Universidade do Minho

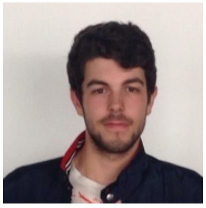
Universidade do Minho

Mestrado Integrado em Engenharia Informática

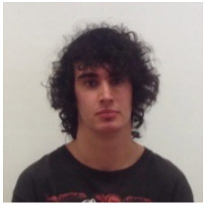
Estruturas Criptográficas

TP2

Autores:



João Cabo A75064



Diogo Soares A74478

10 de Abril de 2019

RSA

April 10, 2019

1 Exercicio 1

Pretende-se implementar uma classe Python que implemente esquemas RSA.

```
In [1]: #Gera um primo random entre 2^(l-1) e 2^l-1
        def rprime(l):
            return random_prime(2**l-1,True,2**(l-1))

        #Mapeia a string dada como argumento para código ASCII
        def mapStr(m):
            m = [ord(x) for x in m]
            m = ZZ(list(reversed(m)),100)
            return m
```

```
In [4]: l=1024

        #http://doc.sagemath.org/html/en/thematic_tutorials/numtheory_rsa.html?fbclid=
        #IwAR0rt-oWkN14K1Y1r9RBTu_-B6nq6bOGy2iusQ2uVMSidaUTF8zhx88R2Bo#cormenetal2001

        def startRSA():

            #Gera 2 primos q e p
            q=rprime(l)
            p=rprime(l+1)

            n = p*q
            phi = (p-1)*(q-1)

            #ZZ.random_element(n) retorna um pseudo-random integer
            #uniformemente distribuido entre o intervalo [0,n-1]
            e = ZZ.random_element(phi)
            #o maior divisor comum entre 'e' e 'phi' deve ser 1
            while gcd(e,phi) != 1:
                e = ZZ.random_element(phi)

            # xgcd retorna 3-tuple (g,s,t) que satisfaz a identidade
            #de bézout: g=gcd(x,y) = sx + ty
            bezout = xgcd(e,phi)
```

```
    #calcula o valor de d usando o algoritmo de Euclides estendido
    # de1(mod (n))
    d = Integer(mod(bezout[1],phi))

    return (d,e,n)

#private key (p,q,d)
#public key (e,n)

def cipher(m,e,n):
    return power_mod(m,e,n)

def decipher(c,d,n):
    return power_mod(c,d,n)
```

2 Assinatura RSA e sua verificação

```
In [49]: #http://www.sagemath.org/files/kohel-book-2008.pdf

         def sign(m,d,n):
             return power_mod(m,d,n)

         def verify(sign,m,e,n):
             s = power_mod(sign,e,n)

             if m==s:
                 return true
             return false
```

```
In [50]: msg1 = "HELLOWORLD"
         msg = mapStr(msg1)

         #Cipher and Decipher
         (d,e,n) = startRSA()
         cip = cipher(msg,e,n)
         plain = decipher(cip,d,n)
         print(msg1 + "--->" + str(msg) + "--->" + str(plain))

         print("Decipher: " + str(mapStr(msg1) == msg))

         #Sign and Verify
         sig = sign(msg,d,n)
         ver = verify(sig,msg,e,n)

         print("Signature: " + str(ver))
```

```
HELLOWORLD--->72697676798779827668--->72697676798779827668
Decipher: True
Signature: True
```

```
In [ ]:
```

ECDSA

April 10, 2019

1 Exercicio 2

Implementar o ECDSA usando uma das curvas elípticas primas definidas no FIPS186-4.

```
In [31]: #s é a inversa multiplicativa modular de a modulo b
         #t é a inversa multiplicativa modular de b modulo a
         def mod_mult_inv(a,b):
             [g,s,t] = xgcd(a,b)
             return Integer(mod(s,b))
```

O ECDSA foi implementado com a curva P-192. Iremos agora apresentar os parâmetros das curvas para melhor perceber esta implementação.

p - Primo que define o campo finito em que da curva
n - Ordem do gerador da curva
G - Ponto da curva
d - Chave Privada
q - Chave Publica (multiplicação de d com G)
k - Gerado aleatoriamente no processo de assinatura
(s1,s2) - Assinatura

```
In [32]: NIST = dict()
         NIST['P-192'] =  {
                  'p': 6277101735386680763835789423207666416083908700390324961279,
                  'n': 6277101735386680763835789423176059013767194773182842284081,
                  'b': '64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1',
                  'Gx' : '188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012',
                  'Gy' : '07192b95ffc8da78631011ed6b24cdd573f977a11e794811'
                 }
```

```
In [37]: c = NIST['P-192']
         p  = c['p']
         n  = c['n']
         b  = ZZ(c['b'],16)
         Gx = ZZ(c['Gx'],16)
         Gy = ZZ(c['Gy'],16)

         #https://www.johannes-bauer.com/compsci/ecc/
         #4.2
```

```
#Geração de chaves
def keyGen():
    #escolhe um inteiro aleatório 'd' tal que 0 < d < p
    #private key d
    d = ZZ.random_element(1,n-1)
    E = EllipticCurve(GF(p),[-3,b])
    G = E((Gx,Gy))
    #public key q
    q = d * G
    return (d,q,G)

#4.4.1
#Assinatura
#nem o s nem o r podem ser 0
def sign(d,q,G,msg):
    s = 0
    while s==0:
        r = 0
        while r==0:
            k = ZZ.random_element(1,n-1)
            r = k*G
        s1 = mod(r[0],n)
        inv = mod_mult_inv(k,n)
        s2 = (msg+d*s1)*inv
        return (s1,s2)

def verify(s1,s2,msg,q,G):
    inv = mod_mult_inv(s2,n)
    u1 = msg*inv
    u2 = s1*inv
    u1 = ZZ(u1)
    u2 = ZZ(u2)
    ponto = (u1*G) + (u2*q)
    #se o módulo p[0] é igual a s1 então a assinatura é válida
    px = ponto[0]
    return mod(px,n) == s1
```

In [38]:
```
msg = "HELLO"
h = hash(msg)

(d,q,G) = keyGen()
(s1,s2) = sign(d,q,G,h)

ver = verify(s1,s2,h,q,G)
```

```
        if ver==True:
            print "Assinatura válida"
        else:
            print "Erro"
```

```
Assinatura válida
```

```
In [ ]:
```

ECDH

April 10, 2019

1 Exercicio 3

O objetivo deste exercício é a implementação de ECDH usando curvas elípticas binárias. A curva é definida pelas raízes em Kˆ2 de um polinómio com a forma y^{2+xy+x}3+xˆ2+b e é necessário garantir que o parâmetro b tenha um grupo de torsão de ordem prima e de tamanho >= 2ˆ(n-1)

1.0.1 Gerar b

Escolhemos um b aleatório se b for 0 então voltamos a escolher outro b. Criamos uma curva definida pelo polinómio em cima. Calculamos o maior fator primo da ordem da curva Verificamos se esse valor cumpre o requisito de o tamanho ser >= 2ˆ(n-1). Se nao for repetimos o processo

1.0.2 Gerar ponto

Selecionamos um ponto aleatório da curva. Se o ponto não tiver ordem máxima voltamos a escolher outro ponto. Percorremos todos os divisores da ordem da curva e verificando se algum cumpre o seguinte requisito: divisor*ponto == ponto_infinito Se não se verificar voltamos ao ínicio. Se o requisito for cumprido podemos calcular o ponto final da seguinte forma: base_Point = (ordem//maior_fator)*ponto. Devolvemos esse ponto.

1.0.3 Gerar as Chaves

Depois de todo este processo de setup podemos finalmente gerar as chaves utilizando o método keyGen e o ponto anteriormente calculado.

```
In [107]: def setup(n):
              p = 2**n
              K = GF(p,name='t')
              #enquanto a curva E/ não tiver um grupo de torsão
              #de ordem prima e tamanho >= 2^(n-1) continuamos
              #a procurar um novo b
              while True:
                  E,facts,big_fact,b,order = bGen(K)

                  if (big_fact < 2^(n-1)):
                      break

              base_point = pointGen(E,big_fact,order)
```

```
        return base_point

    def pointGen(E,big_fact,order):
        Q = E.random_element()
        P_inf = Q*0

        divs = divisors(order)
        flag = True
        while flag:

            while (Q.order() != order):
                Q = E.random_element()

            for div in divs:
                if ((div*Q) == P_inf):
                    flag=False
                    base_point = (order//big_fact)*Q
                    break

        return base_point


    def bGen(K):
        b = K.random_element()
        while b == 0:
            b = K.random_element()
        # Nota :  [a1,a2,a3,a4,a6]  define a curva
        # y^2 + a1*x*y + a3*y = x^3 + a2*x^2 + a4*x + a6
        E = EllipticCurve(K,[1,1,0,0,b]);
        order = E.order()
        facts = list(factor(n))
        (big_fact,_) = facts[-1]

        return E,facts,big_fact,b,order


    def keyGen(point,n):
        priv = ZZ.random_element(2^n)
        pub = priv*point
        return priv,pub
```

In [108]:
```
#Inicialização
n = 23
point = setup(n)
```

```
#Dados Alice
priv_A,pub_A = keyGen(point,n)

#Dados Bob
priv_B,pub_B = keyGen(point,n)

#Calcular shared secret
sharedA = priv_A * pub_B
sharedB = priv_B * pub_A

#Correto?

print(str(sharedA == sharedB))
```

```
(0 : 1 : 0)
True
```

In [ ]: