t1.1

April 5, 2021

1 T1

A85272	Jorge Mota
A83840	Maria Silva

1.1 2

1.1.1 KEM- RSA

Com o auxilio do módulo RSA de python as chaves rsa são geradas com o método `rsa.newkeys` A cifragem e decifragem do inteiro m é feita com o método `pow` sobre os parametros das chaves. O valor de m é gerado aleatóriamente entre o intervalo]1,n[

[75]:
```
import rsa
import hashlib

def big_int_to_bytes(x):
        return (x).to_bytes((x.bit_length() + 7) // 8, 'big')


def bytes_to_big_int(b):
    return int.from_bytes(b, 'big')


def kdf(m):
    h = hashlib.sha256() # 256 for 256 bit long aes keys
    h.update(big_int_to_bytes(m))
    return h.digest()


class KEM_RSA_Error(Exception):
    pass

class KEM_RSA:

    ####### Constants #######

    FULL = 0
```

```python
    PUBLIC_KEY = 1

    ####### Constructors #######

    def __init__(self,N=512):

        if not (N % 128 == 0 and N < 4096):
            raise KEM_RSA("Invalid RSA key size")
        # Nota: este tamanho é so para ser mais rapida a demonstração
        (self.public_key, self.private_key) = rsa.newkeys(N, poolsize=4) #␣
↪4096, poolsize=8)
        self.INITIALIZATION = KEM_RSA.FULL

    # Uma maneira improvisada e nao recomendada de se implementar este método
    # mas python ainda nao fornece multiplos construtores infelizmente
    @classmethod
    def from_public_key(cls,N,pub_key):
        instance = cls(N)
        instance.public_key = pub_key

        instance.INITIALIZATION = KEM_RSA.PUBLIC_KEY
        return instance

    def gen_m(self):
        m = randint(2,self.public_key.n-1)
        return m

    def hide(self,m):
        return pow(m,self.public_key.e,self.public_key.n)

    def recover(self,c):
        if self.INITIALIZATION == KEM_RSA.FULL:
            m = pow(c,self.private_key.d,self.private_key.n)
            return kdf(m)
        raise KEM_RSA_Error("KEM_RSA instance only with public key, recover not␣
↪avaiable")


print("Exemplo 1")
kem = KEM_RSA(512)
m = kem.gen_m()
print("my key:",kdf(m))
c = kem.hide(m)
# Gonna send c to the other side
print("its key:",kem.recover(c))
```

```python
print("Exemplo 2")
# Agent 1
kem1 = KEM_RSA(512)
pub = kem1.public_key
# Agent 2
# Este agente manda ao outro a sua chave publica
kem2 = KEM_RSA.from_public_key(512,pub)
m = kem2.gen_m()
print("my key:",kdf(m))
c = kem2.hide(m)
print("sent c:",c)
# Agent 1
# Agora o primeiro agente recebe o que foi encapsulado:
print("its key:",kem1.recover(c))
```

```
Exemplo 1
my key: b'\xde$3\x12\xddL\xbal\x8d\xacxQp\xd0\x90\x0b\x8e\xe9[,<:\x84\x94\xbd\xf
3e<\xdb\xd5\xb4\x88'
its key: b'\xde$3\x12\xddL\xbal\x8d\xacxQp\xd0\x90\x0b\x8e\xe9[,<:\x84\x94\xbd\x
f3e<\xdb\xd5\xb4\x88'
Exemplo 2
my key: b"\xf9\xb4jPS\xf6H\x89\xde\xfa\xf8V<sP\x02\xaa\xa0\xbe\x7fsS\xe1\x96\xbe
\x05-S\x98\x17'\xec"
sent c: 897072247264821452732020174983274276662201522100552889576464731351496190
705135583550156201121632327802979615731681756922177836066569979237271781671779034
4
its key: b"\xf9\xb4jPS\xf6H\x89\xde\xfa\xf8V<sP\x02\xaa\xa0\xbe\x7fsS\xe1\x96\xb
e\x05-S\x98\x17'\xec"
```

1.1.2 KEM- RSA Fujisaki-Okamoto

1.1.3 DSA

Implementação do *Digital Signature Algorithm* sobre uma classe instanciável por um construtor com a geração de chaves e parametros, ou por uma instanciação pela chave publica (e os parametros necessários)

[51]:
```python
import hashlib

def digest(msg):
    msg = msg.encode("utf-8")
    return Integer('0x' + hashlib.sha1(msg).hexdigest())

class DSA_Error(Exception):
    pass

class DSA:
```

```python
    ####### Constants #######

    FULL = 0
    PUBLIC_KEY = 1

    ####### Constructors #######

    def __init__(self,L,N):

        # FIPS 186-4 Possible L,N combinations (source: wikipedia):
        _LN_COMBINATIONS = [(1024, 160), (2048, 224), (2048, 256), (3072, 256)]
        if not ((L,N) in _LN_COMBINATIONS):
            raise DSA_Error("Invalid key length pair")

        ########### Parameters ###########

        # Choose an N-bit prime q
        #self.q = random_prime(2 ^ N)
        self.q = 1193447034984784682329306571139467195163334221569
        # Choose an L-bit prime p such that p-1 is a multiple of q
        self.p = 
↪8988465674311579674242971140576336446017715169278342980088465244931097926375225352934919545
        #self.p = random_prime(2 ^ N) # TODO: Wrong
        # Choose an integer h randomly from {2...p-2}
        h = randint(2, self.p-2)
        # Compute g = h ^ ((p-1)/p) mod p
        self.g = mod(h ^ ((self.p-1) // self.p), self.p)

        ###### Public & Private Key ######

        # Choose an integer x randomly from {1...q-1}
        self.x = randint(1,self.q-1)
        # Compute y = g ^ x mod p
        self.y = self.g ^ self.x % self.p

        self.INITIALIZATION = DSA.FULL

    @classmethod
    def from_public_key(cls,LN,pqgy):
        instance = cls(LN[0],LN[1])
        instance.p = pqgy[0]
        instance.q = pqgy[1]
        instance.g = pqgy[2]
        instance.y = pqgy[3]
        instance.INITIALIZATION = DSA.PUBLIC_KEY
```

```
    ######## Getters ########

    def parameters(self):
        return (self.p,self.q,self.g)

    def public_key(self):
        return self.y

    ##### Sign & Verify #####

    def sign(self,m):
        if self.INITIALIZATION == DSA.FULL:
            k = randint(1, self.q-1)
            r = 0
            while r == 0:
                r = mod(pow(self.g,k,self.p), self.q)
                s = mod(((digest(m) + self.x*r) // k), self.q)
                k = randint(1, self.q-1)
            return (r,s)
        #else:
        raise DSA_Error("DSA instance only with public key, signing not␣
↪avaiable")

    def verify(self,m,rs):
        r = rs[0]
        s = rs[1]
        if (0 < Integer(r) < Integer(self.q)) and (0 < Integer(s) <␣
↪Integer(self.q)):
            w = mod(1 / s ,self.q)
            u1 = mod(digest(m) * w, self.q)
            u2 = mod(r * w, self.q)
            v = mod((self.g ^ u1) * (self.g ^ u2), self.q)
            return v == r
        return False

dsa = DSA(1024,160)
m = "hello cruel world"
rs = dsa.sign(m)
print(dsa.verify(m,rs))
```

```
True
```

1.1.4 ECDSA

Implementação do *Digital Signature Algorithm* com Curvas Elípticas sobre uma classe instanciável por apenas um construtor, usando uma das curvas especificadas em: NIST FIPS 186 4

[29]:

```python
import hashlib

def digest(msg):
    msg = msg.encode("utf-8")
    return Integer('0x' + hashlib.sha1(msg).hexdigest())


class ECDSA:
    ####### Constructors #######

    def __init__(self):
        # link auxiliar: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.
 ↪pdf
        # Curva e parameterização P192
        self.F = FiniteField(2**192 - 2**64 - 1)
        b  = 0x64210519E59C80E70FA7E9AB72243049FEB8DEECC146B9B1
        E  = EllipticCurve(self.F, [-3, b])
        self.G =␣
 ↪E((0x188DA80EB03090F67CBF20EB43A18800F4FF0AFD82FF1012,0x07192B95FFC8DA78631011ED6B24CDD573F
        # order n
        self.n  = 0xFFFFFFFFFFFFFFFFFFFFFFFF99DEF836146BC9B1B4D22831
        self.Fn = FiniteField(self.n)

        self.d = randint(1, self.n - 1)
        self.Q = self.d * self.G

    ######## Getters ########

    def public_key(self):
        return self.Q

    def sign(self, m):
        r = 0
        s = 0
        while s == 0:
            k = 1
            while r == 0:
                k = randint(1, self.n - 1)
                n_Q = k * self.G
                (x1, y1) = n_Q.xy()
                r = self.Fn(x1)
            kk = self.Fn(k)
            e = digest(m)
            s = kk ^ (-1) * (e + self.d * r)
        return [r, s]

    ##### Sign & Verify #####
```

```python
    def verify(self, m, rs):
        r = rs[0]
        s = rs[1]
        e = digest(m)
        w = s ^ (-1)
        u1 = (e * w)
        u2 = (r * w)
        P1 = Integer(u1) * self.G
        P2 = Integer(u2) * self.Q
        X = P1 + P2
        (x, y) = X.xy()
        v = self.Fn(x)
        return v == r


ecdsa = ECDSA()
m = "hello cruel world"
rs = ecdsa.sign(m)
print(ecdsa.verify(m, rs))
```

```
True
```

[ ]:

```python

```