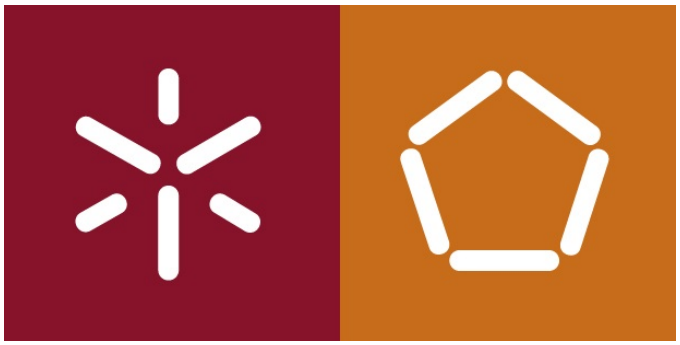
Universidade do Minho

Mestrado Integrado em Engenharia Informática -
Criptografia e Segurança da Informação



Estruturas Criptográficas

Relatório do Trabalho Prático 2

Grupo 1

Pedro Freitas
A80975

André Gonçalves
A80368

April 21, 2020

Contents

1 Exercício1

Neste primeiro exercício tínhamos de implementar uma esquema KEM- RSA-OAEP:

- Inicializar cada instância recebendo como parâmetro obrigatório o parâmetro de segurança (tamanho em bits do módulo RSA-OAEP) e gerando as chaves pública e privada.
- Conter funções para encapsulamento e revelação da chave gerada.
- Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.

Infelizmente não conseguimos implementar todas as funcionalidades.

Neste momento temos implementado um KEM usando encriptação OAEP.

Depois fizemos a transformação disso num IND-CPA seguro, porém não conseguimos transformar em IND-CCA seguro.

Exercício 1 - Esquema KEM- RSA-OAEP

Neste exercício temos de implementar um esquema KEM- RSA-OAEP que deve :

- **Inicializar cada instância recebendo como parâmetro obrigatório o parâmetro de segurança (tamanho em bits do módulo RSA-OAEP) e gerando as chaves pública e privada**
- **Conter funções para encapsulamento e revelação da chave gerada.**
- **Construir, a partir deste KEM e usando a transformação de Fujisaki-Okamoto, um PKE que seja IND-CCA seguro.**

Ao longo deste documento vamos explicar o algoritmo de cada um dos processos acima expostos.

Gerar Parâmetros

Neste processo vamos gerar os parâmetros essenciais que são a base do algoritmo do RSA:

```
- 2 números primos p e q;
- n é o módulo para a chave pública e a chave privada;
- phi é co-prime com n;
```

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

In [2]:
```
l = 1024
q = rprime(l)
p = rprime(l+1)


N = p * q
phi = (p-1)*(q-1)
```

Aplicação OAEP

In [3]:
```
G = IntegerModRing(phi)
R = IntegerModRing(N)


def generateKeys():
    e = G(rprime(512)) #public exponent
    s = 1/e #private exponent
    return (e,s)

e,s = generateKeys()
```

In [4]:
```
def OAEP(pk,m): ##OAEP encrypt
    a = R(m)
    cm = a**pk
    return cm


def OAEPinv(sk,cm): ##OAEP decrypt
    b=R(cm)
    dm = b**sk
    return dm
```

Funções auxiliares para as classes KEM e FOT

In [5]:
```
def generateRandomString(size): # gera uma string de tamanho variáv
el de 0 e 1 aleatórios.
    i = 0
    stream = ""
    while(i<size):
        j = randint(0,1)
        stream = stream + str(j)
        i+=1
    return stream

def generateZeroString(size): # gera uma string de tamanho variável
de zeros.
    i = 0
    stream = ""
    while(i<size):
        stream = stream + str(0)
        i+=1
    return stream

xor = lambda x, y: x.__xor__(y)

def concat(i,j):
    return i +j
```

KEM

Esta classe contém funções para o encapsulamento e revelação da chave gerada.

In [6]:
```
class KEM:
    def encrypt(self,pk,x,n):
        return power_mod(x,ZZ(pk),n)

    def decrypt(self,sk,enc,n):
        return power_mod(enc,ZZ(sk),n)

    def encapsulation(self,pk): #enc
        x = randint(1,N-1)
        #enc = self.encrypt(pk,x,N)
        enc = OAEP(pk,x)
        k = hash(x)
        return (k,enc)

    def reveal(self,sk,enc):
        #x = self.decrypt(sk,enc,N)
        x = OAEPinv(sk,enc)
        k = hash(x)
        return k

    def enc(self,pk):
        a = generateRandomString(l)
        zero = generateZeroString(l)
        (k,enc) = self.encapsulation1(pk,concat(a,zero))
        return (k,enc)

    def encapsulation1(self,pk,a0):
        enc = OAEP(pk,int(a0))
        print('encapsulation1- enc: ' + str(enc))
        k = hash(enc)
        print('encapsulation1- k: ' + str(k))
        return (k,enc)
```

In [7]:
```
kem = KEM()
(k,enc) = kem.encapsulation(e)
print('k: ' + str(k))
print('enc: ' + str(enc))

k1 = kem.reveal(s,enc)
print('k1: ' + str(k1))

teste = kem.enc(e)
print('teste: ' +str(teste))
```

```
k: 512998086484580947
enc: 271537983504884533935799470973946660199515510574792841540322112468406127415940111780264333346779288254565215776785050578840777786241814877996172269434690515778344089167141585043040169558357900920950074747646450116794258152857085043005693744435931887719498947582086346921089303243571433068794456845670162207943952806629299054533050383340056828169196058104965846085933385709331080696851948355063619594675492099873546616328160433720263385829476176332177615849328343824864770931784156054384151175805014205068877418475697859569420338268744407967556639652589830753705884706468944114249197473801780349822216412040150943814841
k1: 512998086484580947
encapsulation1- enc: 952308298281991089891020960425936882524619816682750296253597561896296168514547370018716357952242973582711682975702154199089587915772486529762574414826241607267983632514656042758039223371968006720002055075281640574170984656186928317621196249190013131037227026451616421564619642001001817812856947897407423862461372260959877974031614676766139784377006363486238259010067049797363261483229835547250928358840418415224861022768797437544069319989910115470990972729136015892114667811024409789326411591091891086897119333037383902843428935816630796043977535582045231441350948739392550469519958685769089523248974285317211388813 5
encapsulation1- k: 1166647538513584938
teste: (1166647538513584938, 952308298281991089891020960425936882524619816682750296253597561896296168514547370018716357952242973582711682975702154199089587915772486529762574414826241607267983632514656042758039223371968006720002055075281640574170984656186928317621196249190013131037227026451616421564619642001001817812856947897407423862461372260959877974031614676766139784377006363486238259010067049797363261483229835547250928358840418415224861022768797437544069319989910115470990972729136015892114667811024409789326411591091891086897119333037383902843428935816630796043977535582045231441350948739392550469519958685769089523248974285317211388813 5)
```

PKE

Esta classe implementa um esquema PKE(public key encryption) a partir do esquema KEM

In [8]:
```
class PKE:
    def __init__(self):
        self.kem = KEM()

    def encrypt(self,pk,m):
        (k,enc) = self.kem.encapsulation(pk)
        c = xor(m,k)
        return (enc,c)

    def decrypt(self,sk,c):
        (enc,m1) = c
        k = self.kem.reveal(sk,enc)
        m = xor(m1,k)
        return m
```

FOT

Esta classe contém funções para implementar a transformação de Fujisaki-Okamoto que transforma PKE's que possuem IND-CPA seguro em outros PKE's que possuem IND-CCA seguros

In [9]:
```
class FOT:
    def __init__(self):
        self.kem = KEM()

    def encrypt(self,pk,m):
        a = generateRandomString(l)
        (enc,k) = self.encrypt1(pk,m,a)
        return (enc,k)

    def encrypt1(self,pk,a,m):
        (enc,k) = kem.encapsulation1(pk,concat(a,hash(m)))
        #print('encrypt1 - enc: ' + str(enc))
        #print('encrypt1 - k: ' + str(k))
        aux1 = concat(str(a),str(m))
        #print('encrypt1 - aux1: ' + str(aux1))
        aux2 = xor(int(aux1),int(k))
        #print('encrypt1 - aux2: ' + str(aux2))
        return (enc,aux2)

    def decrypt(self,sk,c):
        (enc,m1) = c
        k = kem.reveal(sk,enc)
        print('decrypt- k: ' + str(k))
        am = xor(m1,k)
        a = am[0:l]
        m = am[l:]
        if(c == encrypt1(pk,a,m)):
            return m
        else:
            return false
```

In [ ]:
```
pke = PKE()

fot = FOT()
pk = e
sk = s

m = 123
c = pke.encrypt(pk,m)
(enc,t) = c
print('t: ' + str(t))
m1 = pke.decrypt(sk,c)
print('m1: ' + str(m1))

c = fot.encrypt(pk,m)
enc,k = c
print('k: '+ str(k))
c2 = fot.decrypt(sk,c)
print(c2)
```

Nota

De realçar que tivemos problemas na implementação de um PKE seja um IND-CCA seguro.Mesmo assim, colocámos o código da nossa tentativa.

2 Exercício2

Neste exercício tínhamos de implementar um DSA. Além de implementarmos um *Esquema DSA* tentamos também implementar um *Esquema de Assinaturas El Gamal*.

2.1 Digital Signature Algorithm

Exercício 2 - DSA

Neste exercício temos de implementar um algoritmo de DSA.

Esta implementação pode-se dividir em 4 processos distintos:

- **Gerar parâmetros**
- **Criação das chaves**
- **Assinatura de uma mensagem**
- **Verificação da assinatura de uma mensagem**

Ao longo deste documento vamos explicar o algoritmo de cada um dos processos acima expostos.

Gerar Parâmetros

Neste processo vamos gerar os 3 parâmetros essenciais que são a base deste algoritmo de DSA: `p`, `q`, e `g`.

Parâmetro q:

- **q** - Um número primo de N-bits

In [1]:
```
def generateQ(N):
    q = random_prime(2^(N-1))
    print("q: "+ str(q))
    return q
```

Parâmetro p:

- **p:**
  - Número primo de L-bits
  - `p-1` é múltiplo de `q`

In [2]:
```
def generateP(L,q):
    while(true):
        p = random_prime(2^(L-1))
        if(mod(ZZ(p-1),q) == 0):
            break
        #print('next')
    print("p: "+ str(p))
    return p
```

Parâmetro g :

- **g**
  - Temos de computar: `h^((p-1)/q) mod p`

Como vemos na expressão em cima é necessário um parâmetro auxiliar: `h` .

De realçar que se com um determinado valor de `h` , a computação de `g` for igual a 1, devemos repetir esta computação, mas assumindo `h=2` .

Parâmetro h :

- **h**
  - Inteiro compreendido entre os valores: 1 < h < p-1

In [3]:
```
def generateH(p):
    h = randint(2,p-2)
    print("h: "+ str(h))
    return h
```

In [4]:
```
def generateG(p,q):
    h = generateH(p)
    exph = (p-1)/q
    g = power_mod(h,ZZ(exph),p)
    if (g == 1):
        print('g = 1 ...')
        h = 2
        g = pow(h,aux,p)
    print('g: ' + str(g))
    return g
```

Gerador de parâmetros:

`generateParameters()` é a função é responsável por gerar os parâmetros:

- **p**
- **q**
- **g**

In [5]:
```
def generateParameters(L,N):
    q = generateQ(16)
    p = generateP(128,q)
    g = generateG(p,q)
    parameters = (p,q,g)
    return parameters
```

In [6]:
```
parameters = generateParameters(2048,256)
```

```
q: 22279
p: 144398932945256214146235797358038236531
h: 681522073872587233592443799031021352555
g: 79311238626828824131157619611774038295
```

Criação das chaves

Agora que temos os parâmetros necessários, podemos passar para o processo de criação das chaves. Assim na função seguinte vamos gerar dois valores:

- **x**
  - Inteiro aleatório compreendido no intervalo: 0 < x < q
  - Corresponde à **Chave Privada**
- **y**
  - Computar `g^x mod p`
  - Corresponde à **Chave Pública**

In [7]:
```
def generateKeys(parameters):
    (p,q,g) = parameters
    #x
    x = randint(1,q-1)
    privateKey = x
    print('Private Key: ' + str(privateKey))
    y = power_mod(g,privateKey,p)
    publicKey = y
    print('Public Key: ' + str(publicKey))
    return privateKey, publicKey
```

In [8]:
```
privateKey, publicKey = generateKeys(parameters)
```

```
Private Key: 3203
Public Key: 19747773144800009831469514182127128013
```

Assinatura de uma mensagem

Com os parâmetros gerados e criadas as chaves, podemos passar à produção de assinaturas de qualquer mensagem `m`. Este processo pode ser dividido em 3 processos mais pequenos:

- **Calcular r** - `(g^k mod p) mod q`. Se `r==0`, gerar outro `k` e repetir este processo
- **Calcular s** - `[ k^(-1) * ( hash(m) + x*r ) ] mod q`
- **Construir assinatura** - Par: `(r,s)`

Calcular r

Este cálculo necessita de um valor auxiliar `k`.

Como foi referido em cima `r` tem de ser diferente de `0`. Por isso no raro caso de isso acontecer, será necessário gerar outro `k` e recalcular `r`.

Calcular k

- **k** - Inteiro aleatório compreendido no intervalo: 0 < k < q

In [9]:
```
def generateK(q):
    k = randint(1,q-1)
    print('k: ' + str(k))
    return k
```

In [10]:
```
def calculateR(parameters,k):
    (p,q,g) = parameters
    r = mod(power_mod(g,k,p),q)
    if (r == 0):
        print('r = 0')
        generateK(q)
        r = calculateR(parameters,k)
    print('r: ' + str(r))
    return r
```

Calcular s

Para calcular `s` precisamos do `k` em cima gerado também.

Para processar a expressão `[ k^(-1) * ( hash(m) + x*r ) ] mod q` vamos dividir em tarefas mais pequenas, devido ao grande esforço computacional que requerem.

Primeiro vamos computar: `k^(-1) mod q` e só depois multiplicar esse valor por `(hash(m) + x*r)`

In [11]:
```
def calculateS(parameters,privateKey,k,m,r):
    (p,q,g) = parameters
    hm = hash(m)
    #print('hm: ' + str(hm))
    s1 = power_mod(k,-1,q)
    xr = privateKey*r
    s2 = hm + xr
    s = s1*s2
    print('s: ' + str(s))
    return s
```

Construir assinatura

Este processo é bastante simples na medida que apenas pega nos valores de `r` e `s` e junta-os como um par: `signature = (r,s)`

Na função em baixo, `signMessage` vamos executar os 3 processos descritos em cima e vamos devolver a assinatura.

In [12]:
```
def signMessage(parameters,privateKey,m):
    (p,q,g) = parameters
    k = generateK(q)
    r = calculateR(parameters,k)
    s = calculateS(parameters,privateKey,k,m,r)
    signature = (r,s)
    print('signature: ' + str(signature))
    return signature
```

In [13]:
```
m = b'teste12'
signature = signMessage(parameters,privateKey,m)
```

```
k: 8868
r: 12542
s: 18608
signature: (12542, 18608)
```

Verificação da assinatura de uma mensagem

Esta operação diz respeito à validação de uma assinatura a uma dada mensagem.

Este processo de verificação também pode ser dividido em algumas etapas:

- **Tamanho de `r` e `s`**
  - 0 < `r` < q
  - 0 < `s` < q
- **Cálculo de w** - `s^(-1) mod q`
- **u1** - `hash(m) * w mod q`
- **u2** - `r*w mod q`
- **v** - `( g^(u1) * y^(u2) mod p ) mod q`
- **Verificar** : `v == r`

In [14]:
```
def verifySignature(publicKey,parameters,signature,m):
    (p,q,g) = parameters
    #print('q: ' + str(q))
    (r,s) = signature
    qr = q-r
    #print('qr: ' + str(qr))
    qs = q-s
    #print('qs: ' + str(qs))
    if(r<=0):
        #print('r negativo')
        return 'ERRO R!'
    if(qr <= 0):
        #print('r>=q')
        return 'ERRO R!'
    if(s<=0):
        #print('s negativo')
        return 'ERRO S!'
    if(qs <= 0):
        #print('s>=q')
        return 'ERRO S!'

    w = power_mod(ZZ(s),-1,q)
    #print('w: ' + str(w))

    mw = hash(m) * w
    #print('mw: ' + str(mw))
    u1 = power_mod(mw,1,q)
    #print('u1: ' + str(u1))

    rw = ZZ(r)*w
    #print('rw: ' + str(rw))
    u2 = mod(rw,q)
    #print('u2: ' + str(u2))

    gu1 = g^u1
    pu2 = publicKey^u2
    gu1pu2=gu1*pu2

    v1 = mod(gu1pu2,p)
    #print('v1: ' + str(v1))

    v = mod(v1,q)
    print('v: ' + str(v))

    if(v == r):
        return 'True'
    else:
        return 'False'
```

In [15]:
```
result = verifySignature(publicKey,parameters,signature,m)
print('Resultado da verificação 1: ' + result)
```

```
v: 12542
Resultado da verificação 1: True
```

In [16]:
```
m2 = b'Teste2'
result2 = verifySignature(publicKey,parameters,signature,m2)
print('Resultado da verificação 2: ' + str(result2))
```

```
v: 2555
Resultado da verificação 2: False
```

2.2 Esquema de Assinaturas El Gamal

Exercício 2 - DSA

Neste exercício temos de implementar um algoritmo de DSA.

Como tal existem vários algoritmos disponíveis. Após uma breve conversa entre os dois elementos do grupo achamos por bem implementar uma definição do algoritmos de assinatura de **El Gamal** visto que já foi bastantes vezes referida e falada tanto na unidade curricular anterior (Tecnologias Criptográficas) como nesta mesma unidade curricular. Esta definição encontra-se nos anexos deste relatório.

Parâmetro p:

- **p** - Um número primo de tamanho 2^(tamanho-1)

In [1]:
```
def generateP(tam):
    p = random_prime(2^(tam-1))
    print("p: "+ str(p))
    return p
```

Parâmetro g:

- **g** - Inteiro compreendido em: 0 < g < p , tal que este será um gerador

In [2]:
```
def generateG(p):
    g = randint(1,p-1)
    print("g: "+ str(g))
    return g
```

Gerador de parâmetros:

`generateParameters()` é a função é responsável por gerar os parâmetros:

- **p**
- **g**

In [3]:
```
def generateParameters(tam):
    p = generateP(tam)
    g = generateG(p)
    return p,g
```

Parâmetro x :

- **x** - Inteiro compreendido entre os valores: 1 < x < p-1

Este parâmetro é o segredo do utilizador, que será usado na chave privada e na geração da chave pública.

In [4]:
```
def generateX(p):
    x = randint(2,p-2)
    print('x: ' + str(x))
    return x
```

Chave privada

Esta é a função que recebe os parâmetros e compõe a chave privada.

In [5]:
```
def createPrivateKey(p,g,x):
    privateKey= (p,g,x)
    print('privateKey: ' + str(privateKey))
    return privateKey
```

Chave pública

Esta função é a que recebe os paramatros e compõe a chave pública

In [6]:
```
def createPublicKey(p,g,y):
    publicKey = (p,g,y)
    print('publicKey: ' + str(publicKey))
    return publicKey
```

Gerador de Chave

`generateKey(p,g)` é uma função que cria a chave pública e a chave privada. Começa-se por gerar a parte do segredo da chave privada, sendo invocada a função `generateX(p)` .

Depois a partir desse valor chega-se à parte pública da chave com a operação: `(g^x) mod p` .

Tendo esses dois valores as chaves privada e pública são compostas com as seguintes funções, respetivamente, `createPrivateKey(p,g,x)` e `createPublicKey(p,g,y)`

In [7]:
```
def generateKey(p,g):
    x = generateX(p)

    y =pow(g,x,p)
    print('y: ' + str(y))

    privateKey = createPrivateKey(p,g,x)
    publicKey = createPublicKey(p,g,y)

    return publicKey,privateKey
```

Parâmetro k:

- **k** - Inteiro compreendido entre os valores: 0<k<p-1 e gcd(k,p-1) = 1

In [8]:
```
def generateK(p):
    while(true):
        aux = randint(1,p-2)
        if(gcd(aux,p-1)== 1):
            print('found')
            k = aux
            break
        print('next')
    print("k: "+ str(k))
    return k
```

Assinatura

Esta função é a responsável pela assinatura de uma mensagem, recebendo como argumentos a chave privada e a mensagem.

Esta começa por decompor a chave privada pelos 3 parâmetros que a compõe: `p,g,x` .

Após isso gera-se um inteiro `k` com a função `generateK(p)` .

A partir deste `k` podemos calcular `r` e `l` :

- **r** : `r = g^k mod p`
- **l** : `l = k^-1 mod(p-1)`

Após isso calculamos `re` :

- **re**: `re = l*(hash(m)-r*x)`

Assim podemos calcular `s` :

- **s**: `s = re mod(p-1)`

Após verificarmos se este é positivo, criamos a assinatura com os parâmetros s e r: `signature=(r,s)`

In [9]:
```python
def signMessage(privateKey,m):
    (p,g,x) = privateKey
    k = generateK(p)
    #g^k mod p
    r = power_mod(g,k,p)
    #print('r: ' + str(r))
    l = power_mod(k,-1,p-1)
    #print('l: ' + str(l))
    re = l*(hash(m)-r*x)
    #print('re: ' + str(re))
    s= power_mod(re,1,p-1)
    #print('s: ' + str(s))
    if (s < 0):
        print('s < 0')
        return ''
    signature = (r,s)
    print('signature: ' + str(signature))
    return signature
```

Verificação da assinatura

Esta função recebe como argumentos a chave pública, a mensagem e assinatura e é responsável se a assinatura é válida com aquela chave e com aquela mensagem.

O primeiro passo está em decompor a chave pública pelos três parâmetros que a compõe: `(p,g,y)`. Faremos o mesmo com a assinatura, obtendo agora: `(r,s)`.

A primeira fase da verificação da assinatura passa por verificar o tamanho dos parâmetros que a compõe:

- **r**: 0<r<p
- **s**: 0<s<p-1

Depois temos que fazer o calculo de dois valores e fazer a sua comparação: `val1 e val2`:

- **val1** - Precisa de 4 operações distintas.
  - **cal1** - `y^r`
  - **cal2** - `r^s`
  - **cal** - `cal1*cal2`

    `val1 = cal mod p <=> val1 = y^r * r^s mod`
- **val2** - `g^(hash(m)) mod`

Depois deste cálculo a vericidade da assinatura depende do resultado da comparação destes dois valores:

- **Verdadeiro** se `val1 == val2`
- **Falso** se `val1 != val2`

In [10]:
```python
def verifySignature(publicKey, m, signature):
    (p,g,y) = publicKey
    (r,s) = signature

    if(r<=0 or r >= p):
        print('Erro no tamanho de r')
        return False
    if(s<=0 or s >= p-1):
        print('Erro no tamanho de s')
        return False

    cal1 = y^r
    #print('tenho cal1')
    cal2 = r^s
    #print('tenho cal2')
    cal = cal1*cal2
    #print('tenho cal')
    #print('cal: ' +str(cal))

    val1= power_mod(cal,1,p)
    print('val1: ' + str(val1))
    val2 = power_mod(g,hash(m),p)
    print('val2: ' + str(val2))

    if(val1==val2):
        return True
    else:
        return False
```

Main

Esta função é a responsável por testar todas as funcionalidades implementadas.

Esta começa por gerar os parâmetros iniciais necessários para todo o processo.

Depois gera as chaves (privada e pública).

Assina uma mensagem com a chave privada e depois vai verificar essa mesma assinatura dessa mensagem.

A `main1` verifica com a mesma mensagem.

a `main2` verifica com uma mensagem diferente, provocando o erro propositado.

In [11]:
```
def main1():
    p,g = generateParameters(32)

    publicKey, privateKey = generateKey(p,g)

    m=b"Vamos assinar esta mensagem"
    signature = signMessage(privateKey,m)
    if(signature == ''):
        print('Erro na assinatura da mensagem')

    value = verifySignature(publicKey,m,signature)
    print('value: ' + str(value))
```

In [12]:
```
main1()
```

```
p: 1691291033
g: 1236029196
x: 494772165
y: 125931399
privateKey: (1691291033, 1236029196, 494772165)
publicKey: (1691291033, 1236029196, 125931399)
next
found
k: 554805665
signature: (1656598076, 86408788)
val1: 716857687
val2: 716857687
value: True
```

In [ ]:
```
def main2():
    p,g = generateParameters(32)

    publicKey, privateKey = generateKey(p,g)

    m=123
    signature = signMessage(privateKey,m)
    if(signature == ''):
        print('Erro na assinatura da mensagem')

    m2 =456
    value = verifySignature(publicKey,m2,signature)
    print('value: ' + str(value))
```

In [ ]:
```
main2()
```

3 Exercício3

Neste exercício tínhamos de implementar um ECDSA. Para tal seguimos o documento: https://www.johannes-bauer.com/compsci/ecc/#anchor23 , mais concretamente os capítulos 4.1, 4.2 e 4.4 .

Exercício 3 - ECDSA

Neste exercício vamos implementar um ECDSA segundo uma curva elíptica definidas no **FIPS186-4**. Numa pesquisa encontramos os valores dos parâmetros de várias curvas, tendo optado pela curva **P-384**, para ir de acordo com a curva utilizado no trabalho prático 1.

Parâmetros

- Os parâmetros desta curva são:

In [1]:
```
FIPS = dict()
FIPS['P-384'] = {
    'p': 3940200619639447921227904010014361380507973927046544666794
829340424572177149687032904726608825893800186160697311231 9,
    'n': 3940200619639447921227904010014361380507973927046544666794
690527962765939911326356939895630815229491355443365394264 3,
    'seed': 'a335926aa319a27a1d00896a6773a4827acdac73',
    'c': '79d1e655f868f02fff48dcdee14151ddb80643c1406d0ca10dfe6fc52
009540a495e8042ea5f744f6e184667cc722483',
    'b': 'b3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5
013875ac656398d8a2ed19d2a85c8edd3ec2aef',
    'Gx': 'aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e0
82542a385502f25dbf55296c3a545e3872760ab7',
    'Gy': '3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113
b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f',
}
```

Chaves

Para gerar as chaves vamos primeiro obter o **Ponto Gerador** (**basepoint**) e a partir daí já podemos calcular as chaves.

Ponto gerador

Este ponto é um ponto que se assemelha ao início da curva.

Para calcular precisamos dos parâmetros `p` e `b` tabelados para a partir disso obtermos a curva elítica correspondente. Sabendo que a curva tem o formato: `y^2 = x^3 - 3x + b mod p` a nossa curva pode ser obttida através: `E = EllipticCurve(GF(p),[-3,b])`

In [2]:
```
def generateGeneratorPoint():
    p = FIPS['P-384']['p']
    b = ZZ(FIPS['P-384']['b'],16)

    E = EllipticCurve(GF(p),[-3,b])
    #print(E)

    gx = ZZ(FIPS['P-384']['Gx'], 16)
    gy = ZZ(FIPS['P-384']['Gy'], 16)
    generatorPoint = E((gx,gy))
    print('GeneratorPoint: ' + str(generatorPoint))
    return generatorPoint
```

Chave privada

A chave privada é facilmente obtida, visto que é apenas necessário que seja um valor aleatório entre 0 e `n`, onde `n` representa a ordem do **basepoint**.

In [3]:
```
def generatePrivateKey(n):
    d = ZZ.random_element(1,n-1)
    print('PrivateKey: ' + str(d))
    return d
```

Chave pública

A chave pública é obtida a partir do **basepoint** e da chave privada : `q = d * G`

In [4]:
```
def generatePublicKey(generatorPoint, d):
    q = d * generatorPoint
    print('PublicKey: ' + str(q))
    return q
```

Na função abaixo geramos o **basepoint** e depois a **chave privada** e a respetiva **chave pública**.

In [5]:
```
def generateKeys(n):
    generatorPoint = generateGeneratorPoint()
    d = generatePrivateKey(n)
    q = generatePublicKey(generatorPoint, d)
    return (generatorPoint,d,q)
```

Assinatura de uma mensagem

Para assinar uma mensagem precisamos da **chave privada**, do **basepoint**, do `n` e da mensagem `m`.

O primeiro passo é calcular `e : e = hash(m)`

Depois temos de calcular os pares de chave temporarios (`k,r`) tal que `r e s` , depois de computados, têm de ser maiores que 0.

Após isso calculamos `s : ( ( e + d*r ) / k ) mod n`

Terminados os cálculos, juntamos estes dois fatores para formar a assinatura: `assinatura = (r,s)`

Pares de chave temporários

- **k** - inteiro compreendido: 0 < k < n-1
- **r** - para calcular `r` precisamos de:
  - **R** - `k * basepoint`
  - **Rx** - Componente x de R
  - **r** - `Rx mod n`

In [6]:
```
def generateEphemeral(n,generatorPoint):
    while(true):
        k = ZZ.random_element(1,n-1)
        R = k*generatorPoint
        Rx = R[0]
        #print('Rx: ' + str(Rx))
        r = mod(Rx,n)
        if(r > 0):
            break
    #print('k: ' + str(k))
    #print('r: ' + str(r))
    return (r,k)
```

Na função em baixo vemos o processo de assinatura de uma mensagem `m` .

In [7]:
```
def signMessage(privateKey,n,generatorPoint,m):
    e = hash(m)

    while(true):
        (r,k) = generateEphemeral(n,generatorPoint)

        dr = privateKey*r
        s1 = e + dr
        s = power_mod(s1/k,1,n)
        if(s > 0):
            break
    signature = (r,s)
    print('Signature: ' + str(signature))
    return signature
```

Verificar Assinatura

Após a assinatura é necessário a operação de verificação de assinatura.

O primeiro passo é verificar os tamanhos de `r e s` :

- `0 < r < n`
- `0 < s < n`

Depois calculamos a hash da mensagem: `e = hash(m)` .

Depois calculamos:

- **w**: `w = s^(-1) mod n`
- **u1** : `u1 = (e*w) mod n`
- **u2** : `u2 = (r*w) mod n`
- **P**: `P = (u1*G) + (u2*q)`

A partir deste `P` , pegamos na sua componente x: `px = P[0]` .

A verificação da assinatura corresponde ao valor de verdade da comparação: `px == r mod n`

In [8]:
```
def verifySignature(signature,n,generatorPoint,publicKey,m):
    (r,s) = signature

    nr = n-r
    ns = n-s
    if(r<=0):
        print('r negativo!')
        return ''
    if(nr<=0):
        print('r maior que n!')
        return ''
    if(s<=0):
        print('s negativo!')
        return ''
    if(ns<=0):
        print('s maior que n!')
        return ''

    e = hash(m)

    w = power_mod(ZZ(s),-1,n)

    u1 = power_mod(e*w,1,n)
    u2 = power_mod(r*w,1,n)

    P1 =(u1*generatorPoint)
    P2 = (ZZ(u2)*(publicKey))
    P = P1+P2
    px = ZZ(P[0])
    #print('px: ' + str(px))
    r2 = mod(r,n)
    if(px==r2):
        return 'Assinatura válida'
    else:
        return 'Assinatura inválida'
```

In [9]:
```
def main():
    n = FIPS['P-384']['n']
    (generatorPoint,privateKey,publicKey) = generateKeys(n)
    # privateKey: d
    #publicKey: q
    m = b'Teste1'
    signature = signMessage(privateKey,n,generatorPoint,m)

    ver1 = verifySignature(signature,n,generatorPoint,publicKey,m)
    print(ver1)

    m2 = b'Teste2'
    ver2 = verifySignature(signature,n,generatorPoint,publicKey,m2)
    print(ver2)
```

In [10]:
```
main()
```

```
GeneratorPoint: (262470350957996892686231567445669818918529234911092133878156159009255188547380500890223880539757197866508724767320887 : 832571096148902998554675128952010817928785304886131559470920590248050319988441922443864376039294733307808651162787 1 : 1)
PrivateKey: 14999611940890415949892677496934663755160619559661734457385406090500285461364294685714177593732828588826292614964 37
PublicKey: (383562395390612805592952530275967764090531735278464343279801164816142349608321462750968173210638853597456579867146551 : 1793371818299485588731286105334920071485131113490719798111646009576835956241194133343264386473170214870272438595721 7 : 1)
Signature: (289421755610322389020752247376200324765651082487487869050649543313657885132314116459913918982409767981419826870687751, 95693885892527136225923859433098470000807621543271472462315167829442739129563586137684220280285276595663923649645061)
Assinatura válida
Assinatura inválida
```

In [ ]:
```

```

Appendices

A partir da próxima página poderemos consultar o documento que expõe o algoritmo de construção do esquema de assinatura El Gamal.

ElGamal signature scheme

The **ElGamal signature scheme** is a digital signature scheme which is based on the difficulty of computing discrete logarithms. It was described by Taher ElGamal in 1984.[1]

The ElGamal signature algorithm described in this article is rarely used in practice. A variant developed at NSA and known as the Digital Signature Algorithm is much more widely used. There are several other variants.[2] The ElGamal signature scheme must not be confused with ElGamal encryption which was also invented by Taher ElGamal.

The ElGamal signature scheme allows that a verifier can confirm the authenticity of a message *m* sent by the signer sent to him over an insecure channel.

System parameters

- Let H be a collision-resistant hash function.
- Let p be a large prime such that computing discrete logarithms modulo p is difficult.
- Let $g < p$ be a randomly chosen generator of the multiplicative group of integers modulo p Z_p^*.

These system parameters may be shared between users.

Key generation

- Choose randomly a secret key x with $1 < x < p - 1$.
- Compute $y = g^x \bmod p$.
- The public key is (p, g, y).
- The secret key is x.

These steps are performed once by the signer.

Signature generation

To sign a message m the signer performs the following steps.

- Choose a random k such that $0 < k < p - 1$ and $\gcd(k, p - 1) = 1$.
- Compute $r \equiv g^k \pmod p$.
- Compute $s \equiv (H(m) - xr)k^{-1} \pmod{p-1}$.
- If $s = 0$ start over again.

Then the pair (r,s) is the digital signature of m. The signer repeats these steps for every signature.

Verification

A signature (r,s) of a message m is verified as follows.

- $0 < r < p$ and $0 < s < p - 1$.
- $g^{H(m)} \equiv y^r r^s \pmod p$.

The verifier accepts a signature if all conditions are satisfied and rejects it otherwise.

Correctness

The algorithm is correct in the sense that a signature generated with the signing algorithm will always be accepted by the verifier.

The signature generation implies

$$H(m) \equiv xr + sk \pmod{p-1}.$$

Hence Fermat's little theorem implies

$$\begin{aligned} g^{H(m)} &\equiv g^{xr} g^{ks} \\ &\equiv (g^x)^r (g^k)^s \\ &\equiv (y)^r (r)^s \pmod{p}. \end{aligned}$$

Security

A third party can forge signatures either by finding the signer's secret key *x* or by finding collisions in the hash function $H(m) \equiv H(M) \pmod{p-1}$. Both problems are believed to be difficult. However, as of 2011 no tight reduction to a computational hardness assumption is known.

The signer must be careful to choose a different *k* uniformly at random for each signature and to be certain that *k*, or even partial information about *k*, is not leaked. Otherwise, an attacker may be able to deduce the secret key *x* with reduced difficulty, perhaps enough to allow a practical attack. In particular, if two messages are sent using the same value of *k* and the same key, then an attacker can compute *x* directly.[1]

References

[1] T. ElGamal (1985). "A public key cryptosystem and a signature scheme based on discrete logarithms" (http://hereford.homeip.net/ElGamal/ElGamal - A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms ElGamal.pdf). *IEEE Trans inf Theo* **31** (4): 469–472. .

[2] K. Nyberg, R. A. Rueppel (1996). "Message recovery for signature schemes based on the discrete logarithm problem" (http://dsns.csie.nctu.edu.tw/research/crypto/HTML/PDF/E94/182.PDF). *Designs, Codes and Cryptography* **7** (1-2): 61–81. doi:10.1007/BF00125076. .

Article Sources and Contributors

ElGamal signature scheme *Source*: http://en.wikipedia.org/w/index.php?oldid=412263425 *Contributors*: ArnoldReinhold, Bbartlog, Ciphergoth, Davidgothberg, Fryed-peach, Gelbard, Intgr, Jafet, John Reaves, Jopsen, JustAGal, Kevinsluckynumbersev3n, Matt Crypto, Maxal, Message From Xenu, Michael Hardy, Reetep, Senojsitruc, Ww, 27 anonymous edits

License