Exercício 2

Implementação do EcDSA com Twisted Edwards Curves (FIPS 186-5)

Descrição do Problema

O objetivo é implementar uma classe Python que realize assinaturas digitais e verificação de assinaturas usando o **EcDSA** conforme definido no **FIPS 186-5**. A implementação deve ser baseada em uma **Twisted Edwards Curve**, escolhida no momento da inicialização da classe. As curvas disponíveis são:

- **edwards25519** (utilizada no Ed25519, segura e eficiente para chaves de 256 bits).
- **edwards448** (utilizada no Ed448, fornecendo maior segurança com chaves de 448 bits).

A classe deve fornecer duas funções principais:

1. **Gerar uma assinatura digital** a partir de uma chave privada e de uma mensagem.
2. **Verificar a assinatura** utilizando a chave pública correspondente.

In [11]:
```
import hashlib, os
from pickle import dumps
from sage.all import * # type: ignore
import time
```

Twisted Edward Curves

Classes Ed e ed fornecidas pelo docente para implementar a curva de Edwards e métodos dos pontos correspondentes.

In [12]:
```
class Ed(object):
    def __init__(self,p, a, d , ed = None):
        assert a != d and is_prime(p) and p > 3
        K        = GF(p)

        A =  2*(a + d)/(a - d)
        B =  4/(a - d)

        alfa = A/(3*B) ; s = B

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

        self.K = K
        self.constants = {'a': a , 'd': d , 'A':A , 'B':B , 'alfa':alfa , 's':s ,
        self.EC = EllipticCurve(K,[a4,a6])

        if ed != None:
            self.L = ed['L']
            self.P = self.ed2ec(ed['Px'],ed['Py'])  # gerador do gru
        else:
            self.gen()
```

```
    def order(self):
        # A ordem prima "n" do maior subgrupo da curva, e o respetivo cofator "h"
        oo = self.EC.order()
        n,_ = list(factor(oo))[-1]
        return (n,oo//n)


    def gen(self):
        L, h = self.order()
        P = O = self.EC(0)
        while L*P == O:
            P = self.EC.random_element()
        self.P = h*P ; self.L = L


    def is_edwards(self, x, y):
        a = self.constants['a'] ; d = self.constants['d']
        x2 = x**2 ; y2 = y**2
        return a*x2 + y2 == 1 + d*x2*y2

    def ed2ec(self,x,y):      ## mapeia Ed --> EC
        if (x,y) == (0,1):
            return self.EC(0)
        z = (1+y)/(1-y) ; w = z/x
        alfa = self.constants['alfa']; s = self.constants['s']
        return self.EC(z/s + alfa , w/s)

    def ec2ed(self,P):        ## mapeia EC --> Ed
        if P == self.EC(0):
            return (0,1)
        x,y = P.xy()
        alfa = self.constants['alfa']; s = self.constants['s']
        u = s*(x - alfa) ; v = s*y
        return (u/v , (u-1)/(u+1))


class ed(object):
    def __init__(self,pt=None,curve=None,x=None,y=None):
        if pt != None:
            self.curve = pt.curve
            self.x = pt.x ; self.y = pt.y ; self.w = pt.w
        else:
            assert isinstance(curve,Ed) and curve.is_edwards(x,y)
            self.curve = curve
            self.x = x ; self.y = y ; self.w = x*y

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

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

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

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

    def soma(self, other):
        a = self.curve.constants['a']; d = self.curve.constants['d']
        delta = d*self.w*other.w
        self.x, self.y  = (self.x*other.y + self.y*other.x)/(1+delta), (self.y*oth
        self.w = self.x*self.y
```

```
    def duplica(self):
        a = self.curve.constants['a']; d = self.curve.constants['d']
        delta = d*(self.w)**2
        self.x, self.y = (2*self.w)/(1+delta) , (self.y**2 - a*self.x**2)/(1 - del
        self.w = self.x*self.y

    def mult(self, n):
        m = Mod(n,self.curve.L).lift().digits(2)   ## obter a representação binári
        Q = self.copy() ; A = self.zero()
        for b in m:
            if b == 1:
                A.soma(Q)
            Q.duplica()
        return A
```

Para implementares o pedido desenvolvemos uma classe composta por vários métodos que tratam da **geração de chaves**, **assinação de mensagens** e **verificação de assinaturas**. O objetivo principal é garantir a autenticidade e integridade das mensagens assinadas.

Fases de Implementação da Classe

1. Definição da Curva e Parâmetros Iniciais

A primeira fase da classe envolve a definição da curva utilizada. Dependendo da escolha entre `ed25519` ou `ed448`, são definidos os parâmetros necessários para operar com essas curvas, incluindo as constantes como o ponto gerador (`G`), a ordem do subgrupo, a cofactor, e o valor de `p` (primo da curva). Esses parâmetros são essenciais para realizar as operações de assinatura e verificação.

2. Função Hash para cada Curva

Em seguida, a classe implementa uma função de hash que é adaptada a cada uma das curvas. A função de hash é fundamental para gerar os valores necessários para a assinatura. Para `ed25519`, é utilizado o SHA-512, enquanto para `ed448`, usa-se o Shake-256. Este processo transforma a mensagem de entrada em um valor fixo (digest) que será utilizado no cálculo da assinatura.

3. Geração de Chaves

A geração de chaves é uma etapa crucial onde a classe gera tanto a chave privada quanto a chave pública. A chave privada é criada de forma aleatória e é processada através de um hash, resultando em um valor modificado que será utilizado para gerar a chave pública. A chave pública é calculada multiplicando o valor da chave privada pelo ponto gerador da curva. A chave pública gerada será usada para a verificação das assinaturas.

4. Processo de Assinatura

Na fase de assinatura, a classe utiliza a chave privada e o hash da mensagem para gerar a assinatura digital. O processo envolve o cálculo de dois valores principais: `r` e `S`. O valor `r` é obtido a partir de um hash da mensagem concatenada com o hash da chave privada. O

valor `S` é calculado utilizando uma equação baseada no valor `r`, o hash da mensagem e a chave privada. Esta assinatura digital é a combinação dos dois valores, `R` e `S`, que serão transmitidos juntamente com a mensagem para a verificação.

5. Verificação da Assinatura

A fase de verificação é onde a autenticidade da assinatura é confirmada. Utilizando a chave pública e a assinatura recebida, a classe realiza cálculos para verificar se a assinatura corresponde à mensagem original. A verificação envolve a comparação de dois lados de uma equação, onde a verificação de que a assinatura foi criada com a chave privada correta garante a autenticidade e integridade da mensagem. Se os lados da equação coincidirem, a assinatura é válida.

6. Codificação e Decodificação de Pontos

A classe também inclui métodos para codificar e decodificar pontos na curva de Edwards, permitindo a conversão entre formatos binários e matemáticos dos pontos, que são usados durante o cálculo de assinaturas e verificação. A codificação de pontos é importante para a comunicação eficiente entre as partes envolvidas na assinatura digital.

Conclusão

No geral, a classe é projetada para realizar de forma eficiente a assinatura e verificação de mensagens utilizando as curvas `ed25519` e `ed448`. Ela abrange todas as fases de geração de chaves, criação de assinaturas, e verificação, além de tratar a codificação e decodificação dos pontos envolvidos.

In [13]:

```
class ECDSA:
    def __init__(self, curve_name):
        # Define curves parameters
        if curve_name == "ed25519":

            p = 2**255-19
            K = GF(p)
            a = K(-1)
            d = -K(121665)/K(121666)

            ed25519 = {
                'b'  : 256,
                'Px' : K(151122213495354007725011514095885315114540126930418572060
                'Py' : K(463168356949264781694283940034751631413079938662562256157
                'L'  : ZZ(2**252 + 27742317777372353535851937790883648493), ## ord
                'n'  : 254,
                'h'  : 8
            }

            self.E = Ed(p,a,d,ed=ed25519)
            self.G = ed(curve=self.E, x=ed25519['Px'], y=ed25519['Py'])
            self.b = ed25519['b']
            self.n = ed25519['L']
            self.p = p
            self.c = 3
            self.securitybits = 128

        elif curve_name == "ed448":
```

```
        p = 2**448 - 2**224 - 1
        K = GF(p)
        a = K(1)
        d = K(-39081)

        ed448= {
        'b'  : 456,     ## tamanho das assinaturas e das chaves públicas
        'Px' : K(224580040295924300187604334099896036246789641632564134246125
        'Py' : K(298819210078481492676017930443930673437544040154080242095928
        'L'  : ZZ(2**446 - 138180668098951153520073867485154268803366924748821
        'n'  : 447,     ## tamanho dos segredos: os dois primeiros bits são 0
        'h'  : 4        ## cofactor
        }

        self.E = Ed(p,a,d,ed=ed448)
        self.G = ed(curve=self.E, x=ed448['Px'], y=ed448['Py'])
        self.b = ed448['b']
        self.n = ed448['L']
        self.p = p
        self.c = 2
        self.securitybits = 224

    else:
        raise ValueError("Curve name must be 'ed25519' or 'ed448'")

    self.curve = curve_name



# hash function for each curve ED2556 and ED448
def hash(self,data):
    if self.curve == 'ed25519':
        return hashlib.sha512(data).digest()
    else:
        return hashlib.shake_256(data).digest(912//8)


def dom4(self, f, c):
    # "SigEd448" as a byte string in ASCII (8 bytes)
    sig_ed448 = b"SigEd448"

    # Convert the value f to a single byte (octet with value f)
    f_octet = bytes([f])

    # Convert the length of the string c to a 2-byte little-endian representat
    c_length = len(c).to_bytes(2, 'little')

    # Ensure c is bytes (convert only if it's a string)
    if isinstance(c, str):
        c = c.encode('ascii')

    # Return the concatenation of "SigEd448" || f || len(c) || c
    return sig_ed448 + f_octet + c_length + c



def encode_point(self, P):
    x_int = int(P.x)
    y_int = int(P.y)

    if self.curve == "ed25519":
        octet_length = 32
    else:
```

```
            octet_length = 57

        # Encode y-coordinate as little-endian bytes
        y_encoded = y_int.to_bytes(octet_length, byteorder='little')

        # Make a mutable copy of the y-encoded bytes
        encoded_point = bytearray(y_encoded)

        # For Ed25519, the most significant bit of the final octet should be zero
        # For Ed448, the final octet should be zero
        if self.curve == 'Ed25519':
            encoded_point[-1] &= 0x7F  # Clear the most significant bit
        elif self.curve == 'Ed448':
            encoded_point[-1] = 0x00  # Set final octet to zero

        # Copy the least significant bit of x to the most significant bit of the f
        if x_int & 1:  # If x is odd (least significant bit is 1)
            encoded_point[-1] |= 0x80  # Set the most significant bit

        return bytes(encoded_point)


    def decode_point(self, encoded_point):

        a = self.E.constants['a']
        d = self.E.constants['d']
        p = self.p

        # Step 1: Interpret as integer in little-endian, extract and clear x0 bit
        encoded_bytes = bytearray(encoded_point)
        x0 = (encoded_bytes[-1] & 0x80) >> 7  # Extract MSB (which is x's LSB)
        encoded_bytes[-1] &= 0x7F  # Clear the MSB

        # Convert to integer (y-coordinate)
        y_int = int.from_bytes(encoded_bytes, byteorder='little')

        # Check if y >= p
        if y_int >= p:
            return None  # Decoding fails

        # Convert to SageMath's IntegerMod type
        y = y_int % p

        # Step 2: Recover the x-coordinate using the curve equation
        # x² = (y² - 1) / (d·y² - a) mod p

        # Calculate y²
        y_squared = y**2

        # Calculate numerator: (y² - 1)
        numerator = y_squared - 1

        # Calculate denominator: (d·y² - a)
        denominator = d * y_squared - a

        # Check if denominator is non-zero
        if denominator == 0:
            return None  # Cannot decode, division by zero

        # Calculate x² using SageMath's division
        x_squared = numerator / denominator

        # Compute square root
        try:
```

```
            x = x_squared.sqrt()
        except ValueError:
            # No square root exists
            return None

        # Convert x to integer to check the least significant bit
        x_int = int(x)

        # Ensure x has the correct least significant bit
        if (x_int & 1) != x0:
            x = -x  # Get the other square root in SageMath

        # Step 3: Return the decoded point
        return (x, y)


    def key_generation(self):

        b = self.b

        # 1. Generate private key `d` (random `b`-bit string)
        d = os.urandom(b // 8)

        # 2. Compute hash `H(d)`
        hdigest = self.hash(d)

        # 3. Extract `hdigest1` (first half of `H(d)`)
        hdigest1 = bytearray(hdigest[:b // 8])  # First `b/8` bytes

        # 3.1 Modify `hdigest1` for Ed25519
        if self.curve == "ed25519":
            hdigest1[0] &= 0b11111000  # Set first 3 bits to 0
            hdigest1[-1] &= 0b01111111  # Set last bit to 0
            hdigest1[-1] |= 0b01000000  # Set second to last bit to 1

        # 3.2 Modify `hdigest1` for Ed448
        else:  # ed448
            hdigest1[0] &= 0b11111100  # Set first 2 bits to 0
            hdigest1[-1] = 0  # Set last octet to 0
            hdigest1[-2] |= 0b10000000  # Set last bit of second to last octet to

        # 4. Convert `hdigest1` to integer `s` (little-endian)
        self.s = int.from_bytes(hdigest1, byteorder="little")

        # 5. Compute public key `Q = [s]G`
        Q = self.G.mult(self.s)

        # Return (private key, public key)
        Q_encoded = self.encode_point(Q)

        return d, Q_encoded


    def sign(self, M, d, Q, context=b""):

        b = self.b  # Bit length (256 for Ed25519, 456 for Ed448)

        # Compute hash H(d)
        hdigest = self.hash(d)

        # Extract the second half of H(d)
        hdigest2 = hdigest[b // 8:]

        # Compute r
```

```
        if self.curve == "ed25519":
            r = hashlib.sha512(hdigest2 + M).digest()
            r = int.from_bytes(r, "little")
        else:  # ed448
            dom = self.dom4(0,context)
            r = hashlib.shake_256(dom + hdigest2 + M).digest(912//8)
            r = int.from_bytes(r, "little")

        # Compute point R = [r]G and encode it
        R = self.G.mult(r)
        R_encoded = self.encode_point(R)

        # Compute `S`
        if self.curve == "ed25519":
            h = hashlib.sha512(R_encoded + Q + M).digest()
            h = int.from_bytes(h, "little")
        else:  # ed448
            h = hashlib.shake_256(dom + R_encoded + Q + M).digest(912//8)
            h = int.from_bytes(h, "little")

        S = (r + h * self.s) % self.n
        S_encoded = int(S).to_bytes(self.b//8, 'little')

        # Return the signature R || S
        return R_encoded + S_encoded




    def verify(self, M, signature, Q, context=b""):

        # Step 1: Decode the signature and public key
        if len(signature) != 2 * (self.b // 8):
            return False  # Invalid signature length

        R_bytes = signature[:self.b // 8]
        S_bytes = signature[self.b // 8:]

        # Decode S as an integer
        s = int.from_bytes(S_bytes, byteorder='little')

        # Verify that S is in range [0, n)
        if s < 0 or s >= self.n:
            return False  # S out of range

        # Decode R as a point
        try:
            R_x, R_y = self.decode_point(R_bytes)
            R = ed(curve=self.E, x=R_x, y=R_y)
        except ValueError:
            return False  # R decoding failed

        # Decode Q as a point
        try:
            Q_x, Q_y = self.decode_point(Q)
            Q_point = ed(curve=self.E, x=Q_x, y=Q_y)
        except ValueError:
            return False  # Q decoding failed

        # Step 2: Form HashData = R || Q || M
        HashData = R_bytes + Q + M

        # Step 3: Compute digest
```

```
        if self.curve == "ed25519":
            # Step 3.1: For Ed25519
            digest = hashlib.sha512(HashData).digest()
        else:  # ed448
            # Step 3.2: For Ed448
            dom = self.dom4(0,context)
            digest = hashlib.shake_256(dom + HashData).digest(912//8)

        # Interpret digest as little-endian integer t
        t = int.from_bytes(digest, byteorder='little')


        # Step 4: Check verification equation [2^c * S]G = [2^c]R + (2^c * t)Q
        # Calculate left side: [2^c * S]G
        left_side = self.G.mult(2 ** self.c * s)

        # Calculate right side: [2^c]R + (2^c * t)Q
        R_scaled = R.mult(2 ** self.c)
        Q_scaled = Q_point.mult(2 ** self.c * t)

        # Add the points
        right_side = R_scaled.copy()
        right_side.soma(Q_scaled)

        # Compare left and right sides
        return left_side.eq(right_side)
```

Exemplos e testes de aplicação

In [14]:

```
def test():
    # Initialize and test both curve types
    test_curves = ["ed25519","ed448"]

    for curve_name in test_curves:
        print(f"\n{'=' * 60}")
        print(f"Testing {curve_name}")
        print(f"{'=' * 60}")

        # Initialize the ECDSA object
        print("Initializing curve...")
        try:
            ecdsa = ECDSA(curve_name)
            print("✓ Successfully initialized curve")
        except Exception as e:
            print(f"✗ Failed to initialize curve: {e}")
            continue

        # Test key generation
        print("\nTesting key generation...")
        try:
            start_time = time.time()
            private_key, public_key = ecdsa.key_generation()
            end_time = time.time()

            print(f"✓ Key generation completed in {end_time - start_time:.4f} secc
            print(f"✓ Private key length: {len(private_key)} bytes")
            print(f"✓ Public key length: {len(public_key)} bytes")
        except Exception as e:
            print(f"✗ Key generation failed: {e}")
            continue

        # Test signing
```

```
print("\nTesting message signing...")
messages = [
    b"This is a test message",
    b"Another test with different content",
    b"A third test message with more data to sign"
]

for i, message in enumerate(messages):
    try:
        start_time = time.time()
        signature = ecdsa.sign(message, private_key, public_key)
        end_time = time.time()

        print(f"✓ Message {i+1} signed in {end_time - start_time:.4f} secc
        print(f"✓ Signature length: {len(signature)} bytes")

        # Test verification
        start_time = time.time()
        verification_result = ecdsa.verify(message, signature, public_key)
        end_time = time.time()

        if verification_result:
            print(f"✓ Signature {i+1} verified successfully in {end_time -
        else:
            print(f"✗ Signature {i+1} verification failed")
    except Exception as e:
        print(f"✗ Signing or verification failed for message {i+1}: {e}")

# Test with context for Ed448
if curve_name == "ed448":
    print("\nTesting with context (Ed448)...")
    context = b"test-context"
    try:
        message = b"Message with context"
        signature = ecdsa.sign(message, private_key, public_key, context)
        verification_result = ecdsa.verify(message, signature, public_key,

        if verification_result:
            print(f"✓ Signature with context verified successfully")
        else:
            print(f"✗ Signature with context verification failed")
    except Exception as e:
        print(f"✗ Signing or verification with context failed: {e}")

# Test with invalid signature
print("\nTesting with invalid signature...")
try:
    message = b"Original message"
    signature = ecdsa.sign(message, private_key, public_key)

    # Modify the signature to make it invalid
    modified_signature = bytearray(signature)
    modified_signature[0] ^= 0xFF  # Flip bits in the first byte
    modified_signature = bytes(modified_signature)

    verification_result = ecdsa.verify(message, modified_signature, public

    if not verification_result:
        print(f"✓ Invalid signature correctly rejected")
    else:
        print(f"✗ Invalid signature incorrectly accepted")
except Exception as e:
    print(f"✗ Invalid signature test failed: {e}")
```

```
        # Test with wrong message
        print("\nTesting with wrong message...")
        try:
            original_message = b"Original message"
            signature = ecdsa.sign(original_message, private_key, public_key)

            wrong_message = b"Wrong message"
            verification_result = ecdsa.verify(wrong_message, signature, public_ke

            if not verification_result:
                print(f"✓ Signature for wrong message correctly rejected")
            else:
                print(f"✗ Signature for wrong message incorrectly accepted")
        except Exception as e:
            print(f"✗ Wrong message test failed: {e}")


# Run the test if this script is executed directly
# if __name__ == "__main__":
#     print("Starting EdDSA implementation test...")
#     test()
```

In [15]:

```
!python3 ex2.py
```

```
Starting EdDSA implementation test...

============================================================
Testing ed25519
============================================================
Initializing curve...
✓ Successfully initialized curve

Testing key generation...
✓ Key generation completed in 0.0057 seconds
✓ Private key length: 32 bytes
✓ Public key length: 32 bytes

Testing message signing...
✓ Message 1 signed in 0.0038 seconds
✓ Signature length: 64 bytes
✓ Signature 1 verified successfully in 0.0088 seconds
✓ Message 2 signed in 0.0034 seconds
✓ Signature length: 64 bytes
✓ Signature 2 verified successfully in 0.0076 seconds
✓ Message 3 signed in 0.0030 seconds
✓ Signature length: 64 bytes
✓ Signature 3 verified successfully in 0.0069 seconds

Testing with invalid signature...
✓ Invalid signature correctly rejected

Testing with wrong message...
✓ Signature for wrong message correctly rejected

============================================================
Testing ed448
============================================================
Initializing curve...
✓ Successfully initialized curve

Testing key generation...
✓ Key generation completed in 0.0076 seconds
✓ Private key length: 57 bytes
✓ Public key length: 57 bytes

Testing message signing...
✓ Message 1 signed in 0.0078 seconds
✓ Signature length: 114 bytes
✓ Signature 1 verified successfully in 0.0155 seconds
✓ Message 2 signed in 0.0078 seconds
✓ Signature length: 114 bytes
✓ Signature 2 verified successfully in 0.0155 seconds
✓ Message 3 signed in 0.0077 seconds
✓ Signature length: 114 bytes
✓ Signature 3 verified successfully in 0.0159 seconds

Testing with context (Ed448)...
✓ Signature with context verified successfully

Testing with invalid signature...
✓ Invalid signature correctly rejected

Testing with wrong message...
✓ Signature for wrong message correctly rejected
```