TP01_P2

April 7, 2021

Solução Para o Trabalho Prático 01

Problema 02:

1. Implementação de um KEM-RSA que inicializa as instâncias com os respetivos paramêtros de segurança, gera chave publica e privada, encapsulamento e revelação da chave gerada;
2. Segundo, ainda baseado no KEM implementado, a usufruindo da transformação de Fujisaki-Okamoto, a criação de um PKE IND-CCA;
3. Em seguida, a implementação de um DSA, onde são definidos na inicialização os parametros dos primos p e q com seus respectivos tamanhos, e ainda assinatura digital e a verificação da mesma;
4. Por fim, a implementação de um ECDSA que utilizará uma das curvas primas definidas pelo FIPS186-4.

[4]:
```
import os
import random
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
from sage.arith.power import generic_power
from sage.functions.log import logb
from sys import getsizeof
```

Implementação KEM-RSA

Temos então definidos a classe KEM-RSA, composta por 9 funções:

- *init*
- *generate_keys*
- *encapsulate*
- *reveal*
- *verify*
- *rsa_encrypt*
- *rsa_decrypt*
- *kdf*
- *print_nums*

[5]:
```
class KEMRSA:
    # construtor da Classe
    def __init__(self, seg):
```

```python
        # parametro de segurança
        self.size = seg
        # tamanho em bits tanto do primo p como do primo q
        self.p_q_size = int(seg/2)
        # salt para o kdf
        self.kdf_salt = os.urandom(16)
        # primo p
        self.p = None
        # primo q
        self.q = None
        # inteiro d (chave privada)
        self.d = None
        # n = p * q
        self.n = None
        # parte da chave pública
        self.pk = None
        # chave em bytes antes da hash
        self.r = None
        # chave revelada em bytes
        self.r_b = None
        # chave como um inteiro
        self.r_as_int = None
        # chave revelada como um inteiro
        self.r_b_as_int = None
        # criptograma como um inteiro
        self.e = None
        # chave simétrica
        self.symetric_key_a = None
        # chave simétrica revelada
        self.symetric_key_b = None

    def generate_keys(self):
        # conjunto dos números primos
        P = Primes()

        # gerar um p, obter um número aleatório com p_q_size bits de tamanho até encontrar um primo
        p_candidate = ZZ.random_element(2^(self.p_q_size - 1) + 1, (2^self.p_q_size) - 1)
        while not (p_candidate in P):
            p_candidate = ZZ.random_element(2^(self.p_q_size - 1) + 1, (2^self.p_q_size) - 1)

        # gerar um q, obter um número aleatório com p_q_size bits de tamanho até encontrar um primo (devia ser menor que p)
        q_candidate = ZZ.random_element(2^(self.p_q_size - 1) + 1, (2^self.p_q_size) - 1)
```

```
        while not (q_candidate in P):
            q_candidate = ZZ.random_element(2^(self.p_q_size - 1) + 1, (2^self.
↪p_q_size) - 1)

        # p e q escolhidos
        self.p = p_candidate
        self.q = q_candidate

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

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

        # gerar um e
        #e_candidate = ZZ.random_element(phi)
        #while gcd(e_candidate, phi) != 1:
        #    e_candidate = ZZ.random_element(phi)
        # e escolhido
        self.pk = 65537

        # algoritmo extendido de Euclides para calcular d
        bezout = xgcd(self.pk, phi)
        d_candidate = Integer(mod(bezout[1], phi))
        self.d = d_candidate

    def encapsulate(self, key_as_int):
        self.r_as_int = int(key_as_int)
        self.r = self.r_as_int.to_bytes(int(self.size/8), "big")
        # a chave é o kdf aplicado à mensagem em bytes
        self.symetric_key_a = self.kdf(self.r)
        # ciframos o texto limpo
        self.rsa_encrypt()

    def reveal(self):
        # deciframos o criptograma
        self.rsa_decrypt()
        self.r_b = int(self.r_b_as_int).to_bytes(int(self.size/8), "big")
        self.symetric_key_b = self.kdf(self.r_b)

    def verify(self):
        print("Chaves iguais:", self.symetric_key_a == self.symetric_key_b)

    def rsa_encrypt(self):
        # o criptograma é m^e mod n
        self.e = power_mod(self.r_as_int, self.pk, self.n)
```

```
    def rsa_decrypt(self):
        # o texto_limpo é c^d mod n
        self.r_b_as_int = power_mod(self.e, self.d, self.n)

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

    def print_nums(self):
        print("p =", self.p,"\n")
        print("q =", self.q,"\n")
        print("d =", self.d,"\n")
        print("n =", self.n,"\n")
        print("e =", self.pk,"\n")
```

Função para executar o processo do KEM-RSA

[6]:
```
def ex1():
    # primeiro inicializamos tudo com o parâmetro de segurança
    a = KEMRSA(1024)
    # depois geramos as chaves e os parâmetros p, q ...
    a.generate_keys()
    # depois encapsulamos esta chave
    a.encapsulate(int(ZZ.random_element(2^(a.size - 2))))
    # depois revelamos a chave
    a.reveal()
    # finalmente verificamos se correu tudo bem
    a.verify()
    #a.print_nums()
```

[18]:
```
ex1()
```

```
Chaves iguais: True
```

Implementação PKE

Temos então definidos a classe PKE, composta por 7 funções:

- *init*
- *hashg*
- *hash*
- *mini_xor*
- *xor*
- *cifrar*

- *decifrar*

```
[14]: class PKE:
    # construtor da classe
    def __init__(self, x):
        self.x = x

    # hash h
    def hashh(self, message):
        digest = hashes.Hash(hashes.SHA256())
        digest.update(message)
        return digest.finalize()

    # hash g
    def hashg(self, message):
        digest = hashes.Hash(hashes.BLAKE2s(32))
        digest.update(message)
        return digest.finalize()

    # xor de um byte a com um byte b (o sagemath faz interferência com o␣
 ↪operador '^')
    def mini_xor(self, a, b):
        tmpa = a
        tmpb = b
        r0 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r1 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r2 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r3 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r4 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r5 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r6 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
        tmpb = int(tmpb//2)
        r7 = tmpa % 2 + tmpb % 2
        tmpa = int(tmpa//2)
```

```
        tmpb = int(tmpb//2)

        soma = 0
        if r0 == 1:
            soma += 1
        if r1 == 1:
            soma += 2
        if r2 == 1:
            soma += 4
        if r3 == 1:
            soma += 8
        if r4 == 1:
            soma += 16
        if r5 == 1:
            soma += 32
        if r6 == 1:
            soma += 64
        if r7 == 1:
            soma += 128

        return soma

    # xor de dois arrays de bytes
    def xor(self, a, b):
        size = len(b)
        if len(a) < len(b):
            size = len(a)

        xored = bytearray(size)
        for i in range(size):
            xored[i] = self.mini_xor(a[i], b[i])
        return xored

    # E'
    def cifrar(self, a):
        # primeiro passo, r <- h
        self.r = self.hashh(b'teste')
        # segundo passo, y <- x XOR g(r)
        self.y = self.xor(self.x, self.hashg(self.r))
        # terceiro passo, r' <- y || r
        self.rl = self. y + self.r
        self.rl_as_int = int.from_bytes(self.rl, "big")
        # quarto passo, KEM(r')
        a.encapsulate(self.rl_as_int)
        # vamos buscar o e
        self.e = a.e
        # finalmente c = k XOR r
```

```
        self.c = self.xor(a.symetric_key_a, self.r)

    # D'
    def decifrar(self, a):
        # KREv e
        a.reveal()
        # verificação não faz mal a ninguém
        a.verify()
        # k <- KREv(e)
        self.k = a.symetric_key_b
        # r <- c XOR k
        self.r_2 = self.xor(self.c, self.k)
        # r' = y || r
        self.rl_2 = self.y + self.r_2
        self.rl_2_as_int = int.from_bytes(self.rl_2, "big")
        # verificação f(rl) == (e, k)
        a.encapsulate(self.rl_2_as_int)
        if(a.e == self.e) & (self.k == a.symetric_key_a):
            # x == y XOR g(r)
            print("Transformação FO deu:", self.x == self.xor(self.y, self.
↪hashg(self.r_2)))
```

Função para executar o processo PKE

[15]:
```
def ex2():
    # inicializamos a classe KEMRSA
    a = KEMRSA(1024)
    # geramos os parâmetros
    a.generate_keys()
    # inicializamos a classe PKE
    b = PKE(b'teste')
    # fazemos E'(x)
    b.cifrar(a)
    # fazemos D' (yec)
    b.decifrar(a)
```

[16]:
```
ex2()
```

```
Chaves iguais: True
Transformação FO deu: True
```

Implementação DSA

A classe DSA é composta pelas funções init, generate__parameters, generate__keys, sign, compute__s, compute__r, hash__init e verify.

[8]:
```
class DSA:
    def __init__(self, seg):
        # parâmetro de segurança
```

```
        self.l = seg
        # tamanho do n de forma a não exceder |H|
        self.n = 256
        # h costuma ser 2
        self.h = 2
        # primo q
        self.q = None
        # primo p
        self.p = None
        # parametro g
        self.g = None
        self.private_key = None
        self.public_key = None
        # r e s, par de inteiros da assinatura
        self.r = None
        self.s = None
        # variaveis usadas na assinatura e verificação
        self.k = None
        self.h_m  = None
        self.w = None
        self.u1 = None
        self.u2 = None
        self.v = None

    def generate_parameters(self):
        # conjunto dos números primos
        P = Primes()

        flag = 1

        # enquanto que não tivermos gerado um p e um q
        while flag:
            # gerar o p, primo de l bits
            p_candidate = ZZ.random_element(2^(self.l - 1) + 1, (2^self.l) - 1)
            while not (p_candidate in P):
                p_candidate = ZZ.random_element(2^(self.l - 1) + 1, (2^self.l)␣
↪- 1)
            self.p = p_candidate

            # para gerar o q temos duas alternativas
            """
            # alternativa 1, começar pelo primo 2^127 - 1 e ver se p-1 é␣
↪múltiplo, senão obter o próximo primo
            # processo caro devido graças ao passo de obter o próximo primo
            p_menos_um = self.p - 1
            q_candidate = (2**127) - 1
```

```
            it = 0
            limite = 2**256
            while ((p_menos_um % q_candidate) != 0) & (q_candidate < limite):
                q_candidate = P.next(q_candidate)
                it += 1

            if q_candidate < 2**256:
                self.q = q_candidate
                flag = 0"""

            # alternativa 2, começar por (p-1)/div e ver se é primo, se não for␣
↪vemos se (p-1)/div++ é primo
            p_menos_um = self.p - 1
            q_candidate = p_menos_um
            # div poderia ser 1 ou 2 mas escolhemos este para q ter no máximo␣
↪256 bits de tamanho
            div = 2**(self.l - 256)
            while ((not (q_candidate in P)) | (p_menos_um % q_candidate) != 0)␣
↪& (q_candidate > 1):
                q_candidate = int(p_menos_um / div)
                div += 1
            if q_candidate < 2**256:
                self.q = q_candidate
                flag = 0

        # gerar o g com o h predefenido
        self.g = power_mod(self.h, int((self.p - 1)/self.q), self.p)
        # se g == 1 geramos um h novo e experimentamos
        while self.g == 1:
            self.h = ZZ.random_element(3, p - 2)
            self.g = power_mod(self.h, int((self.p - 1)/self.q), self.p)

    def generate_key(self):
        # chave privada é um inteiro aleatório (1..q - 1)
        self.private_key = ZZ.random_element(1, self.q-1)
        # chave pública é g^(chave privada) mod p
        self.public_key = power_mod(self.g, self.private_key, self.p)

    def sign(self, message):
        # primeiro escolhemos um k, importante, k deveria ser gerado por um␣
↪PRNG criptográficamente forte
        self.k = ZZ.random_element(1, self.q-1)
        # calculamos r
        self.compute_r()
        # calculamos s
        self.compute_s(message)
        # enquanto que qualquer um deles for 0 repetimos o processo
```

```python
        while (self.r == 0) | (self.s == 0):
            self.k = ZZ.random_element(1, self.q-1)
            self.compute_r()
            self.compute_s(message)

    def compute_r(self):
        # r = (g^k  mod p) mod q
        tmp = power_mod(self.g, self.k, self.p)
        self.r = tmp % self.q

    def compute_s(self, message):
        # primeiro fazemos H(m)
        self.h_m = self.hash_int(message)
        # depois H(m) + xr
        tmp = self.h_m + (self.private_key * self.r)
        # dividimos por k
        tmp = tmp/self.k
        # mod q
        self.s = tmp % self.q

    def hash_int(self, message):
        # hash da mensagem convertida num inteiro
        digest = hashes.Hash(hashes.SHA256())
        digest.update(message)
        return int.from_bytes(digest.finalize(), "big")

    def verify(self, message):
        # como estamos num ambiente controlado sabemos que 0 < r < q e 0 < s < q
        # w = s^-1 mod q
        self.w = power_mod(self.s, -1, self.q)
        # u1 = (H(m) * w) mod q
        self.u1 = (self.h_m * self.w) % self.q
        # u2 = (r * w) mod q
        self.u2 = (self.r * self.w) % self.q
        # agora fazemos g^u1 mod q
        tmp = power_mod(self.g, self.u1, self.p)
        # a esse resultado multiplicamos x^u2 mod p
        tmp = tmp * power_mod(self.public_key, self.u2, self.p)
        # a isso tudo fazemos mod p
        tmp = tmp % self.p
        # finalmente v é igual ao anterior mod q
        self.v = tmp % self.q
        # verificamos se v é igual a r
        print(self.v == self.r)

    def test_parameters(self):
```

```
        self.p =␣
↪1444966394841696567029138850532091776179192779938718039682665565793385805638893814827047687
        self.q = 1422218247344634743859933275335176438393052153679
        self.g =␣
↪96310659971562742116469593861258820096385438737507853418033235652927122210740787446849366651
```

Função para executar o processo do DSA

[9]:
```python
def ex3():
    # inicializamos o DSA
    a = DSA(1024)
    # geramos os parametros
    #a.generate_parameters()
    # ou então usamos os pre gerados
    a.test_parameters()
    # geramos um par de chaves
    a.generate_key()
    # mensagem aleatória
    m = os.urandom(16)
    # assinamos a mensagem
    a.sign(m)
    # verificamos a assinatura
    a.verify(m)
```

[10]:
```python
ex3()
```

Implementação DSAEC

A classe DSAEC é composta pelas funções init, generate_key, hashm, sign e verify.

[214]:
```python
class DSAEC:
    # construtor da classe
    def __init__(self):
        # a curva
        self.curve = None
        # o Ponto G
        self.G = None
        self.gx = None
        self.gy = None
        # ordem n
        self.n = None
        # módulo p
        self.p = None
        # coeficiente b
        self.b = None
        # chave privada (d)
        self.private_key = None
        # chave pública (Q)
```

```
        self.public_key = None
        # inteiros da assinatura
        self.r = 0
        self.s = 0

    # funções com curvas pré definidas
    def curva_p192(self):
        self.gx = int(0x188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012)
        self.gy = int(0x07192b95ffc8da78631011ed6b24cdd573f977a11e794811)
        self.n = 6277101735386680763835789423176059013767194773182842284081
        self.p =  6277101735386680763835789423207666416083908700390324961279
        self.b = int(0x64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1)
        self.curve = EllipticCurve(GF(self.p), [-3,self.b])
        self.G = self.curve(self.gx, self.gy)

    def curva_p224(self):
        self.gx =␣
↪int(0xb70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21)
        self.gy =␣
↪int(0xbd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34)
        self.n =␣
↪26959946667150639794667015087019625940457807714424391721682722368061
        self.p =␣
↪26959946667150639794667015087019630673557916260026308143510066298881
        self.b = int(0xb4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4)
        self.curve = EllipticCurve(GF(self.p), [-3,self.b])
        self.G = self.curve(self.gx, self.gy)

    def curva_p521(self):
        self.gx =␣
↪int(0xc6858e06b70404e9cd9e3ecb662395b4429c648139053fb521f828af606b4d3dbaa14b5e77efe75928fe1
        self.gy =␣
↪int(0x11839296a789a3bc0045c8a5fb42c7d1bd998f54449579b446817afbd17273e662c97ee72995ef42640c5
        self.n =␣
↪686479766013060971498190079908139321726943530014330540939446345918554318339765539424505774
        self.p =␣
↪686479766013060971498190079908139321726943530014330540939446345918554318339765605212255964
        self.b =␣
↪int(0x051953eb9618e1c9a1f929a21a0b68540eea2da725b99b315f3b8b489918ef109e156193951ec7e937b16
        self.curve = EllipticCurve(GF(self.p), [-3,self.b])
        self.G = self.curve(self.gx, self.gy)

    # geração da chave
    def generate_key(self):
        # chave privada é um inteiro (1..n-1)
        self.private_key = int(ZZ.random_element(1, int(self.n)-1))
```

```
        # chave pública é d X G (multiplicação do ponto por um escalar)
        self.public_key = self.private_key*self.G

    # função genérica para uma hash que devolve já os n bits à esquerda
    def hashm(self, message):
        digest = hashes.Hash(hashes.SHA256())
        digest.update(message)
        digest = digest.finalize()
        # ln bytes
        ln = floor(int(self.n).bit_length()/8)
        # à esquerda
        return int.from_bytes(digest[ln:], "big")

    # assinatura de uma mensagem
    def sign(self, message):
        # z são os Ln bits à esquerda de H(m)
        z = self.hashm(message)

        # se r for 0 ou s for 0 temos de repetir o processo apartir do passo em␣
↪que geramos o k
        while self.r == 0 | self.s == 0:
            # k é um número aleatório (criptográficamente seguro idealmente)
            k = int(ZZ.random_element(1, self.n-1))

            # calcular o ponto k X G mod p
            (x1, y1, Z) = k * self.G
            x1 = int(x1)
            y1 = int(y1)

            # r = x1 mod n
            self.r = x1 % self.n

            # s = ((z + r*dA)/k)
            inverse_k = inverse_mod(k, self.n)
            self.s = int(((z + (self.r * self.private_key)) * inverse_k) % self.
↪n)

    # verificação de uma assinatura de uma mensagem, já sabemos que tanto s␣
↪como r são válidos porque estamos num ambiente controlado
    def verify(self, message):
        # z são os Ln bits à esquerda de H(m)
        z = self.hashm(message)

        # u1 = (z/s) mod n
        inverse_s = inverse_mod(self.s, self.n)
        u1 = int(int(inverse_s * z) % self.n)
        # u2 = (r/s) mod n
```

```
        u2 = int(int(self.r * inverse_s) % self.n)

        # x1,y1= u1 X G + u2 X Qa mod n
        (x1, y1, Z) = (u1 * self.G) + (u2 * self.public_key)
        x1 = int(x1) % self.n
        y1 = int(y1) % self.n

        # x == r (tanto x como r já estão em mod n)
        print(x1 == self.r)
```

Função para realizar a chamada dos metodos atribuidos na classe DSAEC

[215]:
```
def ex4():
    # inicializar o objecto
    a = DSAEC()
    # escolher uma curva
    a.curva_p224()
    # gerar o par de chaves
    a.generate_key()
    # mensagem aleatória
    m = os.urandom(16)
    # assinatura
    a.sign(m)
    # verificação
    a.verify(m)
```

[216]:
```
ex4()
```

```
True
```