RSA

O objectivo desta fase é a criação de uma classe que implementa o algoritmo RSA, a partir de um parâmetro de segurança que se utiliza para calcular os restantes valores necessários à implementação do RSA. São definidos também métodos de encapsulamento (key_wrap(self)) e revelação (key_unwrap(self,wrap)) de uma chave aleatoriamente gerada.

In [6]:
```
import random
import hashlib


class RSA:
    def __init__(self, sec_param):
        self.sec_param = sec_param #Parametro de segurança #512 p.e.

        self.p = next_prime(ZZ.random_element(2^self.sec_param))
        self.q = next_prime(ZZ.random_element(2^self.sec_param))
        self.n = self.p*self.q
        self.phi = (self.p-1)*(self.q-1)
        self.e = ZZ.random_element(self.phi)
        while(gcd(self.e, self.phi)!=1): #coprime e and phi
            self.e = ZZ.random_element(self.phi)
        self.d = inverse_mod(self.e, self.phi) #private key

        self.publicKey = (self.n, self.e)
        self.privateKey = (self.q, self.p, self.d)


    def key_wrap(self):
        m = ZZ.random_element(self.n)
        k = hashlib.sha256(str(m).encode())
        return (pow(m,self.e,self.n), k) #wrapped

    def key_unwrap(self, wrap):
        r = pow(wrap[0],self.d,self.n)
        k = hashlib.sha256(str(r).encode())
        print(k.digest() == wrap[1].digest())
        return k
```

Teste

In [7]:
```
obj = RSA(512)
a = obj.key_wrap()
b = obj.key_unwrap(a)
print(a[0], a[1].digest())
print(b.digest())
```

```
True
660444719126486397744847412122161595658280409675526960534524849554220723812189071192874065927975554354370948955093790
182423518768412193125202446975704312079674841557030730184902188189203358052521433058668573974618613281599420398223154
412141058701739906151559935110905950108504294651536555339466888397889145 27 b'\x05\x02\xcc+\x89O\x88\x17\x13s\xbc\xa6
\x93\xb6}\xf8\x1a\xce\xa4)\xf4\xb3\x08\xc8\x8a\x1c\xd0\xe1 \x1f\xce\xd3'
b'\x05\x02\xcc+\x89O\x88\x17\x13s\xbc\xa6\x93\xb6}\xf8\x1a\xce\xa4)\xf4\xb3\x08\xc8\x8a\x1c\xd0\xe1 \x1f\xce\xd3'
```

ECDSA

Aqui encontra-se definida uma classe em Python que implementa o algoritmo ECDSA. Para a construção deste algoritmo serão utilizados parâmetros a partir da curva NIST P-224. Isto vai permitir gerar a curva elíptica associada, bem como o seu ponto gerador e chaves pública e privada. O uso de uma instância desta classe irá permitir assinar uma mensagem e respectiva verificação da assinatura da mensagem.

As 3 funções apresentadas neste esquema são as seguintes:

A função ***init(self)*** tem como objetivo inicializar os parâmetros necessários para que seja, posteriormente, possível assinar e verificar mensagens.

A função *sign(self,message)* tem como objetivo assinar digitalmente a mensagem message.

A função verify(self,message,signature) tem como objetivo verificar a assinatura signature tendo em conta a mensagem.

In [1]:
```
import hashlib
from sage.crypto.util import ascii_to_bin, bin_to_ascii

def convert_to_ZZ(message):
    raw = ascii_to_bin(message)
    return ZZ(int(str(raw),2))
```

In [9]:
```
# Curva P-224 FIPS 186-4

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

    def __init__(self):
        p224dic = NIST['P-224']
        p = p224dic['p']
        self.n = p224dic['n']
        b = ZZ(p224dic['b'],16)
        Gx = ZZ(p224dic['Gx'],16)
        Gy = ZZ(p224dic['Gy'],16)

        print(type(n))

        self.E = EllipticCurve(GF(p),[-3,b])
        self.G = self.E((Gx,Gy))
        self.private_key = ZZ.random_element(1,self.n-1)
        self.public_key = self.private_key * self.G



    def sign(self,msg):
        m = msg.encode('utf-8')
        digest = hashlib.sha256(m).hexdigest()
        digest = convert_to_ZZ(digest)
        loop_again1 = False
        while not loop_again1:
            print("break1")
            loop_again2 = False
            k = ZZ.random_element(1,self.n-1)
            r_point = k * self.G
            r = Mod(r_point[0],self.n)
            if r > 0:
                while not loop_again2:
                    print("break2")
                    k_inverse = inverse_mod(k,self.n)
                    temp_calc = k_inverse * (digest + (r*self.private_key))
                    s = ZZ(Mod(temp_calc,self.n))
                    if s > 0 :
                        loop_again1 = True
                        loop_again2 = True

        return r,s



    def verify(self,msg,sig):
        m = msg.encode('utf-8')
        sig_r = sig[0]
        sig_s = sig[1]
        if (sig_r < 1 or sig_r > self.n -1 or sig_s < 1 or sig_s > self.n - 1):
            return False
        else:
            digest = digest = hashlib.sha256(m).hexdigest()
            digest = convert_to_ZZ(digest)
            w = inverse_mod(sig_s,self.n)
            u1 = ZZ(Mod(digest*w,self.n))
            u2 = ZZ(Mod(sig_r*w,self.n))
            cp = u1*self.G + u2*self.public_key
            if Mod(cp[0],self.n) == Mod(sig_r,self.n):
                return True
            else:
                return False
```

Teste

In [10]:
```
e = myECDSA()
msg = "Mensagem super secreta impossível de decifrar"
r,s = e.sign(msg)
if e.verify(msg,(r,s)):
    print('Assinatura: OK')
else:
    print('Assinatura: Not OK')
```

```
<class 'function'>
break1
break2
Assinatura: OK
```