Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Implementação de módulo para geração e verificação de assinaturas digitais EdDSA

Autor: Lucas Penido Antunes
Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF
2021

Lucas Penido Antunes

Implementação de módulo para geração e verificação de assinaturas digitais EdDSA

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Tiago Alves da Fonseca

Brasília, DF

2021

Lucas Penido Antunes
Implementação de módulo para geração e verificação de assinaturas digitais EdDSA/ Lucas Penido Antunes. – Brasília, DF, 2021-
83 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Tiago Alves da Fonseca

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2021.

1. Assinatura digital. 2. Curva elíptica. 3. Criptografia. 4. ICP-Brasil. 5. EdDSA. 6. Dart. I. Prof. Dr. Tiago Alves da Fonseca. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação de módulo para geração e verificação de assinaturas digitais EdDSA

CDU 02:141:005.6

Lucas Penido Antunes

Implementação de módulo para geração e verificação de assinaturas digitais EdDSA

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 04 de novembro de 2021:

Prof. Dr. Tiago Alves da Fonseca
Orientador

Dr. José Antônio Carrijo Barbosa
Convidado 1

Prof. Dr. Edson Alves da Costa Junior
Convidado 2

Brasília, DF
2021

Nós só podemos ver um pouco do futuro,
mas o suficiente para perceber que há muito a fazer.
(Alan Turing)

Resumo

Considerando a importância dos meios digitais e o grande número de atividades que as pessoas realizam por meio da *internet*, como pagar contas, realizar compras e transações *online*, acessar sites, entre outras, nada disso seria possível sem a utilização de assinaturas digitais. Com elas pode-se garantir a autoria de uma mensagem e que essa mensagem não foi alterada durante o momento em que é enviada pelo remetente até ser recebida pelo destinatário. Por exemplo, em uma transação *online*, pode-se garantir que nem o valor da transação, nem o destinatário foram alterados. E para ser possível a criação de assinaturas digitais, vários algorítimos foram criados e um deles, proposto inicialmente em 2011 e aprimorado posteriormente em 2015, é o algoritmo de assinatura digital da curva de Edwards, que utiliza curvas torcidas de Edwards. Por ser um algoritmo novo, ele ainda não foi implementado em algumas linguagens de programação, tornando seu uso impossibilitado por pessoas que desejam utilizá-lo em seus projetos. Uma dessas linguagens de programação que não possui muitas implementações é a Dart, uma linguagem cuja demanda tem crescido devido à possibilidade de criação de aplicações para *web*, *mobile* e *desktop*. Neste trabalho, propõe-se a implementação, na linguagem de programação Dart, dos algoritmos Ed448 e Ed521, os quais foram, respectivamente, utilizados nas Cadeias V6 e V7 da Infraestrutura de Chaves Públicas Brasileira. Dessa forma, serão elaboradas documentações detalhadas dos módulos para que futuros pesquisadores que queiram realizar trabalhos parecidos possam utilizá-los como um guia e assim facilitar a implementação do algoritmo em outras linguagens de programação. Ao finalizar a implementação em Dart, serão realizadas validações cruzadas de um algoritmo já implementado em Python de modo a certificar que os módulos foram implementados corretamente.

Palavras-chave: assinatura digital. curva elíptica. criptografia. ICP-Brasil. EdDSA. Dart

Abstract

Considering the importance of digital media and the large number of activities that people do through the internet, such as paying bills, making purchases and online transactions, accessing websites, and more, none of this would be possible without the use of digital signatures. With them, it is possible to guarantee the authorship of a message and that this message has not been altered during the moment it is sent by the sender until it is received by the recipient. For instance, in an online transaction, one can ensure that neither the transaction amount nor the recipient has been changed. And to make it possible to create digital signatures, several algorithms have been created and one of them, initially proposed in 2011 and later improved in 2015, is the Edwards curve digital signature algorithm, which uses Edwards twisted curves. As it is a new algorithm, it has not been implemented in some programming languages yet, making its use impossible for people who want to use it in their projects. One of those programming languages that doesn't have many implementations is Dart, a language whose demand has grown due to the possibility of creating applications for web, mobile and desktop. In this work, it is proposed the implementation, in Dart programming language, of the Ed448 and Ed521 algorithms, which were used, respectively, in the V6 and V7 chains of Infraestrutura de Chaves Públicas Brasileira. In this way, the detailed documentation of the modules will be elaborated so that future researchers who want to carry out similar work can use them as a guide and thus facilitate the implementation of the algorithm in other programming languages. When finishing the Dart implementation, cross validations of an algorithm already implemented in Python will be performed in order to certify that the modules were implemented correctly.

Key-words: digital signature. elliptical curve. cryptography. ICP-Brasil. EdDSA. Ed488. Ed521. Dart.

Lista de ilustrações

Figura 1 – Comparação de popularidade entre Flutter e React Native. Adaptado de Google (2021). 23
Figura 2 – Representação simplificada dos elementos essenciais do processo de assinatura digital (NAKOV, 2018, tradução nossa). 26
Figura 3 – Hierarquia da ICP-Brasil (Benefícios e Aplicações da Certificação Digital, c2013). 32
Figura 4 – Fluxo de trabalho segundo o GitFlow 34
Figura 5 – Adicionando suporte ao Ed521 na biblioteca ECPy. 38

Lista de tabelas

Tabela 1 – Comparação do tempo médio de execução de cada função do Ed448 em diferentes linguagens 39
Tabela 2 – Comparação do tempo médio de execução de cada função do Ed521 em diferentes linguagens 40

Lista de abreviaturas e siglas

AC-Raiz	Autoridade Certificadora Raiz
DSA	*Digital Signature Algorithm*
ECDSA	*Elliptic Curve Digital Signature Algorithm*
EdDSA	*Edwards-curve Digital Signature Algorithm*
ICP-Brasil	Infraestrutura de Chaves Públicas Brasileira
RSA	Rivest–Shamir–Adleman

Lista de símbolos

μs	Microssegundos

Sumário

1 INTRODUÇÃO 21
1.1 Contextualização 21
1.2 Justificativa 22
1.3 Objetivos 23
1.3.1 Objetivo Geral 23
1.3.2 Objetivos Específicos 23
1.4 Estrutura do Trabalho 24

2 FUNDAMENTAÇÃO TEÓRICA 25
2.1 Sistemas Criptográficos de Chave Pública 25
2.2 Assinaturas Digitais 25
2.2.1 Elliptic Curve Digital Signature Algorithm (ECDSA) 27
2.2.1.1 Geração de Par de Chaves ECDSA 27
2.2.1.2 Assinatura ECDSA 27
2.2.1.3 Verificação ECDSA 28
2.2.2 Edwards-curve Digital Signature Algorithm (EdDSA) 28
2.2.2.1 Geração de Par de Chaves EdDSA 31
2.2.2.2 Assinatura EdDSA 31
2.2.2.3 Verificação EdDSA 31
2.3 Infraestrutura de Chaves Públicas Brasileira 32

3 METODOLOGIA 33
3.1 Metodologia de Desenvolvimento 33
3.2 Políticas do Repositório 33
3.3 Linguagens e Ambiente de Desenvolvimento 34
3.4 Testes 35
3.5 Integração Continuada e Implantação Continuada 35

4 RESULTADOS E DISCUSSÕES 37

5 CONCLUSÕES E PROPOSIÇÕES 41

REFERÊNCIAS 43

APÊNDICES 45

APÊNDICE A – DOCUMENTAÇÃO ED448 47

APÊNDICE B – DOCUMENTAÇÃO ED521 61

APÊNDICE C – VETOR DE TESTES PARA ED521 75

1 Introdução

1.1 Contextualização

A criptografia, um dos campos da criptologia, diz respeito à troca privada de mensagens entre dois participantes, em que um adversário não deve saber nada sobre o conteúdo dessas mensagens (RIVEST, 1991). Ela, porém, engloba não somente a confidencialidade mas também a integridade, autenticação e o não repúdio da mensagem.

Apesar de ser um assunto bastante discutido atualmente devido ao grande volume de dados e informações circulando a todo o momento, a história da criptografia começa há muito tempo. Os sistemas criptográficos na criptografia clássica consistiam em realizar a troca de letras por símbolos ou a mudança de posição entre os caracteres.

Um dos sistemas mais antigos que se tem conhecimento, datado em torno de 2000 antes de Cristo, era utilizado pelos egípcios para dificultar a leitura de sua escrita e consistia em trocas de símbolos (COHEN, 1990). Outro sistema é a cifra de César, criada pelo Imperador Júlio César, que consistia na substituição de uma letra do alfabeto pela terceira adiante.

Posteriormente, os sistemas criptográficos começaram a utilizar uma chave secreta para realizar a cifração de uma mensagem, um desses sistemas é a Cifra de Vigenère. Essa cifra consiste em deslocar cada letra do alfabeto um número fixo de lugares e esse deslocamento é determinado por uma chave (BRUEN; FORCINITO, 2011). Por exemplo, na cifração de uma mensagem pela chave “abc” a primeira letra não seria deslocada, pois, o “a” equivale a 0, a segunda seria deslocada uma posição, “b” equivale a 1, a terceira duas posições e assim sucessivamente até todas as letras da mensagem fossem deslocadas.

Com o passar do tempo, a cifração de uma mensagem foi sendo aprimorada junto ao desenvolvimento de máquinas, formas mais elaboradas de realizar permutações e substituições foram sendo criadas e deram início à criptografia moderna. Essa nova forma de cifração foi utilizada durante a Segunda Guerra Mundial e ficou bastante conhecida pela máquina Enigma criada pelos alemães.

Com o avanço da computação, algoritmos de permutação e substituição foram se tornando cada vez mais complexos e difíceis de serem quebrados, dessa forma, iniciou-se o uso dos sistemas criptográficos de chave simétrica, que, utilizando uma mesma chave criptográfica, consistem em cifrar e decifrar uma mensagem. Exemplos de criptografia simétrica são: *Data Encryption Standard*, *Advanced Encryption Standard* e Salsa20. Contudo, esses sistemas não garantem o armazenamento das chaves de forma confiável, além de não permitirem a verificação da identidade do remetente da mensagem.

Dessa forma, deu-se início ao desenvolvimento da criptografia de chave assimétrica ou chave pública, que consiste na cifração e decifração por chaves distintas. Sendo assim, foi possível a criação de algoritmos que resolveram os problemas dos sistemas de chave simétrica. Alguns desses sistemas fornecem métodos de criptografia para realizar, seguramente, a trocas de chaves em canais públicos, como o algoritmo de Diffie–Hellman, outros fornecem o serviço de assinatura digital, como o DSA, ECDSA e EdDSA, e alguns proveem ambos, além de realizar cifração de mensagens, como o RSA.

1.2 Justificativa

A assinatura digital é um dos campos de estudo dos sistemas criptográficos de chave pública utilizada como garantia de proteção em transações eletrônicas dentre outros serviços na Internet. A assinatura digital trouxe uma série de benefícios, como a criação de Certificados Digitais, assegurando e facilitando a identificação de pessoas físicas, jurídicas, ou até mesmo de máquinas na Internet.

No âmbito das assinaturas digitais, o Algoritmo de Assinatura Digital da Curva Elíptica de Edwards (EdDSA) tem se tornado um algoritmo promissor, visto que tem se demonstrado mais simples, mais seguro e mais rápido do que outros algoritmos (BERNSTEIN *et al.*, 2012), além de depender da dificuldade de se calcular logaritmos discretos em curvas elípticas.

A Infraestrutura de Chaves Públicas Brasileira (ICP-Brasil) optou por recomendar a utilização do algoritmo EdDSA na sua busca pela implementação de algoritmos e suítes de assinaturas digitais. Essa abordagem tem a finalidade de desburocratizar e agilizar, com a segurança técnica e jurídica necessária, todos os procedimentos e segmentos da sociedade brasileira, diminuindo os custos e fraudes, tornando melhor a vida do cidadão (BRASIL, 2018).

Este trabalho tem como uma de suas propostas a implementação dos algoritmos de assinatura digital Ed448 e Ed521, respectivamente utilizados nas Cadeias V6 e V7 da ICP-Brasil na linguagem de programação Dart. Essa linguagem foi lançada em 2011 pela Google e vem tomando espaço no mundo da programação dado que sua demanda tem aumentado devido ao seu uso pelo *framework* Flutter que é capaz de criar aplicações web, mobile e desktop.

A Figura 1 mostra como a popularidade do *framework* tem crescido desde seu lançamento, que ocorreu em maio de 2017, em relação ao React Native, lançado em março de 2015, outro *framework* utilizado para a criação de aplicativos mobile. Neste gráfico a ordenada representa o nível de popularidade dos termos Flutter e React Native pesquisados no Google, onde 100 é onde houve um pico de popularidade e 50 teve metade da popularidade.

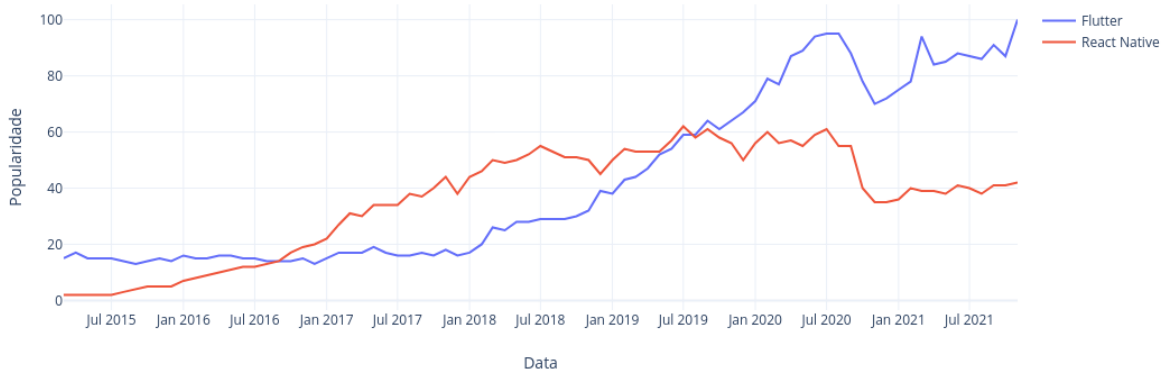

Figura 1 – Comparação de popularidade entre Flutter e React Native. Adaptado de Google (2021).

A outra proposta deste trabalho é a criação de um guia detalhado sobre o EdDSA, onde será descrito um passo a passo do algoritmo. Para isso, será feito um roteiro do código implementado neste trabalho, mostrando o estado da aplicação para cada linha de código executada. Dessa forma, será possível ter uma melhor compreensão de como o EdDSA funciona e também facilitar a conversão do algoritmo para outra linguagem de programação desejada.

1.3 Objetivos

1.3.1 Objetivo Geral

O objetivo principal do trabalho é desenvolver um módulo na linguagem de programação Dart que implementa os algoritmos de assinatura digital da curva de Edwards utilizados pela ICP-Brasil.

1.3.2 Objetivos Específicos

- Desenvolver um módulo para assinar e verificar as assinaturas digitais utilizando os algoritmos Ed448 e Ed521 na linguagem de programação Dart.
- Realizar uma prova de conceito da implementação na linguagem Dart e na linguagem Python.
- Implementar um guia do algoritmo EdDSA.

1.4 Estrutura do Trabalho

Este presente trabalho está dividido da seguinte forma: no Capítulo 1, é realizada uma introdução à história da criptografia, assim como os algoritmos desenvolvidos. Também é apresentada as motivações para a realização deste trabalho. No Capítulo 2 será realizado um breve estudo dos sistemas criptográficos de chave pública, sobre as assinaturas digitais e uma breve explicação sobre a Infraestrutura de Chaves Públicas Brasileira. No Capítulo 3 é apresentada as metodologias utilizadas para conduzir este trabalho, assim como as políticas e ferramentas para o seu desenvolvimento. No Capítulo 4 são feitas discussões e apresentações dos resultados obtidos ao longo do desenvolvimento. E finalmente no Capítulo 5 são apresentadas as conclusões deste trabalho.

2 Fundamentação Teórica

2.1 Sistemas Criptográficos de Chave Pública

A proposta de chave pública foi criada a partir de dois problemas centrais. O primeiro seria o problema de distribuição de chaves, onde duas pessoas só teriam uma comunicação segura caso elas já compartilhassem uma chave anteriormente distribuída entre si. O segundo problema, que aparentemente não havia relação com o primeiro, era o problema de assinaturas digitais. Com a popularização das transações de mensagens e documentos via Internet, viu-se a necessidade da criação de uma assinatura que fosse análoga às assinaturas convencionais. Desta maneira seria possível comprovar a outras pessoas que uma mensagem digital partiu de uma pessoa em particular. Para Diffie (1988, p. 560) existiam duas questões sobre os problemas mencionados:

> No primeiro caso, se duas pessoas pudessem de alguma forma comunicar uma chave secreta de uma para a outra sem nunca terem se conhecido, por que não poderiam comunicar sua mensagem em segredo? O segundo não é melhor. Para ser eficaz, a assinatura deve ser impressa. Como então pode uma mensagem digital, que pode ser copiada perfeitamente, ter uma assinatura?

Diffie e Hellman (1976) descobriram uma solução para essas questões, propondo assim uma forma de duas pessoas realizarem a troca de chave por um canal não seguro, chamada criptografia de chave pública. Essa descoberta proporcionou uma grande mudança na história da criptografia, que antes era realizada sob ferramentas elementares da substituição e permutação, e passou a ser baseada em funções matemáticas. Com essa nova descoberta passou-se a utilizar duas chaves separadas, ao invés de apenas uma chave, como era realizado na criptografia simétrica (STALLINGS, 2017).

Após essas descobertas, foram criados vários sistemas criptográficos de chave pública, os quais providenciavam geração de chaves, algoritmos de cifração, de troca de chaves e de assinatura digital, que será discutido no próximo capítulo.

2.2 Assinaturas Digitais

O processamento de mensagens com um algoritmo de criptografia de chave pública pode oferecer a autenticação da mensagem, garantindo a origem da mensagem ao receptor e que nenhum terceiro está tentando se passar por alguma das partes. Não é garantido, porém, que alguma das duas partes que trocam a mensagem realizem uma disputa. Isso poderia ocorrer caso o receptor alterasse a mensagem e alegasse que ela veio do emissor ou

caso o emissor negasse a mensagem enviada para o receptor. Nesses casos a autenticação não é suficiente.

Uma solução proposta é a utilização de assinaturas digitais, que provê: a autenticação da mensagem, garantindo que um emissor criou e assinou uma mensagem; a integridade da mensagem, provando que a mensagem não foi alterada após ter sido assinada; e o não repúdio, prevenindo que o emissor negue a autoria da mensagem (JOHNSON; MENEZES; VANSTONE, 2001).

É importante notar que, por conta do processo de assinatura e de verificação serem lentos devido às cifrações e decifrações, torna-se muito custosa a implementação de uma assinatura em uma mensagem por completo. Além disso, o receptor teria que armazenar o texto cifrado por completo o verificando sempre que necessário (DIFFIE, 1988).

Para a solução desse problema, o processo de assinatura é baseado no resumo da mensagem, mais conhecido como *hash* da mensagem, para criar a assinatura. Isso trouxe a vantagem de transmitir a assinatura independentemente da mensagem, além de permitir a criação de protocolos onde a mensagem não precisa ser transmitida, visto que já é conhecida por todas as partes (DIFFIE, 1988).

A Figura 2 representa um esquema simplificado do funcionamento de uma assinatura digital. Para criar a assinatura de uma mensagem, deve-se primeiramente passar a mensagem por uma função *hash* para simplificar o processo da assinatura conforme comentado anteriormente, o resultado da função *hash* é então processado com uma chave privada, dessa forma é obtida uma assinatura da mensagem. Para realizar a verificação da assinatura é preciso passar a mensagem pela mesma função *hash* utilizada para criar a assinatura, então esse valor com a assinatura digital e a chave pública são processados e uma comparação é feita de modo a validar ou não a assinatura.

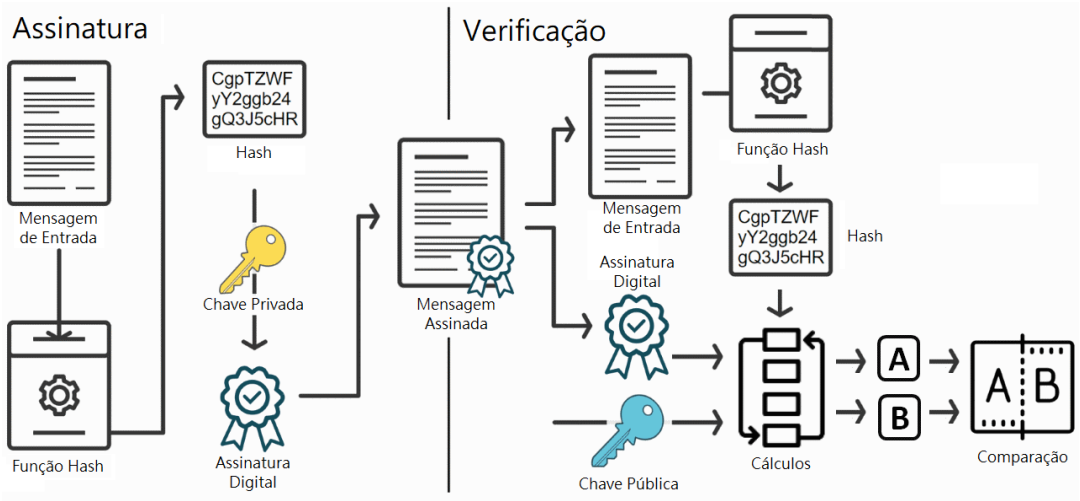

Figura 2 – Representação simplificada dos elementos essenciais do processo de assinatura digital (NAKOV, 2018, tradução nossa).

Vários esquemas de assinatura digital foram desenvolvidos visando garantir a autenticação, integridade e o não repúdio. Alguns exemplos desses esquemas são: assinaturas RSA, assinaturas ElGamal, assinaturas Schnorr, DSA, ECDSA e EdDSA. Esses dois últimos esquemas serão discutidos a seguir.

2.2.1 Elliptic Curve Digital Signature Algorithm (ECDSA)

O algoritmo de assinatura digital com curvas elípticas utiliza cálculos sob o grupo de pontos de uma curva elíptica sobre um corpo finito, sendo um corpo cujo conjunto de seus elementos são finitos. Os cálculos matemáticos que geram segurança para operações com essas curvas se baseiam na inviabilidade computacional do problema do logaritmo discreto da curva elíptica (ECDLP).

Um algoritmo ECDSA deve seguir uma série de parâmetros de domínio adequados de modo a prevenir diversos ataques que poderiam comprometer a segurança do algoritmo e a detecção de codificação inadvertida ou erros de transmissão. Segundo Johnson, Menezes e Vanstone (2001) as restrições para esses parâmetros consistem em um inteiro p especificando o corpo finito $\mathbb{F}_p$, dois elementos $a, b \in \mathbb{F}_p$ especificando uma curva elíptica $E(\mathbb{F}_p)$ podendo ser definida pela equação

$$E : y^2 = x^3 + ax + b \pmod{p}, \tag{2.1}$$

um ponto gerador $G = (x_G, y_G)$ em $E(\mathbb{F}_p)$, um primo n que é a ordem de G, e o cofator $h = \#E(\mathbb{F}_p)/n$, onde $\#E(\mathbb{F}_p)$ é a ordem de E.

2.2.1.1 Geração de Par de Chaves ECDSA

Após a validação dos parâmetros de domínio da curva, ela estará apta a ser utilizada para gerar pares de chaves, as quais serão necessárias para realizar as assinaturas digitais.

Segundo Johnson, Menezes e Vanstone (2001) para a geração de pares de chaves do algoritmo ECDSA primeiramente é necessário definir a chave privada, escolhendo de forma aleatória ou pseudoaleatória um inteiro d no intervalo $[1, n-1]$. A chave pública Q é computada realizando a multiplicação da chave privada d pelo ponto gerador G, e esta pode ser comprimida escolhendo uma das coordenadas e somando um bit de paridade.

2.2.1.2 Assinatura ECDSA

A assinatura ECDSA se baseia no esquema de assinatura de ElGamal. Para realizar a assinatura de uma mensagem m é preciso seguir os seguintes passos (JOHNSON; MENEZES; VANSTONE, 2001):

1. Escolha um número aleatório ou pseudoaleatório k no intervalo $[1, n-1]$;

2. Calcule $kG = (x_1, y_1)$ e converta x_1 em um inteiro $\bar{x}_1$;

3. Calcule $r = \bar{x}_1 \bmod n$;

4. Calcule o *hash* da mensagem, utilizando uma função *hash* criptográfica como SHA-256 e o converta para um número inteiro e;

5. Calcule $s = k^{-1}(e + dr) \bmod n$;

6. A assinatura digital da mensagem m se dá por (r, s);

2.2.1.3 Verificação ECDSA

Para ser possível verificar a assinatura (r, s) da mensagem m é necessário ter conhecimento dos parâmetros de domínio da curva, assim como a chave pública Q. Após isso é realizado os seguintes passos (JOHNSON; MENEZES; VANSTONE, 2001):

1. Verifique se r e s estão no intervalo $[1, n-1]$;

2. Calcule o *hash* da mensagem m, utilizando uma função *hash* criptográfica como SHA-256 e o converta para um número inteiro e;

3. Calcule $w = s^{-1} \bmod n$;

4. Calcule $u_1 = ew \bmod n$ e $u_2 = rw \bmod n$;

5. Calcule $X = u_1G + u_2Q$;

6. Se X for o ponto no infinito, recuse a assinatura. Caso contrário, converta a coordenada x de X em um inteiro $\bar{x}_1$ e calcule $v = \bar{x}_1 \bmod n$;

7. Aceite a assinatura apenas se e somente se $v = r$;

O principal objetivo na verificação da assinatura é recalcular o ponto X por meio da chave pública fornecida e comparar com o ponto gerado no processo de assinatura pela multiplicação do número aleatório k com o ponto gerador G.

2.2.2 Edwards-curve Digital Signature Algorithm (EdDSA)

O Algoritmo de Assinatura Digital da curva de Edwards se baseia no esquema de assinatura de Schnorr e utiliza curvas de Edwards, possivelmente sendo curvas torcidas de Edwards. As curvas de Edwards tem a forma

$$x^2 + y^2 = c^2(1 + dx^2y^2) \tag{2.2}$$

Já as curvas torcidas de Edwards possuem a forma

$$ax^2 + y^2 = 1 + dx^2y^2 \tag{2.3}$$

Os cálculos matemáticos que geram segurança para operações com essas curvas fundamenta-se na inviabilidade computacional do problema do logaritmo discreto da curva elíptica (ECDLP).

Esse esquema de assinaturas por chave pública traz uma série de benefícios como: verificação rápida de uma única assinatura, verificação em lote ainda mais rápida, assinaturas muito rápidas, rápida geração de chaves, alto nível de segurança, chaves de sessão infalíveis, resiliência à colisão e chaves pequenas (BERNSTEIN *et al.*, 2012).

Assim como nos algoritmos ECDSA, é necessária uma série restrições para os parâmetros a serem seguidos com a finalidade de prevenir ataques que poderiam comprometer a segurança do algoritmo. Definido por Bernstein *et al.* (2015) e posteriormente citado por Josefsson e Liusvaara (2017) na RFC8032, essas restrições consistem nos seguintes critérios:

1. Um expoente primo ímpar q especificando o corpo finito $\mathbb{F}_q$;
2. Um inteiro b onde $2^{b-1} > q$, com a recomendação de que b seja um múltiplo de 8;
3. Uma codificação de $(b-1)$ bits dos elementos de $\mathbb{F}_q$;
4. Uma função *hash* H que produza um valor de tamanho $2b$ bits, com a recomendação de se utilizar funções *hash* conservativas, que minimizem a possibilidade de ocorrer colisões;
5. Um inteiro $c \in \{2, 3\}$;
6. Um inteiro n onde $c \leq n \leq b$;
7. Um elemento quadrado não nulo a de $\mathbb{F}_q$, com a recomendação de que $a = -1$ se $q \bmod 4 \equiv 1$, e $a = 1$ se $q \bmod 4 \equiv 3$;
8. Um elemento que não seja resíduo quadrático d de $\mathbb{F}_q$;
9. Um elemento B, que não seja o ponto no infinito, nesse caso denotado como $(0, 1)$, pertencendo à curva definida pela equação 2.3;
10. Um primo ímpar l de tal modo que $lB = 0$ e $2^c l = \#E(\mathbb{F}_q)$;
11. Uma função *hash* que poderá ser aplicada à mensagem ou não chamada função *prehash* H';

Um exemplo de algoritmo EdDSA é o Ed448, que utiliza a forma de curva torcida de Edwards edwards448. Ed448 utiliza uma chave privada de 57 *bytes*, uma chave pública de 57 *bytes* e gera uma assinatura de 114 *bytes*. Seus parâmetros discutidos por Josefsson e Liusvaara (2017) na RFC8032 consistem em:

1. Um primo $q = 2^{448} - 2^{224} - 1$;
2. $b = 456$;
3. A codificação 455 bits de $\mathbb{F}_q$ é a codificação *little-endian* em $\{0, 1, \ldots, 2^{448} - 2^{224} - 2\}$;
4. A função *hash* utilizada é a SHAKE256;
5. $c = 2$;
6. $n = 448$;
7. $a = 1$;
8. $d = -39081$;
9. B é o ponto (224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710, 298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660) $\in E$;
10. l é o primo $2^{446} -$ 138180668098951153520073867485154268803366924748821786098945475038 85;
11. H' é a função identidade, ou seja, $H'(M) = M$;

Outro algoritmo é o Ed521, que utiliza a forma de curva de Edwards E-521. Esse algoritmo utiliza uma chave privada de 66 *bytes*, uma chave pública de 66 *bytes* e gera uma assinatura de 132 *bytes*. Seus parâmetros citados por Aranha *et al.* (2013) e posteriormente definidos por Brasil (2019) consistem em:

1. Um primo $q = 2^{521} - 1$;
2. $b = 528$;
3. A codificação 455 bits de $\mathbb{F}_q$ é a codificação *little-endian* em $\{0, 1, \ldots, 2^{448} - 2^{224} - 2\}$;
4. A função *hash* utilizada é a SHAKE256;
5. $c = 2$;
6. $n = 521$;

7. $a = 1$;

8. $d = -376014$;

9. B é o ponto (15710548941849953875359397498943175686452973504029058214376251811523049943811885296325911960676041007726739279151142671933899050032766737490120511483560413 24, 12) $\in E$;

10. l é o primo $2^{519} - 337554763258501705789107630418782636071904961214051226618635150085779108655765$;

11. H' é a função identidade, ou seja, $H'(M) = M$;

2.2.2.1 Geração de Par de Chaves EdDSA

Com os valores dos parâmetros de domínio da curva escolhidos e validados, é possível gerar pares de chaves que serão utilizados para assinar mensagens e verificar assinaturas digitais.

Segundo Bernstein *et al.* (2015), a chave privada k é um número aleatório de tamanho b bits. A função *hash* $H(k) = (h_0, h_1, \ldots, h_{2b-1})$ determina um inteiro $s = 2^n \sum_{c \leq i < n} 2^i h_i$ que por sua vez determina o múltiplo $A = sB$, onde $\underline{\text{A}}$ é a chave pública.

2.2.2.2 Assinatura EdDSA

O algoritmo de assinatura EdDSA utiliza uma mensagem M e uma chave privada k. Dessa forma, define-se $r = H(h_b, \ldots, h_{2b-1}, M) \in \{0, 1, \ldots, 2^{2b} - 1\}$; $R = rB$; $S = (r + H(\underline{R}, \underline{A}, M)s) \bmod l$. A assinatura de M sob k é o par de inteiros $(\underline{R}, \underline{S})$ de tamanho $2b$ bits, onde $\underline{\text{R}}$ possui um ponto comprimido e um inteiro $\underline{\text{S}}$ que indica que o assinante conhece a chave privada k e a mensagem M garantindo o não repúdio (BERNSTEIN *et al.*, 2015).

No esquema de assinatura ECDSA, é necessário realizar a criação do valor r para cada mensagem de forma aleatória. Caso r seja o mesmo para duas mensagens então a chave privada poderá ser descoberta (ELGAMAL, 1985). Um caso concreto dessa falha ocorreu no esquema de assinatura ECDSA implementado para a assinatura de código do console PlayStation3 da empresa Sony (FAILOVERFLOW, 2010).

Para evitar esse problema, o algoritmo EdDSA gera um valor de r dependente da mensagem, resultando em um r diferente para cada mensagem assinada.

2.2.2.3 Verificação EdDSA

A verificação é feita utilizando a mensagem M, a chave pública do assinante $\underline{A}$ e a assinatura $(\underline{R}, \underline{S})$. Primeiramente recuperam-se os pontos da curva elíptica A e R dos

pontos comprimidos $\underline{A}$ e $\underline{R}$. Depois calcula-se $H(\underline{R}, \underline{A}, M)$ e verifica-se o grupo de equação $2^cSB = 2^cR + 2^cH(\underline{R}, \underline{A}, M)A$ em E. A assinatura deve ser rejeitada caso a recuperação dos pontos falhe ou se a comparação divergir.

2.3 Infraestrutura de Chaves Públicas Brasileira

A Infraestrutura de Chaves Públicas Brasileira, mais comumente conhecida por ICP-Brasil, é a cadeia hierárquica regulamentada responsável pela emissão de certificados digitais no Brasil. Criada em 2001 pela Medida Provisória 2.200-2, a ICP-Brasil surge com a finalidade de normatizar e estabelecer os requisitos técnicos necessários para realizar a identificação virtual de entidades e de cidadãos brasileiros.

O Instituto Nacional de Tecnologia da Informação é a autarquia federal responsável por operar e supervisionar a ICP-Brasil de modo a garantir sua segurança. Ao ITI compete a responsabilidade de fiscalizar, credenciar, descredenciar as demais entidades da cadeia, além de ser a Autoridade Certificadora Raiz (AC-Raiz), a qual executa as Políticas de Certificados, emite, expede, distribui, revoga e gerencia os certificados das autoridades certificadoras imediatamente abaixo (BRASIL, 2020).

À AC-Raiz estão credenciadas as Autoridades Cerificadoras, cujo objetivo é gerenciar os certificados digitais associando pares de chaves criptográficas ao seu titular, podendo ser outra Autoridade Certificadora ou um usuário final. A Figura 3 descreve detalhadamente cada nível da hierarquia da ICP-Brasil.

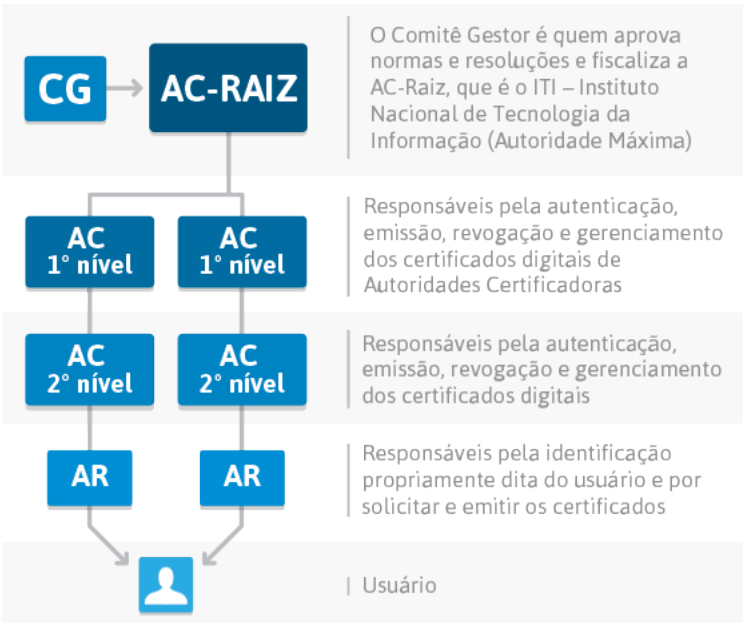

Figura 3 – Hierarquia da ICP-Brasil (Benefícios e Aplicações da Certificação Digital, c2013).

3 Metodologia

3.1 Metodologia de Desenvolvimento

Para a concepção do trabalho, foi feita uma junção de diferentes metodologias ágeis. Essa abordagem visa selecionar as melhores práticas de cada metodologia de modo a otimizar o desenvolvimento das tarefas que fizeram parte da elaboração desse trabalho.

Com o intuito de realizar um desenvolvimento mais produtivo, foram utilizados alguns elementos da metodologia Scrum. Dessa forma, as tarefas que foram planejadas para serem desenvolvidas foram organizadas em um *Product Backlog*. As *Sprints* tiveram a duração de uma semana, e no início de cada uma era realizado o *Sprint Planning* e, ao final, era executada o *Sprint Review* e o *Sprint Restrospective* .

Para melhorar o controle e visualização das tarefas realizadas e assim otimizar o desenvolvimento, a agilidade e eficácia, foi utilizada a metodologia KanBan com algumas modificações. O quadro foi composto por 5 diferentes colunas: *Product Backlog*, *Sprint Backlog*, Em Progresso, Revisão e Fechadas.

As tarefas a serem desenvolvidas foram representadas no formato de *User Story*. Os cartões do quadro continham o seguinte conteúdo: um identificador único, uma descrição da tarefa no padrão de sentença em inglês "*I am <who>, I want <what>*", sua categoria, as atividades a serem realizadas e a indicação da *Sprint*.

De modo a desenvolver um código com boa qualidade, foi imprescindível a adoção de boas práticas de programação, dessa forma, a metodologia *Extreme Programming* define alguns elementos que serão utilizados. O primeiro é a adoção de uma folha de estilo para ser possível realizar padronizações de codificação e o segundo é a realização de refatorações do código. A folha de estilo escolhida foi a do próprio Dart[1].

3.2 Políticas do Repositório

Todo o Software foi armazenado na plataforma GitHub em repositórios abertos, A biblioteca DartFFIedLibdecaf está no repositório `pbad-pades/dartffiedlibdecaf` e a SignDart está no repositório `pbad-pades/SignDart`. As tarefas a serem desenvolvidas foram cadastradas como *issues* e utilizadas como cartões no quadro KanBan disponível na aba *Projects* do repositório da SignDart.

O repositório do projeto seguiu o fluxo de trabalho conhecido como GitFlow, con-

[1] Regras definidas pelo Dart Lint, disponível em: https://dart-lang.github.io/linter/lints/

forme a Fig. 4 e possui as seguintes ramificações: a *Main* que será a ramificação principal e possui a versão estável do código; a *Develop*, criada a partir da *Main*, e serve para integração de novos recursos; a *Feature* é dedicada para a criação de novas funcionalidades e após finalizada e revisada será integrada à *Develop*; a *Release* é uma ramificação da *Develop* onde é criada ao final de uma *Sprint*, quando estiver pronta para o lançamento, ela deve ser mesclada com a *Main* e com a *Develop*; a *Hotfix* é criada sempre que algum erro em produção for detectado.

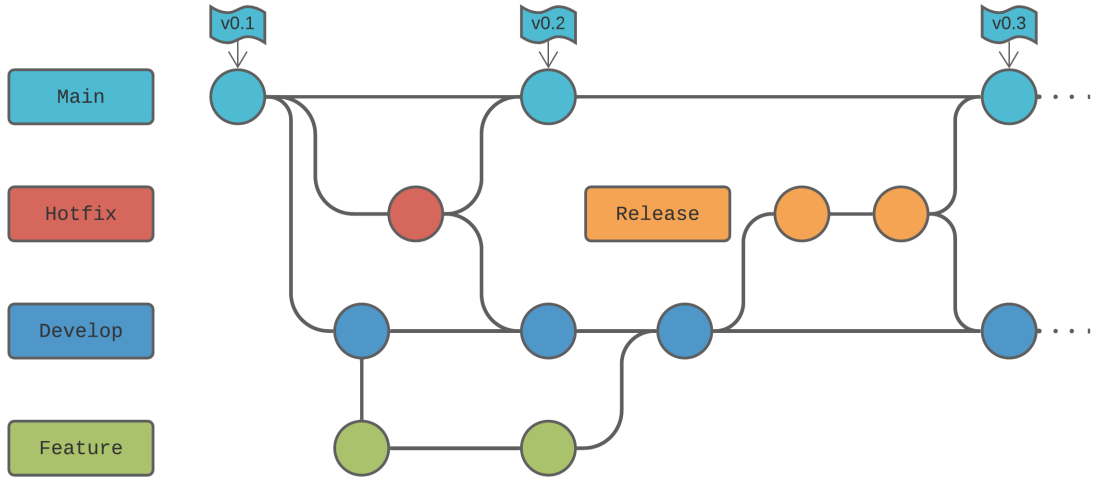

Figura 4 – Fluxo de trabalho segundo o GitFlow

Os *commits* descrevem sucintamente o que foi realizado, são atômicos, escritos no gerúndio, em inglês, e deverão ser identificados pelo código da *issue* a que se referem, como no seguinte exemplo: "#1 — Creating project".

3.3 Linguagens e Ambiente de Desenvolvimento

Para a criação dos módulos, foi utilizada a linguagem de programação Dart, uma linguagem que tem se popularizado devido ao *framework* Flutter, que permite o desenvolvimento de aplicações nativas para web, mobile e desktop. Nessa linguagem de programação, foi constatada a falta de uma biblioteca que implementasse os algoritmos Ed448 e Ed521, devido a isso, essa linguagem foi utilizada para a implementação do módulo que realizará a assinatura e verificação conforme o algoritmo EdDSA.

De modo a realizar uma prova de conceito e validar as funções intermediárias que escritas em Dart, foi realizada uma validação cruzada com uma biblioteca mais madura na linguagem Python.

Tanto para o estudo dessas linguagens quanto para a elaboração dos módulos mencionados, será utilizada o editor de código *Visual Studio Code*, uma ferramenta de desenvolvimento produzida pela Microsoft.

3.4 Testes

Para realizar testes nas implementações, foram desenvolvidos testes de integração, que visam validar a saída final do programa dado diversas entradas. Para realizar a validação do Ed448, utilizou-se o vetor de testes disponibilizado na RFC8032, já para validar o Ed521, utilizou-se o vetor de testes criado a partir da implementação do Ed521 na biblioteca ECPy, o qual está disponível no Apêndice C.

Junto aos vetores de testes mencionados anteriormente, foi feito uma comparação entre as saídas de cada função da biblioteca ECPy de modo a verificar se as implementações retornavam o valor esperado.

3.5 Integração Continuada e Implantação Continuada

A integração continuada e a implantação continuada foram realizadas por meio do GitHub Actions, e, dessa forma, é possível executar um fluxo de trabalho totalmente automático que analisa a qualidade e a padronização do código, realiza testes, empacota os módulos e os publicam no sistema de gerenciamento de pacotes pub.dev.

Assim que um *Pull Request* é aberto, a qualidade e a padronização do código são analisadas por uma própria ferramenta do Dart e um fluxo de testes é realizado e caso sejam aprovados, um *merge* do código estará liberado para ser realizado na ramificação *Main*. Assim que o *merge* for feito, uma nova versão do pacote será publicada.

4 Resultados e Discussões

Uma primeira abordagem adotada para a criação dos módulos que implementassem os algoritmos Ed448 e Ed521 em Dart foi pesquisar outras bibliotecas escritas em linguagens de programação diferentes e que fossem mais maduras para serem utilizadas como base de desenvolvimento. Dessa forma, foi verificada uma biblioteca codificada na linguagem C que implementa algoritmos EdDSA, incluindo o Ed448, chamada libdecaf, a qual está disponível no repositório raisty/libdecaf[1] no GitHub. Por estar codificada em C, decidiu-se realizar uma interoperabilidade com um código em Dart utilizando a biblioteca FFI, que é uma biblioteca do Dart cujo objetivo é realizar chamadas de funções em C, além de ser possível ler, gravar, alocar e desalocar memória nativa.

Feita a interoperabilidade utilizando a FFI a partir da libdecaf já implementada em C, foi implementada uma nova biblioteca que cria uma interface em Dart possibilitando chamadas de funções em C responsáveis pelo algoritmo Ed448. Essa biblioteca foi nomeada DartFFIedLibdecaf e se encontra disponível no repositório pbad-pades/dartffiedlibdecaf[2] no GitHub. No repositório há instruções de como importar a biblioteca. Na pasta "`\example`" é possível encontrar um exemplo de como utilizar a biblioteca. Já na pasta "`\test`" foram implementados testes que validam chaves públicas e privadas, assinaturas de mensagens e a verificação dessas assinaturas. Para realizar essas validações, foi utilizado um vetor de testes criado por Josefsson e Liusvaara (2017). As mesmas validações foram feitas a partir de chaves privadas geradas de forma aleatória e utilizando-se as mensagens do mesmo vetor de teste para gerar assinaturas e verificá-las.

Após o desenvolvimento da nova biblioteca, foi realizado um estudo para verificar a viabilidade da implementação do algoritmo Ed521 na libdecaf, de modo a utilizá-la por meio da interoperabilidade. Concluiu-se, porém, que seria necessário um alto nível de modificação, devido a alguns parâmetros específicos utilizados na biblioteca que demandariam um esforço de desenvolvimento incompatível com o escopo desse trabalho. Dessa forma, foi preciso recorrer à alguma outra biblioteca que possuísse um nível adequado de abstração e que correspondesse com o que Josefsson e Liusvaara (2017) descrevem na RFC8032.

Após analisar algumas bibliotecas em diferentes linguagens de programação, escolheu-se a ECPy, que está disponível no repositório cslashm/ECPy[3] do GitHub. Ela está codificada em Python, possui a implementação do EdDSA e é bastante flexível. Dessa forma, iniciou-se a implementação do Ed521 nessa biblioteca de modo a gerar um vetor de teste

[1] Disponível em: https://github.com/raisty/libdecaf
[2] Disponível em: https://github.com/pbad-pades/dartffiedlibdecaf
[3] Disponível em: https://github.com/cslashm/ECPy

que pudesse ser utilizado futuramente pelos módulos implementados em Dart. Assim, foi possível adicionar o algoritmo Ed521 descrito por Aranha *et al.* (2013), a ela. Essas modificações ocorreram de forma simples, pelo fato de o algoritmo seguir os mesmos passos que o Ed448 segue na RFC8032, provavelmente devido a ambos utilizarem curvas com o mesmo cofator. Para validação, foi utilizado o certificado digital V7[4] disponibilizado pela ICP-Brasil, onde foi possível, por meio da sua chave pública, verificar sua assinatura. Também foi possível gerar chaves privadas e públicas, realizar a assinatura de mensagens de testes e verificá-las.

Como é possível observar na Figura 5, o suporte ao Ed521 foi aceito pelo mantenedor da biblioteca ECPy e está disponível para a comunidade usar. Dessa forma, esse trabalho não só criou uma biblioteca, mas também contribuiu para a comunidade *open source*.

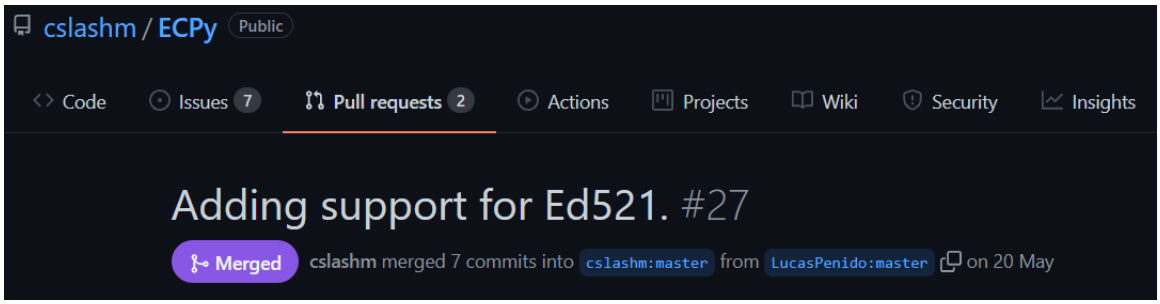

Figura 5 – Adicionando suporte ao Ed521 na biblioteca ECPy.

Em seguida, deu-se início então à implementação da biblioteca que provê o Ed448 e o Ed521 em Dart utilizando como base a ECPy. Dessa forma, buscou-se implementar o algoritmo o mais similar possível com o que é descrito na RFC8032 por Josefsson e Liusvaara (2017) de modo a facilitar o entendimento do algoritmo. A nova biblioteca implementada foi nomeada de SignDart e está disponível no repositório pbad-pades/SignDart[5] no GitHub.

No repositório, na pasta "`\example`" é possível encontrar exemplos de como utilizar os algoritmos Ed448 e Ed521 implementados na biblioteca. Já na pasta "`\test`" foram implementados testes que validam chaves públicas e privadas, a assinatura de mensagens e a verificação de assinaturas. Para realizar essas validações com o Ed448, foi utilizado o vetor de testes criado por Josefsson e Liusvaara (2017). Já para o Ed521, utilizou-se o vetor de testes gerado a partir do algoritmo implementado na biblioteca ECPy, que está disponível no Apêndice C. Para ambos os algoritmos, as mesmas validações também foram feitas a partir de chaves privadas geradas de forma aleatória e utilizando-se as mensagens

[4] Disponível em: https://www.gov.br/iti/pt-br/assuntos/repositorio/repositorio-ac-raiz
[5] Disponível em: https://github.com/pbad-pades/SignDart

do mesmo vetor de teste para gerar assinaturas e verificá-las. A biblioteca também pode ser encontrada no repositório de pacotes do Dart *pub.dev* com o nome de *sign_dart*[6]

De modo a realizar uma comparação entre as bibliotecas mencionadas anteriormente, foram realizadas medições dos tempos de execução de cada função do algoritmo (criação de chave privada e de chave pública, assinatura de uma mensagem e sua verificação). Essas medições foram realizadas no Notebook Lenovo IdeaPad Gaming 3 15IMH05 com um processador Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz. Assim, foram realizadas, em todas as bibliotecas, 10.000 medições de tempo em microssegundos (μs), onde, para cada medição de tempo das funções de assinar e verificar foram gerados chaves públicas e privadas, e mensagens diferentes, essas sempre de 64 *bytes* de tamanho. É importante mencionar que a primeira medição em Dart foi descartada visto que seu tempo de execução demorava mais que as demais medições, provavelmente por questões de *caching*.

Sendo assim, foi possível registrar na Tabela 1 os resultados das médias de cada função do Ed448 nas bibliotecas DartFFIedLibdecaf, SignDart e ECPy. Dessa forma é possível notar que a biblioteca DartFFIedLibdecaf realizou as operações mais rapidamente que as demais, visto que essa utiliza uma interface para realizar chamadas de funções em C, exceto na criação da chave privada. Também é importante constatar que a função de criação da chave privada foi mais lenta do que nas demais bibliotecas, o que não foi possível averiguar a causa. Outro ponto a se notar é que a biblioteca implementada em Dart puro foi mais rápida que a implementada em Python.

Tabela 1 – Comparação do tempo médio de execução de cada função do Ed448 em diferentes linguagens

Biblioteca	**Criar chave privada** [μs]	**Criar chave pública** [μs]	**Assinar uma mensagem** [μs]	**Verificar uma assinatura** [μs]
DartFFIedLibdecaf	95,65	56,81	144,29	158,28
SignDart	11,26	5443,55	10844,85	11969,26
ECPy	1,10	8010,96	15921,94	17662,95

Já os resultados das médias das medições de tempo de cada função do Ed521 estão registrados na Tab. 2. Assim é possível considerar que, como no Ed448, a biblioteca implementada em Dart executou as funções de mais rapidamente que a em Python. Como era esperado, as operações para a curva utilizada pelo Ed521 são mais complexas computacionalmente do que aquelas da Ed448.

[6] Disponível em: https://pub.dev/packages/sign_dart

Tabela 2 – Comparação do tempo médio de execução de cada função do Ed521 em diferentes linguagens

Biblioteca	**Criar chave privada** $[\mu s]$	**Criar chave pública** $[\mu s]$	**Assinar uma mensagem** $[\mu s]$	**Verificar uma assinatura** $[\mu s]$
SignDart	12,37	9098,71	18196,12	20380.59
Python	1,09	13168.58	25274.41	26073.90

Com a SignDart implementada, foi possível criar uma documentação com um passo a passo dos algoritmos de assinatura Ed448 e Ed521. Para isso, foi feito um roteiro do código, mostrando o estado da aplicação para cada linha de código executada. Com essa documentação é possível ter uma melhor compreensão de como o EdDSA funciona e também facilitar a conversão do algoritmo para outra linguagem de programação desejada. Essa documentação está disponível na Wiki[7] do repositório `pbad-pades/SignDart` e nos Apêndices A e B.

[7] Ed448: https://github.com/pbad-pades/SignDart/wiki/Ed448
Ed521: https://github.com/pbad-pades/SignDart/wiki/Ed521

5 Conclusões e Proposições

Conforme apresentado ao longo do presente trabalho, as assinaturas digitais estão cada vez mais em estudo em razão do aumento das interações via internet e a necessidade de torná-las mais seguras, lançando mão dessas estruturas, pois já proveem a autenticação, integridade e o não repúdio de mensagens.

A proposta desse trabalho foi devidamente concluída, visto que os algoritmos Ed448 e Ed521 foram implementados de forma satisfatória, dado a prova de conceito realizada com a validação cruzada da biblioteca em Python com a biblioteca criada em Dart, além dos diversos testes realizados. A biblioteca tem se demonstrado de fácil uso e compreensão, visto que seguiu o paradigma da Orientação à Objetos e os princípios do *Clean Code*, além de estar de acordo com o que está descrito na RFC8032.

Outros insumos gerados por esse trabalho foram as documentações passo a passo dos algoritmos Ed448 e E521, disponíveis nos Apêndices A e B, assim como o vetor de testes para o algoritmo Ed521, disponível no Apêndice C.

Vale ressaltar haver outro documento escrito por Hamburg (2015) que trata da Ed448, mas que chama a atenção por especificar outro valor para o ponto gerador, diferente do especificado por Josefsson e Liusvaara (2017) na RFC8032.

O objetivo de criar uma documentação que descrevesse um passo a passo dos algoritmos Ed448 e Ed521 foi realizado com sucesso e estão disponíveis nos apêndices A e B. Neles, está descrito um roteiro do código implementado, mostrando o estado da aplicação para cada linha executada. Dessa forma, será possível ter uma melhor compreensão de como o EdDSA funciona e também facilitar a conversão do algoritmo para outra linguagem de programação desejada.

Os resultados expostos no presente trabalho demonstram que muitas outras pesquisas ainda podem ser realizadas sobre as assinaturas digitais, mais especificamente as que utilizam curvas elípticas. Um desses estudos, e o principal que este trabalho facilitará, é a conversão do algoritmo para outra linguagem de programação desejada, bem como sua contribuição direta para a ampliação de conhecimentos na área.

Outros estudos podem ser feitos a partir do código desenvolvido. Podendo ser modificado para atender requisitos de desempenho, velocidade e de segurança, dado que a biblioteca foi desenvolvida com o foco apenas no algoritmo, o EdDSA. Uma indicação é o repositório do GitHub veorq/cryptocoding[1], que contém recomendações gerais e de melhores práticas para escrever um código mais seguro. Também podem ser adicionados

[1] Disponível em: https://github.com/veorq/cryptocoding

novos testes que garantam o funcionamento do algoritmo e do vetor de teste criado. Há também a possibilidade da modificação do algoritmo para que ele seja capaz de realizar assinaturas a partir de um hash da mensagem, chamado procedimento de assinatura com pre-hash. Outra possibilidade é adicionar um contexto ao algoritmo, utilizado para separar os usos do protocolo entre protocolos diferentes e entre diferentes usos no mesmo protocolo.

Referências

ARANHA, D. F. *et al.* **A note on high-security general-purpose elliptic curves**. 2013. Cryptology ePrint Archive, Report 2013/647. Disponível em: <https://eprint.iacr.org/2013/647.pdf>. Acesso em: 22/03/2021. Citado 2 vezes nas páginas 30 e 38.

Benefícios e Aplicações da Certificação Digital. **Hierarquia da ICP-Brasil**. c2013. Disponível em: <http://www.beneficioscd.com.br/cartilha_online/?pagina=oq06>. Acesso em: 21/04/2021. Citado 2 vezes nas páginas 11 e 32.

BERNSTEIN, D. J. *et al.* High-speed high-security signatures. **J Cryptogr Eng**, Springer, v. 2, p. 77–89, 2012. Disponível em: <http://ed25519.cr.yp.to/ed25519-20110926.pdf>. Acesso em: 22/03/2021. Citado 2 vezes nas páginas 22 e 29.

BERNSTEIN, D. J. *et al.* Eddsa for more curves. **Cryptology ePrint Archive**, IACR, v. 2015, 2015. Disponível em: <http://ed25519.cr.yp.to/eddsa-20150704.pdf>. Acesso em: 22/03/2021. Citado 2 vezes nas páginas 29 e 31.

BRASIL. **Novo marco da ICP-Brasil: emitidas Cadeias V6 e V7 em curvas elípticas**. 2018. Disponível em: <https://www.gov.br/iti/pt-br/assuntos/noticias/indice-de-noticias/novo-marco-da-icp-brasil-emitidas-cadeias-v6-e-v7-em-curvas-elipticas>. Acesso em: 22/03/2021. Citado na página 22.

BRASIL. **Padrões Algoritmos Criptográficos da ICP-BRASIL DOC ICP-01.01 Versão 4.2**. 2019. Disponível em: <https://www.gov.br/iti/pt-br/centrais-de-conteudo/doc-icp-01-01-v-4-2-padroes-e-algoritmos-criptograficos-da-icp-brasil-copy-pdf>. Acesso em: 05/08/2021. Citado na página 30.

BRASIL. **ICP-Brasil**. 2020. Disponível em: <https://www.gov.br/iti/pt-br/acesso-a-informacao/perguntas-frequentes/icp-brasil>. Acesso em: 21/04/2021. Citado na página 32.

BRUEN, A.; FORCINITO, M. **Cryptography, information theory, and error-correction: a handbook for the 21st century**. Nova Jersey: John Wiley & Sons, 2011. v. 68. Citado na página 21.

COHEN, F. **A short history of cryptography**. 1990. Disponível em: <http://www.all.net/books/ip/Chap2-1.html>. Acesso em: 22/06/2021. Citado na página 21.

DIFFIE, W. The first ten years of public-key cryptography. **Proceedings of the IEEE**, v. 76, n. 5, p. 560–577, 1988. Disponível em: <http://www.cs.virginia.edu/~evans/greatworks/diffie.pdf>. Acesso em: 03/04/2021. Citado 2 vezes nas páginas 25 e 26.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. **IEEE transactions on Information Theory**, IEEE, v. 22, n. 6, p. 644–654, 1976. Disponível em: <https://ee.stanford.edu/~hellman/publications/24.pdf>. Acesso em: 03/04/2021. Citado na página 25.

ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. **IEEE Transactions on Information Theory**, v. 31, n. 4, p. 469–472, 1985. Disponível em: <https://bitlybr.com/xrDjpA>. Acesso em: 28/03/2021. Citado na página 31.

FAILOVERFLOW. **PS3 epic fail**. 2010. Disponível em: <https://fahrplan.events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf>. Acesso em: 28/03/2021. Citado na página 31.

GOOGLE. **Google Trends**. 2021. Disponível em: <https://bitlybr.com/S4bSD>. Acesso em: 06/11/2021. Citado 2 vezes nas páginas 11 e 23.

HAMBURG, M. Ed448-goldilocks, a new elliptic curve. **IACR Cryptol. ePrint Arch.**, v. 2015, p. 625, 2015. Disponível em: <https://eprint.iacr.org/2015/625.pdf>. Acesso em: 25/04/2021. Citado na página 41.

JOHNSON, D.; MENEZES, A.; VANSTONE, S. The elliptic curve digital signature algorithm (ECDSA). **International journal of information security**, Springer, v. 1, n. 1, p. 36–63, 2001. Citado 3 vezes nas páginas 26, 27 e 28.

JOSEFSSON, S.; LIUSVAARA, I. **Edwards-Curve Digital Signature Algorithm (EdDSA)**. RFC Editor, 2017. RFC 8032. Disponível em: <https://rfc-editor.org/rfc/rfc8032.txt>. Acesso em: 06/08/2021. Citado 5 vezes nas páginas 29, 30, 37, 38 e 41.

NAKOV, S. **Practical Cryptography for Developers**. Svetlin Nakov, 2018. E-Book. Disponível em: <https://cryptobook.nakov.com/>. Acesso em: 20/03/2021. Citado 2 vezes nas páginas 11 e 26.

RIVEST, R. L. Cryptography. In: **Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity**. Cambridge, MA, USA: MIT Press, 1991. p. 617–755. Citado na página 21.

STALLINGS, W. **Cryptography and Network Security Principles and Practice**. 7. ed. Harlow: Pearson Education Limited, 2017. Citado na página 25.

Apêndices

APÊNDICE A – Documentação Ed448

1. Sobre o Ed448

O algoritmo Ed448 utiliza a forma de Edward da curva edwards448 definida na RFC7748. Os parâmetros dessa curva são:

$p : 2^{448} - 2^{224} - 1 = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$

$l(ordem) : 2^{446} - 138180668098951153520073867485154268803366924748821786098945475 03885 = 0x3FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF7CCA23E9C44EDB49AED63690216CC2728DC58F552378C292AB5844F3$

$a : 1$

$d : -39081$

X(G) (coordenada x do ponto gerador) : $0x4F1970C66BED0DED221D15A622BF364DA9E146570470F1767EA6DE324A3D3A46412AE1AF72AB66511433B80E18B00938E2626A82BC70CC05E$

Y(G) (coordenada y do ponto gerador) : $0x693F46716EB6BC248876203756C9C7624BEA73736CA3984087789C1E05A0C2D73AD3FF1CE67C39C4FDBD132C4ED7C8AD9808795BF230FA14$

cofator : 4

Função *hash* utilizada: Shake256

Essa curva é definida na classe `TwistedEdwardCurve`. Para utilizá-la é preciso instanciar desta forma:

```
var ed448 = TwistedEdwardCurve.ed448();
```

2. Geração de chaves

Ed448 utiliza uma chave pública e privada de 57 *bytes*.

2.1. Chave privada

A chave privada é composta por 57 *bytes* gerados de forma criptograficamente aleatória e segura. Ela é definida pela classe `EdPrivateKey` e para gerá-la de forma aleatória é preciso chamar o método `generate` e passar como parâmetro a curva que está sendo utilizada:

```
EdPrivateKey privateKey = EdPrivateKey.generate(ed448);
```

Outra forma de instanciar uma chave privada é utilizando o método `fromBytes` e passando como parâmetro uma cadeia de *bytes* (Uint8List) e a curva que está sendo utilizada:

```
EdPrivateKey privateKey = EdPrivateKey.fromBytes(privateKeyBytes, ed448);
```

Para esse exemplo vamos utilizar a chave privada em hexadecimal: $0xC4EAB05D357007C632F3DBB48489924D552B08FE0C353A0D4A1F00ACDA2C463AFBEA67C5E8D2877C5E3BC397A659949EF8021E954E0A12274E$

2.2. Chave pública

A chave pública é definida pela classe `EdPublicKey` e pode ser obtida a partir da classe `EdPrivateKey` utilizando o método `getPublicKey`:

```
EdPublicKey publicKey = privateKey.getPublicKey();
```

Sua geração acontecem em 4 etapas que serão demonstradas a seguir.

2.2.1. Hash

Realize o *hash* da chave privada utilizando SHAKE256 e recupere os 114 primeiros *bytes*, o resultado é armazenado em um *buffer* (h) de 114 *bytes*. Apenas os 57 *bytes* inferiores são utilizados para gerar a chave pública. Esses passos são realizados no método `getPublicKey`.

```
EdPublicKey getPublicKey() {
  // privateKey = C4EAB05D357007C632F3DBB48489924D552B08FE0C353A0D4A1F00
    ACDA2C463AFBEA67C5E8D2877C5E3BC397A659949EF8021E954E0A12274E

  Uint8List h = this._hashPrivateKey();
  // h = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA955151
    AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276804F295792C6751240FFB
    31682EFC31B6FF55C731C387C41F1126419059F73D4199F539E559EE1B4DBA8E6316CB0
    C288304B6AAF61EA2CA8E2

  Uint8List leftHalf = h.sublist(0, this.curve.keySize);
  // leftHalf = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276

  ...
}
```

2.2.2. Recuperando o escalar secreto (s)

A recuperação do escalar privado é feita pelo método `_getPrivateScalar`. Para recuperá-lo é necessário ajustar alguns elementos do *buffer*, que é realizado no método `_pruningBuffer`. Os dois *bits* menos significativos do primeiro *byte* são zerados, todos os

oito *bits* do último *byte* são zerados e o *bit* mais alto do penúltimo *byte* é ajustado/setado. O resultado é, então, interpretado como *little endian*.

```
Uint8List _pruningBuffer(Uint8List buffer) {
  // buffer = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA9
    55151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276

  if (this.curve.curveName == 'Ed448' ||
      this.curve.curveName == 'Ed521') {
    buffer[0] &= 0xFC;
    // buffer = 88A95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276

    buffer[buffer.length - 1] = 0;
    // buffer = 88A95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF200

    buffer[buffer.length - 2] |= 0x80;
    // buffer = 88A95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF200
  } else {
    throw Exception("Curve not supported");
  }

  // buffer = 88A95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA9
    55151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF200
  return buffer;
}

Uint8List _getPrivateScalar(leftHalf) {
  // leftHalf = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276

  Uint8List a = _pruningBuffer(leftHalf);
  // a = 88A95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA955151
    AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF200

  Uint8List s = toLittleEndian(a);
  // s = 00F2FE3AD28FAD21358FF9C369C24B14DC010E8E041603DEAF515195AAC6DC63
    F745ECFE4B76E07715C6C0BA822C7C79C3234F7035905EA988

  return s;
}

EdPublicKey getPublicKey() {
  ...

  // leftHalf = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6A
```

```
    A955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276
  Uint8List s = _getPrivateScalar(leftHalf);
  // s = 00F2FE3AD28FAD21358FF9C369C24B14DC010E8E041603DEAF515195AAC6DC63
    F745ECFE4B76E07715C6C0BA822C7C79C3234F7035905EA988

  ...
}
```

2.2.3. Ponto da chave pública

Para recuperar o ponto da chave pública, é preciso realizar a multiplicação do escalar secreto (s) pelo ponto gerador da curva. Essa multiplicação é realizada no método `_getPublicKeyPoint`.

```
Point _getPublicKeyPoint(s) {
  // s = 00F2FE3AD28FAD21358FF9C369C24B14DC010E8E041603DEAF515195AAC6DC63
    F745ECFE4B76E07715C6C0BA822C7C79C3234F7035905EA988

  Point publicKeyPoint = this.curve.generator.mul(s); // generator * s
  /*
  publicKeyPoint = {
    x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B9
    69BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
    y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37
    C058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
  }
  */
  return publicKeyPoint;
}

EdPublicKey getPublicKey() {
  ...

  // s = 00F2FE3AD28FAD21358FF9C369C24B14DC010E8E041603DEAF515195AAC6DC63
    F745ECFE4B76E07715C6C0BA822C7C79C3234F7035905EA988
  Point publicKeyPoint = _getPublicKeyPoint(s);
  /*
  publicKeyPoint = {
    x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B9
    69BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
    y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37
    C058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
  }
  */

  ...
}
```

2.2.4. Encoding do ponto da chave pública

A chave pública A é o *encoding* do ponto da chave pública. Primeiro codifique a coordenada y (no intervalo 0 <= y <p) como um *little endian* de 57 *bytes*. O *bit* mais significativo do último *byte* é sempre zero. Para finalizar a codificação, copie o *bit* menos significativo da coordenada x para o *bit* mais significativo do último *byte*. O resultado é a chave pública. O *encoding* é realizado pelo método `encodePoint` definido na classe `TwistedEdwardCurve`.

```
1  // Método da classe TwistedEdwardCurve
2  Uint8List encodePoint(Point point) {
3    /*
4    point = {
5      x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B9
       69BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
6      y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37
       C058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
7    }
8    */
9
10   Uint8List encodedString = toLittleEndian(encodeBigInt(point.y, keySize));
11   // encodedString = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BF
       A0C6C6798C0866AEA01EB00742802B8438EA4CB82169C235160627B4C3A9400
12
13   if (point.x & BigInt.one == BigInt.one) {
14     encodedString[encodedString.length - 1] |= 0x80;
15   }
16   // encodedString = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0
       C6C6798C0866AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480
17
18   return encodedString;
19 }
20
21 EdPublicKey getPublicKey() {
22   ...
23
24   /*
25   publicKeyPoint = {
26     x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B9
       69BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
27     y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37
       C058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
28   }
29   */
30   Uint8List A = curve.encodePoint(publicKeyPoint);
31   // A = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C6C6798C0866
       AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480
```

```
}
```

2.3 Resultado da chave pública

A partir da chave privada é possível gerar a seguinte chave pública:

$0x43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C6C6798C0866AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480$

3. Assinatura

O algoritmo de assinatura do Ed448 recebe como entrada uma chave privada (57 *bytes*) e uma mensagem de tamanho arbitrário e produz como saída um par de inteiros R e S. A assinatura é definida no método `sign` da classe `EdPrivateKey`.

Para esse exemplo será utilizada a seguinte mensagem:

$0x03$

A assinatura de uma mensagem é feita da seguinte forma:

3.1. Hash, Chave pública e prefixo

Realize o *hash* da chave privada utilizando SHAKE256 e recupere os 114 primeiros *bytes*, o resultado é armazenado em um *buffer* (h) de 114 *bytes*, depois construa o escalar privado (s) a partir da primeira metade do resultado do *hash* e a chave pública (A) como descrito em 2.2.3.. O prefixo é a segunda metade do resultado do *hash*.

```
Uint8List sign(Uint8List message) {
  // message = 03
  // privateKey = C4EAB05D357007C632F3DBB48489924D552B08FE0C353A0D4A1F00ACDA2C463AFBEA67C5E8D2877C5E3BC397A659949EF8021E954E0A12274E

  Uint8List h = this._hashPrivateKey();
  // h = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276804F295792C6751240FFB31682EFC31B6FF55C731C387C41F1126419059F73D4199F539E559EE1B4DBA8E6316CB0C288304B6AAF61EA2CA8E2

  Uint8List leftHalf = h.sublist(0, this.curve.keySize);
  // leftHalf = 8BA95E9035704F23C3797C2C82BAC0C61577E0764BFEEC45F763DCC6AA955151AFDE0316048E0E01DC144BC269C3F98F3521AD8FD23AFEF276

  Uint8List s = _getPrivateScalar(leftHalf);
  // s = F2FE3AD28FAD21358FF9C369C24B14DC010E8E041603DEAF515195AAC6DC63F745ECFE4B76E07715C6C0BA822C7C79C3234F7035905EA988

  Point A = _getPublicKeyPoint(s);
  /*
  A = {
    x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B96
```

```
    9BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
     y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37C
    058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
  }
  */

  Uint8List prefix = _getSignaturePrefix(h);
  // prefix = 804F295792C6751240FFB31682EFC31B6FF55C731C387C41F1126419059F
    73D4199F539E559EE1B4DBA8E6316CB0C288304B6AAF61EA2CA8E2

  ...
}
```

3.2. Gerar o inteiro secreto (r)

Realize o *hash* da *string* "SigEd448" + 0 + 0, do prefixo e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 114 primeiros *bytes*, o resultado deve ser interpretado como *little endian*.

```
Uint8List sign(Uint8List message) {
  ...

  String m = 'SigEd448' + String.fromCharCodes([0x00, 0x00]);
  curveSigner = Uint8List.fromList(m.codeUnits);
  // curveSigner = 53696745643434380000

  hash.update(curveSigner);
  hash.update(prefix);
  hash.update(message);
  Uint8List r = hash.digest(curve.signatureSize);
  // r = 766E43478DD919F909FF39B15103CE7326A61D359F0AAAE99C3BB41726E7AF3A8
    BF7CEAA047928E6A43C67CD77A49D1A7F12BCA83D0C03CA2687741F026FBF0D2932AF95
    27E52EE8E0C38DD1FCE03E11F21CD5731477BA3EDAE8FE157C2B1115078CCC583CF0C7B
    AF9EF6B199203884B0274

  r = toLittleEndian(r);
  // r = 74024B880392196BEFF9BAC7F03C58CC8C0715112B7C15FEE8DA3EBA771473D51
    CF2113EE0FCD18DC3E0E82EE52795AF32290DBF6F021F748726CA030C3DA8BC127F1A9D
    A477CD673CA4E6287904AACEF78B3AAFE72617B43B9CE9AA0A9F351DA62673CE0351B13
    9FF09F919D98D47436E76

  ...
}
```

3.3. Calcular o inteiro R

Primeiro calcule a multiplicação de r pelo ponto gerador da curva, depois realize o *encoding* desse ponto, o resultado é a parte R da assinatura.

```
Uint8List sign(Uint8List message) {
  ...

  BigInt rBigInt = decodeBigInt(r);
  rBigInt = rBigInt % order;
  // rBigInt = E7A03ACFB682C7F764D8299F5F5BE8C0903A78218594852AB68F9A45B51
  4881EE844F3BCEFF185BDD0B8618561F29C9AABFD0DFBC2078D9

  r = encodeBigInt(rBigInt, curve.signatureSize);

  Point pointR = generator.mul(r);
  /*
  pointR = {
    x: 18CF2A2A946C7F9E3789A406436C808C775806269E475CF3DE74593E168581CB256
    7DF845963E686C01E4B339C9E13C5CCC2706649080AF7,
    y: 79D77CAB37F01BFC33F6CD8CDE628FE1B781B9C443B1412535F4107808D00BCB352
    38FD410C6B9EF77C343EB415EF17A8962BD2717F9B826
  }
  */

  Uint8List R = curve.encodePoint(pointR);
  // R = 26B8F91727BD62897AF15E41EB43C377EFB9C610D48F2335CB0BD0087810F4352
  541B143C4B981B7E18F62DE8CCDF633FC1BF037AB7CD77980

  ...
}
```

3.4. Calcular o inteiro S

Realize o *hash* da *string* "SigEd448" + 0 + 0, do inteiro R, do *encoding* do ponto da chave pública A e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 114 primeiros *bytes*, interprete o resultado como *little endian* (k).

```
Uint8List sign(Uint8List message) {
  ...
  /*
  A = {
    x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B96
    9BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
    y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37C
    058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
  }
  */

  Uint8List eA = curve.encodePoint(A);
  // eA = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C6C6798C086
  6AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480

```

```
13   String m = 'SigEd448' + String.fromCharCodes([0x00, 0x00]);
14   curveSigner = Uint8List.fromList(m.codeUnits);
15   // curveSigner = 53696745643434380000
16
17   hash.update(curveSigner);
18   hash.update(R);
19   hash.update(eA);
20   hash.update(message);
21   Uint8List k = hash.digest(curve.signatureSize);
22   // k = A1E2CB4E8B7DD00631D36979C28C729B5DFED35ED27BA5C351EA2EC9FBFEC332E
       6F091CF2E1453D9C9536A2B5B96EE16BBBA8B52D597B817F1D949834A77046D5E044428
       77D8C18FF7067AB08E9BBB57013A19555D8227967206FFD4E18CCFC753BC1FB19A07F8A
       2003019B3AE911634A49F
23
24   k = toLittleEndian(k);
25   // k = 9FA4341691AEB3193000A2F8079AB11FBC53C7CF8CE1D4FF06729627825D55193
       A0157BB9B8EB07A06F78FC1D8772844045E6D04774A8349D9F117B897D5528BBABB16EE
       965B2B6A53C9D953142ECF91F0E632C3FEFBC92EEA51C3A57BD25ED3FE5D9B728CC279
       69D33106D07D8B4ECBE2A1
26
27   ...
28 }
```

3.5 Calcular o inteiro S e retornar a assinatura

Calcule $S = (r + k \times s) \mod L$, onde L é a ordem da curva. Para eficiência, reduza k módulo L primeiro. Após isso, a assinatura é a concatenação de R com a interpretação *little endian* de S.

```
1  Uint8List sign(Uint8List message) {
2    ...
3
4    BigInt reducedK = decodeBigInt(k) % order;
5    // reducedK = 23A737EF5F95AEAFB2F327B8453E18E3AC96091F3C13DD13F02EBDB010
       929101084904A963E38C32F7304D3F3E337FB9588BCA22C61745EC
6
7    Uint8List S = encodeBigInt(
8        (decodeBigInt(r) + (reducedK * decodeBigInt(s))) % order,
9        curve.keySize);
10   /*
11   reducedK * s = 21D77AFAFDB6F88942E45930FC6C5760B861C5EA23AF7D2439A0267B
       C4D97B274C0C22B472B49AC7207B2960C7B73C9053045F9DA6D4100650D5DFCE6D34135
       F9EB9D3C63C0B4C2AD76E8447E29AD62CE082A65EC1874C3EF4376448B9A8B012303DE5
       8D23D189522422E4FB062DF160
12
13   r + reducedK * s = 21D77AFAFDB6F88942E45930FC6C5760B861C5EA23AF7D2439A0
       267BC4D97B274C0C22B472B49AC7207B2960C7B73C9053045F9DA6D410065F4FE37B689
       C3FDF1507566032010AB6E0722BC9FAF41E7F8BEBA0031CD894C0E2BBB38488A7C86E0D
```

```
    496BA579F0B31BCEE2B5DAC24E6A39

  S = (r + reducedK * s) mod order = 3
    AC007F81C4434D2D1AFADA41AB28A34F30F2A3D95F199E7052947A6E0F70A9FC1364315
    BFECDC04BC36DF27E0B2AFE1CECBE1AAC0BC0D5E
  */

  S = toLittleEndian(S);
  // S = 5E0DBCC0AAE1CBCEE1AFB2E027DF36BC04DCECBF154336C19F0AF7E0A6472905E
    799F1953D2A0FF3348AB21AA4ADAFD1D234441CF807C03A00

  return Uint8List.fromList(R + S);
}
```

3.6 Resultado da assinatura

A partir da chave privada e da mensagem é possível gerar a seguinte assinatura:

$0x26B8F91727BD62897AF15E41EB43C377EFB9C610D48F2335CB0BD0087810F4352541B143C4B981B7E18F62DE8CCDF633FC1BF037AB7CD779805E0DBCC0AAE1CBCEE1AFB2E027DF36BC04DCECBF154336C19F0AF7E0A6472905E799F1953D2A0FF3348AB21AA4ADAFD1D234441CF807C03A00$

4. Verificação

A verificação do Ed448 recebe como parâmetro a chave pública (57 *bytes*), uma mensagem de tamanho arbitrário e uma assinatura R, S (114 *bytes*) e produz como saída um booleano da verificação se a assinatura é válida ou não para a mensagem. A verificação é definida no método `verify` da classe `EdPublicKey`.

A verificação é feita da seguinte forma:

4.1. Decoding da assinatura e da chave pública

Divida a assinatura em duas metades de 57 *bytes*. Após a divisão da assinatura, decodifique o R e a chave pública (A) como pontos. O método para realizar esse procedimento é o `decodePoint` definido na classe `TwistedEdwardCurve`.

```
// decode R
Point decodePoint(Uint8List encodedPoint) {
  // encodedPoint = 26B8F91727BD62897AF15E41EB43C377EFB9C610D48F2335CB0BD0
    087810F4352541B143C4B981B7E18F62DE8CCDF633FC1BF037AB7CD77980

  Uint8List eP = Uint8List.fromList(encodedPoint); // object copy
  int size = eP.length;
  // size = 57

  int sign = eP[size - 1] & 0x80;
  // sign = 128
```

```
11
12    eP[size - 1] &= ~0x80;
13    // eP = 26B8F91727BD62897AF15E41EB43C377EFB9C610D48F2335CB0BD0087810F435
        2541B143C4B981B7E18F62DE8CCDF633FC1BF037AB7CD77900
14
15    BigInt y = decodeBigInt(toLittleEndian(eP));
16    // y = 79D77CAB37F01BFC33F6CD8CDE628FE1B781B9C443B1412535F4107808D00BCB3
        5238FD410C6B9EF77C343EB415EF17A8962BD2717F9B826
17
18    BigInt x = _xRecover(y, sign);
19    // x = 18CF2A2A946C7F9E3789A406436C808C775806269E475CF3DE74593E168581CB2
        567DF845963E686C01E4B339C9E13C5CCC2706649080AF7
20
21    return Point(x, y, this);
22  }
23
24  // decode A
25  Point decodePoint(Uint8List encodedPoint) {
26    // encodedPoint = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C
        6C6798C0866AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480
27
28    Uint8List eP = Uint8List.fromList(encodedPoint); // object copy
29    int size = eP.length;
30    // size = 57
31
32    int sign = eP[size - 1] & 0x80;
33    // sign = 128
34
35    eP[size - 1] &= ~0x80;
36    // eP = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C6C6798C086
        6AEA01EB00742802B8438EA4CB82169C235160627B4C3A9400
37
38    BigInt y = decodeBigInt(toLittleEndian(eP));
39    // y = 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B3
        7C058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43
40
41    BigInt x = _xRecover(y, sign);
42    // x = D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B
        969BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857
43
44    return Point(x, y, this);
45  }
46
47  bool verify(Uint8List message, Uint8List signature) {
48    Curve curve = this.curve;
49    BigInt order = curve.order;
50    int signatureSize = curve.signatureSize;
```

```
  Uint8List A = this.publicKey;

  Uint8List R = signature.sublist(0, signatureSize >> 1);
  // R = 26B8F91727BD62897AF15E41EB43C377EFB9C610D48F2335CB0BD0087810F4352
    541B143C4B981B7E18F62DE8CCDF633FC1BF037AB7CD77980

  Uint8List S = toLittleEndian(signature.sublist(signatureSize >> 1));
  // S = 3AC007F81C4434D2D1AFADA41AB28A34F30F2A3D95F199E7052947A6E0F70A9FC
    1364315BFECDC04BC36DF27E0B2AFE1CECBE1AAC0BC0D5E

  Point pointR = curve.decodePoint(R);
  /*
  pointR = {
    x: 18CF2A2A946C7F9E3789A406436C808C775806269E475CF3DE74593E168581CB256
    7DF845963E686C01E4B339C9E13C5CCC2706649080AF7,
    y: 79D77CAB37F01BFC33F6CD8CDE628FE1B781B9C443B1412535F4107808D00BCB352
    38FD410C6B9EF77C343EB415EF17A8962BD2717F9B826,
  }
  */

  Point pointA = curve.decodePoint(A);
  /*
  pointA = {
    x: D2FFD9B53D3BFF141F68CF0248020A14E2F1ABA433258C33C290607F78EAB2C6B96
    9BDF226775BA7DB74BCF20FFDDAB9D3ADA6B23A0EE857,
    y: 943A4C7B626051239C1682CBA48E43B802287400EB01EA6A86C098676C0CFA2B37C
    058935DA534C80ACD7E5F5431E56A45FFCD30F428BA43,
  }
  */
  ...
}
```

4.2. Calculando k

Realize o *hash* da *string* "SigEd448" + 0 + 0, do R, da chave pública e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 114 primeiros *bytes*, interprete em *little endian* o resultado como um inteiro k.

```
bool verify(Uint8List message, Uint8List signature) {
  ...

  String m = 'SigEd448' + String.fromCharCodes([0x00, 0x00]);
  curveSigner = Uint8List.fromList(m.codeUnits);
  // curveSigner = 53696745643434380000

  hash.update(curveSigner);
  hash.update(R);
  hash.update(A);
```

```
  hash.update(message);

  Uint8List k = hash.digest(signatureSize);
  // k = A1E2CB4E8B7DD00631D36979C28C729B5DFED35ED27BA5C351EA2EC9FBFEC332E
    6F091CF2E1453D9C9536A2B5B96EE16BBBA8B52D597B817F1D949834A77046D5E044428
    77D8C18FF7067AB08E9BBB57013A19555D8227967206FFD4E18CCFC753BC1FB19A07F8A
    2003019B3AE911634A49F

  k = toLittleEndian(k);
  // k = 9FA4341691AEB3193000A2F8079AB11FBC53C7CF8CE1D4FF06729627825D55193
    A0157BB9B8EB07A06F78FC1D8772844045E6D04774A8349D9F117B897D5528BBABB16EE
    965B2B6A53C9D953142ECF91F0E632C3FEFBC92EEA51C3A57BD25ED3FE5D9B728CC2796
    9D33106D07D8B4ECBE2A1
  ...
}
```

4.3. Verificando o grupo da equação $[S]Generator = pointR + [k]pointA$

Para validar a assinatura, é preciso verificar se S vezes o ponto gerador é igual à soma do ponto R com a multiplicação de S com o ponto A.

```
bool verify(Uint8List message, Uint8List signature) {
  ...

  BigInt reducedK = decodeBigInt(k) % order;
  k = encodeBigInt(reducedK, curve.keySize);
  // k = 23A737EF5F95AEAFB2F327B8453E18E3AC96091F3C13DD13F02EBDB0109291010
    84904A963E38C32F7304D3F3E337FB9588BCA22C61745EC

  Point left = pointA.mul(k).add(pointR); // (pointA * k) + pointR
  /*
  left = {
    x: C6BAC9BABF158F9B04AA9E2198C78F12ED271B0E4454FCEC287DE925BAFFFE0264B
    18376532927D55A4497EAF511F11B425F6474CE9B5BB2,
    y: A1777B30B1848EFF749BF26F7C304F97C9B94BAC3D48D1A8EBC975F0079CFAD59A3
    D2ECC6C992415B6D127EE9E36CB979295B8FE28F054DB
  }
  */

  Point right = curve.generator.mul(S); // generator * S
  /*
  right = {
    x: C6BAC9BABF158F9B04AA9E2198C78F12ED271B0E4454FCEC287DE925BAFFFE0264B
    18376532927D55A4497EAF511F11B425F6474CE9B5BB2,
    y: A1777B30B1848EFF749BF26F7C304F97C9B94BAC3D48D1A8EBC975F0079CFAD59A3
    D2ECC6C992415B6D127EE9E36CB979295B8FE28F054DB
  }
  */
```

```
23
24   return left == right;
25 }
```

APÊNDICE B – Documentação Ed521

1. Sobre o Ed521

O algoritmo Ed521 utiliza a forma de Edward da curva E-521 definida no documento EdDSA for more curves. Os parâmetros dessa curva são:

$p : 2^{521} - 1 = 0x1FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$
$FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$
$FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$

$l(ordem) : 2^{519} - 337554763258501705789107630418782636071904961214051226618635150085779108655765 = 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF$
$FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD15B6C64746FC85F736B8A$
$F5E7EC53F04FBD8C4569A8F1F4540EA2435F5180D6B$

a : 1

d : -376014

X(G) (coordenada x do ponto gerador) : 0x752CB45C48648B189DF90CB2296B2878A3BFD9F42FC6C818EC8BF3C9C0C6203913F6ECC5CCC72434B1AE949D568FC99C6059D0FB13364838AA302A940A2F19BA6C

Y(G) (coordenada y do ponto gerador) : 0x0C

cofator : 4

Função *hash* utilizada: Shake256

Essa curva é definida na classe `TwistedEdwardCurve`. Para utilizá-la é preciso instanciar dessa forma:

```
var ed521 = TwistedEdwardCurve.ed521();
```

2. Geração de chaves

Ed521 utiliza uma chave pública e privada de 66 *bytes*.

2.1. Chave privada

A chave privada é composta por 66 *bytes* gerados de forma criptograficamente aleatória e segura. Ela é definida pela classe `EdPrivateKey` e para gerá-la de forma aleatória é preciso chamar o método `generate` e passar como parâmetro a curva que está sendo utilizada:

```
EdPrivateKey privateKey = EdPrivateKey.generate(ed521);
```

Outra forma de instanciar uma chave privada é utilizando o método `fromBytes` e passar como parâmetro uma cadeia de *bytes* (Uint8List) e a curva que está sendo utilizada:

```
EdPrivateKey privateKey = EdPrivateKey.fromBytes(privateKeyBytes, ed521);
```

Para esse exemplo vamos utilizar a chave privada em hexadecimal: $0x2367B29BCEACA04D6FE50D959DEE3D706F3A5F605CE5AAC0205C54CDDA59145C573B932814C7405770CD46171A0EB44C911F8E851ADEEFCC77E5841BF86E0B6486DD$

2.2. Chave pública

A chave pública é definida pela classe `EdPublicKey` e pode ser obtida a partir da classe `EdPrivateKey` utilizando o método `getPublicKey`:

```
EdPublicKey publicKey = privateKey.getPublicKey();
```

Sua geração acontecem em 4 etapas que serão demonstradas a seguir.

2.2.1. Hash

Realize o *hash* da chave privada utilizando SHAKE256 e recupere os 132 primeiros *bytes*, o resultado é armazenado em um *buffer* (h) de 132 *bytes*. Apenas os 66 *bytes* inferiores são utilizados para gerar a chave pública. Esses passos são realizados no método `getPublicKey`.

```
EdPublicKey getPublicKey() {
  // privateKey = 2367B29BCEACA04D6FE50D959DEE3D706F3A5F605CE5AAC0205C54CD
    DA59145C573B932814C7405770CD46171A0EB44C911F8E851ADEEFCC77E5841BF86E0B6
    486DD

  Uint8List h = this._hashPrivateKey();
  // h = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03
    746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C440380F
    590C13657792D96FA5BF0379507CDD781EB47126F240B015D0C7ADBB297722E1B4D37C
    106CE7C7E1596708B333AAA9557E1937B3A6E6C7565B8CA9EAD764B506

  Uint8List leftHalf = h.sublist(0, this.curve.keySize);
  // leftHalf = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB
    83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C
    440

  ...
}
```

2.2.2. Recuperando o escalar secreto (s)

A recuperação do escalar privado é feita pelo método `_getPrivateScalar`. Para recuperá-lo é necessário ajustar alguns elementos do *buffer*, que é realizado no método `_pruningBuffer`. Os dois *bits* menos significativos do primeiro *byte* são zerados, todos os

oito *bits* do último *byte* são zerados e o *bit* mais alto do penúltimo *byte* é ajustado/setado. O resultado é então interpretado como *little endian*.

```
Uint8List _pruningBuffer(Uint8List buffer) {
  // buffer = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C440

  if (this.curve.curveName == 'Ed448' || this.curve.curveName == 'Ed521') {
    buffer[0] &= 0xFC;
    // buffer = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C440

    buffer[buffer.length - 1] = 0;
    // buffer = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C400

    buffer[buffer.length - 2] |= 0x80;
    // buffer = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C400
  } else {
    throw Exception("Curve not supported");
  }

  // buffer = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C400
  return buffer;
}

Uint8List _getPrivateScalar(leftHalf) {
  // leftHalf = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C440

  Uint8List a = _pruningBuffer(leftHalf);
  // a = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C400

  Uint8List s = toLittleEndian(a);
  // s = 00C4B260A50FD887D8B41BD7F5D34F07FB2FBFD3F54BE2AD72E63A862CF7B8DD4637A0E683EB46F67EE317084510BDDEDF6548878431AC903B16C2EDB52839A96EF8

  return s;
}

```

```
33 EdPublicKey getPublicKey() {
34   ...
35
36   // lefthalf = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646
       EB83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560
       B2C440
37   Uint8List s = _getPrivateScalar(leftHalf);
38   // s = 00C4B260A50FD887D8B41BD7F5D34F07FB2FBFD3F54BE2AD72E63A862CF7B8D
       D4637A0E683EB46F67EE317084510BDDEDF6548878431AC903B16C2EDB52839A96EF8
39
40   ...
41 }
```

2.2.3. Ponto da chave pública

Para recuperar o ponto da chave pública, é preciso realizar a multiplicação por escalar do escalar secreto (s) com o ponto gerador da curva. Essa multiplicação é realizada no método `_getPublicKeyPoint`.

```
1  Point _getPublicKeyPoint(s) {
2    // s = 00C4B260A50FD887D8B41BD7F5D34F07FB2FBFD3F54BE2AD72E63A862CF7B8D
       D4637A0E683EB46F67EE317084510BDDEDF6548878431AC903B16C2EDB52839A96EF8
3
4    Point publicKeyPoint = this.curve.generator.mul(s); // generator * s
5    /*
6    publicKeyPoint = {
7      x: A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B93D
       E27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
8      y: 1F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC4875
       7B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
9    }
10   */
11   return publicKeyPoint;
12 }
13
14 EdPublicKey getPublicKey() {
15   ...
16
17   // s = 00C4B260A50FD887D8B41BD7F5D34F07FB2FBFD3F54BE2AD72E63A862CF7B8D
       D4637A0E683EB46F67EE317084510BDDEDF6548878431AC903B16C2EDB52839A96EF8
18   Point publicKeyPoint = _getPublicKeyPoint(s);
19   /*
20   publicKeyPoint = {
21     x: 00A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530
       B93DE27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
22     y: 01F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC4
       8757B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
23   }
```

```
  */

  ...
}
```

2.2.4. Encoding do ponto da chave pública

A chave pública A é o encoding do ponto da chave pública. Primeiro codifique a coordenada y (no intervalo $0 \leq y < p$) como um *little endian* de 66 *bytes*. O *bit* mais significativo do último *byte* é sempre zero. Para fazer a codificação, copie o *bit* menos significativo da coordenada x para o *bit* mais significativo do último *byte*. O resultado é a chave pública. O encoding é realizado pelo método `encodePoint` definido na classe `TwistedEdwardCurve`.

```
// Método da classe TwistedEdwardCurve
Uint8List encodePoint(Point point) {
  /*
  point = {
    x: 00A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B93DE27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
    y: 01F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC48757B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
  }
  */

  Uint8List encodedString = toLittleEndian(encodeBigInt(point.y, keySize));
  // encodedString = 72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF001

  if (point.x & BigInt.one == BigInt.one) {
    encodedString[encodedString.length - 1] |= 0x80;
  }
  // encodedString = 72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF081

  return encodedString;
}

EdPublicKey getPublicKey() {
  ...

  /*
  publicKeyPoint = {
    x: 00A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B93DE27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
```

```
    y: 01F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC487
    57B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
  }
  */
  Uint8List A = curve.encodePoint(publicKeyPoint);
  // A = 43BA28F430CDFF456AE531545F7ECD0AC834A55D9358C0372BFA0C6C6798C0866
    AEA01EB00742802B8438EA4CB82169C235160627B4C3A9480
}
```

2.3 Resultado da chave pública

A partir da chave privada é possível gerar a seguinte chave pública:

$0x72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF081$

1.3. Assinatura

O algoritmo de assinatura do Ed521 recebe como entrada uma chave privada (66 *bytes*) e uma mensagem de tamanho arbitrário e produz como saída um par de inteiros R e S. A assinatura é definida no método `sign` da classe `EdPrivateKey`.

Para esse exemplo será utilizada a seguinte mensagem:

$0x03$

A assinatura de uma mensagem é feita da seguinte forma:

3.1. Hash, Chave pública e prefixo

Realize o *hash* da chave privada utilizando SHAKE256 e recupere os 132 primeiros *bytes*, o resultado é armazenado em um *buffer* (h) de 132 *bytes*, depois construa o escalar privado (s) a partir da primeira metade do resultado do *hash* e a chave pública (A) como descrito em 2.2.3.. O prefixo é a segunda metade do resultado do *hash*.

```
Uint8List sign(Uint8List message) {
  // message = 03
  // privateKey = 2367B29BCEACA04D6FE50D959DEE3D706F3A5F605CE5AAC0205C54CD
    DA59145C573B932814C7405770CD46171A0EB44C911F8E851ADEEFCC77E5841BF86E0B6
    486DD

  Uint8List h = this._hashPrivateKey();
  // h = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB83E6A03
    746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C440380F
    590C13657792D96FA5BF0379507CDD781EB47126F240B015D0C7ADBB297722E1B4D37C1
    06CE7C7E1596708B333AAA9557E1937B3A6E6C7565B8CA9EAD764B506

  Uint8List leftHalf = h.sublist(0, this.curve.keySize);
  // leftHalf = F86EA93928B5EDC2163B90AC3184874865DFDEBD10450817E37EF646EB
```

```
    83E6A03746DDB8F72C863AE672ADE24BF5D3BF2FFB074FD3F5D71BB4D887D80FA560B2C
    440

  Uint8List s = _getPrivateScalar(leftHalf);
  // s = 00C4B260A50FD887D8B41BD7F5D34F07FB2FBFD3F54BE2AD72E63A862CF7B8DD4
    637A0E683EB46F67EE317084510BDDEDF6548878431AC903B16C2EDB52839A96EF8

  Point A = _getPublicKeyPoint(s);
  /*
  A = {
    x: 00A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B9
    3DE27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
    y: 01F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC487
    57B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
  }
  */

  Uint8List prefix = _getSignaturePrefix(h);
  // prefix = 380F590C13657792D96FA5BF0379507CDD781EB47126F240B015D0C7ADBB
    297722E1B4D37C106CE7C7E1596708B333AAA9557E1937B3A6E6C7565B8CA9EAD764B506

  ...
}
```

3.2. Gerar o inteiro secreto (r)

Realize o *hash* da *string* "SigEd521" + 0 + 0, do prefixo e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 132 primeiros *bytes*, o resultado deve ser interpretado como *little endian*.

```
Uint8List sign(Uint8List message) {
  ...

  String m = 'SigEd521' + String.fromCharCodes([0x00, 0x00]);
  curveSigner = Uint8List.fromList(m.codeUnits);
  // curveSigner = 393900767670058993385472

  hash.update(curveSigner);
  hash.update(prefix);
  hash.update(message);
  Uint8List r = hash.digest(curve.signatureSize);
  // r = A1A28A471B985043C9AC0743CBEEC1F564995AD6F144C2691056DEFCE6B85FD57
    0F735C8515730A4F24E20ED19C49B530EC7BA14304DFCD6A6AABC52C9994996FC2E3396
    ADD56BB3290C826D5D9B8247780C8AE34378ECBC58CF2F95C09BA9C95A4EE71AF78B435
    380577B83400B2040189A70FF5E8DCBEA4221303B25EE5112309DE826

  r = toLittleEndian(r);
  // r = 26E89D301251EE253B302142EACB8D5EFF709A1840200B40837B578053438BF71
```

```
    AE74E5AC9A99BC0952FCF58BCEC7843E38A0C7847829B5D6D820C29B36BD5AD96332EFC
    964999C952BCAAA6D6FC4D3014BAC70E539BC419ED204EF2A4305751C835F770D55FB8
    E6FCDE561069C244F1D65A9964F5C1EECB4307ACC94350981B478AA2A1

  ...
}
```

3.3. Calcular o inteiro R

Primeiro calcule a multiplicação de r pelo ponto gerador da curva, depois realize o encoding desse ponto, o resultado é a parte R da assinatura.

```
Uint8List sign(Uint8List message) {
  ...
  BigInt rBigInt = decodeBigInt(r);
  rBigInt = rBigInt % order;
  // rBigInt = 492350657D690F6AE3B36994A0F340E54B3556310D0E999585C81A8F17F
    27C06A4FE94DAEA1A55AF5598EA5D2278C36D5894D9D3031E880529F125AEE1392F18CD

  r = encodeBigInt(rBigInt, curve.signatureSize);
  // r = 007C964999C952BCAAA6D6FC4D3014BAC70E539BC419ED204EF2A4305751C835F
    87FF1F9B1023321AB4188AA8F9DC8B2B49578029B551D0950A0AE4170813DCDC2C2

  Point pointR = generator.mul(r);
  /*
  pointR = {
    x: 010476E7F95E187A0E03B2C1F3710B5953A060FFA1F784A3EA9BF2BAAB4687DC3EB
    EF403D32BBF0A34A977B3BD4FA44C858D3F2FC70A2AC2EF18A2BCE2FAAF2A6C6F,
    y: 000B88FEE36016F0F4653C262D4C4553EEE3D0012CB26E92C9CBDFB373185651935
    207ADF5712E06604E92D8D611C81421056C8BCD05583210D3B62FA293FBE3F659
  }
  */

  Uint8List R = curve.encodePoint(pointR);
  // R = 59F6E3FB93A22FB6D310325805CD8B6C052114C811D6D8924E60062E71F5AD075
    29351561873B3DFCBC9926EB22C01D0E3EE53454C2D263C65F4F01660E3FE880B80

  ...
}
```

3.4. Calcular o inteiro k

Realize o *hash* da *string* "SigEd521" + 0 + 0, do inteiro R, do encoding do ponto da chave pública A e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 132 primeiros *bytes*, interprete o resultado como *little endian* (k).

```
Uint8List sign(Uint8List message) {
  ...
  /*
```

```
  A = {
    x: A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B93D
    E27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
    y: 1F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC4875
    7B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
  }
  */

  Uint8List eA = curve.encodePoint(A);
  // eA = 72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B
    7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF081

  String m = 'SigEd521' + String.fromCharCodes([0x00, 0x00]);
  curveSigner = Uint8List.fromList(m.codeUnits);
  // curveSigner = 393900767670058993385472

  hash.update(curveSigner);
  hash.update(R);
  hash.update(eA);
  hash.update(message);
  Uint8List k = hash.digest(curve.signatureSize);
  // k = FFC0888CBC9D36737C7DED5FF22A5BFBB80CFEC1D1E24E5043DD5C8E17FAD80A7
    7ACDE025B50920F47AB243F67A804DCD335053E4905CA7FF6D25F32A85CD6A3E1BAA527
    8A6AA332FF6669D5CE2AFB8B983E8227195AFEE0471889393153411947D58F0EC9ABA8A
    EEC2613FC29CA2A73F5C2EFF1B6F16C65861CFDF529B53E07D1083471

  k = toLittleEndian(k);
  // k = 713408D1073EB529F5FD1C86656CF1B6F1EFC2F5732ACA29FC1326ECAEA8ABC90
    E8FD5471941533139891847E0FE5A1927823E988BFB2ACED56966FF32A36A8A27A5BAE1
    A3D65CA8325FD2F67FCA05493E0535D3DC04A8673F24AB470F92505B02DEAC770AD8FA1
    78E5CDD43504EE2D1C1FE0CB8FB5B2AF25FED7D7C73369DBC8C88C0FF

  ...
}
```

3.5. Calcular o inteiro S e retornar a assinatura

Calcule $S = (r + k \times s) \mod L$, onde L é a ordem da curva. Para eficiência, reduza k módulo L primeiro. Após isso, a assinatura é a concatenação de R com a interpretação *little endian* de S.

```
Uint8List sign(Uint8List message) {
  ...

  BigInt reducedK = decodeBigInt(k) % order;
  // reducedK = 5E957D1EE76C185C99CB9BE712822D600B59210CDC743460EFB194948D1
    4BD4451407FB48A1A263BE565F5855FAE010F068B193E86544F5B2052333A748848FA25

```

```
  Uint8List S = encodeBigInt(
      (decodeBigInt(r) + (reducedK * decodeBigInt(s))) % order,
      curve.keySize);
  /*
  reducedK * s = 48AC5B71B3CBAA2ACF3C1C9256ED46F6F72CDA6B68F9D3D2893C9F07D
   D76DAA9E95CAD625038BEC31BC721EB883DEDFE9BF95A527DD1CA469414E1C4080019E3
   117209C324C7BE3D8E2DD5C0D1C74CF38F5CF80FE851D45AE431E1FBCFB2FA6E1304181
   5714017F6122AEE637B747EE946A40044288CB6E6DC55B0892A1AC19B39D8

  r + reducedK * s = 48AC5B71B3CBAA2ACF3C1C9256ED46F6F72CDA6B68F9D3D2893C9
   F07DD76DAA9E95CAD625038BEC31BC721EB883DEDFE9BF95A527DD1CA469414E1C40800
   19E311BB2D138A45274CF911892A6668403474A82D66195EE2F479B7AA165ECAECEA19A
   916AA4C2A324BC180874DD896F7ACB3FC951DFB8FD56EE17FA1AED8FBFACA52A5

  S = (r + reducedK * s) mod order = 3
   AC007F81C4434D2D1AFADA41AB28A34F30F2A3D95F199E7052947A6E0F70A9FC1364315
   BFECDC04BC36DF27E0B2AFE1CECBE1AAC0BC0D5E
  */

  S = toLittleEndian(S);
  // S = 1B23A076975FCBB327E756A3A486E791745BF5120E69C4D908CA3F04BEA5E9587
   74BAB073A8DD7B7D47737CEE5FC3F4D7D7881B13A7540AF8E40BE02722D5DEBC4

  return Uint8List.fromList(R + S);
}
```

3.6. Resultado da assinatura

A partir da chave privada e da mensagem é possível gerar a seguinte assinatura:

$0x59F6E3FB93A22FB6D310325805CD8B6C052114C811D6D8924E60062E71F5AD07529351561873B3DFCBC9926EB22C01D0E3EE53454C2D263C65F4F01660E3FE880B80C4EB5D2D7202BE408EAF40753AB181787D4D3FFCE5CE3777D4B7D78D3A07AB4B7758E9A5BE043FCA08D9C4690E12F55B7491E786A4A356E727B3CB5F9776A0231B00$

4. Verificação

A verificação do Ed521 recebe como parâmetro a chave pública (66 *bytes*), uma mensagem de tamanho arbitrário e uma assinatura R, S (132 *bytes*) e produz como saída um booleano da verificação se a assinatura é válida ou não para a mensagem. A verificação é definida no método `verify` da classe `EdPublicKey`.

A verificação é feita da seguinte forma:

4.1. Decoding da assinatura e da chave pública

Divida a assinatura em duas metades de 66 *bytes*. Após a divisão da assinatura, decodifique o R e a chave pública (A) como pontos. O método para realizar esse procedi-

mento é o `decodePoint` definido na classe `TwistedEdwardCurve`.

```
// decode R
Point decodePoint(Uint8List encodedPoint) {
  // encodedPoint = 59F6E3FB93A22FB6D310325805CD8B6C052114C811D6D8924E60062E71F5AD07529351561873B3DFCBC9926EB22C01D0E3EE53454C2D263C65F4F01660E3FE880B80

  Uint8List eP = Uint8List.fromList(encodedPoint); // object copy
  int size = eP.length;
  // size = 66

  int sign = eP[size - 1] & 0x80;
  // sign = 132

  eP[size - 1] &= ~0x80;
  // eP = 59F6E3FB93A22FB6D310325805CD8B6C052114C811D6D8924E60062E71F5AD07529351561873B3DFCBC9926EB22C01D0E3EE53454C2D263C65F4F01660E3FE880B00

  BigInt y = decodeBigInt(toLittleEndian(eP));
  // y = B88FEE36016F0F4653C262D4C4553EEE3D0012CB26E92C9CBDFB373185651935207ADF5712E06604E92D8D611C81421056C8BCD05583210D3B62FA293FBE3F659

  BigInt x = _xRecover(y, sign);
  // x = 10476E7F95E187A0E03B2C1F3710B5953A060FFA1F784A3EA9BF2BAAB4687DC3EBEF403D32BBF0A34A977B3BD4FA44C858D3F2FC70A2AC2EF18A2BCE2FAAF2A6C6F

  return Point(x, y, this);
}

// decode A
Point decodePoint(Uint8List encodedPoint) {
  // encodedPoint = 72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF081

  Uint8List eP = Uint8List.fromList(encodedPoint); // object copy
  int size = eP.length;
  // size = 66

  int sign = eP[size - 1] & 0x80;
  // sign = 132

  eP[size - 1] &= ~0x80;
  // eP = 72C86F2D561E5D3DB6E350A92B0287A2434207B3869CD013CCBF1034C82B647B7548BC927975DA76E750C991A443A11D23A264888B36C73E6A48A2602F777D3FF001

  BigInt y = decodeBigInt(toLittleEndian(eP));
```

```
39    // y = 1F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC48
        757B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872
40
41    BigInt x = _xRecover(y, sign);
42    // x = A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B9
        3DE27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5
43
44    return Point(x, y, this);
45  }
46
47  bool verify(Uint8List message, Uint8List signature) {
48    Curve curve = this.curve;
49    BigInt order = curve.order;
50    int signatureSize = curve.signatureSize;
51    Uint8List A = this.publicKey;
52
53    Uint8List R = signature.sublist(0, signatureSize >> 1);
54    // R = 59F6E3FB93A22FB6D310325805CD8B6C052114C811D6D8924E60062E71F5AD075
        29351561873B3DFCBC9926EB22C01D0E3EE53454C2D263C65F4F01660E3FE880B80
55
56    Uint8List S = toLittleEndian(signature.sublist(signatureSize >> 1));
57    // S = 1B23A076975FCBB327E756A3A486E791745BF5120E69C4D908CA3F04BEA5E9587
        74BAB073A8DD7B7D47737CEE5FC3F4D7D7881B13A7540AF8E40BE02722D5DEBC4
58
59    Point pointR = curve.decodePoint(R);
60    /*
61    pointR = {
62      x: 10476E7F95E187A0E03B2C1F3710B5953A060FFA1F784A3EA9BF2BAAB4687DC3EBE
        F403D32BBF0A34A977B3BD4FA44C858D3F2FC70A2AC2EF18A2BCE2FAAF2A6C6F,
63      y: B88FEE36016F0F4653C262D4C4553EEE3D0012CB26E92C9CBDFB373185651935207
        ADF5712E06604E92D8D611C81421056C8BCD05583210D3B62FA293FBE3F659,
64    }
65    */
66
67    Point pointA = curve.decodePoint(A);
68    /*
69    pointA = {
70      x: A7104267A1A89C35214F37DB17948CA9BB9F18671277BEAC3B49112BACD1530B93D
        E27BF67B952C8C8543619E9C3E6D625BBFEBB797730298C68AC2E33C61369E5,
71      y: 1F03F7D772F60A2486A3EC7368B8864A2231DA143A491C950E776DA757992BC4875
        7B642BC83410BFCC13D09C86B3074243A287022BA950E3B63D5D1E562D6FC872,
72    }
73    */
74    ...
75  }
```

Realize o *hash* da *string* "SigEd521" + 0 + 0, do R, da chave pública e da mensagem a ser assinada utilizando o SHAKE256 e recupere os 132 primeiros *bytes*, interprete em *little endian* o resultado como um inteiro k.

```
bool verify(Uint8List message, Uint8List signature) {
  ...

  String m = 'SigEd521' + String.fromCharCodes([0x00, 0x00]);
  curveSigner = Uint8List.fromList(m.codeUnits);
  // curveSigner = 393900767670058993385472

  hash.update(curveSigner);
  hash.update(R);
  hash.update(A);
  hash.update(message);

  Uint8List k = hash.digest(signatureSize);
  // k = FFC0888CBC9D36737C7DED5FF22A5BFBB80CFEC1D1E24E5043DD5C8E17FAD80A7
    7ACDE025B50920F47AB243F67A804DCD335053E4905CA7FF6D25F32A85CD6A3E1BAA527
    8A6AA332FF6669D5CE2AFB8B983E8227195AFEE0471889393153411947D58F0EC9ABA8A
    EEC2613FC29CA2A73F5C2EFF1B6F16C65861CFDF529B53E07D1083471

  k = toLittleEndian(k);
  // k = 713408D1073EB529F5FD1C86656CF1B6F1EFC2F5732ACA29FC1326ECAEA8ABC90
    E8FD5471941533139891847E0FE5A1927823E988BFB2ACED56966FF32A36A8A27A5BAE1
    A3D65CA8325FD2F67FCA05493E0535D3DC04A8673F24AB470F92505B02DEAC770AD8FA1
    78E5CDD43504EE2D1C1FE0CB8FB5B2AF25FED7D7C73369DBC8C88C0FF
  ...
}
```

4.3. Verificando o grupo da equação $[S]Generator = pointR + [k]pointA$

Para validar a assinatura, é preciso verificar se S vezes o ponto gerador é igual à soma do ponto R com a multiplicação de k com o ponto A.

```
bool verify(Uint8List message, Uint8List signature) {
  ...

  BigInt reducedK = decodeBigInt(k) % order;
  k = encodeBigInt(reducedK, curve.keySize);
  // k = 5E957D1EE76C185C99CB9BE712822D600B59210CDC743460EFB194948D14BD445
    1407FB48A1A263BE565F5855FAE010F068B193E86544F5B2052333A748848FA25

  Point left = pointA.mul(k).add(pointR); // (pointA * k) + pointR
  /*
  left = {
    x: 107DF403D2FF3040955DD1D746444571EEB1353DFE5260696FDCFA805D10CCC06A0
    CA10F0BCD9E966B4829B8AB78DA31DA7891238361AFBA1248FE8B6952131D55C,
```

```
12      y: BB8CC0A41EA8D62B3E0E54A63A92C6F94E53F9DED3E45DBCAA2D4C87B2BB91220E0
        012ED78F59CB6342392F16CB9C68EC23AF847FD62B76F7441CAB31E5DE1C76
13    }
14    */
15
16    Point right = curve.generator.mul(S); // generator * S
17    /*
18    right = {
19      x: 107DF403D2FF3040955DD1D746444571EEB1353DFE5260696FDCFA805D10CCC06A0
        CA10F0BCD9E966B4829B8AB78DA31DA7891238361AFBA1248FE8B6952131D55C,
20      y: BB8CC0A41EA8D62B3E0E54A63A92C6F94E53F9DED3E45DBCAA2D4C87B2BB91220E0
        012ED78F59CB6342392F16CB9C68EC23AF847FD62B76F7441CAB31E5DE1C76
21    }
22    */
23
24    return left == right;
25  }
```

APÊNDICE C – Vetor de testes para Ed521

-----Em branco

ALGORITMO:
Ed521

CHAVE PRIVADA:
ea1738712e2080f0bfef3599334f225f
dcdc9fd4db056d5ed626b26a7b970359
e55fcb48c134aba7825d14a9f7109473
462976c95fd09ce2a5b74416a5977f71
04ad

CHAVE PÚBLICA:
c1922a80cfad9eec691b7909e8bef922
e40c0f0f86a4793fac84d44238859c5d
63c7e33b03c21fe9ad2ac3c4d2282ba4
45dc5140de186a9c588c43a67d87f384
d980

MENSAGEM (tamanho de 0 bytes):

ASSINATURA:
d201597855eb6d3d378932b1dff2e043
cc7690a0ac25c4181f16b98f13d646f4
1af3a6e7e3973bb60fcd55efab480124
368bb5427e5be340c0f6f1a5da6eedbf
5d80360b82f184c137e5bd741fbb6116
50c385414d683fae55720bf3cb0f6942
a6f2925cc9b90e4e6b8e747e9a0baf5a
c9b61527cadd397ecb831b2c9c223942
b5ea5600

-----1 octeto

ALGORITMO:

Ed521

CHAVE PRIVADA:
319e75376adf3818a134bad4efefa797
f62a5bb248af880db698442ffd01b1d6
872ff680c428f1e6829bdc6e0788469c
592ff4677415082cc837b5dd15f535e6
d922

CHAVE PÚBLICA:
e95be773f7f383233130b89e89b949b8
6f051def83ad7a1f79b1866653799310
4ba52e58c89b59427a2f7cbe38a3bd3a
3841413f3f63bcec33d6480fbfedc0ff
6501

MENSAGEM (tamanho de 1 byte):
03

ASSINATURA:
31a8284aaf6738974a981aa88066954f
d4bb82442ea7b7ab45a237eb7f177a11
28cf980529e03633b332c48e245e7b7d
e12ac1a2ae053e2d37467222b72d12da
d681ea0523cc32724afbaf27c0973e78
1ffb44a0b7667efd5b6066514088dfed
0358095e34e7d8105513f46d99376160
f5ef97aba918ee8db819bd35c3273d24
b63e3d00

-----11 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
69cf5662ba62353f7f20380f6ff06459
82a10c2365eabfe628f4ef3374673e01
cf75e28d3fd4da3c4a76ed45b52c81b5

bcfd8e44279edb2f9bcc2baaddf4517f
7791

CHAVE PÚBLICA:
dffe7001fa2af809dcc6ab9419cce63f
a38168fd43e04335907676cf94b68b94
f2ed6248ea82c443dfa05e2e50401081
f8a1d9531141e6ef0167c2cfd6095c2b
b081

MENSAGEM (tamanho de 11 bytes):
0c3e544074ec63b0265e0c

ASSINATURA:
c4dd236387bb9b35d67f785296b8f6e2
898cc031dd8785e71d9000253263cacf
4bc09ab37b20184d6d5fd1c7e8472551
a0ee45b948eb41ba544a362ae8354cf5
58018988aa06e6ab7d233d4ded0d2eb6
0943fe2d02fe257bb787d4cd86b18378
63f77589249fbf46b71e034bce0564ce
d7149ca46e5c402d62723f6ae9c4b1a4
6abe2300

-----12 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
cf6aafa11216e020117c72fc47391eba
5257e5e024ffa670057197b5742d4303
dbfb1e99ae57efaa3d50a925e60fcec7
32932e7f18e55c7436908a4a26441b30
08ad

CHAVE PÚBLICA:
6a00c488ab20eea14c2e081fbb964eaf
d9ad482f8d0e2f334dc0378c0e6dd704

```
b63e26a1d1c74ef621968694ae3a252a
598b2b3451ef6d08b858b13a4e83c9df
3900

MENSAGEM (tamanho de 12 bytes):
64a65f3cdedcdd66811e2915

ASSINATURA:
2865677a2e6bafb6c5c6abe3804a4c05
d4b0ddfe6d1cd77ea9fe9159703d662f
c633f7f76475f00cb850523e76d178fc
20c0ae714bdc08647388d933e8eef4d6
6b802604cefb93a579342268b3801134
a4303a0d806776e7800ec8e434ad5e81
f236ba27cd85ff021b52467b4b1965ee
d6319f0f10350c0bd971aac4ec82e11c
e3cb3500

-----13 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
db58193646ce4c6519e845bfde9187ee
de698c6e67085a77c008b05d87dbc8f5
70fb63bec88424a49a32fa6705204448
b5b697f48f209857bad9d5ff5d69d872
a94b

CHAVE PÚBLICA:
9e85be73d735e5e81fd7cb016e5401f5
b1c07000841248ac3f78dec321307740
e6e22c10e656efaa7103a1f1873f68ef
5a1ed629741e32531d7769c546b978b8
2380

MENSAGEM (tamanho de 13 bytes):
64a65f3cdedcdd66811e2915e7
```

```
ASSINATURA:
4673b4ff1e31e0505208e0dfd1e64504
5b223f37d3d0a3a941f62d79b72d9cae
145318f96701a827e2bb7e7cb4657685
bbd12f22bf67e29400873dbb4d3ed745
f8802b2994a7609c2d011741d29085fc
c2a45db0e0a9e9a191a0a3eaf7619d02
5fb6e9c30eff9efd81272c93d7940f7e
b0dfd3a96c8f385a6f8429656b063168
47d13d00

-----64 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
22bc5974fc08e7683344e7f4f2f8ae65
f8178c2d71bbc7a7a69a015885add97d
be0c2737ecfd264d7140b8c72167d9dc
648acca1d7975b1b996d46af0d784b76
9c29

CHAVE PÚBLICA:
986f869dd91ef50f7996a2816a611853
1c470b6e8641caf8cbe0235bf94b62e5
32c1b2a419c05aee5b6916dd70401be0
5279086cb0ddc181cdd7f8854c542012
a201

MENSAGEM (tamanho de 64 bytes):
bd0f6a3747cd561bdddf4640a332461a
4a30a12a434cd0bf40d766d9c6d458e5
512204a30c17d1f50b5079631f64eb31
12182da3005835461113718d1a5ef944

ASSINATURA:
78bc83c72f986731f4b9099e84be8847
```

c5346c131caba644fc1d7fd7ecd4b9dc
47b5a15ee288b2bc42a07cb95ec74898
a060b325b2e8c2ca6e4121b390690b64
8c8122db7d333499e1ba1f7a2ad8d4d7
2b7eb3e25e7596d4ae2ef22561d599c2
b3e69b487d569de94abc6e6fc9891d1f
bafd8b15b7207a87e5e5de40fd57fbff
9fb21a00

-----256 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
a68f0c18545b3fdeeba5df3723d2773c
6a70cb366a85a4de5c6db803d2844135
f90019fd06bbce5be764daa17e00239d
8ae27ff506990d88d0b466ce18a8b5b2
f256

CHAVE PÚBLICA:
6d6e8104c8f7dd79180c4293eb723a46
6ea03269e9acfb3cf21d18044a863c7f
7e60ec0342687eeb4780ced5526b24de
7e52324b0fd6139ef9aef3d9825a3736
8a00

MENSAGEM (tamanho de 256 bytes):
15777532b0bdd0d1389f636c5f6b9ba7
34c90af572877e2d272dd078aa1e567c
fa80e12928bb542330e8409f31745041
07ecd5efac61ae7504dabe2a602ede89
e5cca6257a7c77e27a702b3ae39fc769
fc54f2395ae6a1178cab4738e543072f
c1c177fe71e92e25bf03e4ecb72f47b6
4d0465aaea4c7fad372536c8ba516a60
39c3c2a39f0e4d832be432dfa9a706a6
e5c7e19f397964ca4258002f7c0541b5

90316dbc5622b6b2a6fe7a4abffd9610
5eca76ea7b98816af0748c10df048ce0
12d901015a51f189f3888145c03650aa
23ce894c3bd889e030d565071c59f409
a9981b51878fd6fc110624dcbcde0bf7
a69ccce38fabdf86f3bef6044819de11

ASSINATURA:
78a6a4b89819b729c3a018413d7553f9
c89120a36e280ed6afabf26e56328b12
1e11aa42924c0fdddd1c9867bc95ee7f
aabb51bc884c862beef7b74ead700627
5500c1b9ba9c3762c80cc02f71cd8e6b
61b558569b6f1bb37f232a9d0ec94661
ffa046aa9ce0631703dfb59786af68b9
17281337e225f5f6d8c2d97bd7e17f0d
83052d00

-----1023 octetos

ALGORITMO:
Ed521

CHAVE PRIVADA:
440bb8c4a8ea54a51f9312f052e4dc1d
3f2110a2125a2043120c673daaa60191
358a1328f2b37f4d96b5209f0497de8b
60befe93ddd3460fbc2812a38d380bec
8726

CHAVE PÚBLICA:
d718433f5da0173024accaf78ce3518e
b3b5a0bc35616b02438de37378b7429c
1ac07bd844f96d68991ab8384e5e8f69
343c13014f8439a4dc9400e0a00a0a4b
ab81

MENSAGEM (tamanho de 1023 bytes):
6ddf802e1aae4986935f7f981ba3f035

1d6273c0a0c22c9c0e8339168e675412
a3debfaf435ed651558007db4384b650
fcc07e3b586a27a4f7a00ac8a6fec2cd
86ae4bf1570c41e6a40c931db27b2faa
15a8cedd52cff7362c4e6e23daec0fbc
3a79b6806e316efcc7b68119bf46bc76
a26067a53f296dafdbdc11c77f7777e9
72660cf4b6a9b369a6665f02e0cc9b6e
dfad136b4fabe723d2813db3136cfde9
b6d044322fee2947952e031b73ab5c60
3349b307bdc27bc6cb8b8bbd7bd32321
9b8033a581b59eadebb09b3c4f3d2277
d4f0343624acc817804728b25ab79717
2b4c5c21a22f9c7839d64300232eb66e
53f31c723fa37fe387c7d3e50bdf9813
a30e5bb12cf4cd930c40cfb4e1fc6225
92a49588794494d56d24ea4b40c89fc0
596cc9ebb961c8cb10adde976a5d602b
1c3f85b9b9a001ed3c6a4d3b1437f520
96cd1956d042a597d561a596ecd3d173
5a8d570ea0ec27225a2c4aaff26306d1
526c1af3ca6d9cf5a2c98f47e1c46db9
a33234cfd4d81f2c98538a09ebe76998
d0d8fd25997c7d255c6d66ece6fa56f1
1144950f027795e653008f4bd7ca2dee
85d8e90f3dc315130ce2a00375a318c7
c3d97be2c8ce5b6db41a6254ff264fa6
155baee3b0773c0f497c573f19bb4f42
40281f0b1f4f7be857a4e59d416c06b4
c50fa09e1810ddc6b1467baeac5a3668
d11b6ecaa901440016f389f80acc4db9
77025e7f5924388c7e340a732e554440
e76570f8dd71b7d640b3450d1fd5f041
0a18f9a3494f707c717b79b4bf75c984
00b096b21653b5d217cf3565c9597456
f70703497a078763829bc01bb1cbc8fa
04eadc9a6e3f6699587a9e75c94e5bab
0036e0b2e711392cff0047d0d6b05bd2
a588bc109718954259f1d86678a579a3

120f19cfb2963f177aeb70f2d4844826
262e51b80271272068ef5b3856fa8535
aa2a88b2d41f2a0e2fda7624c2850272
ac4a2f561f8f2f7a318bfd5caf969614
9e4ac824ad3460538fdc25421beec2cc
6818162d06bbed0c40a387192349db67
a118bada6cd5ab0140ee273204f628aa
d1c135f770279a651e24d8c14d75a605
9d76b96a6fd857def5e0b354b27ab937
a5815d16b5fae407ff18222c6d1ed263
be68c95f32d908bd895cd76207ae7264
87567f9a67dad79abec316f683b17f2d
02bf07e0ac8b5bc6162cf94697b3c27c
d1fea49b27f23ba2901871962506520c
392da8b6ad0d99f7013fbc06c2c17a56
9500c8a7696481c1cd33e9b14e40b82e
79a5f5db82571ba97bae3ad3e0479515
bb0e2b0f3bfcd1fd33034efc6245eddd
7ee2086ddae2600d8ca73e214e8c2b0b
db2b047c6a464a562ed77b73d2d841c4
b34973551257713b753632efba348169
abc90a68f42611a40126d7cb21b58695
568186f7e569d2ff0f9e745d0487dd2e
b997cafc5abf9dd102e62ff66cba87

ASSINATURA:
fb7891494cb93da8a40a3395469e2f35
3027f8c82c7621f23862962b0579ff4c
124b8b203249ce13cb72ff373394ec8a
92a82cf977a38c7136a9e3e55f4449cd
04017c98bb40b87e82a83c7aa2707487
b2b73448a5ca92e282ecda40cc6236da
0f443a05f09433c1bb52ad6c96b7c0e5
c72a83320553b98377ca97ab567833fa
79326e00