter a mensagem original.

[5]:
```
#Funçao: decifragem de acordo com a transformaçao de Fujisaki-Okamoto
#D (y,e,c) k←KREv(e) r←c k if(e,k) f(y r)then else y g(r)
def fot_decrypt(pubk,privk,xored,cipher,key_encaps):
    #parametros da pubkey
    e, n = pubk
    #parametros da privatekey
```

```
    d,p,q = privk

    #Instancia de classe KEM_RSA: utilizaçao do decrypt para revelaçao de chave
    decrypt = kem_rsa.decrypt(cipher,d,n)
    key = hash(str(decrypt))
    k = binascii.hexlify(str(key).encode('utf-8'))

    #XOR'ing entre a chave de encapsulamento e o "k" para descobrir "r" -␣
↪randomgenerated
    random_generated = kem_rsa.xor(key_encaps,k)
    random_generated = binascii.unhexlify(random_generated.decode('utf-8'))

    g = hash(random_generated)
    g = binascii.hexlify(str(g).encode('utf-8'))

    #concatenacao do xored recebido e do "r"-random-generated e cifragem com␣
↪KEM_RSA.encrypt para futura verificaçao
    xored_int = int.from_bytes(xored,"big")
    concatenate = str(xored_int) + str(int(random_generated))
    cipher_verify = kem_rsa.encrypt(decrypt,e,n)
    #verificaçao
    if cipher != cipher_verify:
        print("ERROR: The cipher doesn't match!")
        return
    else:
        # XOR entre o output do xored e a hash(r)-g para obter o plaintext
        plaintext = kem_rsa.xor(xored,g)
        #print('Plain: ',plaintext)
        plaintext = binascii.unhexlify(plaintext.decode('utf-8')).
↪decode('utf-8')
        return plaintext
```

Exemplo de Teste

[6]:
```
mensagem = "msg secreta"
msg_encaps, cipher, key_encaps = fot_encrypt(pubk,mensagem)
print('Message encapsulation: ',msg_encaps)
print('Ciphertext: ',cipher)
print('Key encapsulation: ',key_encaps)

plaintext = fot_decrypt(pubk,privk,msg_encaps,cipher,key_encaps)
print('Plaintext: ',plaintext)
```

```
Message encapsulation:  b'\x05P\x04\x05\x05\x05\x01\t\x04\x04\x05\x03\x05\x07\x0
4\n\x05\r\x04\x03\x05\x01'
Ciphertext:  107484350034965314761708792572357669372516293186280451854112947655
000595389786571964966486540024032732396811245157408142832868917266514297407923 79
```

```
20188023005474948676999025525792669631197061258050701134383782830075535234051442
51101841101946427834748406368522815617213731935642570845342483603520404693127312
93
Key encapsulation:  b'\x00\x0c\x00\x01\x00\x05\x00\x00\x00\x0e\x00\x0c\x00\x05\x
00\x0b\x00\x0f\x00\x07\x00\x0c\x00\x06\x00\t\x00\x0f\x00\x00\x00\x06\x00\x03\x00
\x06'
Plaintext:  msg secreta
```