KEM-RSA-OAEP

Nesta parte do trabalho é pedido que seja criado uma classe RSA que:

- Seja inicializada com o parametro de segurança (tamanho em bits do modulo RSA) e geração de chaves
- Contenha funções para encapular e revelar a chave gerada
- Construir a partir do KEM, e usando a transformação de Fujisaki-Okamoto, um PKE IND-CCA seguro

Funções auxiliares

Nesta secção vamos falar das funçoes auxiliares aqui implementadas, sendo estas:

- secure_prime(length): Gera um primo de tamanho arbitrário, em principio seguro
- coprime(n): lista todos os coprimos de n entre 3 e 2**16 e escolhe um aleatoriamente
- I2OSP(x,xLen): Converte um int em bytes
- mgf1 e KDF1: Mesma função, nomes diferentes Função para derivar chaves atraves de outra informação
- bxor(b1,b2): xor para bytes
- OS2IP(x): Converte bytes para int

In [129]:

```
from sage.crypto.util import carmichael_lambda
from sage.misc.prandom import choice,randrange,getrandbits
from sage.crypto.util import ascii_to_bin,bin_to_ascii
import hashlib

def secure_prime(length):
    while True:
        prime = random_prime(2**length-1, False, 2**(length-1))
        if ZZ((prime+1)/2).is_prime():
            return prime


"""def rand_prime(length):
    return random_prime((sqrt(2)*2**length-1).n(digits=9),True, 2**length)"""
def rand_prime(length):
        return random_prime(2**length-1, True, 2**(length-1))

def coprime(n):
    list = [i for i in range(3,(2**16)+1) if gcd(i, n) == 1]
    return choice(list)



def I2OSP(x: int, xLen: int) -> bytes:
    #if(x >= 256**xLen):
    #    raise ValueError("integer too large")
     #   exit()
    #else:
        return int(x).to_bytes(xLen,'big')



#retirado de https://en.wikipedia.org/wiki/Mask_generation_function#MGF1
"""def mgf1(input_str: bytes, length: int, hash=hashlib.sha256) -> bytes:
    #Mask generation function
    counter = 0
    output = b''
    while (len(output) < length):
        C = I2OSP(counter, 4)
        output += hash(input_str + C).digest()
        counter += 1
    return output[:length]"""

#https://tools.ietf.org/html/rfc3447#page-55
def mgf1(mgfSeed: bytes, maskLen: int, hash=hashlib.sha256) -> bytes:
    if(maskLen < 0):
        raise ValueError("maskLen cant be a negative integer")
    output = b''
    hLen = hash().digest_size
```

```
    for counter in range(0,ceil(maskLen / hLen)):
        C = I2OSP(counter, 4)
        output += hash(mgfSeed + C).digest()
    return output[:maskLen]

#retirado de https://stackoverflow.com/questions/23312571/fast-xoring-bytes-in-python-3?rq=1
def bxor(b1, b2): # use xor for bytes
    parts = []
    for b1, b2 in zip(b1, b2):
        #xor no sage: ^^
        parts.append(bytes([b1 ^^ b2]))
    return b''.join(parts)

def bxor2(var, key):
    key = key[:len(var)]
    int_var = int.from_bytes(var, "big")
    int_key = int.from_bytes(key, "big")
    int_enc = int_var ^^ int_key
    return int_enc.to_bytes(len(var), "big")

def xor(data, mask):
    '''Byte-by-byte XOR of two byte arrays'''
    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 OS2IP(X: bytes) -> int:
    return int.from_bytes(X,'big')


#mesmo que mgf1
def KDF1(x:bytes,l:int,hash=hashlib.sha256):
    if(l < 0):
        raise ValueError("l cant be a negative integer")
    result =b''
    for k in range (0,ceil(l/hash().digest_size)):
        result += hash(x+I2OSP(k,4)).digest()
    return result[:l]
```

RSA

Definição da classe RSA

Na classe RSA encotramos as seguintes funções:

init(self, security):

Inicia a classe RSA, recebendo como pârametro o tamanho do modulo em bits. Tem tambem como responsabilidade gerar as chaves publicas e privadas

A inicialização da classe passa pelos seguintes processos:

- geração dos dois primos p e q aleatoriamente, com tamanho security/2
- calculo de n, tal que n = p * q
- calcular totiente de euler m, tal que φ(n) = (p-1)*(q-1)
- gerar um coprimo *e* , entre 3 e 2**16 -1 ,tal que *e* e φ(n) sejam primos entre si
- calcular d, sendo este o valor da multiplicativa inversa modular de *e* e φ(n)
- chave publica (n,e) e chave privada (n,d)

padOAEP(self, msg, L,Hash)

Tem como objetivo a implementação do AOEP com está referido em rfc8017. A função recebe a mensagem que receberá padding, e L (label) (que poderá ser uma string vazia), e retorna uma octet string Nesta função os seguintes passos são implementados de

L (label) (que poderá ser uma string vazia), e retorna uma octet string. Nesta função os seguintes passos são implementados de maneira a introduzir "padding" na mensagem a encriptar

Para tal fazemos:

- k, que será o tamanho do modulo de RSA em octetos
- Hash, que por padrão será sha256
- M, mensagem a sofrer padding
- mLen, tamanho da mensagem
- hLen, tamanho em bytes da hash
- lHash, hash de L
- PS, uma byte string composta por zeros , tal que PS tenha (k - mLen - 2hLen - 2) * b'\0' octetos
- DB, um bloco de dados (Data Block), que consiste na concatenação de lHash, PS ,um unico byte b'\x01' e M, que dará origem a um data block de tamanho k -hLen - 1
- seed, tal que seed seja uma octet string aleatória segura de tamanho hLen
- dbMask, o resultado da aplicação de um MGF (mask generation function) sobre seed, de tamanho k - hLen - 1
- maskedDB, o resultado de XORing entre DB e dbMask
- seedMask, o resultado o da aplicação de um MGF sobre maskedDB, de tamanho hLen
- maskedSeed, o resultado de XORing entre seed e seedMask
- EM, que será o reultado final do padding, e é construido através da concatenação de um único octeto de valor 0x00 com maskedSeed e maskedDB, que dará origem a uma mensagem de tamanho k

unpadOAEP(self, msg, L,Hash)

Será a função que reverte o padding da mensagem, receberá a mensagem com padding, L (Label) que por padrão é uma byte string vazia

Nesta função os seguintes passos são implementados de maneira a remover "padding" na mensagem

Para tal fazemos o reverso de padOAEP:

- Separamos o conteudo da mensagem em três partes:
  - Em que Y corresponde ao primeiro octeto da mensagem (que será o octeto com valor 0x00)
  - maskedSeed corresponde ao hLen + 1 octet string da mensagem, que terá hLen de tamanho
  - maskedDb corresponderá ao resto da mensagem
- Para obter novamente o seedMask aplica-se MGF1 (mask generation function, que é basicamente um kdf) sobre maskedDB
- seed será devolvido quando é realizado XORing entre maskedSeed e seedMask
- dbMask será o resultado da aplicação do MGF sobre seed
- DB será devolvido quando é realizado XORing entre maskedDB e dbMask

Para verificar a validade da mensagem, verifica-se primeiro se a Hash da Label recebida no unpad é igual aquela que está concatenada na mensagem. Caso não seja a desncriptação/unpading falha. Esta hash a ser verificada encotra-se nos primeiros hLen octetos do DB obtido anteriormente.

Verifica-se também se a mensagem contêm o '0x01' octeto, e se não a desencriptação/unpading falha

- Remove-se os zeros e o octeto '0x01' e retornamos a mensagem sem padding

RSAEP(self,P:tuple,m):

A função trata de encriptar a mensagem m utilizando a chave publica recebida P.
Para tal recorremos a função pow() que recebe como argumento a base o expoente e o modulo,em que a base será a mensagem (um inteiro) o expoente será o *e* (P(1)), coprimo de n , e o modulo será n (P(0))
Retorna a mensagem encriptada.

RSADP(self,K:tuple,m):

A função trata de desencriptar a mensagem m utilizando a chave privada recebida K.
Para tal recorremos a função pow() que recebe como argumento a base,que será o criptograma , o expoente, que será o d (K(1)), que corresponde à multiplicativa inversa modular do coprimo de n e o valor do totiente de euler , e o modulo será n (K(0))
Retorna a mensagem desencriptada.

def EncRsaOAEP(self,pubK:tuple,msg:bytes,label = b''):

Fujisaki-okamoto Transformation

Geramos um "a" aleatorio começando primeiramente por gerar uma string com n bytes aleatorios, sobre a qual é passado um KDF
Chamamos a função padOAEP para dar padding a mensagem recebida.
É chamada a função Encap(), que vai aplicar o KEM e a transformação de FO

def Encap(self,PubK:tuple,msg,a,Hash=hashlib.sha1) -> tuple:

Começamos primeiramente por passar um Hash pela mensagem, o tal one way compressor, depois pela qual vamos somar "a"
Convertemos o valor resultante para um inteiro e aplicamos a Função RSAEP, que nos retorna a encriptação, chamado de C.
Fazemos XOR entre o valor de "a" mais a mensagem e um "k", que neste caso é uma hash resultante de "a" mais o valor da Hash da mesagem.
Será returnado C e o valor de XOR, chamaremos XORResult.

def DecRsaOAEP(self,privK:tuple,msg:bytes,label = b''):

O reverso do que foi falado anteriormente.
Nesta função é chamado Decap(), que vai reverter o processo de encapsulamento e a transformação, e retorna-nos a mensagem com padding.
Depois chamamos a função UnpadOAEP(), que nos retornara o texto limpo

def Decap(self,PrivK:tuple,C:bytes,KEKLen=1024):

Como falado anteriormente, esta função reverte o que foi aplicado anteriormente.
Primeiramente obtemos o valor de C e de XORResult.
Convertemos o valor de C para um inteiro, e aplicamos RSADP(), que nos retornara a desencriptação que chamaremos aqui de "k" (no codigo "msgint")
Apartir daqui consiguimos obter o valor de "a" e da mensagem com padding novamente ao realizar um XOR entre XORResult e k
Retornamos a mensagem caso as condiçoes sejam cumpridas, sendo que cyfer == self.Encap(PubK,msg,a), sendo cyfer um tuple de C e XORResult

In [130]:

```
class RSA:

    def __init__(self,security):
        #gerar numeros primos
        self.size = security
        p = secure_prime(self.size/2)
        q = secure_prime(self.size/2)

        #Calcular n
        n = p*q
        #m = carmichael_lambda(n)
        #m = euler_phi(n)
        #Calcular phi (euler)
        m= (p-1)*(q-1)
        #coprimo
        e = coprime(m)
        #alt
        #coprimos
        #G = IntegerModRing(m)
        #e = G(rand_prime(10))
        #modular multiplicative inverse
        d = inverse_mod(e,m)
        #chave publica
        self.public_key = (n,e)
        #chave privada
        self.private_key = (n,d)
        #podes ver isto em https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Key_generation

        #https://tools.ietf.org/html/rfc3447#section-7.1.1
    def padOAEP(self,msg: bytes, L = b'',Hash = hashlib.sha1 ):

        #k denotes the length in octets of the RSA modulus n
        k = self.size//8
        #keyinbyte =
int(self.public_key[0]).to_bytes(int(self.public_key[0]).bit_length()//8,"big")
        #k = len(keyinbyte)

        #message to be encrypted, an octet string of length mLen, where mLen <= k - 2hLen - 2
        M = msg
        mLen = len(M)

        #hLen denotes the length in octets of the hash function output
        hLen = Hash().digest_size

        #If mLen > k - 2hLen - 2, output "message too long" and stop
        if mLen > k - 2*hLen - 2:
```

```
            raise ValueError('message too long')
            exit()

        #If the label L is not provided, let L be the empty string. Let
        #lHash = Hash(L), an octet string of length hLen (see the note
        #below)

        lHash = Hash(L).digest()

        #Generate an octet string PS consisting of k - mLen - 2hLen - 2 zero octets.  The length o
f PS may be zero.
        #PS = k - mLen - 2*hLen zeros
        PS = b''
        PS = b'\0' * (k - mLen - 2*hLen -2)
        #while(len(PS) < k - mLen - 2*hLen):
        #    PS += int(0).to_bytes(1, 'big')

        #Concatenate lHash, PS, a single octet with hexadecimal value 0x01, and the message M
         #to form a data block DB of length k -hLen - 1 octets as
        DB = lHash + PS + b'\x01' + M

        #Generate a random octet string seed of length hLen.
        #seed = bytes(randrange(256) for i in range(hLen))
        #seed = i2osp(getrandbits(hLen*8), hLen)
        seed = os.urandom(hLen)

        #Let dbMask = MGF(seed, k - hLen - 1)
        dbMask = mgf1(seed, (k - hLen - 1))

        #Let maskedDB = DB \xor dbMask
        maskedDB = bxor(DB,dbMask)


        #Let seedMask = MGF(maskedDB, hLen)
        seedMask = mgf1(maskedDB, hLen)

        #Let maskedSeed = seed \xor seedMask
        maskedSeed =bxor(seed,seedMask)

        #Concatenate a single octet with hexadecimal value 0x00,
        #maskedSeed, and maskedDB to form an encoded message EM of
        #length k octets as
        EM = b'\x00' + maskedSeed + maskedDB

        return (EM)

    def unpadOAEP(self,msg:bytes,Hash = hashlib.sha1,L = b''):

        #k denotes the length in octets of the RSA modulus n
        k = self.size//8
        #keyinbyte =
int(self.public_key[0]).to_bytes(int(self.public_key[0]).bit_length()//8,"big")
        #k = len(keyinbyte)

        #message to be encrypted, an octet string of length mLen, where mLen <= k - 2hLen - 2
        EM = msg
        mLen = len(EM)

        #hLen denotes the length in octets of the hash function output
        hLen = Hash().digest_size


        #If the label L is not provided, let L be the empty string. Let
        #lHash = Hash(L), an octet string of length hLen (see the note
        #below)
        lHash = Hash(L).digest()

        #Separate the encoded message EM into a single octet Y, an octet
         #string maskedSeed of length hLen, and an octet string maskedDB
         #of length k - hLen - 1 as
            #EM = Y || maskedSeed || maskedDB.

        Y = EM[:1]
        maskedSeed = EM[1:hLen+1]
        maskedDB = EM[hLen+1:]

        #Let seedMask = MGF(maskedDB, hLen).
```

```
        seedMask = mgf1(maskedDB, hLen)

        #Let seed = maskedSeed \xor seedMask
        seed = bxor(maskedSeed,seedMask)

        #Let dbMask = MGF(seed, k - hLen - 1)
        dbMask = mgf1(seed, k - hLen - 1)

        #Let DB = maskedDB \xor dbMask.
        DB = bxor(maskedDB,dbMask)

        #Separate DB into an octet string lHash' of length hLen, a
         #(possibly empty) padding string PS consisting of octets with
         #hexadecimal value 0x00, and a message M as
            #DB = lHash' || PS || 0x01 || M.

        lHashO = DB[:hLen]
        if lHashO != lHash:
            raise TypeError("decryption error,bye")
            exit()

        rest_msg = DB[hLen:]
        zeronemsg = rest_msg.replace(b'\x00',b'')

        if zeronemsg.find(b'\x01') is None:
            raise TypeError("decryption error,bye")
            exit()

        msg = zeronemsg.replace(b'\x01',b'')
        return msg

    def RSAEP(self,P:tuple,m):
        return pow(m,P[1],P[0])

    def RSADP (self,K:tuple, c):
        return pow(c,K[1],K[0])


    #https://tools.ietf.org/html/rfc5990#page-12
    def Encap(self,PubK:tuple,msg,a,Hash=hashlib.sha1) -> tuple:
        lHash = Hash().digest_size
        hmsg = Hash(msg).digest()
        hmsgz = Hash(a + hmsg).digest()
        msgint = OS2IP(hmsgz)
        c = self.RSAEP(PubK,msgint)
        msgxor = a+msg
        C = I2OSP(c,len(msgxor))
        xoredxit = bxor2(msgxor,hmsgz)
        return (C,xoredxit)

    def Decap(self,PubK:tuple,PrivK:tuple,cyfer,Hash=hashlib.sha1):
        lHash = Hash().digest_size
        C, bxoreslot = cyfer
        c = OS2IP(C)
        msgint = self.RSADP(PrivK,c)
        k = I2OSP(msgint,len(bxoreslot))
        amsg = bxor2(bxoreslot,k)
        print("amsg  ", amsg)
        a = amsg[:lHash]
        print("aaaaaa   ", a)
        msg = amsg[lHash:]
        print("mmmmmm",msg)
        if cyfer == self.Encap(PubK,msg,a):
            return msg

    def EncRsaOAEP(self,pubK:tuple,msg:bytes,Hash=hashlib.sha1,label = b''):
        EM = self.padOAEP(msg,label)
        rand = os.urandom(Hash().digest_size)
        a = KDF1(rand,Hash().digest_size)
        return self.Encap(pubK,EM,a)

    def DecRsaOAEP(self,PubK:tuple,privK:tuple,cyfer,label = b''):
        paddedmsg = self.Decap(PubK,privK,cyfer)
        unpaddedmsg = self.unpadOAEP(paddedmsg)
        return unpaddedmsg

rsa = RSA(512)
```

```
#stuff = rsa.Encap(rsa.public_key,b'dwdwddwdw',KDF1(b'dwdwddwdw', 20))
#print(stuff)
#print(rsa.Decap(rsa.public_key,rsa.private_key,stuff))
encryptedmsg = rsa.EncRsaOAEP(rsa.public_key,b'ola mae')
decryptedmsg = rsa.DecRsaOAEP(rsa.public_key,rsa.private_key,encryptedmsg)
print(encryptedmsg)
print(decryptedmsg)
#print(rsa.padOAEP(encryptedmsg))
#print(rsa.decrypt(encryptedmsg))
#rsaC,rsaKEK = rsa.Encap(rsa.public_key)
#print(rsaKEK)
#print (rsa.Decap(rsa.private_key,rsaC))
```

```
amsg
b'\x84\x9b\xc2\xdd\xac\xd7\xcf\xd7|G\xfdw\xbbX\xc0U\x1b\\\xd39\x00\x88\xfe\xa1\xaf\x95\\\x1a\x85L\x
\xd7\x8e\x18\xc0k\xe32\x8f$\n\xd9\x14fl\xb1\x910K\x90\x805\xa3\xdd\xbfo\x0b|\xcf8\x8c\xb8\xdb/a[F\x
a0\x0b\xa6\xa7\x07\xd5,\xf4\x9dn\xe6(fk'
aaaaaa    b'\x84\x9b\xc2\xdd\xac\xd7\xcf\xd7|G\xfdw\xbbX\xc0U\x1b\\\xd39'
mmmmmm
b'\x00\x88\xfe\xa1\xaf\x95\\\x1a\x85L\xdex~\xd7\x8e\x18\xc0k\xe32\x8f$\n\xd9\x14fl\xb1\x910K\x90\x8
a3\xdd\xbfo\x0b|\xcf8\x8c\xb8\xdb/a[F\xcb\xa0\x0b\xa6\xa7\x07\xd5,\xf4\x9dn\xe6(fk'
(b"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x11\xb7\xf1\xe7
5\t\x97\x02\x8c*\xe3&\x9e\x86\xac5'\x90\x1f}\xa1G\x1f\xe33\x05E\x8b\x9f\x8bo\xaaCv\xf4\xf1\xfc\xfe
\xb9>R0J\xa3G\xe0\x93(\x12\x13\x8aR9\xa1\xa8\x07\x8f\xe9N*\xff",
b'\x84\x9b\xc2\xdd\xac\xd7\xcf\xd7|G\xfdw\xbbX\xc0U\x1b\\\xd39\x00\x88\xfe\xa1\xaf\x95\\\x1a\x85L\x
\xd7\x8e\x18\xc0k\xe32\x8f$\n\xd9\x14fl\xb1\x910K\x90\x805\xa3\xdd\xbfo\x0b|\xcf8\x8c\xb8\x05\xfb\x
xe9\x881R3L\xd2\xa2v\x88\x11\xe1\x16\xa8k\x81')
b'ola mae'
```

DSA - Digital Signature Algorithm

O DSA (Digital Signature Algorithm)é baseado no conceito matemático de exponenciação modular e no problema de logaritmo discreto. Presente no ElGamal e no Schnorr. O DSA simplifica o esquema de assinatura digital do ElGamal trabalhando num subgrupo de F p^p* que tendo em conta o algoritmo de Pohlig-Hellman. Este subgrupo vesse obrigado a ter maior ordem possivel prima, sendo ele um dos algoritmos mais eficientes para computar o logaritmo discreto.

Geração dos parametros

1. Gerar pseudo-aleatoriamente os números primos (p e q), podendo alterar o tamanho dos mesmos(1024,160)(2048,224),(2048,256) ou (3072,256).
2. Verificar se realmente são primos
3. Escolher aleatoriamente um número inteiro h de {2...p-2}
4. Calcular g := h^(p-1)/q mod p. Se acontecer que g = 1, voltar para o passo anterior e gerar um h diferente.(h=2)
5. Gerar a Chave Privada, que será usada no processo de assinatura. privKey = {1..q-1}
6. Gerar a Chave Pública que é usada no processo de verificação da assinatura. pubKey = g^privKey mod p

Os parametros do algoritmo são (p,q,g), podem ser compartilhados entre diferentes usuários.

Geração da Assinatura

O assinante deve publicar a chave pública. Esta chave deve ser enviada por meio de um mecanismo confiavel. Já achave privada deve manter-se secreta.

1. Calcular a hash da mensagem que se pretende assinar.
2. Escolher um numero inteiro k aleatoriamente de {1...q-1}
3. Calcular r = (g^k mod p) mod q, se r=0 voltar a gerar um novo k.
4. Calcular s = ((k^(-1))*(H(m)+privKey*r))mod q, se s = 0 voltar a gerar um novo k.

Assinatura é (r,s).

Fator mais caro:
1° A exponenciação modular a computar r, podendo ser calculada antes que a mensagem seja co

nhecida.
2° Calculo do inverso modular k^-1 mod q, podendo ser calculada antes que a mensagem seja conhecida.
teorema de Fermat - k^(q-2) mod q.

Verificação da Assinatura

Uma assinatura (r,s) é valida para uma mensagem m do seguinte modo:

1. Verificar se 0<r<q e 0<s<q.
2. Calcular w = s¯1 mod q.
3. Calcular u1 = H(m)*w mod q
4. Calcular u2 = r*w mod q
5. Calcular v = ((g^u1)(y^u2) mod p) mod q.
6. Verificar se assinatura é valida se v = r.

Observação: Não deverão ser utilizados tamanhos de primos não recomendados. A seguinte implementação é apenas para fins académicos e de aprendizagem.

In [143]:

```
import random

class DSA:

    def __init__(self,pT,qT):
        self.p = random_prime ( 2^pT , proof=True , lbound=2^(pT-1))
        self.q = random_prime ( 2^qT , proof=True , lbound=2^(qT-1))
        #Verificamos se realmente são primos
        #Se sim devolvemos o p e n
        if mod(self.p,2) and mod(self.q,2):
            print ('primos p: ', self.p)
            print ('primo q: ', self.q)
            #return p,q
        #Se não voltamos a gerar
        else:
            print ('nao primos')
            geraPrimos(1024,160)

        #x e k sao usados na geraçao da assinatura, e precisam ser segredo
        h = random.randint(2,self.p-2)
        print('h: ',h)
        #
        e = (self.p-1)/self.q
        #menor número inteiro não inferior a x.
        self.g = pow(int(h),int(e),self.p)
        #self.g = ((h^((self.p-1)/self.q))%self.p)
        print('g: ',self.g)
        #A chave privada é usada no processo de geraçao da assinatura
        self.privKey = randint(1,self.q+1)
        #A chave publica é usada na verificaçao da assinatura
        self.pubKey=mod((self.g^self.privKey),self.p)


    def sign(self, message):
        dM = hash(message)
        # K precisa de ser gerado em cada assinatura
        k = random.randint(1,self.q-1)
        r = Mod(Mod((self.g^k),self.p), self.q)
        s = Mod(((int(k)^-1)*(dM + (self.privKey*int(r)))),self.q)
        sig = (r,s)
        print('k: ', k)
        print('r: ',r)
        print('s: ',s)
        return r,s


    def verify(self, message, signature):
        sigR = signature[0]
        sigS = signature[1]
```

```
        if (sigR > 0 or sigR < self.q or sigS > 0 or sigS < self.q ):
            #Calculo a hash
            dM = hash(message)
            #Calulo w = s^-1 mod q
            w = Mod((sigS^(-1)),self.q)
            #Calculo u1= hash da mensagem * w mod q
            u1 = Mod((dM * w),self.q)
            #Calculo u2 = r * w mod q
            u2 = Mod((sigR * w), self.q)
            #Calculo V = ((g^u1)*(y^u2) mod p) mod q
            v = Mod(Mod(((self.g^u1)*(self.pubKey^u2)),self.p),self.q)
            #v = (((self.g^u1)*((self.pubKey)^u2))%self.p)%self.q
            print ('v:',v)
            print ('r', sigR)
            if v == sigR:
                print ("Assinatura verificada.")
            else:
                print ("Assinatura não verificada.")
            #r = (g^k mod p)mod q
            #  = (g^u1 y^u2 mod p)mod q
            #  = v
```

In [144]:

```
e = DSA(1024,160)
msg = "DSA2020"
signature = e.sign(msg)
print ('sig:',signature)
```

```
primos p:
177078454327232093669265571085421768287902992705438649004689402619718988122239392903842423386145638
009486987182614578038143049685825904911331586210014660790573690983541045871935144311989408834109342
106413691439037235669050812293700049852557947971403016918407662405328666035417841292154804318703356

primo q:  919743501908547256091838367391043799732784864603
h:
227742617753231117691240619766094557701759134380122991200112614831424895738102178045960957252495873
788803550661601447431342043816735929921450166622441559013775733943433237843012749239391712983542992
436102341434112625499504873272252869007290733869187341770546045859382212523650877452337669033345347

g:
847532375706249015404150005280909223594402919917057408143756078860877969777872327191345014099135160
423264792706626572068266312248806375983184753244155554722217205401341350954239326273342872393140284
418862089853364707101726477441577819911864320552394090812443142379261613533844457570698728262860459

k:  492835928744838440404875625901514393507440049 52
r:  798703877818910385737530174316845392501312118860
s:  576004839730089720083561751436470497312380322401
sig: (798703877818910385737530174316845392501312118860,
576004839730089720083561751436470497312380322401)
```

In [145]:

```
e.verify(msg, signature)
```

```
v: 376309464517312376578010566911619857627905914261
r 376309464517312376578010566911619857627905914261
Assinatura verificada.
```

ECDSA - Elliptic Curve Digital Signature Algorithm

Trabalho Prático 2 - Alinea 3 Nesta alinea foi implementado o ECDSA, utilizando a curva eliptica prima P-384(FIPS186-4).

Primeiramente, as curvas elipticas se não forem nem 2 nem 3, podem ser representadas pela equação: $y^2 = x^3 - px - q$, onde p e q são elementos de K tais que o polinômio do membro direito $x^3 - px - q$ não tenha nenhuma raiz dupla. Além disto, deve-se verificar um ponto "no" infinito. Com tais condições, conseguimos formar o tal grupo abeliano munido de uma operaçao de soma.

Inicialização

Nesta função são estabelecidos todos os parametros necessarios, tais como os parametro retirados do documento do NIST,por tanto fez se uso da curva eliptica prima P-384. Este valores são usados posteriormente no processo de assinatura e na verificação. p - primo responsável por determinar o dominio da curva. n - Ordem da curva b - termo independente da Curva Gx - Ponto base da curva G, nas abcissas Gy - Ponto base da curva G, nas coordenadas Chave privada - Gerar um inteiro $x \\in [1,q-1]$ Chave publica - Q = xG

Geração da Assinatura

1. Começar por gerar o valor hash da mensagem que se pretende assinar
   e = HASH(m), neste caso usou-se o SHA-256
2. Gerar um inteiro k ∈ [1, q-1], ou seja 1<=k<n. Este valor não poderá ser reutilizado.
3. Calcular kP = (x1,y1)
   1. Calcular o ponto de curvatura(x1,y1)
   2. Calcular r = x1 mod n, se r = 0, volta ao passo 2 e gera outro k.
4. Calcular a Inversa de k mod n.
5. Calcular s = k_inversa *(hash + (r*private_key))mod n
7. A assinatura é o par(r,s). sendo que E(r,-s mod n) também é uma assinatura válida.

A ter em atenção: O valor r funciona como uma chave efémera, escondendo o valor do k. k tem de ser sempre descartado no da execuçao da assinatura, deste modo é impossivle gerar a mesma chave r, tornando-se possivel transportar o nonce de forma invertível. Se k for gerado por um gerador de numeros aleatoreos defeituoso, pode ocorrer das chaves privadas sejam descobertas.

Verificação da Assinatura

Esta função deve receber uma mensagem, assinatura (r,s) e claro a chave pública associada a chave privada utilizada no processo anterior da mensagem.

1. Começar por dividir assinatura (r,s)
2. Verificar se 1<=r<b e 1<=sig<n, se sim prosseguir, se não assinatura inválida.
3. Calcular hash da mensagem
4. Calcular w = inversa de s mod n
5. Calcular u1 = zw mod n , z = dM
6. Calcular u2 = rw mod n , r = sigR
7. Calcular ponto da curva(x1,y1)= u1 * G + u2 * Chave pública
8. Verificar se (cp_x mod n) == (r mod n), cp_x é a coordenada de x no ponto cp (O identidade)
   Se isto se verificar assinatura é valida,
   Caso contrário é inválida.

In [150]:

```
import hashlib
from sage.crypto.util import ascii_to_bin, bin_to_ascii

class ECDSA:


    def __init__(self):
        #Parametros retirados do documento do NIST - P-384 - Maximo cofator(h)=2²⁴
        p =
3940200619639447921227904010014361380507973927046544666794829340424572177149687032904726608825893
1606973112319
        self.n =
3940200619639447921227904010014361380507973927046544666794690527962765939911326356939895630815229
4433653942643
        b =
Integer(0xb3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8ed
2aef)
        Gx =
Integer(0xaa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55296c3a545e38
0ab7)
        Gy =
Integer(0x3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d7c
0e5f)
        #Criar curva eliptica
        self.E = EllipticCurve(GF(p),[-3,b])
```

```
        #Criar o tal ponto gerador da curva ou podiamos gerar um aleatoriamente
        #self.G = self.E.random_point()
        self.G = self.E((Gx,Gy))
        #Chave privada - valor aleatorio zz entre menor que 1 e valor n
        self.privKey = ZZ.random_element(1,self.n)
        #Chave publica - Chave privada * ponto gerador da curva
        self.pubKey = self.privKey * self.G


    def sign(self,message):
        #Sha256 - Calcular a hash da mensagem
        sha256 = hashlib.sha256()
        sha256.update(message)
        #Converter para inteiro
        dM = int(sha256.hexdigest(), 16)

        r = 0
        s = 0

        while not (r and s):
            #gerar valor aleatorio, maior ou igual a 1 e menor que n

            #Gerar um inteiro k aleatorio 1<=k<n
            #k tem de ter valores diferentes para assinaturas diferentes
            #Se k for gerado por um gerador de numeros aleatoreos defeituoso, pode ocorrer das
chaves privadas sejam descobertas
            #Soluçao seria derivar k tanto na mensagem quanto na chave privada
            k = ZZ.random_element(1,self.n)
            #Calcular ponto da curva(x1,y1)
            pontoC = k*self.G
            #Calcular r = x1 mod n, se r=0, voltar a gerar k
            r = Mod(pontoC[0],self.n)
            #r = mod (pontoC,self.n)
            if r:
                #Calcular a inversa de k mod n
                kInversa = inverse_mod(k,self.n)
                s = mod((kInversa * (dM + self.privKey * r)), self.n)

        sig = (r,s)
        return r,s

    #Metodo verificar
    def verify(self,message,signature):
        sigR = signature[0]
        sigS = signature[1]
        #1-Se 1<=r<b e 1<=sig<n prosseguir, se nao assinatura invalida
        if (sigR < 1 or sigR > self.n -1 or sigS < 1 or sigS > self.n - 1):
            return False
        else:
            #2-Calcular e=HASH(m) e converter para inteiro
            dM = int(hashlib.sha256(message).hexdigest(), 16)
            #3-Calcular w = 1/sig mod n
            w = mod(sigS^(-1),self.n)#inverse_mod(sigS,self.n)
            #4-Calcular u1 = zw mod n , z = dM
            u1 = ZZ(mod(dM*w,self.n))
            #4-Calcular u2 = rw mod n , r = sigR
            u2 = ZZ(mod(sigR*w,self.n))
            #5-Calcular ponto da curva(x1,y1)= u1 * G + u2 * Qa,
            pontoC = u1*self.G + u2*self.pubKey
            #  se o ponto de curvatura (x1,y1) = O(elemneto identidade) entao assinatura invalida
            if Mod(pontoC[0],self.n) == sigR: #Mod(sigR,self.n):
                print ("Assinatura verificada.")
            else:
                print ("Assinatura não foi verificada.")
```

In [151]:

```
e = ECDSA()
msg = b"Teste2"
signature = e.sign(msg)
```

In [152]:

```
e.verify(msg,signature)
```

```
```

```
Assinatura verificada.
```

In [ ]:

```
```