TP2 - Ex.1

Abril 1, 2024

Estruturas Criptográficas

PG53886, Ivo Miguel Alves Ribeiro

A95323, Henrique Ribeiro Fernandes

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
   A. A implementação deve conter funções para assinar digitalmente e verificar a assinatura.
   B. 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".

In [ ]:
```
import hashlib
from hashlib import sha512
from sage.all import GF, power_mod
import random

def RGB(key_size, security_strength):
    # Gera uma lista de bits com valores aleatórios (0 ou 1)
    return ''.join(str(random.randint(0, 1)) for _ in range(key_size))


class EdDSA:
    def __init__(self, curve_type):
        if curve_type == "edwards25519":
            self.p = 2^255 - 19
            self.K = GF(self.p)
            self.a = -1
            self.d = self.K(-121665/121666)
            self.key_size = 256
            self.security_strength = 128
            self.c = 3
            self.n = 254
            self.B = (15112221349535400772501151409588531511454012693041857206046
                      46316835694926478169428394003475163141307993866256225615783
            self.L = 2**252 + 27742317777372353535851937790883648493
        elif curve_type == "edwards448":
            self.p = pow(2,448) - pow(2,224) - 1
            self.K = GF(self.p)
            self.c = 2
            self.n = 447
            self.d = -39081
            self.a = 1
            self.B = (22458004029592430018760433409989603624678964163256413424612
                      29881921007848149267601793044393067343754404015408024209592
            self.L =  pow(2,446) - 13818066809895115352007386748515426880336692474
            self.key_size = 456
            self.security_strength = 224
        else:
            raise ValueError("Invalid curve. Supported curves: 'edwards25519' or
```

```
    def pointAddition(self, P, Q):
        #twisted edwards  curve: (a*x^2 + y^2) mod mod = (1 + d*x^2*y^2) mod mod
        x1 = P[0]; y1 = P[1]; x2 = Q[0]; y2 = Q[1]

        x3 = ((x1*y2 + y1*x2) / (1 + self.d*x1*x2*y1*y2)) % self.p
        y3 = ((y1*y2 - self.a*x1*x2) / (1 - self.d*x1*x2*y1*y2)) % self.p

        assert (self.a*x3*x3 + y3*y3) % self.p == (1 + self.d*x3*x3*y3*y3) % self
        return x3, y3


    def applyDoubleAndAddMethod(self, P, k):
        # Pk = P x k
        additionPoint = (P[0], P[1])
        kAsBinary = bin(k)[2:]
        for i in range(1, len(kAsBinary)):
            currentBit = kAsBinary[i: i+1]
            #always apply doubling
            additionPoint = self.pointAddition(additionPoint, additionPoint)
            if currentBit == '1':
                #add base point
                additionPoint = self.pointAddition(additionPoint, P)
        return additionPoint


    def computeG(self, dP):
        return  self.applyDoubleAndAddMethod(self.B, dP)


    def encode_integer(S, b):
        # Convert integer S to little-endian binary string of length b
        return format(S, '0' + str(b) + 'b')[::-1]


    def encode_curve_element(self, x, y):
        x = int(x)
        y = int(y)
        if not (0 <= self.p-x and 0 <= self.p-y):
            raise ValueError("Point coordinates must be within the range 0 ≤ x, y
        # Encode the y-coordinate
        y_encoded = format(y, '0' + str(self.key_size - 1) + 'b')

        # Encode the sign of x (1 if odd, 0 if even)
        x_sign = 1 if x%2 != 0 else 0

        return (y_encoded + str(x_sign)).encode()


    def decode_curve_element(self, encoded_str):
        encoded_str = encoded_str.decode()
        # Extrair a parte do componente y
        y_encoded = encoded_str[:self.key_size-1]
        # Extrair a sinalização do componente x
        x_sign = int(encoded_str[self.key_size-1])
        # Reconstruir o valor do componente y
        y = int(y_encoded, 2)
        # Step 2: Recover the x-coordinate using the curve equation
        u = y*y - 1
        v = self.d*y*y - self.a
        if self.key_size == 256:
            uv = u/v
            exp = (self.p+3)/8
            w = power_mod(uv, exp, self.p)

            if (v*w*w) % self.p == u % self.p:
```

```
                x_final = w
            elif (v*w*w) % self.p == -u % self.p:
                x_final = (w * power_mod(2, (self.p-1)/4, self.p)) % self.p
            else:
                raise ValueError("decoding fails.")
        elif self.key_size == 456:
            uv = u/v
            exp = (self.p+1)/4
            w = power_mod(uv, exp, self.p)

            if (v*w*w) % self.p == u % self.p:
                x_final = w
            else:
                raise ValueError("decoding fails.")
        if x_final == 0 and x_sign == 1:
            raise ValueError("decoding fails.")
        elif int(x_final) % 2 != x_sign:
            return self.p - x_final, y
        else:
            return x_final, y


    def get_s(self, h_pk):
        h_len = len(h_pk)/2
        b = self.key_size
        if b == 256:
            hdigest1 = h_pk[:h_len]
            h1_list = list(hdigest1)
            h1_list[0]  &= 0b00011111
            # Ensure the last bit of the last octet is 0
            h1_list[-1] &= 0b01111110
            # Ensure the second to last bit of the last octet is 1
            h1_list[-1] |= 0b01000000
        elif b == 456:
            hdigest1 = h_pk[:h_len]
            h1_list = list(hdigest1)
            h1_list[0]  &= 0b00111111
            h1_list[-2] |= 0b01000000
            h1_list[-1] &= 0b11111111

        else:
            raise ValueError("Unsupported curve")

        # Step 4: Determine an integer s from hdigest1 using little-endian conver
        return int.from_bytes(bytes(h1_list), 'little')


    def key_gen(self):
        # Step 1: Generate a private key
        private_key_bits = RGB(self.key_size, self.security_strength)
        # Step 2: Compute the hash of the private key
        if self.key_size == 256:
            h_pk = sha512(private_key_bits.encode()).digest()
        elif self.key_size == 456:
            h_pk = hashlib.shake_256(private_key_bits.encode()).digest(912)
        else:
            raise ValueError("Unsupported hash function")
        # Step 3: Generate the public key
        s = self.get_s(h_pk)
        # Step 5: Compute the point [s]G
        Ec_point = self.computeG(s)
        #x3 = Ec_point[0]
        #y3 = Ec_point[1]
        #print("pk:",(x3,y3))
        public_key_bits =  self.encode_curve_element(Ec_point[0], Ec_point[1])
```

```
        # Return private key and public key
        return private_key_bits.encode(), public_key_bits


    def sign(self, text, private_key, public_key, context=''):
        # Step 1: Compute the hash of the private key
        if self.key_size == 256:
            h_pk = sha512(private_key).digest()
        elif self.key_size == 456:
            h_pk = hashlib.shake_256(private_key).digest(912)
        else:
            raise ValueError("Unsupported hash function")
        # Step 2: Compute the hash of the message and hdigest2
        byte_length = len(h_pk)//2
        hdigest1 = h_pk[:byte_length]
        hdigest2 = h_pk[byte_length:]

        if self.key_size == 256:
            r_octet = sha512(hdigest2 + text).digest()
        elif self.key_size == 456:
            context_bytes = context.encode('utf-8')
            dom4_prefix = b"SigEd448" + bytes([0]) + bytes([len(context_bytes)])
            r_octet = hashlib.shake_256(dom4_prefix + context_bytes + hdigest2 +
        else:
            raise ValueError("Error")

        # Step 3: Compute the point [r]G
        r = int.from_bytes(r_octet, 'little')
        Ec_point = self.computeG(r)
        R = self.encode_curve_element(Ec_point[0], Ec_point[1])

        # Step 4: Derive s from H(d) as in the key pair generation algorithm
        s = self.get_s(h_pk)

        # Step 5: Compute S
        if self.key_size == 256:
            hashed_data = sha512(R + public_key + text).digest()
            t = int.from_bytes(hashed_data, byteorder='little')

            # Calcular S
            S = (r + t * s) % self.L
            S = format(S, '0' + str(self.key_size) + 'b').encode()
        elif self.key_size == 456:
            hashed_data = hashlib.shake_256(dom4_prefix + R + public_key + text).
            t = int.from_bytes(hashed_data, byteorder='little')

            # Calcular S
            S = (r + t * s) % self.L
            S = format(S, '0' + str(self.key_size) + 'b').encode()
        else:
            raise ValueError("Unsupported curve")

        #print("t: ",t)
        #print("R: ", Ec_point)
        #print("s: ",s)
        # Step 6: Return signature
        return R + S


    def verify(self, text, Sign, publicKey, context=''):
        # Decode first half of signture as a point R
        half_len = len(Sign)/2
        h1_sign = Sign[:half_len]
        h2_sign = Sign[half_len:]
```

```
        R = self.decode_curve_element(h1_sign)
        s = int(h2_sign.decode(), 2)

        # Decode public key Q into a point
        Q = self.decode_curve_element(publicKey)

        # 2
        if self.key_size == 256:
            hashed_data = sha512(h1_sign + publicKey + text).digest()
            t_int = int.from_bytes(hashed_data, byteorder='little')
        else:
            context_bytes = context.encode('utf-8')
            dom4_prefix = b"SigEd448" + bytes([0]) + bytes([len(context_bytes)])
            hashed_data = hashlib.shake_256(dom4_prefix + h1_sign + publicKey + t
            t_int = int.from_bytes(hashed_data, byteorder='little')
        # 3
        #print("pk:",Q)
        #print("t: ",t_int)
        #print("R: ", R)
        #print("s: ",s)
        # 4 verification
        check_p1 = self.computeG(s)
        check_p2 = self.pointAddition(R, self.applyDoubleAndAddMethod(Q, t_int))
        #print((check_p1[0], check_p1[1]))
        #print((check_p2[0] , check_p2[1]))
        if check_p1[0] == check_p2[0]%self.p and check_p1[1] == check_p2[1]%self.
            return True
        else:
            return False
```

teste para Ed25519

In [ ]:
```
# Ed25519 is a special form of this curve where a = -1,
# d = -121665/121666.
# It handles over prime fields where p = 2^255 — 19.
# -x^2 + y^2 (mod 2255 — 19) = 1 — (121665/121666) * x^2 * y^2 (mod 2255 — 19)
# Ed25519, curve 25519
Ed25519 = EdDSA("edwards25519")

Mensagem = b"Ola Mundo"

sk, pk = Ed25519.key_gen()
print("SK: "+str(sk) + " len: "+str(len(sk)))
print("PK: "+str(pk) + " len: "+str(len(pk)))

Sign = Ed25519.sign(Mensagem, sk, pk)
print("Sign: "+str(Sign) + " len: "+str(len(Sign)))


is_valid = Ed25519.verify(Mensagem, Sign, pk)

if is_valid:
    print("Assinatura válida.")
else:
    print("Assinatura inválida.")
```

```
SK: b'0010101001010100111000001010010011010111000100111100010001011011000110010110011010110100111001101110010010000011100001101111111100100111001010000101000110010001000101010011001101001001101110111011101101011101010101111000010011010000001110000010011000101000' len: 256
PK: b'10001110111100000101101001100110111111100000100011110011000101010011001010000000001011101001010001111110011011100111000001110110110000010010100010001010110000001011010100111111011110111110111100010110001110100110000111111111011011110100010100010101100101011' len: 256
Sign: b'11010111101101111110001100001100001101011011010000101000010000111000001011110101000111001010010111011011011100111010001000110111001100011001011000000011011101001000000111110101111000010110011000011100011001110010011000110100111100001000101111000010101100011000001000001001001110100010010001100110101001100011000001000000000111001101001011101111100111111110101110110111111111010110000100000010010010001100111101011110000101010100010110010111110111011011010000000000111100001111001011100100101100101000000001111010000' len: 512
Assinatura válida.
```

teste para ED448

In [ ]:

```python
# Ed25519 is a special form of this curve where a = -1,
# d = -121665/121666.
# It handles over prime fields where p = 2^255 − 19.
# -x^2 + y^2 (mod 2255 − 19) = 1 − (121665/121666) * x^2 * y^2 (mod 2255 − 19)
Ed448 = EdDSA("edwards448")

Mensagem = b"Ola Mundo"
sk, pk = Ed448.key_gen()
print("SK: "+str(sk) + "len: "+str(len(sk)))
print("PK: "+str(pk) + "len: "+str(len(pk)))

Sign = Ed448.sign(Mensagem, sk, pk)
print("Sign: "+str(Sign) + "len: "+str(len(Sign)))


is_valid = (Ed448.verify(Mensagem, Sign, pk))

if is_valid:
    print("Assinatura válida.")
else:
    print("Assinatura inválida.")
```

```
SK: b'0100110111101010010000101010000110101000000111111001110011010100011110000000
0000001011001101110011110101011001010001001110110111100100011000000100101000100000
0111010110011011100001111010101110000000111100110101000111101011100001101101111000
0000001001110010101111100100001000001100100101010110011110100010101011011001100011
1000111000011101101101110111101000010001011000101011101101111111111111001000101111
01101001110110101110101001111010101101010010111111111101'len: 456
PK: b'0000000010000010100010001011110010101000111111101010000110111011101010111 0
0001111101111101111000101001101011110111101010101100100010111100110011001110111111
0010110110010111100010111100100001110001010000101100000000101011110000101110001110
0100010001110010001001100000111101100001111100111011100101001110101110110000001000
1011010100100101001100001010010010000110100000101101000110100011001100111111110011
0001011100010000001001111000110001110001101000100001110011'len: 456
Sign: b'00000001101110011011111111110100000011000010000100110101111110101001011 00
1001110001010011011001001000101110011000000101111111110000010011110000101010101 01
1000001100000110111000000110111010110011011101010100011000101011101011110011100000
0101101000101000110101011111101011110100010110111010010111000111000010100101101 10
1110000011011011101110000110001010111010010100101011011111110101001111111001000110
0111110110001100000100100100000001011100101111011111111101100000000000000001101000111
0001011001000101110110101011011000111011001100101111110110110011011000100110101111
0100000000010110100011100001010010111110010101111011101010110011100010000000110000
1011010010111101111111001101000101111011111100000110011101111111011100101010011010 1
0110000110001110111111101000000010001110000100111111100000111000110111111010011100
0000101000000111101111011011100001101000100101000000001000101010111011101000111101
10101110001000011001100001001'len: 912
Assinatura válida.
```