Estruturas criptográficas 2024-2025

Grupo 02

Pg55986: Miguel Ângelo Martins Guimarães

Pg55997: Pedro Miguel Oliveira Carvalho

Universidade do Minho, Março 2025

Exercicio 2

Enunciado do exercicio:

1. Construir uma classe Python que implemente o EcDSA 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”.

Interpretação do enunciado:

O “standard” *FIPS186-5* especifica o Digital Signature Standard (DSS), que define os algoritmos que são utilizados para implementar assinaturas digitais, incluindo o EdDSA, bem como as curvas elípticas que devem ser utilizadas para a sua implementação.

> **Note:** Durante a interpretação do enunciado, surgiu uma dúvida em relação ao que é pedido. O exercício solicita a implementação do EcDSA, no entanto, nas alíneas seguintes, é indicado que devem ser utilizadas curvas de Edwards na implementação, o que não é compatível com o ECDSA, já que este normalmente utiliza curvas de Weierstrass. Assumimos então que a intenção do enunciado era pedir a implementação do EdDSA (Edwards-Curve Digital Signature Algorithm) com base no “standard” *FIPS186-5*.

Analise do FIPS186-5:

Contexto geral da utilização de assinaturas:

Um algoritmo de assinatura digital permite garantir a autenticidade, integridade e não repúdio de uma mensagem. Isto significa que é possível verificar que o emissor da mensagem é realmente quem afirma ser (autenticidade), que a mensagem não foi alterada durante a transmissão (integridade), e que o autor da mensagem não pode negar que a enviou (não repúdio).

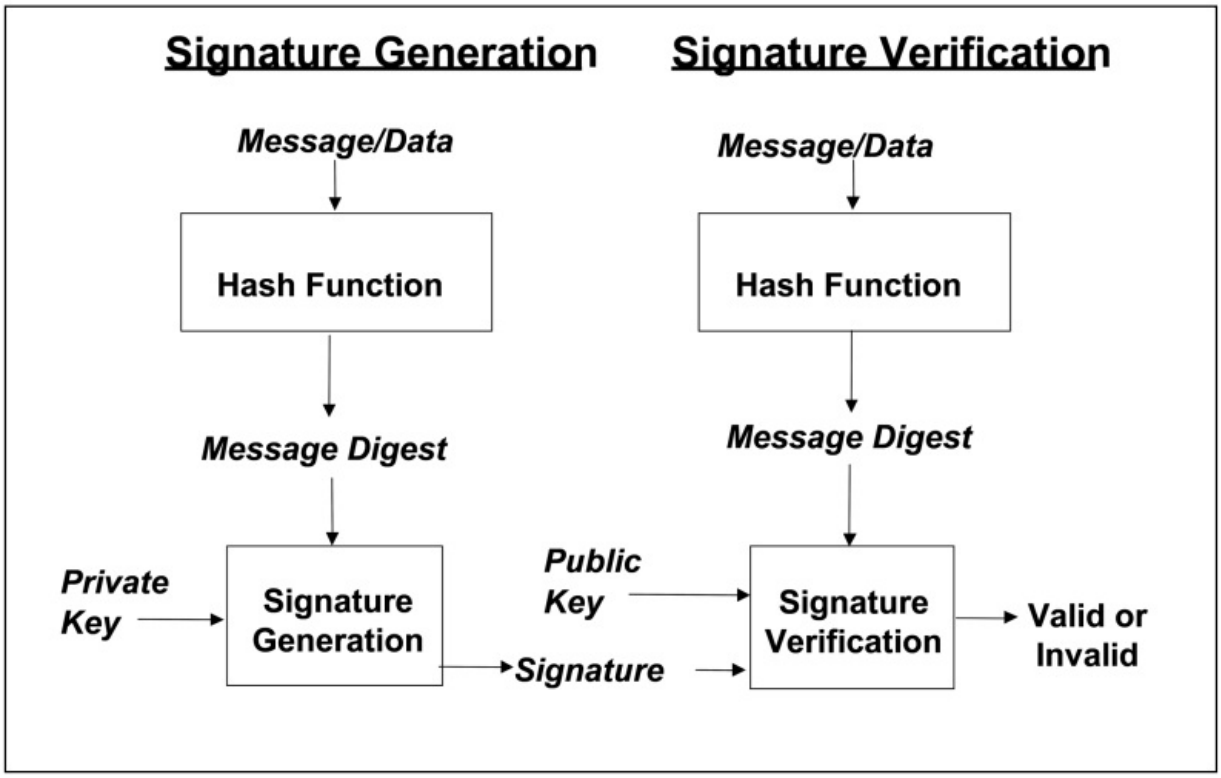

Figure 1: Digital Signature Processes

Nesta figura é possível observar o funcionamento das assinaturas digitais. Durante o processo de geração da assinatura, o emissor utiliza a sua chave privada e a mensagem que pretende enviar para criar uma assinatura digital.

Posteriormente, no processo de verificação da assinatura, o recetor utiliza a chave pública do emissor juntamente com os dados recebidos (mensagem + assinatura) para determinar se a assinatura é válida ou não.

Caso a assinatura seja considerada **inválida**, pode indicar uma das seguintes situações:

- O emissor não é um emissor reconhecido (poderá ter sido utilizada uma chave privada diferente da esperada);
- A integridade da mensagem foi comprometida, ou seja, a mensagem original foi alterada após ter sido assinada pelo emissor.

Deste modo, uma *assinatura digital* válida assegura simultaneamente a *autenticidade* do emissor e a *integridade* da mensagem recebida.

Escolha da curva:

De acordo com o *FIPS186-5* pode ser utilizada a curva **edwards-25519** ou **edwards-448** para implementar o **edDSA**.

A curva que decidimos escolher na implementação do algoritmo é a curva **ed-25519**,

Após uma leitura do *FIPS186-5* é possivel destacar que os algoritmos de hash utilizados na implementação dos algoritmos de assinatura digital devem utilizar funções de hash aprovadas. Para a implementação do edDSA com a curva **ed-25519** é utilizada a função de hash **SHA-512**.

Implementação do edDSA

Criação da curva ed-25519

Para a utilização/implementação da curva **ed-25519** foram utilizadas as classes implementadas pelo professor que se encontravam disponiveis no ficheiro *"Edwards0.ipynb"* fornecido nos anexos da UC.

O ficheiro contém duas classes, a classe **"Ed"** que permite criar uma instancia de uma curva de edwards permitindo obter um ponto gerador na curva de edwards, verificar se um ponto pertence à curva de edwards, entre outras operações. Esta classe implementa a curva de edwards recorrendo a curva de weiterstrass uma vez que esta já se encontra implementada no sage math por default.

A classe **"ed"** permite criar uma instancia de um ponto da curva de edwards, fornecendo metodos que permitem comparar pontos, duplicar, somar e a parte mais importante: multiplicar o ponto por um escalar. Esta operação de multiplicação de um ponto por um escalar é extremamente importante pois encontra-se na base de todo o algoritmo de edDSA.

Encoding

No EdDSA é importante representar de forma compacta e eficiente um ponto na curva de Edwards como uma sequência de bytes, reduzindo o espaço necessário para a sua representação.

7.2 Encoding

Parameter values used in EdDSA are coded as octet strings, and integers are coded using little-endian convention (i.e., a 32-octet string $h=h[0],...h[31]$ represents the integer $h[0] + 2^8 \bullet h[1] + ... + 2^{248} * h[31]$). The most significant byte is $h[31]$, and the least significant byte $h[0]$.

For a curve point (x,y) with coordinates in the range $0 \le x, y < p$, first encode the y-coordinate as a little-endian string of 32 octets for Ed25519 or 57 octets for Ed448. For Ed25519, the most significant bit of the final octet is always zero, while for Ed448, the final octet is always zero. To form the encoding of the point, copy the least significant bit of the x-coordinate to the most significant bit of the final octet.

The encoding of GF(p) is used to define "negative" elements of GF(p)—specifically, x is negative if the (b-1)-bit encoding of x is lexicographically larger than the (b-1)-bit encoding of $-x$.

- **Passo 1**: Interpretar o valor *y* como uma sequencia de bytes em formato little-endian.
- **Passo 2**: Copiar o bit menos significativo da coordenada x e colocá-lo no bit mais significativo do último byte de y.

Com esta operação é possivel representar um ponto usando apenas 32 bytes ao invés de 64 (porque o valor *x* não é armazenado de forma explicita).

O bit menos significativo do *x* permite distinguir entre os dois valores de *x* possiveis na curva de *edwards* durante a descodificação.

Assim é possivel obter o valor *x* original a partir de um formato compacto.

Decoding

No decoding será recebido como input um ponto da curva de edwards no formato codificado, e é desejado obter a coordenada *x* e *y* do ponto.

1. Interpret the octet string as an integer in little-endian representation. The most significant bit of this integer is the least significant bit of the x-coordinate, denoted as x_0. The y-coordinate is recovered simply by clearing this bit. If the resulting value is $\ge p$, decoding fails.

- **Passo 1**: Interpretar o valor codificado como uma sequencia de bytes em formato little-endian.
- **Passo 2**: Extrair o bit menos significativo do *x*, o valor *x0*, a partir do bit mais significativo do valor codificado. **Nota:** Relembramos que este bit permite distinguir entre os dois valores de *x* possiveis na curva de edwards durante o processo de descodificação.
- **Passo 3**: Obter a coordenada *y* colocando o bit mais significativo a 0. **Nota:** Nesta posição era onde se encontrava o valor *x0*.

2. To recover the x-coordinate, the curve equation requires $x^2 = (y^2 - 1) / (d\,y^2 - a) \pmod p$. The denominator is always non-zero mod p. Compute a square root to obtain x. Square roots can be computed using the Tonelli-Shanks algorithm (see NIST SP 800-186, Appendix E).

- **Passo 4**: Para obter o valor de *x* teremos que resolver a equação demonstrada na figura. Essa equação é obtida a partir da equação da curva de edwards. De forma a obter esta equação foi utilizado o algoritmo de Tonelli-Shanks tal como sugerido na figura. Após a aplicação do algoritmo de tonelli-shanks o valor de x é obtido. O algoritmo encontra-se explicado na secção abaixo.

Consoante o valor de x obtido e o valor x0 é necessário extrair o valor de *x* correto, como tal é realizada uma operação de `p-x` para obter o outro valor de *x* caso a paridade seja diferente.

Tonelli-shanks

Seguindo o **"NIST SP 800-186"**, tal como sugerido na secção do decoding, é possivel implementar o algoritmo de tonelli-shanks que nos permite obter um valor de *x* a partir de uma equação no formato `x2 ≡ n mod p`.

Find q and s (with q odd), such that $p - 1 = q2^s$ by factoring out the powers of 2.

- **Passo 1**: Descobrir os valores q e s, de forma a que a equação `p-1 = q*2^s` seja verdade. Para resolver esta equação primeiro foi inicializado o valor *q* igual a `p-1`, e o valor *s* igual a 0. Depois o valor Q foi iterado, isto é, para o valor de Q atual verificamos qual o resto da sua divisão por 2 a cada iteração. Sempre que o valor *Q* é divisivel por 2 o valor *s* é incrementado em 1 unidade. Assim foi possivel simplificar o valor `p-1` para o formato `p-1 = q*2^s` obtendo assim os valores finais de *q* e de *s*.

Check to see if $n^q = 1$. If so, then the root $x = n^{(q+1)/2} \bmod p$.

- **Passo 2**: Verificar se `n^q = 1`. Caso isso acontecer então possuimos imediatamente um valor de *x*.

Otherwise, select a z, which is a quadratic non-residue modulo p. The Legendre symbol $\left(\frac{a}{p}\right)$, where p is an odd prime and a is an integer, can be used to test candidate values for z to see if a value of -1 is returned.

- **Passo 3**: Para esta etapa utilizamos a identidade de euler `(z/p) == z^(p-1/2)`. O algoritmo utilizado nesta secção é relativamente simples, iteramos todos os valores de *z* de 2 até *p*, e assim que a equação se tornar verdade possuiamos um candidato valido de z.

Search for a solution as follows:

1. Set $x = n^{(q+1)/2} \bmod p$.
2. Set $t = n^q \bmod p$.
3. Set $m = s$.
4. Set $c = z^q \bmod p$.

- **Passo 4**: Definir as variaveis x, t, m e c para os valores que se encontram na figura.

5. While $t \neq 1$, repeat the following steps:
   a) Using repeated squaring, find the smallest i, such that $t^{2^i} = 1$, where $0 < i < m$.
   For example:
   Let $e = 2$.
   Loop for $i = 1$ until $i = m$.
   If $t^e \bmod p = 1$, then exit the loop.

- **Passo 5**: Iterar todos os valores de i possiveis até que a equação `t^2^i = 1` seja verdade.

b) Update values:

$$b = c^{2^{m-i-1}} (\bmod p)$$
$$x = xb (\bmod p)$$
$$t = tb^2 (\bmod p)$$
$$c = b^2 (\bmod p)$$
$$m = i$$

Durante este processo as variaveis *x*, *t*, *m* e *c* são atualizadas.

Todo este processo é repetido enquanto *t* for diferente de 1. Assim que *t* for igual a 1 possuimos os valores de x que procuravamos.

Gerar um par de chaves

A seguir encontra-se descrito como foi implementada a etapa de gerar um par de chaves valido para utilização com o edDSA.

Inputs:

1. b: for Ed25519 $b = 256$, while for Ed448 $b = 456$.
2. *requested_security_strength*: 128 bits of security strength for Ed25519 or 224 bits of security strength for Ed448.
3. H: SHA-512 for Ed25519 or SHAKE256 for Ed448.

Output: Valid public-private key pair (d, Q) for domain parameters D.

Na figura acima encontram-se os dados de entrada utilizados para a criação da chave privada e da chave publica.

Para a criação da chave privada e publica foram seguidos os passos identificados no *FIPS186-5*:

Process:

1. Obtain a string of b *bits* from an **approved RBG** with a security strength of *requested_security_strength* or more. The private key d is this string of b bits.
2. Compute the hash of the private key d, $H(d) = (h_0, h_1, ..., h_{2b-1})$ using SHA-512 for Ed25519 and SHAKE256 for Ed448 ($H(d)$= SHAKE256(d, 912)). $H(d)$ may be pre-computed. Note $H(d)$ is also used in the EdDSA signature generation; see Section 7.6.
3. The first half of $H(d)$, (i.e. *hdigest1* $= (h_0, h_1, \ldots, h_{b-1})$) is used to generate the public key. Modify *hdigest1* as follows:
   - 3.1 For Ed25519, the first three bits of the first octet are set to zero; the last bit of the last octet is set to zero; and the second to last bit of the last octet is set to one. That is, $h_0 = h_1 = h_2 = 0$, $h_{b-2} = 1$, and $h_{b-1} = 0$.
   - 3.2 For Ed448, the first two bits of the first octet are set to zero; all eight bits of the last octet are also set to zero; and the last bit of the second to last octet is set to one. That is, $h_0 = h_1 = 0$, $h_{b-9} = 1$, and $h_i = 0$ for $b - 8 \leq i \leq b - 1$.
4. Determine an integer s from *hdigest1* using little-endian convention (see Section 7.2).
5. Compute the point $[s]G$. The corresponding EdDSA public key Q is the encoding (See Section 7.2) of the point $[s]G$.

Implementação:

- **Passo 1**: É necessário obter uma string de b bits proveniente de um gerador de números aleatórios aprovado, que ofereça um nível de segurança compatível com o parâmetro *requested_security_strength*. Como o gerador de *bits* considerado mais robusto é aquele fornecido pelo sistema operacional, conforme mencionado pelo professor, optou-se por utilizar a biblioteca *secrets* do Python, que utiliza as fontes de entropia do próprio sistema para gerar valores aleatórios de forma segura.
- **Passo 2**: Calcular o hash da chave privada utilizando **SHA-512**. Foi utilizado o **SHA-512** utilizando o *cryptography*.
- **Passo 3**: Extrair a primeira metade do hash criado.
- **Passo 3.1**: Aplicar transformações na metade extraída, garantindo que o valor esteja no formato correto para uso na curva Ed25519:
  - Definir os três primeiros bits do primeiro octeto como zero (foi implementado utilizando a operação *AND* com `248`, que equivale a `11111000` em binário).
  - Definir o último bit do último octeto como zero (foi implementado utilizando a operação *AND* com `127`, que equivale a `01111111` em binário).
  - Definir o penúltimo bit do último octeto como um (foi implementado utilizando a operação *OR* com `64`, que equivale a `01000000` em binário).
- **Passo 4**: Interpretar o resultado final como um inteiro *s*, assumindo uma codificação *little-endian*.
- **Passo 5**: Calcular o ponto **[s]G**, onde *G* é o gerador da curva. Essa operação consiste em multiplicar *G* pelo escalar *s*, ou seja, calcular *G* somado a si mesmo *s* vezes, utilizando a operação `G.mult(s)`, que implementa multiplicação escalar em curvas elípticas.

Criação de uma assinatura

A seguir iremos expor como foi implementada a criação de assinaturas edDSA.

Inputs:

1. Bit string M to be signed
2. Valid public-private key pair (d, Q) for domain parameters D
3. H: SHA-512 for Ed25519 or SHAKE256 for Ed448
4. For Ed448, a string *context* set by the signer and verifier with a maximum length of 255 octets; by default, *context* is the empty string

Na figura acima é possivel visualizar quais são os valores de entrada utilizados na criação de uma assinatura digital recorrendo ao **edDSA**.

- É necessário fornecer uma mensagem que será assinada.
- Um par de chaves publica/privada que serão utilizadas no processo de assinatura.
- E é exigida a utilização de **SHA-512** como função de *hash*.

Implementação:

1. Compute the hash of the private key d, $H(d) = (h_0, h_1, ..., h_{2b-1})$ using SHA-512 for Ed25519 and SHAKE256 for Ed448 ($H(d)$= SHAKE256(d, 912)). $H(d)$ may be pre-computed.

- **Passo 1**: Calcular a hash da chave privada recorrendo ao SHA-512. Este passo foi implementado recorrendo à biblioteca *cryptography*.

2. Using the second half of the digest $hdigest2 = h_b \| ... \| h_{2b-1}$, define:

- **Passo 2**: Extrair a segunda metade do hash calculado. **Nota:** Relembro que a primeira metade do hash da chave privada é utilizado para gerar a chave publica.

2.1 For Ed25519, $r = SHA\text{-}512(hdigest2 \| M)$; r will be 64-octets.

- **Passo 2.1**: Concatenar a segunda metade do hash com a mensagem. Aplicar a função de hash *SHA-512* ao resultado da concatenação. O *hash* obtido é o valor *r* que vai ser usado no proximo passo.

3. Compute the point $[r]G$. The octet string R is the encoding of the point $[r]G$.

- **Passo 3**: Calcular o ponto **[r]G**, onde *G* é o gerador da curva. O ponto obtido é apelidado de ponto *R*. **Nota:** Relembro que esta operação consiste em multiplicar *G* pelo escalar *r*, obtendo assim um novo ponto sobre a curva de *edwards*.

4. Derive s from $H(d)$ as in the key pair generation algorithm. Use octet strings R, Q, and M to define:

- **Passo 4**: Obter um valor **s** a partir do ponto obtido, utilizando o mesmo processo que na geração das chaves. Para replicar essa etapa iremos utilizar novamente a primeira metade do hash, e com ela vamos realizar as mesmas manipulações aos bits dessa metade tal como na fase de criação de chaves.

  Ou seja:

  - Definir os três primeiros bits do primeiro octeto como zero (foi implementado utilizando a operação *AND* com `248`, que equivale a `11111000` em binário).
  - Definir o último bit do último octeto como zero (foi implementado utilizando a operação *AND* com `127`, que equivale a `01111111` em binário).
  - Definir o penúltimo bit do último octeto como um (foi implementado utilizando a operação *OR* com `64`, que equivale a `01000000` em binário).

    Depois passamos este valor para inteiro garantindo que respeitamos a notação little-endian.

4.1 For Ed25519, $S = (r + SHA\text{-}512(R \| Q \| M) * s) \bmod n$.

- **Passo 4.1**: Vamos calcular o valor **S** recorrendo à equação demonstrada na figura acima. Primeiramente devemos concatenar o valor **R** (valor codificado obtido no passo 3) com o valor **Q** (chave publica) e o valor **M** que representa a mensagem. Aplicamos a função de hash **SHA-512** ao resultado da concatenação e multiplicarmos esse valor por **s** (obtido no passo anterior). Somamos o valor **r** (obtido no passo 2.1) ao resultado da operação anterior, finalizando

a equação com `mod n`.

The octet string S is the encoding of the resultant integer.

O resultado final é obtido codificando o valor S.

5. Form the signature as the concatenation of the octet strings R and S.

- **Passo 5**: Para finalizar o processo de criação da assinatura concatenamos o valor **R** e **S** e assim obtemos a assinatura da mensagem **M**.

Verificação da assinatura

A seguir será demonstrado como foi implementada a verificação de assinaturas **edDSA**.

Inputs:

1. Message M
2. Signature $R \| S$ where R and S are octet strings
3. Purported signature verification key Q that is valid for domain parameters D
4. For Ed448, a string *context* set by the signer and verifier with a maximum length of 255 octets; by default, *context* is the empty string

Na figura acima encontram-se representados os dados de entrada da função de verificação de assinaturas **edDSA**. Pode-se verificar que os valores de entrada são:

- Uma mensagem que será utilizada no processo de verificação. **Nota:** Para a assinatura ser considerada valida a mensagem utilizada tem de ser a mesma mensagem que foi utilizada durante a criação da assinatura.
- Uma assinatura que será alvo do processo de verificação.
- Uma chave publica. **Nota:** Para a assinatura ser considerada valida a chave publica utilizada tem de ser a mesma que na criação da assinatura.

Implementação

1. Decode the first half of the signature as a point R and the second half of the signature as an integer s. Verify that the integer s is in the range of $0 \leq s < n$. Decode the public key Q into a point Q'. If any of the decodings fail, output "reject".

- **Passo 1**:
  - O primeiro é extrair a primeira metade da assinatura para obter o valor **R**, a segunda metade da assinatura fornece o valor **s**. **Nota:** É necessario descodificar os valores *S* e *R*.
  - Depois verifica-se se o valor **s** se encontra no intervalo de `0<=s<=n`. **Nota:** Se este passo falhar o processo de verificação é rejeitado. **Nota:** Se este passo falhar o processo de verificação é rejeitado.
  - Descodificamos a chave publica para obter um ponto. **Nota:** Se este passo falhar o processo de verificação é rejeitado.

2. Form the bit string *HashData* as the concatenation of the octet strings R, Q, and M (i.e., $HashData = R \| Q \| M$).

- **Passo 2**: Concatenar os valores **R**, **Q** (chave publica) e **M** (mensagem).

3.1 For Ed25519, compute $digest = SHA\text{-}512(HashData)$.

- **Passo 3**: Aplicar a função de hash SHA-512 ao resultado da concatenação. Nota: Este passo foi implementado recorrendo à biblioteca *cryptography*.

Interpret *digest* as a little-endian integer t.

É necessário interpretar este resultado como um little-endian. Este é o valor *t*.

4. Check that the verification equation $[2^c * S]G = [2^c]R + (2^c * t)Q$. Output "reject" if verification fails; output "accept" otherwise.

- **Passo 3**: Para finalizar o processo é realizada uma verificação da equação demonstrada na figura. Primeiramente vamos calcular o lado direito, onde calculamos `2^c`. O valor **c** é 3 tal como enunciado no standard. Multiplicamos 2^c por S e com esse resultado multiplicamos o gerador **G** por esse mesmo escalar. Do lado direito da equação vamos multiplicar o ponto **R** pelo escalar `2^c` e somar esse resultado com a aplicação do escalar resultante de `(2^c * t)` ao ponto **Q**.

Deste processo surge o resultado final que é "Accept" ou "Reject" (devolvemos true ou false na nossa implementação), que indicam se assinatura é valida ou invalida.

Testes:

Para validar este exercicio foram realizados os seguintes testes:

Teste de assinatura valida

```
edDSA = EdDSA()

print("----------- Gerar chaves -----------")
private_key, public_key = edDSA.genKeyPair()

print("----------- Gerar assinatura -----------")
sig = edDSA.sign(b"mensagem_fixe", public_key, private_key)

print("----------- Verificacao assinatura -----------")
ver = edDSA.verify(b"mensagem_fixe", sig, public_key)
```

```
Ponto gerador utilizado:
[EDDSA]:15112221349535400772501151409588531511454012693041857206046113283949847762202
[EDDSA]:46316835694926478169428394003475163141307993866256225615783033603165251855960
----------- Gerar chaves -----------
[ENCODE] Recebido x: 11876353165678250154733634158265054388096120154669940963723586372808421327085
[ENCODE] Recebido y: 16144327453819489836664927387551180944395011801231305214195643594154550573156
[ENCODE] Output: b'd8mE\xdc6\xd5$\x81E\xcd\xbb\xe8\xa2\n\xf8\xb06}?\xcb\xfb\x92\x87\x18\x07y\xe7\x91
[KEYGEN] Public key (hex): 64386d45dc36d5248145cdbbe8a20af8b0367d3fcbfb9287180779e7925db1a3
[KEYGEN] Private key (hex): dc6e524d179e0e063a36b42de8d03cf1cf533081774909c725370ad7976c15df
----------- Gerar assinatura -----------
[ENCODE] Recebido x: 23969913185232678346035397139966937068722292456995493348556688842435495914212
[ENCODE] Recebido y: 26172591146296764729295708849520502381279329963788129633096033377893511046048
[ENCODE] Output: b'\xa0\xa33\xae\xd7\xec\x89\n\xcc\x9a0C\x1ft\xa7j[\xaep\xc9\xfe\xf3\xde\xe0\xde\xe9
[SIGNATURE] Signature (hex):b"\xa0\xa33\xae\xd7\xec\x89\n\xcc\x9a0C\x1ft\xa7j[\xaep\xc9\xfe\xf3\xde
----------- Verificacao assinatura -----------
[DECODE] Obtido x: 23969913185232678346035397139966937068722292456995493348556688842435495914212
[DECODE] Obtido y: 26172591146296764729295708849520502381279329963788129633096033377893511046048
[DECODE] Obtido x: 11876353165678250154733634158265054388096120154669940963723586372808421327085
[DECODE] Obtido y: 16144327453819489836664927387551180944395011801231305214195643594154550573156
[VERIFY] Assinatura VÁLIDA!
```

Teste de assinatura invalida com mensagem manipulada/diferente

```
edDSA = EdDSA()

print("----------- Gerar chaves -----------")
private_key, public_key = edDSA.genKeyPair()

print("----------- Gerar assinatura -----------")
sig = edDSA.sign(b"mensagem_fixe", public_key, private_key)

print("----------- Verificacao assinatura -----------")
ver = edDSA.verify(b"mensagem_fixe_manipulada", sig, public_key)
```

```
Ponto gerador utilizado:
[EDDSA]:15112221349535400772501151409588531511454012693041857206046113283949847762202
[EDDSA]:46316835694926478169428394003475163141307993866256225615783033603165251855960
----------- Gerar chaves -----------
[ENCODE] Recebido x: 499805838448616718082463092025608112595382107905508832778911758197383330539O1
[ENCODE] Recebido y: 594850166314304166992390080383112681590416650365254843320809104449169055380З
[ENCODE] Output: b'\xcb\xb1\xd8k\xacC[\x99*+\xd3[9=\x92\x90\x128\x00\xd2\xfa\xfej\x0c\xe7\xbb#\x90\x8d\xbb&\x8d'
[KEYGEN] Public key (hex): cbb1d86bac435b992a2bd35b393d9290123800d2fafe6a0ce7bb23908dbb268d
[KEYGEN] Private key (hex): d28260ae2911713cc6cfa8ce48481b869910f876763fd68f8cea4e24ad485d11
----------- Gerar assinatura -----------
[ENCODE] Recebido x: 492759597914742372066567441928591763313883869728636245637881642804668148913Ol
[ENCODE] Recebido y: 222144972271553518184633416857188925763173509260860711164091502817796004245O7
[ENCODE] Output: b';\xfe\x8f\xdfWI\xe0J\xf4\xbbzRoq\xbf\x9a\rf\x16\ns\x91lP^dr~\xb9\xf5\x1c\xb1'
[SIGNATURE] Signature (hex):b';\xfe\x8f\xdfWI\xe0J\xf4\xbbzRoq\xbf\x9a\rf\x16\ns\x91lP^dr~\xb9\xf5\x1c\xb1\xbcyt\x
----------- Verificacao assinatura -----------
[DECODE] Obtido x: 492759597914742372066567441928591763313883869728636245637881642804668148913Ol
[DECODE] Obtido y: 222144972271553518184633416857188925763173509260860711164091502817796004245O7
[DECODE] Obtido x: 499805838448616718082463092025608112595382107905508832778911758197383330539O1
[DECODE] Obtido y: 594850166314304166992390080383112681590416650365254843320809104449169055380З
[VERIFY] Assinatura INVÁLIDA!
```

Teste de assinatura invalida com uma chave publica diferente

```python
edDSA = EdDSA()

print("----------- Gerar chaves -----------")
private_key, public_key = edDSA.genKeyPair()

print("----------- Gerar assinatura -----------")
sig = edDSA.sign(b"mensagem_fixe", public_key, private_key)

print("----------- Verificacao assinatura -----------")
try:
    ver = edDSA.verify(b"mensagem_fixe", sig, b"chave_falsa") # AQUI ENCONTRA-SE UTILIZADA A CHAVE PRIVADA COMO SE FOSSE PUBLICA (Tem de falhar)
except Exception as e:
    print("Erro capturado:", e)
```

```
Ponto gerador utilizado:
[EDDSA]:15112221349535400772501151409588531511454012693041857206046113283949847762202
[EDDSA]:46316835694926478169428394003475163141307993866256225615783033603165251855960
----------- Gerar chaves -----------
[ENCODE] Recebido x: 475116705312651022762741815368068380332278635163557006566681292412679986095Ol
[ENCODE] Recebido y: 171002690575645941325855361270017108830945680792728343638524967801965665537Ol
[ENCODE] Output: b"\xc1T|s'd\x13a\x08\xd73\xa9\xf9\xcbKU\x8f\x13\xe72\\8\xc9\xb5\xa8\x1d\xd3\xaa\xc7h\xce\xa5"
[KEYGEN] Public key (hex): c1547c732764136108d733a9f9cb4b558f13e7325c38c9b5a81dd3aac768cea5
[KEYGEN] Private key (hex): e8831340da3cf614c3d6b69161a5fbb532aa075323714ac7eb55e42f7930498e
----------- Gerar assinatura -----------
[ENCODE] Recebido x: 23794640295059121021515562141055073398269372490198460042197622561616251520830
[ENCODE] Recebido y: 53430956194289666268695706185829335624016581371512154456938178898751428745268
[ENCODE] Output: b"4\xd8\x13\xed\x1e\xf8M)\xd0\xb4y'u\x0e=d\xfa3\xcf\xb8U`\x01\x88\xd1\x07\x9a\x87y\xd9 v"
[SIGNATURE] Signature (hex):b'4\xd8\x13\xed\x1e\xf8M)\xd0\xb4y\'u\x0e=d\xfa3\xcf\xb8U`\x01\x88\xd1\x07\x9a\x87y
----------- Verificacao assinatura -----------
[DECODE] Obtido x: 23794640295059121021515562141055073398269372490198460042197622561616251520830
[DECODE] Obtido y: 53430956194289666268695706185829335624016581371512154456938178898751428745268
[DECODE] Obtido x: 458250255518362832479197804050293779519605656231172089178501744452420165415Ol
[DECODE] Obtido y: 11781087591491095846357616З
[VERIFY] Assinatura INVÁLIDA!
```

Teste de assinatura invalida com um certificado manipulado/diferente

```python
edDSA = EdDSA()

print("------------ Gerar chaves ------------")
private_key, public_key = edDSA.genKeyPair()

print("------------ Gerar assinatura ------------")
sig = edDSA.sign(b"mensagem_fixe", public_key, private_key)

# MANIPULAÇÃO DA ASSINATURA
sig_arr = bytearray(sig)
sig_arr[10] = (sig_arr[10] + 1) % 256  # Incrementar um byte
sig = bytes(sig_arr)

print("------------ Verificacao assinatura ------------")
try:
    ver = edDSA.verify(b"mensagem_fixe", sig, public_key)
except Exception as e:
    print("Erro capturado:", e)
```

```
Ponto gerador utilizado:
[EDDSA]:15112221349535400772501151409588531511454012693041857206046113283949847762202
[EDDSA]:46316835694926478169428394003475163141307993866256225615783033603165251855960
----------- Gerar chaves -----------
[ENCODE] Recebido x: 31187398514261303534992145108944009227888261313106575083414035335906982037333
[ENCODE] Recebido y: 90266935180613402608278879716178692335123663006969987682738754407652479380 7
[ENCODE] Output: b'\xcf/\x9d\xf6\x1e\xd4\x1a\xd2\x18\xc8\x89\rS\x83\xa2! 1\xe6\xc00\xeb\x8a\xd1\x94\xa2\x18\x1e\x8b\xe4\xfe\x81'
[KEYGEN] Public key (hex): cf2f9df61ed41ad218c8890d5383a2212031e6c030eb8ad194a2181e8be4fe81
[KEYGEN] Private key (hex): 94f012a4836127f5309c678c2f1b1f9301410c4e8a79e1cad996ef3a4c539fd3
----------- Gerar assinatura -----------
[ENCODE] Recebido x: 560135430924251469763847636055405604260497946509010295758932112475835611065 35
[ENCODE] Recebido y: 185367060345370614493835453892610075843935192040563718551088934475820833866 31
[ENCODE] Output: b'\x07\r\x13\xd5\xb0\xae6\x9fP\xa4\xf5d;\xc9\xcc\xcd,X\xe1\xb7=\xdd\xdf\xc2^_.^`g\xfb\xa8'
[SIGNATURE] Signature (hex):b'\x07\r\x13\xd5\xb0\xae6\x9fP\xa4\xf5d;\xc9\xcc\xcd,X\xe1\xb7=\xdd\xdf\xc2^_.^`g\xfb\xa8\xe5\x02\x02
----------- Verificacao assinatura -----------
[DECODE] Obtido x: 11109774849297359599128886124046214086098283191711261322884053205393068763947
[DECODE] Obtido y: 18536706034537061449383545389261007584393519204056373064034713062211258092807
[DECODE] Obtido x: 31187398514261303534992145108944009227888261313106575083414035335906982037333
[DECODE] Obtido y: 90266935180613402608278879716178692335123663006969987682738754407652479380 7
[VERIFY] Assinatura INVÁLIDA!
```

Referencias:

[1] NIST, "FIPS 186-5: Digital Signature Standard (DSS)," National Institute of Standards and Technology, Gaithersburg, MD, USA, 2023. [Online]. Disponível em: https://csrc.nist.gov/publications/fips/fips186-5

[2] NIST, "SP 800-186: Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Cryptography," National Institute of Standards and Technology, Gaithersburg, MD, USA, 2010. [Online]. Disponível em: https://csrc.nist.gov/publications/detail/sp/800-186/final

[3] S. Josefsson and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", Internet Research Task Force (IRTF), Request for Comments (RFC) 8032, Dec. 2017. [Online]. Disponível em: https://doi.org/10.17487/RFC8032