Universidade do Minho

Mestrado Integrado em Engenharia Informática - Criptografia e Segurança da Informação

Estruturas Criptográficas

Relatório do Trabalho Prático 2 - Grupo 9

Carla Cruz

A80564

Adriana Meireles

A82582

April 21, 2020

Ex1-TP2

April 21, 2020

1 TP2 : Implementação do RSA - OAEP

1.1 Função auxiliar

Foi utilizada uma função auxiliar para gerar um número primo dado o seu tamanho, retornando o primo com o respetivo tamanho indicado.

[1]:
```
def rprime(l):
    """
    Gera um número primo
    Argumentos:
        - l - tamanho do primo
    Valor de retorno: primo com o tamanho indicado
    """
    return random_prime(2**l-1,True,2**(l-1))
```

1.2 Definição do problema

1.2.1 Alinea a)

Para a parte inicial da primeira parte, tinhamos como objetivo implementar um esquema RSA - OAEP em que recebe como parâmetro de entrada um número de bits do módulo que irá ser utilizado para gerar as chaves pública e privada respetivamente.

1.2.2 Alinea b)

OAEP - Optimal Asymmetric Encryption Padding - é um esquema de preenchimento frequentemente usado em conjunto com a criptografia RSA. Este algoritmo usa um par de oráculos aleatórios G e H para processar o texto sem formatação antes da criptografia assimétrica.

OAEP satisfaz os seguintes objetivos:

- Adiciona um elemento aleatório que possa ser usado para converter um esquema criptográfico determinístico (por exemplo, RSA tradicional) num esquema criptográfico probabilístico.
- Evita a descriptografia parcial de textos cifrados (ou outro vazamento de informações), garantindo que um adversário não pode recuperar nenhuma parte do texto simples.

1.2.3 Alinea c)

Fujisaki-Okamoto apresentaram uma abordagem para transformar PKE's que fossem IND-CCA seguros noutros PKE's que fossem IND-CCA seguros. Este processo aumenta muito a complexidade espacial pois são criados criptogramas maiores para o mesmo parâmetro.

Esta transformação FOT segue o princípio de separar a geração e aleatoriedade do núcleo determinístico do algoritmo. O algoritmo determinístico é o porquê deste IND-CCA ser seguro.

É possível dizer que se a 1ª versão do PKE for IND-CPA seguro e se o OWN ("One Way Compressor") for um hash seguro, então a 2ª versão do PKE é um IND-CCA seguro. Desta forma, assumindo que tanto o KEM como o OWN são seguros, podemos escolher entre ter um PKE mais eficiente mas com segurança mais fraca (IND-CPA) ou ter um PKE menos eficiente mas mais seguro (IND-CCA).

[2]:
```python
import random
import string
import hashlib

class RSA_OAEP:
    def __init__(self,nbits):
        self.nbits=int(nbits)

    def keygen(self):
        p=rprime(int(self.nbits/2) +1)
        q=rprime(int(self.nbits/2))
        while p<=2*q:
            p=rprime(int(self.nbits/2) +1)
            q=rprime(int(self.nbits/2))
        n=p*q #parâmetro n
        phin=(p-1)*(q-1) #Cálculo de phi de n para primos
        e=randint(2,phin)
        while gcd(phin,e)!=1:
            e=randint(2,phin) #e tem de ser tal que exista inverso módulo phi␣
↪de n
        d=power_mod(e,-1,phin) #inverso de e
        PubKey=(n,e)
        PrivKey=d
        return PubKey,PrivKey

    def xor(self, data, mask):
        masked = b''
        ldata = len(data)
        lmask = len(mask)
        for i in range(max(ldata, lmask)):
            if i < ldata and i < lmask:
                masked += (data[i] ^^ mask[i]).to_bytes(1, byteorder='big')
            elif i < ldata:
                masked += data[i].to_bytes(1, byteorder='big')
```

```
            else:
                break
        return masked

    def encrypt(self, message, public_k, label):
        l= label.encode('utf-8')
        a =  hashlib.sha256(l).hexdigest()
        nm = str(a+message)
        nm = nm.encode('utf-8')
        k = int(hashlib.sha256(nm).hexdigest(),16)
        (n,e) = public_k
        enc = power_mod(k , e, n)
        x = k.to_bytes(len(nm), byteorder='big')
        encf = self.xor(x,nm)
        return (enc,encf)

    def decrypt(self, ctxt, private_k, public_k, label):
        enc, encf = ctxt
        (n,e) = public_k
        k = power_mod(enc, private_k, n)
        m = int (k).to_bytes(len(encf), byteorder='big')
        txt = self.xor (m, encf)
        l= label.encode('utf-8')
        size = len (hashlib.sha256(l).hexdigest())
        a, message = txt[:size], txt[size:]
        msg = message.decode('ascii')
        if ctxt == self.encrypt(msg, public_k, label):
            return message.decode('ascii')
        else: return false
```

[3]:
```
rsa_oaep = RSA_OAEP(512)
Pub,Priv=rsa_oaep.keygen()
msg = "Ola mundo"
label= "EstCripto"
print ("Mensagem:", msg)

#para cifrar uma mensagem usa-se a chave pública do RSA
ctxt = rsa_oaep.encrypt(msg,Pub,label)
print ("CipherText:", ctxt)

txt = rsa_oaep.decrypt(ctxt,Priv, Pub, label)
print ("Message:", txt)
```

```
Mensagem: Ola mundo
CipherText: (16437189225653521043834031068658980919832617048640424240412121967202
3878446082647855700726454382105586735366610412506093636127639201164473712856321
5093572, b'898f73a20b73f9963b6f7fde394930c3fd8657fbd#\xc121\x05m\x9f\xcd\xf0\xe3
```

```
\x83\x9c_\xd5\xcc\x9e\x0etz.H1\xb428\xa0\xd2\xc5X\x86\xd7S')
Message: Ola mundo
```

Ex2-TP2

April 21, 2020

1 TP2 : Construir uma classe Python que implemente o DSA

1.1 Função auxiliar

Foi utilizada uma função auxiliar para gerar um número primo dado o seu tamanho, retornando o primo com o respetivo tamanho indicado.

[106]:
```
def rprime(l):
        return random_prime(2^l-1,True,2^(l-1))
```

1.2 Definição do problema

1.2.1 Alinea a)

Os primos **p** e **q** são passados como parâmetros como é pedido e para a sua geração começamos por gerar **q** aleatoriamente e,posteriormente, calcular para sucessivos valores de "t" o valor *p = 1 + q « t* até que p seja primo e tenha o tamanho certo.

1.2.2 Alinea b)

Antes de se proceder à implementação das funçoes para assinar e verificar a assinatura foi necessário guardar as variáveis que seriam utilizadas:

- **x** : *Private Key* (inteiro) gerada aleatoriamente entre [1,q]
- **y** : *Public Key* gerada a partir da private key (g^x % p)
- **g** : é o nosso gerador para as chaves
- **(r,sig)** : Assinatura do documento (inteiro,inteiro)

A primeira variável é utilizada para assinar a mensagem, sendo o resultado retornado na forma (r,sig). As variáveis y,g,p e q são utilizadas na verificação da assinatura.

Para o algoritmo de assinar e verificar baseamo-nos num documento encontrado na internet.

[107]:
```
from random import randint
from fractions import gcd
from sage.rings.all import Integer


class DSA:
    def __init__(self, pbit=1024, qbit=160):
        q = rprime(qbit)
```

```
        t=1
        p = 1+2^t * q
        while not is_prime(p) and p.nbits() < pbit: #enquanto nao é primo nem
↪tem o tamanho pretendido
            t+=1
            p = 1+2^t * q

        P = Integers(p) #anel de inteiro modulo p
        Q = Integers(q) #anel de inteiro modulo q

        for x in Primes(): #testar a primalidade de x
            a = P(x)
            if a.multiplicative_order() == p-1:
                break

        g = pow(a,2**t)#gerador

        x = Q.random_element()#chave privada
        y = pow(g,x,p)#chave publica

        self.sign_k = x
        self.verify_k = (y,g,p,q)

    def sign_key(self):
        return self.sign_k

    def verify_key(self):
        return self.verify_k

    def sign(self,key,msg) :
        (_,g,p,q) = self.verify_k
        P = Integers(p)
        Q = Integers(q)
        r=0
        while r==0: #enquanto o r for 0 calcular k novamente
            k = Q.random_element()#nonce
            r = Q(P(g^k))
        Z = IntegerRing() #retorna o anel inteiro
        m = Z(msg.encode("hex"), base=16) #hash
        sig = Q((1/k) *(m + key * r))
        return r,sig

    def verify(self,verkey,signature,msg) :
        (y,g,p,q) = verkey
        P = Integers(p)
        Q = Integers(q)
        r,sig = signature
```

```
        w = Q(1/sig)
        Z = IntegerRing()
        m = Z(msg.encode("hex"), base=16) #hash
        u1 = Q(m * w)
        u2 = Q(r*w)
        v = Q(P((g^u1) * (y^u2)))
        if v != Q(r):
            raise ValueError("Assinatura Invalida")
```

[108]:
```
dsa = DSA()
msg = "Esta mensagem vai ser assinada e verificada com sucesso"
print "Mensagem:", msg

sign_key = dsa.sign_key()
verify_key = dsa.verify_key()

signature = dsa.sign(sign_key,msg)
print "Signature:", signature

msg = dsa.verify(verify_key,signature, msg)
print "Verificação Bem Sucedida"
```

```
Mensagem: Esta mensagem vai ser assinada e verificada com sucesso
Signature: (434973058844166455056698842466393492032303506118,
327697310434326153499622947746244301059307580694)
Verificação Bem Sucedida
```

[ ]:
```

```

Ex3-TP2

April 21, 2020

1 TP2 : Construir uma classe Python que implemente o ECDSA usando uma das curvas elípticas primas definidas no FIPS186-4

A alinea 3 do segundo trabalho prático pretende que se implemente um mecanismo de assinatura digital, recorrendo para isso às curvas elíticas. São oferecidas curvas para criptografia de 192, 224, 256, 384 e 521 bits definida no **FIPS186-4** cuja forma é E : y^2 = x^3 - 3x. Destas a que escolhemos foi a *P-224*. Depois de recolhermos a definição das mesmas, procedemos à implementação do algoritmo com base em documentos encontrados na internet.

De modo a facilitar a compreensão do algoritmo implementado, é explicado o significado das variáveis utilizadas:

- **s** : *Secret Key* (inteiro) gerada aleatoriamente
- **V** : A nossa *Public Key* (ponto da curva) obtida da multiplicação da *secret key* com G
- **G** : Ponto pertecente à nossa curva elitica (constituido por gx e gy)
- **p** : Primo que define o corpo finito da curva elitica
- **n** : Ordem do primo
- **e** : Chave (inteiro) gerada aleatoriamente e de forma segura
- **(s1,s2)** : Assinatura do documento (inteiro,inteiro)

[50]:
```
#tabelamento da curva P-224
NIST = dict()
NIST['P-224'] = {
    'p': 26959946667150639794667015087019630673557916260026308143510066298881,
    'n': 26959946667150639794667015087019625940457807714424391721682722368061,
    'seed': 'bd71344799d5c7fcdc45b59fa3b9ab8f6a948bc5',
    'c': '5b056c7e11dd68f40469ee7f3c7a7d74f7d121116506d031218291fb',
    'b': 'b4050a850c04b3abf54132565044b0b7d7bfd8ba270b39432355ffb4',
    'Gx': 'b70e0cbd6bb4bf7f321390b94a03c1d356c21122343280d6115c1d21',
    'Gy': 'bd376388b5f723fb4c22dfe6cd4375a05a07476444d5819985007e34'
}

class ECDSA:

    def __init__(self):
        c = NIST['P-224']
        p  = c['p']
        n  = c['n']
```

```
        b  = ZZ(c['b'],16)
        Gx = ZZ(c['Gx'],16)
        Gy = ZZ(c['Gy'],16)
        E = EllipticCurve(GF(p),[-3,b]) #curva eliptica
        self.G = E((Gx,Gy)) #Ponto pertecente à nossa curva elitica␣
↪(constituido por gx e gy)
        self.s = ZZ.random_element(1,p-1) #selecionar numero aleatorio entre 1␣
↪e p-1 -secret key
        self.V= self.s*self.G

    def sign_key(self):
        return self.s

    def verify_key(self):
        return self.V, self.G


    def sign(self,key,msg):
        Q = GF(self.G.order())

        #hash
        Z = IntegerRing()
        m = Z(msg.encode("hex"), base=16)
        d = Q(m)

        e = Q.random_element()
        (x,y,z) = int(e) * self.G #calcular o ponto na curva E:(x,y) = k*G
        s1 = Q(x) #r= x (mod n)
        s2 = Q((d + key * s1)* 1/e)#s= k-1(z+rd)(mod n)
        return s1,s2

    def verify(self, verify_key, signature, msg):
        V, G = verify_key
        s1, s2 = signature
        Q = GF(G.order())

        Z = IntegerRing()
        m = Z(msg.encode("hex"), base=16)
        d = Q(m)

        v1 = Q(d * 1/s2)
        v2 = Q(s1 * 1/s2)
        (x,y,z) = (int(v1)*self.G) + (int(v2)*V)
        if Q(x)!=s1:
            raise ValueError("Assinatura Invalida")
```

[51]:
```python
ecdsa = ECDSA()
msg = "Esta mensagem vai ser assinada e verificada com sucesso"
print "Mensagem:", msg

sign_key = ecdsa.sign_key()
verify_key = ecdsa.verify_key()

signature = ecdsa.sign(sign_key,msg)
print "Signature:", signature

msg = ecdsa.verify(verify_key, signature, msg)
print "Verificação Bem Sucedida"

print "-----------------------"
```

```
Mensagem: Esta mensagem vai ser assinada e verificada com sucesso
Signature:
(12652478553964903363027975263199374617819263728858452934699004753604,
14506590897245168958621185621731102383861829011454942948399060541106)
Verificação Bem Sucedida
-----------------------
```

[ ]:
```python

```