

# Ex1

April 2, 2024

## 1 Estruturas Cripográficas - Criptografia e Segurança da Informação

### 1.1 TP2 - Exercício 1

Tratado realizado por:

Número	Nome
PG54177	Ricardo Alves Oliveira
PG54236	Simão Oliveira Alvim Barroso

#### 1.1.1 Enunciado

Estes problemas destinam à iniciação do uso do SageMath em protótipos de esquemas clássicos de chave pública.

1. Construir uma classe Python que implemente o EdDSA a partir do “standard” [FIPS186-5](#)
  1. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.
  2. A implementação da classe deve usar uma das “Twisted Edwards Curves” definidas no standard e escolhida na iniciação da classe a curva “edwards25519” ou “edwards448”.

Para a resolução deste trabalho, foram feitas 2 versões : 1. Com SageMath (edwards25519) 2. Sem SageMath (edwards25519 e edwards448)

Na versão do sagemath só definimos o “edwards25519”, para facilitar a impletação destas. São no fundo as duas a mesma coisa.

Os dois estão estruturados da mesma forma, começando com uma explicação antes de cada pedaço de código e uma demonstração no final

#### 1.1.2 Versão do SageMath

Como foi dito anteriormente, esta versão é feita com o SageMath e implementa a curva “edwards25519”.

Como seria de esperar, os import necessários são:

- sage.all para utilizar o sagemath
- hashlib para utilizar a função de hash especificada no RFC8032

- os para gerar aliatóriamente a chave privada (e de forma mais “segura” com o método `urandom`)

```
[ ]: from sage.all import *
import os
import hashlib
import re
```

Em primeiro lugar, a existência de uma classe que implementa um ponto de uma Curva de Edwards.

Um ponto para ser iniciado por 3 parâmetros, a curva e as coordenadas  $x$  e  $y$  do ponto. Para além destes parametros, existem outros métodos auxiliares que fazem operações com o ponto, seja adicionar, calcular a simétrica, etc

```
[ ]: class EdPoint(object):
    def __init__(self, curve=None, x=None, y=None):
        self.curve = curve
        self.x = x
        self.y = y

    def eq(self, other):
        return self.x == other.x and self.y == other.y

    def copy(self):
        return EdPoint(curve=self.curve, x=self.x, y=self.y)

    def zero(self):
        return EdPoint(curve=self.curve, x=0, y=1)

    def sim(self):
        return EdPoint(curve=self.curve, x=-self.x, y=self.y)

    def soma(self, other):
        a = self.curve.a
        d = self.curve.d
        delta = d * (self.x*self.y) * (other.x*other.y)

        # https://en.wikipedia.org/wiki/Twisted_Edwards_curve
        self.x, self.y = (self.x * other.y + self.y * other.x) / (1 + delta), (self.
↪ y * other.y - a * self.x * other.x) / (1 - delta)

    def duplica(self):
        a = self.curve.a
        d = self.curve.d
        delta = d*(self.x*self.y)**2

        self.x, self.y = (2 * (self.x*self.y)) / (1 + delta), (self.y**2 - a * self.
↪ x**2) / (1 - delta)
```

```

def mult(self, n):
    m = Mod(n, self.curve.L).lift().digits(2)
    Q = self.copy()
    A = self.zero()

    for b in m:
        if b == 1:
            A.soma(Q)
            Q.duplica()

    return A

```

Funções auxiliares para a implementação do EdDSA. A maioria destas funções são retiradas do RFC8032, que é o standard que define o EdDSA.

```

[ ]: def sha512(x):
    return hashlib.sha512(str(x).encode()).digest()

def sha512_modq(s):
    return int.from_bytes(sha512(s), "little") % ZZ(2**252 +
↳ 27742317777372353535851937790883648493)

def modp_inv(x, p):
    return pow(x, p-2, p)

def recover_x(y, sign, p, d):
    if y >= p:
        return None

    x2 = (y*y-1) * modp_inv(d*y*y+1, p)

    if x2 == 0:
        if sign:
            return None
        else:
            return 0

    x = pow(x2, (p+3) // 8, p)
    x = int(x.lift())

    if (x*x - x2) % p != 0:
        x = x * pow(2, (p-1) // 4, p) % p

    if (x*x - x2) % p != 0:
        return None

```

```

    if (int(x) & 1) != sign:
        x = p - x

    return x

def point_decompress(s,p,d):
    if len(s) != 32:
        raise Exception("Invalid input length for decompression")

    y = int.from_bytes(s, "little")
    sign = y >> 255
    y &= (1 << 255) - 1

    x = recover_x(y, sign, p, d)

    if x is None:
        return None
    else:
        return (x, y, 1, x*y % p)

def point_compress(P, p):
    p_x = P.x
    p_y = P.y
    p_z = 1

    zinv = modp_inv(p_z,p)
    x = p_x * zinv % p
    y = p_y * zinv % p

    return int.to_bytes(int(int(p_y) | ((int(p_x) & 1) << 255)), 32, "little")

```

Classe que implementa o EdCDSA fazendo uso de twisted edwards curves edwards25519. É de relevar que está tudo a ser feito consoante o FIPS 185-5 da NIST e segundo o RF8032. Para além de tratar de tudo o que é assinaturas, verificação e geração de chaves, também tem como função transformar a curva ED2556 na curva elíptica de edwards bi-racional Curve25519. Desta forma é possível que apartir dos parâmetros da curva ED2556 criar uma curva eliptica isomórfica.

É composta pelos seguintes métodos:

- `__init__` : Inicializa a classe
- `public_key` : Método que calcula a chave pública
- `private_key` : Gera uma private a ser utilizada nos outros métodos.
- `sign` : Assina uma mensagem a ser enviada.
- `verify` : Verifica se a mensagem ou a assinatura é válida, isto é, não foram alteradas.
- `ed2ec` e `ec2ed` : Funções auxiliares para mapear curvas edwards em elíticas e vice versa.

```

[ ]: class Ed25519:
    def __init__(self):
        self.p = 2**255 - 19
        self.b = 256
        self.n = 254

        self.K = GF(self.p)
        self.a = self.K(-1)
        self.d = self.K(-121665)/self.K(121666)
        self.Px = self.
↪K(15112221349535400772501151409588531511454012693041857206046113283949847762202)
        self.Py = self.
↪K(46316835694926478169428394003475163141307993866256225615783033603165251855960)

        self.L = ZZ(2**252 + 27742317777372353535851937790883648493)
        self.c = 3

        assert self.a != self.d and is_prime(self.p) and self.p > 3

        A = 2 * (self.a + self.d)/(self.a - self.d)
        B = 4/(self.a - self.d)
        self.alfa = A/(3*B)
        self.s = B

        a4 = self.s**(-2) - 3*self.alfa**2
        a6 = (-self.alfa)**3 - a4*self.alfa

        self.EC = EllipticCurve(self.K, [a4, a6])

        #self.P = self.ed2ec(self.Px, self.Py)

    #Calculate private key
    def private_key(self, seed):
        if (len(seed) != 32):
            raise Exception("Bad seed length")

        hashed_seed = sha512(seed)
        a = int.from_bytes(hashed_seed[:32], "little")
        a &= (1 << 254) - 8
        a |= (1 << 254)

        return (a, hashed_seed[32:])

    #Calculate public key
    def public_key(self, seed):
        (a, dummy) = self.private_key(seed)

```

```

point = EdPoint(curve=self, x=self.Px, y=self.Py)
point = point.mult(a)

return point_compress(point, self.p)

#Sign message
def sign(self, message, seed):
    (a,prefix) = self.private_key(seed)
    #Public key calculation
    point_A = EdPoint(curve=self, x=self.Px, y=self.Py)
    point_A = point_A.mult(a)
    A = point_compress(point_A, self.p)
    #Secret number calculation
    r = sha512_modq(prefix + message)

    #Ponto R
    point_R = EdPoint(curve=self, x=self.Px, y=self.Py)
    point_R = point_R.mult(r)
    Rs = point_compress(point_R, self.p)

    h = sha512_modq(Rs + A + message)
    s = (r + h * a) % self.L

    return Rs + int.to_bytes(int(s), 32, "little")

#Verify message signature
def verify(self, message, signature, public_key):
    if len(public_key) != 32:
        raise Exception("Bad public key length")
    if len(signature) != 64:
        Exception("Bad signature length")

    #Obter ponto representativo da public key
    A = point_decompress(public_key, self.p, self.d)
    if not A:
        return False

    #Obter ponto r
    Rs = signature[:32]
    R = point_decompress(Rs, self.p, self.d)
    point_r = EdPoint(curve=self, x=R[0], y=R[1])

    if not R:
        return False

    s = int.from_bytes(signature[32:], "little")
    if s >= self.L:

```

```

        return False

    h = sha512_modq(Rs + public_key + message)

    point_s = EdPoint(curve=self, x=self.Px, y=self.Py)
    point_s = point_s.mult(s)

    point_h = EdPoint(curve=self, x=A[0], y=A[1])
    point_h = point_h.mult(h)

    point_r.soma(point_h)

    #Compara os pontos
    return point_s.eq(point_r)

def ed2ec(self,x,y):
    if (x,y) == (0,1):
        return self.EC(0)

    z = (1 + y)/(1 - y)
    w = z/x

    return self.EC(z/self.s + self.alfa , w/self.s)

def ec2ed(self,P):
    if P == self.EC(0):
        return (0,1)

    x,y = P.xy()
    u = self.s * (x - self.alfa)
    v = self.s * y

    return (u / v, (u - 1) / (u + 1))

```

**Testes** Temos agora uma secção com 2 testes:

- O primeiro é o funcionamento normal desta primitiva, onde testamos a assinatura de uma mensagem com uma chave privada e verificamos a assinatura com a chave pública correspondente, para ver se uma das duas foi alterada. Como era de esperar a verificação ocorreu com sucesso.
- O segundo teste é um teste de falha, onde alteramos a mensagem e verificamos a assinatura. Como era de esperar a verificação falhou.

```

[ ]: print("----- TESTE_1
      ↳SUCEDIDO-----")

#Gerar chave privada

```

```

private_key = os.urandom(32)

#Criar objeto
E = Ed25519()
message = b"Isto e uma mensagem exemplo que e muito importante, recomendada!"

#Calcular chave publica
public_key = E.public_key(private_key)
print("\tChave Pública: ", public_key)

#Assinar mensagem
signature = E.sign(message, private_key)
print("\tAssinatura: ", signature)

#Verificar que assinatura corresponde à mensagem
print("Foi verificado ? ", E.verify(message, signature, public_key))

print("\n----- TESTE FALHADO - Mudanca da mensagem_
↳-----\n")

#Gerar chave privada
private_key = os.urandom(32)

#Criar objeto
E = Ed25519()
message = b"Isto e uma mensagem exemplo que e muito importante, recomendada!"

#Calcular chave publica
public_key = E.public_key(private_key)
print("\tChave Pública: ", public_key)

#Assinar mensagem
signature = E.sign(message, private_key)
print("\tAssinatura: ", signature)

#Modificar mensagem
message = b"VAMOS ALTERAR A MENSAGEM :)"

#Verificar que assinatura corresponde à mensagem
print("Foi verificado ? ", E.verify(message, signature, public_key))

print("\n----- TESTE FALHADO - Mudanca da Assinatura_
↳-----\n")

#Gerar chave privada
private_key = os.urandom(32)

```



```

#Criar objeto
E = Ed25519()
message = b"Mensagem numero 3."

#Calcular chave publica
public_key = E.public_key(private_key)
print("\tChave Pública: ", public_key)

#Assinar mensagem
signature = E.sign(message, private_key)
print("\tAssinatura: ", signature)

signature = signature[:len(signature)-1]
#print(signature)
signature += b'0'

#Verificar que assinatura corresponde à mensagem
print("Foi verificado ? ", E.verify(message, signature, public_key))

```

```

----- TESTE SUCEDIDO-----
Chave Pública: b'a\xde\x1d_\x9c\xc2\x0cyz\xe4\x990\x8b\r\xe71\x0b\x8c.\
xbf\xa2\x84\xf0ed]Y,}\x19\xdb\xc7'
Assinatura: b'\x9fuX[H\x95"n\x1d\x1d8u\xf8\xac !\x86c\xcb\xc8w7\xed<\xad
3xi5b:\x04\x19\xc1\xe0\xa7\x9e\xa1\xb1\xb8\xa3\x1f]r]\r\xc9:]\x9b\xc1A\xf5\x05\x
e2\xec$s*\x0b\x9c\x80\x15E\t'
Foi verificado ? True

```

```

----- TESTE FALHADO - Mudanca da mensagem
-----

```

```

Chave Pública: b'\xbe\xa9\xe4\x1d\xcc\xdd\xee#G\x91\x9b\xf0\xa2\x0b\xf8
.{\xb1\xe4\x16\x03\xb2\xdbP\x10\xf9\xd7G\x83\xe6\x08v'
Assinatura: b'\xab\xca~2\x1b\xf2\xc3\x88\xc2q\\GT\xa9\xca\xb1\xaf\x96ag
\xcb\x84\xda\x1d\x1d8u\xf8\xac !\x86c\xcb\xc8w7\xed<\xad
3xi5b:\x04\x19\xc1\xe0\xa7\x9e\xa1\xb1\xb8\xa3\x1f]r]\r\xc9:]\x9b\xc1A\xf5\x05\x
e2\xec$s*\x0b\x9c\x80\x15E\t'
Foi verificado ? False

```

```

----- TESTE FALHADO - Mudanca da Assinatura
-----

```

```

Chave Pública: b'B6\xc6\xb2\xcb\x02\xa3\xaa_\xfd\xf2v\x16b\x92\xf3\x941
q\x07noh\x13*\xec\xec\x88g\x14M'
Assinatura: b'x,\xa5\x8a*\xf0\x8e\xc1)\x10?\x0e\xf1\xb2d?B\xe0\xc7\x12\
x80\x98\x1d\x1d8u\xf8\xac !\x86c\xcb\xc8w7\xed<\xad
3xi5b:\x04\x19\xc1\xe0\xa7\x9e\xa1\xb1\xb8\xa3\x1f]r]\r\xc9:]\x9b\xc1A\xf5\x05\x
e2\xec$s*\x0b\x9c\x80\x15E\t'
Foi verificado ? False

```

### 1.1.3 Sem SageMath

Tal como a versão anterior, baseia-se no código fornecido standard RFC8032, mas desta vez implementa as curvas “edwards25519” e “edwards448”. E no código do ficheiro `eddsa2.py` fornecido na dropbox da unidade curricular.

Imports necessários. A razão por estes imports é a mesma do que a versão com sagemath

```
[ ]: import hashlib;
import os;
```

Estão aqui representadas algumas funções auxiliares necessárias para o bom funcionamento. São funções auxiliares presente no ficheiro em cima enunciado. Uma coisa que queremos enunciar aqui é a utilização da função de hash SHA-512 para as Ed25519 e a função de hash SHAKE256 para Ed448.

```
[ ]: def from_le(s):
    return int.from_bytes(s, byteorder="little")

#Decode a hexadecimal string representation of the integer.
def hexi(s): return int.from_bytes(bytes.fromhex(s), byteorder="big")

def shake256(data, olen):
    hasher = hashlib.shake_256()
    hasher.update(data)

    return hasher.digest(olen)

def sha512(data):
    return hashlib.sha512(data).digest()

def ed448_hash(data):
    dompfx = b"SigEd448" + bytes([0, 0])

    return shake256(dompfx + data, 114)

def sqrt4k3(x,p):
    return pow(x, (p + 1) // 4, p)

def sqrt8k5(x,p):
    y = pow(x, (p + 3) // 8, p)

    if (y * y) % p == x % p:
        return y
    else:
```

```

z = pow(2, (p - 1) // 4, p)
return (y * z) % p

```

A classe field presente no ficheiro. Tem as mesmas propriedades necessárias para a implementação das curvas. É uma representação do GF do sagemath.

```

[ ]: class Field:

    def __init__(self, x, p):
        self.x = x % p
        self.p = p

    def check_fields(self, y):
        if type(y) is not Field or self.p != y.p:
            raise ValueError("ERROR fields don't match!!!!")

    def __add__(self, y):
        self.check_fields(y)
        return Field(self.x + y.x, self.p)

    def __sub__(self, y):
        self.check_fields(y)
        return Field(self.p + self.x - y.x, self.p)

    def __neg__(self):
        return Field(self.p - self.x, self.p)

    def __mul__(self, y):
        self.check_fields(y)
        return Field(self.x * y.x, self.p)

    def __truediv__(self, y):
        return self * y.inv()

    def inv(self):
        return Field(pow(self.x, self.p - 2, self.p), self.p)

    def sqrt(self):
        if self.p % 4 == 3:

```

```

        y = sqrt4k3(self.x, self.p)

    elif self.p % 8 == 5:
        y = sqrt8k5(self.x, self.p)
    else:
        raise NotImplementedError("ERROR sqrt calculation!")

    _y = Field(y, self.p)

    return _y if _y * _y == self else None

def make(self, ival):
    return Field(ival, self.p)

def iszero(self):
    return self.x == 0

def __eq__(self, y):
    return self.x == y.x and self.p == y.p

def __ne__(self, y):
    return not (self == y)

def tobytes(self, b):
    return self.x.to_bytes(b // 8, byteorder="little")

def frombytes(self, x, b):
    rv = from_le(x) % (2 ** (b - 1))
    return Field(rv, self.p) if rv < self.p else None

def sign(self):
    return self.x % 2

```

Classe que define um ponto de uma curva de Edwards. Esta classe é semelhante à classe que anteriormente.

```

[ ]: class EdwardsPoint:

    def initpoint(self, x, y):
        self.x = x

```

```

self.y = y
self.z = self.base_field.make(1)

def decode_base(self, s, b):
    if len(s) != b // 8:
        return (None, None)

    xs = s[(b - 1) // 8] >> ((b - 1) & 7)

    y = self.base_field.frombytes(s, b)
    if y is None:
        return (None, None)

    x = self.solve_x2(y).sqrt()
    if x is None or (x.iszero() and xs != x.sign()):
        return (None, None)

    if x.sign() != xs:
        x = -x

    return (x, y)

def encode_base(self, b):
    xp, yp = self.x / self.z, self.y / self.z

    s = bytearray(yp.tobytes(b))

    if xp.sign() != 0:
        s[(b - 1) // 8] |= 1 << (b - 1) % 8

    return s

def __mul__(self, x):
    r = self.zero_elem()
    s = self

    while x > 0:
        if (x % 2) > 0:
            r = r + s
        s = s.double()
        x = x // 2

    return r

```

```

def __eq__(self, y):
    xn1 = self.x * y.z
    xn2 = y.x * self.z
    yn1 = self.y * y.z
    yn2 = y.y * self.z

    return xn1==xn2 and yn1==yn2

def __ne__(self,y):
    return not (self == y)

```

Classe que faz a criação de um ponto de uma curva de Edwards25519.

```

[ ]: class Edwards25519Point(EdwardsPoint):

    base_field = Field(1, 2 ** 255 - 19)
    d = -base_field.make(121665) / base_field.make(121666)
    f0 = base_field.make(0)
    f1 = base_field.make(1)
    xb = base_field.
    ↪make(15112221349535400772501151409588531511454012693041857206046113283949847762202)
    yb = base_field.
    ↪make(46316835694926478169428394003475163141307993866256225615783033603165251855960)

    @staticmethod
    def stdbase():
        return Edwards25519Point(Edwards25519Point.xb, Edwards25519Point.yb)

    def __init__(self, x, y):
        if y * y - x * x != self.f1 + self.d * x * x * y * y:
            raise ValueError("[ERROR] invalid point")
        self.initpoint(x, y)
        self.t = x * y

    def decode(self, s):
        x, y = self.decode_base(s, 256)
        return Edwards25519Point(x, y) if x is not None else None

    def encode(self):
        return self.encode_base(256)

```



```

        return 3

    def n(self):
        return 254

    def b(self):
        return 256

    def is_valid_point(self):
        x, y, z, t = self.x, self.y, self.z, self.t
        x2 = x * x
        y2 = y * y
        z2 = z * z
        lhs = (y2 - x2) * z2
        rhs = z2 * z2 + self.d * x2 * y2
        assert(lhs == rhs)
        assert(t * z == x * y)

```

Classe que faz a criação de um ponto de uma curva de Edwards448.

```

[ ]: class Edwards448Point(EdwardsPoint):

    base_field = Field(1, 2 ** 448 - 2 ** 224 - 1)
    d = base_field.make(-39081)
    f0 = base_field.make(0)
    f1 = base_field.make(1)
    xb = base_field.
↪make(22458004029592430018760433409989603624678964163256413424612546168695041546740603290902
    yb = base_field.
↪make(29881921007848149267601793044393067343754404015408024209592824137233150618983587600353

    @staticmethod
    def stdbase():
        return Edwards448Point(Edwards448Point.xb, Edwards448Point.yb)

    def __init__(self, x, y):
        if y * y + x * x != self.f1 + self.d * x * x * y * y:
            raise ValueError("[ERROR] invalid point")
        self.initpoint(x, y)

```



```

def decode(self, s):
    x, y = self.decode_base(s, 456)
    return Edwards448Point(x, y) if x is not None else None

def encode(self):
    return self.encode_base(456)

def zero_elem(self):
    return Edwards448Point(self.f0, self.f1)

def solve_x2(self, y):
    return ((y*y-self.f1)/(self.d*y*y-self.f1))

def __add__(self, y):
    tmp = self.zero_elem()
    xcp, ycp, zcp = self.x * y.x, self.y * y.y, self.z * y.z
    B = zcp * zcp
    E = self.d * xcp * ycp
    F, G = B - E, B + E
    tmp.x = zcp * F * ((self.x + self.y) * (y.x + y.y) - xcp - ycp)
    tmp.y, tmp.z = zcp * G * (ycp - xcp), F * G

    return tmp

def double(self):
    tmp = self.zero_elem()
    x1s, y1s, z1s = self.x * self.x, self.y * self.y, self.z * self.z
    xys = self.x + self.y
    F = x1s + y1s
    J = F - (z1s + z1s)
    tmp.x, tmp.y, tmp.z = (xys * xys - x1s - y1s) * J, F * (x1s - y1s), F * J

    return tmp

#def l(self): return
1817096810739017226373309519720011335884103401718295150703725497951460039615395857161957552
def l(self):
    return hexi("3fffffffffffffffffffffffffffffffffffffffffffffffff"+\
                "fffffffff7cca23e9c44edb49aed63690216cc2728dc58f552378c2"+\
                "92ab5844f3")

```

```

def c(self):
    return 2

def n(self):
    return 447

def b(self):
    return 456

def is_valid_point(self):
    x, y, z = self.x, self.y, self.z
    x2 = x * x
    y2 = y * y
    z2 = z * z
    lhs = (x2 + y2) * z2
    rhs = z2 * z2 + self.d * x2 * y2
    assert(lhs == rhs)

```

Classe para criar uma EdDSA dependo do tipo de curva que pretendemos.

```

[ ]: class EdDSA:

    def __init__(self, curve):
        if curve == 'edwards25519':
            self.B = Edwards25519Point.stdbase()
            self.H = sha512

        elif curve == 'edwards448':
            self.B = Edwards448Point.stdbase()
            self.H = ed448_hash

        else:
            raise ValueError("ERROR not accepted curve name!!!!!!!")

        self.l = self.B.l()
        self.n = self.B.n()
        self.b = self.B.b()
        self.c = self.B.c()

    def clamp(self, a):
        _a = bytearray(a)
        for i in range(0, self.c):

```

```

        _a[i // 8] &= ~(1 << (i % 8))
    _a[self.n // 8] |= 1 << (self.n % 8)

    for i in range(self.n + 1, self.b):
        _a[i // 8] &= ~(1 << (i % 8))

    return _a

def keygen(self, privkey):
    if privkey is None:
        privkey = os.urandom(self.b // 8)

    khash = self.H(privkey)
    a = from_le(self.clamp(khash[:self.b // 8]))

    return privkey, (self.B * a).encode()

def sign(self, privkey, pubkey, msg):
    khash = self.H(privkey)
    a = from_le(self.clamp(khash[:self.b // 8]))
    seed = khash[self.b // 8:]
    r = from_le(self.H(seed + msg)) % self.l
    R = (self.B * r).encode()
    h = from_le(self.H(R + pubkey + msg)) % self.l
    S = ((r + h * a) % self.l).to_bytes(self.b // 8, byteorder="little")

    return R + S

def verify(self, pubkey, msg, sig):
    if len(sig) != self.b // 4:
        return False

    if len(pubkey) != self.b // 8:
        return False

    Rraw, Sraw = sig[:self.b // 8], sig[self.b // 8:]
    R, S = self.B.decode(Rraw), from_le(Sraw)
    A = self.B.decode(pubkey)

    if (R is None) or (A is None) or S >= self.l:
        return False

    h = from_le(self.H(Rraw + pubkey + msg)) % self.l

```

```

        rhs = R + (A * h)
        lhs = self.B * S
        for i in range(0, self.c):
            lhs = lhs.double()
            rhs = rhs.double()

        return lhs == rhs

```

**Testes** Criamos alguns testes para provar o bom funcionamento do código

```

[ ]: print("----- edd448 -----")
ed448 = EdDSA('edwards448')
private_key, public_key = ed448.keygen(None)
print('\tChave Privada:', private_key)
print('\tChave Publica:', public_key)

sign = ed448.sign(private_key, public_key, b'Mensagem muito boa para assinar')
print('\tAssinatura:', sign)

print('Verificado?', ed448.verify(public_key, b'Mensagem muito boa para_
↳assinar', sign))

print("\tAlterando a mensagem")
print('Verificado?', ed448.verify(public_key, b'Mensagem muito alterada para_
↳verificar', sign))

print("----- ed25519_
↳-----")

ed25519 = EdDSA('edwards25519')
private_key, public_key = ed25519.keygen(None)
print('\tChave Privada:', private_key)
print('\tChave Publica:', public_key)

sign = ed25519.sign(private_key, public_key, b'Mensagem muito boa para assinar')
print('\tAssinatura:', sign)

print('Verificado?', ed25519.verify(public_key, b'Mensagem muito boa para_
↳assinar', sign))

print("\tAlterando a mensagem")
print('Verificado?', ed25519.verify(public_key, b'Mensagem muito alterada para_
↳verificar', sign))

```

```

----- edd448 -----
    Chave Privada: b'\xa8\xae\xd4\x9c0\xdf!\x17\\5+\x17\xb6Kn\x99\x1d\x94q`Y
\x3CkCt\x00\x1e\xbf\xc0d>+<%\xd9\xe0V\x08\x93\x04d\xdb&\xde\x88\x8b\x8e\x8by{\x
00<\x8b\x8199'
    Chave Publica: bytearray(b"\xdc\x86\xf6R\xc9'\x88\xb70\xe2\x9d4\xe6M \n
?4_b}\xd59\xb8m\xa1\x8fk\xc7\xf6\xce\x1d\x92\x0e\xfdV\xb6\xc30o/\xa6\x9d8*\xc9F6
qM]~\x83f\xbd\x1c\x00")
    Assinatura: bytearray(b']\tX\x83\x8c<\xe2\x89\xdd\xa5w\xe4(\x90\xaa\x08\
xc6\xc5\x9f\x94M\xf1S4k\xc6ch\x8bo\xca\xe7\xf6\xb3\xf3\x10\x89\x90&6\xa3\xf6\xea
\x8fSMP\x85\x0bZ\xd9\xff\r8\x83\xb8\x80\xf4,\xfd\xf6\xb5\xda\xf4\xdf\xa4\xcb>\x9
2\x00k __\xab\x05\x9b\x11\xb0nr>\xfc\xfd\x89\x12\x0f\xa2\xff\xfc\x9f\xaf\xdb\xf4
K|\xde\x84\x99\xf7X\xc18\x9e\xbc\xa3L\xe1\xf2\x9c\xb8\xb5(\x00')
Verificado? True
    Alterando a mensagem
Verificado? False
----- ed25519 -----
    Chave Privada: b'\xf2\x94\xbbL-\xd7\xb2\xfe\xf9S\xfd\xec\x18\xeb*\xb3\xf
3\xc6d\xd6\xb0*CK0-W\xc7\x9dZ\xd9\xba'
    Chave Publica: bytearray(b"\xed\xb5EZ\xc8~\xbe\xe5'\xe3k\x07\xa8\x13\x1
d\x9b\xa9\x00\x1fy\x86|\xd6\x03\xbd\xdcM:\x07 \x8c")
    Assinatura: bytearray(b'&Q\x88M\xfd\xfbE\xb8~\xde\x06\xfdC\xbf\xb8u\xe0\
x0b_g\xd9j\xd3\xd0\x91,\xea\xcc\x05aM\x7f\x00?i\xc9\x16V\xd1\xa9\xb1s;\xe8;\\%W\
x15*\xcb\x91\xbe/~\xfa\xa4Q\xb5\x8f\xc0-}\x0c')
Verificado? True
    Alterando a mensagem
Verificado? False

```