Estruturas Criptográficas

Trabalho Prático 2 - Exercício 1

José de Matos Moreira - PG53963

Pedro Freitas - PG52700

Enunciado do problema

Construir uma classe **Python** que implemente o **EdDSA** a partir do *standard* **FIPS186-5**.

- a implementação deve conter funções para assinar digitalmente e verificar a assinatura
- 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”**

Resolução

Em primeiro lugar, mostra-se necessário importar os módulos a utilizar:

In [1]:
```
import hashlib
import os
```

Funções úteis

Neste presente espaço, explica-se, assim, todas as funções utilizadas que não pertencem a nenhuma classe específica:

- **sqrt4k3**: função que calcula a raiz quadrada modular de x em relação ao módulo p, onde p é um primo congruente a 3 (mod 4)
- **sqrt8k5**: função que calcula a raiz quadrada modular de x em relação ao módulo p, onde p é um primo congruente a 5 (mod 8)
- **from_le**: função responsável por converter um conjunto de *bytes* num número inteiro, na ordem *little-endian*
- **sha512**: função que aplica o algortimo de *hash* criptográfico denominado **SHA-512** a um conjunto de *bytes*

- **shake256**: função que calcula um *hash* seguro, recorrendo ao algoritmo **SHAKE256**, de tamanho definido em argumento
- **ed448_hash**: função característica de esquemas de assinatura **Edwards-Curve Digital Signature Algorithm** (EdDSA), com a curva de **edwardsed448**, que cria um *hash*, recorrendo à função anteriormente mencionada

In [2]:
```
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


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


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


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

    return hasher.digest(olen)


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

    return shake256(dompfx + data, 114)
```

Field

De seguida, passou-se à implementação da classe **Field**. Esta mesma classe representa um elemento de um campo finito e inclui diversos métodos, responsáveis por: iniciação da classe, verificações relativas ao elemento, operações aritméticas, negação, cálculo do inverso multiplicativo e da raiz quadrada do elemento, criação de novos elementos e conversões.

In [3]:
```
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")

        _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
```

EdwardsPoint

Aqui, apresenta-se a classe **EdwardsPoint**. Esta é a que se assume como um ponto numa curva de **Edwards**. Na mesma, encontram-se diversas funções capazes de: inicializar um ponto, descompactar uma representação compactada de um ponto, compactar um ponto e efetuar operações aritméticas sobre pontos.

In [4]:
```
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)
```

Edwards25519Point

Nesta classe e, recorrendo-se à classe anteriormente explicada, definem-se as várias características e as várias funções relativas, exclusivamente, a um ponto de uma curva **edwards25519**.

In [5]:
```
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(15112221349535400772501151409588531511454012693041857
    yb = base_field.make(46316835694926478169428394003475163141307993866256225

    @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)

    def zero_elem(self):
        return Edwards25519Point(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()
        zcp = self.z * y.z
        A = (self.y - self.x) * (y.y - y.x)
        B = (self.y + self.x) * (y.y + y.x)
        C = (self.d + self.d) * self.t * y.t
        D = zcp + zcp
        E, H = B - A, B + A
        F, G = D - C, D + C
        tmp.x, tmp.y, tmp.z, tmp.t = E * F, G * H, F * G, E * H

        return tmp

    def double(self):
        tmp = self.zero_elem()
        A = self.x * self.x
        B = self.y * self.y
        Ch = self.z * self.z
        C = Ch + Ch
        H = A + B
        xys = self.x + self.y
        E = H - xys * xys
```

```
        G = A - B
        F = C + G
        tmp.x, tmp.y, tmp.z, tmp.t = E * F, G * H, F * G, E * H

        return tmp
```

Edwards448Point

Analogamente ao que foi descrito anteriormente, nesta classe apresentam-se as várias particularidades de um ponto que, desta vez, pertence a uma curva **edwards448**.

In [6]:
```
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(224580040295924300187604334099896036246789641632564
    yb = base_field.make(298819210078481492676017930443930673437544040154080


    @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),

        return tmp
```

EdDSA

Em último lugar, surge a classe que reúne tudo aquilo que foi implementado até ao momento e que é, portanto, a classe mãe de todo o trabalho desenvolvido. Na **EdDSA**, encontra-se implementado o esquema de assinatura digital **Edwards-Curve Digital Signature Algorithm** para duas curvas específicas: **edwards25519** e **edwards448**. Tal como pedido pelo enunciado do projeto, esta mesma classe possui funções capazes de assinar digitalmente e de verificar a assinatura.

In [7]:
```
class EdDSA:

    def __init__(self, curve):
        if curve == 'edwards25519':
            self.B = Edwards25519Point.stdbase()
            self.H = sha512
            self.l = 723700557733226221397318656304299424085711635937990760
            self.n = 254
            self.b = 256
            self.c = 3

        elif curve == 'edwards448':
            self.B = Edwards448Point.stdbase()
            self.H = ed448_hash
            self.l = 181709681073901722637330951972001133588410340171829515
            self.n = 447
            self.b = 456
            self.c = 2

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


    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 _ in range(0, self.c):
            lhs = lhs.double()
            rhs = rhs.double()

        return lhs == rhs
```

Testes de aplicação

Edwards25519

In [12]:
```
ed25519 = EdDSA('edwards25519')
priv25519, pub25519 = ed25519.keygen(None)
message = b'lionel messi'

sign = ed25519.sign(priv25519, pub25519, message)
verify = ed25519.verify(pub25519, b'lionel messi', sign)

if verify == True:
    print('Signature accepted!')

else:
    print('Error verifying the signature!')
```

```
Signature accepted!
```

In [13]:
```
ed25519 = EdDSA('edwards25519')
priv25519, pub25519 = ed25519.keygen(None)
message = b'lionel messi'

sign = ed25519.sign(priv25519, pub25519, message)
verify = ed25519.verify(pub25519, b'not the goat', sign)

if verify == True:
    print('Signature accepted!')

else:
    print('Error verifying the signature!')
```

```
Error verifying the signature!
```

Edwards448

In [14]:
```
ed448 = EdDSA('edwards448')
priv448, pub448 = ed448.keygen(None)
message = b'the goat'

sign = ed448.sign(priv448, pub448, message)
verify = ed448.verify(pub448, b'the goat', sign)

if verify == True:
    print('Signature accepted!')

else:
    print('Error verifying the signature!')
```

```
Signature accepted!
```

In [15]:
```
ed448 = EdDSA('edwards448')
priv448, pub448 = ed448.keygen(None)
message = b'the goat'
```

```
sign = ed448.sign(priv448, pub448, message)
verify = ed448.verify(pub448, b'not messi', sign)

if verify == True:
    print('Signature accepted!')

else:
    print('Error verifying the signature!')
```

```
Error verifying the signature!
```