Ex1

April 2, 2024

1 Trabalho Prático 2

André Freitas PG54707

Bruna Macieira PG54467

1.1 Exercício 1

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".

[ ]:
```
%pip install sagemath-standard
from hashlib import sha512
from sage.misc.prandom import randint
from sage.misc.randstate import current_randstate
```

```
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: sagemath-standard in /usr/lib/python3/dist-
packages (9.5)
Requirement already satisfied: cysignals>=1.10.2 in
/home/fura/.sage/local/lib/python3.10/site-packages (from sagemath-standard)
(1.11.4)
Note: you may need to restart the kernel to use updated packages.
```

Esta classe é responsável por implementar o esquema de assinatura digital EdDSA usando a curva edwards25519. Contém métodos para gerar chaves, assinar mensagens e verificar assinaturas. Usa métodos auxiliares para realizar operações na curva elíptica, como multiplicação escalar de pontos e adição de pontos. Usa a função de hash SHA-512 para calcular o hash das mensagens.

[ ]:
```
class EdDSA:
    def __init__(self):
        # Parâmetros da curva edwards25519
        # Prime modulus of the finite field over which the elliptic curve␣
 ↪operates
        self.p = 2**255 - 19
        # Coefficient 'a' of the elliptic curve equation
        self.a = -1
```

```
        # Coefficient 'd' of the elliptic curve equation
        self.d =␣
↪37095705934669439343138083508754565189542113879843219016388785533085940283555
        # Prime order of the base point on the elliptic curve
        self.q = 2**252 + 27742317777372353535851937790883648493
        # Coordinates of the base point on the elliptic curve
        self.base_point =␣
↪(15112221349535400772501151409588531511454012693041857206046113283949847762202,␣
↪46316835694926478169428394003475163141307993866256225615783033603165251855960)
        # Secure random number generator state
        self.random = current_randstate()

    # Calcula o hash SHA-512 de uma mensagem e retorna um inteiro no intervalo␣
↪[0, q-1], onde q é a ordem do ponto base
    def hash_message(self, message):
        return int(sha512(message).hexdigest(), 16) % self.q

    def generate_keypair(self):
        # Gerar uma chave privada aleatória
        private_key = randint(1, self.q - 1)

        # Calcular a chave pública correspondente, multiplicando a chave␣
↪privada pelo ponto base na curva
        public_key = self.scalar_multiply(private_key, self.base_point)

        return private_key, public_key

    def sign(self, private_key, public_key, message):
        # Gerar um valor de hash a partir da mensagem
        hash_msg = self.hash_message(message)

        # Gerar um valor aleatório k
        k = randint(1, self.q - 1)

        # Calcular R = k * base_point
        R = self.scalar_multiply(k, self.base_point)

        # Calcular S = (r + hash_msg * private_key) * (k^-1) % q
        k_inv = pow(k, -1, self.q)
        S = (k + hash_msg * private_key) * k_inv % self.q

        return R, S

    def verify(self, public_key, message, R, S):
        # Verificar se R é um ponto válido na curva
        if not self.is_on_curve(R):
            return False
```

```
        # Verificar se S está dentro do intervalo [1, q-1]
        if not (1 <= S < self.q):
            return False

        # Gerar um valor de hash a partir da mensagem
        hash_msg = self.hash_message(message)

        # Calcular u = hash_msg * S * base_point + S * R
        u = self.point_addition(self.scalar_multiply(hash_msg * S, self.
↪base_point), self.scalar_multiply(S, R))

        # Calcular v = -u
        v = (u[0], -u[1])

        # Verificar se R = S * base_point + v * public_key
        return R == self.point_addition(self.scalar_multiply(S, self.
↪base_point), self.scalar_multiply(v, public_key))

    def scalar_multiply(self, scalar, point):
        # Multiplicação escalar de um ponto na curva
        if scalar == 0:
            return (0, 1)

        if scalar == 1:
            return point

        if scalar % 2 == 0:
            return self.scalar_multiply(scalar // 2, self.point_addition(point,␣
↪point))
        else:
            return self.point_addition(self.scalar_multiply(scalar // 2, self.
↪point_addition(point, point)), point)

    def point_addition(self, point1, point2):
        # Adição de dois pontos na curva
        x1, y1 = point1
        x2, y2 = point2

        # Ponto no infinito
        if x1 == 0 and y1 == 1:
            return point2
        if x2 == 0 and y2 == 1:
            return point1

        if x1 == x2 and y1 == -y2:
            return (0, 1)
```

```
        lam = (y2 - y1) * pow(x2 - x1, -1, self.p)
        x3 = (lam**2 - self.a - x1 - x2) % self.p
        y3 = (lam * (x1 - x3) - y1) % self.p

        return (x3, y3)

    def is_on_curve(self, point):
        # Verificar se um ponto está na curva
        x, y = point
        return (y**2 - x**2 - self.a*x - self.d) % self.p == 0
```