Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Uma implementação do esquema de multi-assinaturas MuSig no cenário m-de-n com árvores de Merkle e suas aplicações ao Bitcoin

Vitor Satoru Machi Matsumine

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Pedro Antônio Dourado de Rezende

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

# Uma implementação do esquema de multi-assinaturas MuSig no cenário m-de-n com árvores de Merkle e suas aplicações ao Bitcoin

Vitor Satoru Machi Matsumine

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Pedro Antônio Dourado de Rezende (Orientador)
CIC/UnB

Prof. Dr. João José Costa Gondim
CIC/UnB

Prof. Dr. Jan Mendonça Correa
CIC/UnB

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 06 de julho de 2019

# Dedicatória

*Dedico este trabalho a minha mãe Méria, meu pai Makoto, meus irmãos Gustavo e Júlia, minhas avós e avôs Maria, Seijun, Jerônimo e Hikaru, por todo o suporte, carinho e esforços ao longo dos anos que me permitiram chegar até aqui. Dedico este trabalho também a minha namorada Amanda, cujo apoio e companhia nos longos dias na biblioteca me ajudaram a não desistir e chegar ao final desta jornada. Agradeço profundamente a todos, muito obrigado.*

# Agradecimentos

*Agradeço ao meu orientador, professor Pedro Rezende, por todo suporte nesta jornada pela criptografia, iniciada há quatro semestres atrás e que irá se perdurar pelos diversos anos que estão por vir.*

# Resumo

*Neste trabalho é proposto uma implementação prova de conceito do esquema de multi-assinatura MuSig no cenário m-de-n com validação de chaves públicas agregadas através de árvores de Merkle, expandindo a descrição original do MuSig neste cenário. Foi utilizado a linguagem de programação Python e curvas elípticas (com destaque para a curva secp256k1) para a construção da implementação. As multi-assinaturas produzidas mantêm o tamanho de uma assinatura individual de Schnorr e podem ser verificadas utilizando uma única chave pública agregada calculada a partir das chaves públicas dos signatarios, tal situação traz expectativas positivas quanto a melhorias no desempenho e privacidade do Bitcoin. Melhorias futuras incluem a implementação de uma prova de conceito que integre o MuSig diretamente ao protocolo Bitcoin e a utilização do MuSig para construção de um esquema de assinatura agregada interativa (IAS).*

**Palavras-chave:** MuSig, multi-assinaturas, assinaturas de Schnorr, Bitcoin, assinatura digital, curvas elípticas, criptografia

# Abstract

*This work presents a proof-of-concept implementation of the multi-signature scheme MuSig in the m-of-n scenario with aggregated key validation using Merkle trees, expanding the original MuSig description in this scenario. Python was the programming language of choice and elliptic curves (mainly secp256k1) were used as the basis of the implementation. The multi-signatures generated by the scheme can keep the same size as a single Schnorr signature and can be verified with a single aggregated public key computed from the individual public keys of the signers, this scenario brings positive expectations for performance and privacy improvements in Bitcoin. Future works includes a proof-of-concept implementation that integrates the MuSig scheme directly into the Bitcoin protocol and the implementation of an interactive signature scheme (IAS) with the MuSig scheme as its basis.*

**Keywords:** MuSig, multi-signatures, Schnorr signatures, Bitcoin, digital signature, elliptic curves, cryptography

# Sumário

# Lista de Figuras

# Capítulo 1

# Introdução

As **criptomoedas** descentralizadas, com destaque ao **Bitcoin**, estabelecem cada vez mais a sua posição na sociedade como uma ferramenta poderosa, com aplicações que vão além das financeiras, como evidenciado pelo **Ethereum** e **ZeroNet**.

O Bitcoin é considerada a primeira criptomoeda descentralizada, com sua implantação realizada em 2009, e atualmente ocupa o posto de criptomoeda mais utilizada e mais valorizada no mercado financeiro. Ele utiliza o esquema **ECDSA** (*Elliptic Curve Digital Signature Algorithm*) com a curva elíptica ***secp256k1*** para realizar assinaturas digitais, sendo estas utilizadas na autentificação das transações de sua rede.

Dentro deste cenário, são encontradas novas possibilidades de melhorias nos protocolos já utilizados por essas criptomoedas, principalmente em termos de escalabilidade e privacidade, a medida que novos esquemas criptográficos são desenvolvidos e suas aplicações testadas.

O esquema **MuSig**, publicado em 2018, é um esquema de multi-assinaturas baseado na **assinatura de Schnorr**, com segurança comprovável no modelo *plain public-key model*. Por meio deste esquema é possível produzir uma assinatura conjunta válida para um grupo de signatários e uma mensagem em comum. Esta multi-assinatura mantém o mesmo tamanho de uma assinatura simples de Schnorr, e sua validade pode ser verificada utilizando-se uma única chave pública agregada construída a partir das chaves públicas dos signatários do grupo.

A substituição do ECDSA pelo esquema de assinatura de Schnorr, traz diversas vantagens ao Bitcoin, sendo destacado o suporte nativo a multi-assinaturas eficientes por meio do MuSig. A implementação de multi-assinaturas no Bitcoin até então, ocorre por meio da requisição de múltiplas assinaturas individuais produzidas a partir de diferentes chaves públicas, sendo estas assinaturas validadas individualmente, uma por vez. Com a utilização do MuSig, seriam produzidas e validadas apenas uma única assinatura conjunta, que por seu caráter linear, mantém o mesmo custo de armazenamento e processamento

na verificação, independente do número de signatários que a produziram, além de prover maior privacidade para estes signatários ao se utilizar somente a chave pública agregada do grupo para a verificação da multi-assinatura.

Este trabalho tem como objetivo principal a implementação do esquema MuSig no cenário $m$-de-$n$ utilizando curvas elípticas.

Serão também apresentadas as descrições e implementações do esquema de assinatura de Schnorr, esquema *Naive Schnorr Multi-signature*, *Rogue-Key Attack*, esquema Bellare-Neven, MuSig no cenário $n$-de-$n$ e Árvores de Merkle. Assim como uma análise das vantagens da utilização do MuSig como esquema de multi-assinatura no Bitcoin.

# Capítulo 2

# Metodologia

## 2.1 Introdução

Nesta seção será discutida a metodologia utilizada no desenvolvimento deste trabalho, sendo esta divida em duas partes principais: o estudo teórico dos temas abordados e o desenvolvimento da implementação baseada nesses estudos.

## 2.2 Estudo Teórico

Primeiramente, foi realizado o estudo de criptografia de curvas elípticas a partir do livro "*Guide to Elliptic Curve Cryptography*"[1] de Alfred Menezes, D.C. Hankerson, e S.A. Vanstone. O capítulo 3 "Criptografia de curvas elípticas"foi desenvolvido tomando esses estudos como base.

Em seguida, iniciou-se o estudo do principal artigo no qual este trabalho se baseia: "*Simple Schnorr Multi-Signatures with Applications to Bitcoin*"de Gregory Maxwell, Andrew Poelstra, Yannick Seurin e Pieter Wuille. Durante a leitura inicial foram identificados as principais bases necessárias para o entendimento do esquema **MuSig** (descritos a seguir).

A análise do esquema de assinatura simples de Schnorr, no qual se baseia o capítulo 4, ocorreu por meio do artigo "*Efficient Signature Generation by Smart Cards*"[4] de C. P. Schnorr e da descrição apresentada em [2]. A partir das descrições iniciais o esquema foi adaptado para curvas elípticas.

O capítulo 5, foi baseado nas descrições do *Naive Schnorr Multi-signature* e *Rogue-Key Attack* em [2] adaptadas para curvas elípticas.

Para o capítulo 6, tomou-se como base o artigo "*Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma*"[5], de Mihir Bellare e Gregory Neven. O

esquema foi adaptado para curvas elípticas e foi dado um enfoque, também, ao estudo do *General Forking Lemma*.

Foi então realizado estudo do esquema **MuSig** no cenário $n$-de-$n$ que resultou na primeira metade do capítulo 7 (descrição do esquema MuSig).

Uma vez solidificado o entendimento do esquema MuSig no cenário $n$-de-$n$, iniciou-se o estudo do **MuSig** no cenário $m$-de-$n$. Para tal, foi utilizada a sugestão dos autores ao final do artigo [2], de utilizar Árvores de Merkle para realizar a verificação da validade de chaves agregadas.

O estudo das árvores de Merkle foi baseado na análise de implementações já existentes utilizadas para garantir a integridade de dados em redes descentralizadas e experimentos durante a implementação. Os conhecimentos adquiridos foram adaptados para o contexto de multi-assinaturas (obtenção e cálculo das possibilidades de chaves agregadas, produção e verificação de provas de pertencimento) e os resultados desses estudos estão presentes no capítulo 8.

Dessa forma, foi possível analisar as aplicações das assinaturas de Schnorr e do esquema MuSig ao Bitcoin, sendo assim desenvolvido o capítulo 9 baseado nas discussões apresentadas [2] e [3].

Foi então realizado o estudo da prova de segurança do esquema $MuSig$ e finalizado capítulo 7.

## 2.3 Implementação

A implementação foi realizada de forma intercambiada com o estudo teórico, isto é, a cada esquema estudado foi realizado a sua implementação, assim como descrito em cada seção do capítulo 10.

O primeiro esquema implementado foi o ECDSA, como forma de adquirir uma maior familiarização com implementações de esquemas criptográficos utilizando curvas elípticas. O ECDSA foi escolhido devido a sua utilização como o esquema de assinatura do Bitcoin e pela vasta documentação disponível a respeito.

Foram então realizadas as implementações do esquema de assinatura de Schnorr, *Naive Schnorr Multi-signature*, uma simulação do *Rogue-key attack* e uma prova de conceito do esquema Bellare-Neven, intercambiadas com os seus respectivos estudos teóricos.

A implementação do esquema $MuSig$ ocorreu em quatro etapas. Na primeira, foi implementado o esquema $MuSig$ no cenário $n$-de-$n$ sem a interação entre signatários (o usuário faz o papel de todos os signatários). Foi então desenvolvido um módulo de rede para realizar a comunicação entre signatários por meio de uma rede *peer-to-peer*. Este módulo foi integrado à primeira implementação do **MuSig**, com as modificações neces-

sárias, e assim foi obtido uma implementação funcional do protocolo de multi-assinatura MuSig no cenário $n$-de-$n$.

Na terceira etapa foi desenvolvida a implementação da Árvore de Merkle aplicada a verificação da validade de chaves agregadas. Este módulo foi integrado a implementação MuSig realizada até então, sendo realizadas as modificações necessárias.

Dessa forma, foi obtida a implementação final do protocolo de multi-assinatura MuSig no cenário $m$-de-$n$. Sua interface gráfica foi subsequentemente desenvolvida.

# Capítulo 3

# Criptografia de curvas elípticas

## 3.1 Introdução

O estudo das curvas elípticas por matemáticos é datado desde o século XIX e desde então vem sendo aprimorado, criando-se uma vasta literatura sobre o tema. É possível dizer que, a descoberta de H. W. Lenstra, Jr em 1984 de um algoritmo para fatorar números inteiros a partir de propriedades de curvas elípticas, promoveu o interesse de pesquisadores na utilização de curvas elípticas em criptografia.[1] A construção inicial do que viria a ser conhecido como criptografia de chaves públicas, foi desenvolvido inicialmente por Ralph Merkle em 1974 com seus Enigmas de Merkle (*Merkle's Puzzle*), e ganhou sua popularidade e reconhecimento através do esquema de distribuição de chaves de Whitfield Diffie e Martin Hellman (Diffie–Hellman key exchange) em 1976. Sucessivamente, a primeira realização prática da criptografia de chaves públicas se deu com o conhecido criptossistema RSA (Rivest–Shamir–Adleman), desenvolvido por Ron Rivest, Adi Shamir e Leonard Adleman em 1978, o qual tem sido extensivamente utilizado até os dias de hoje. A segurança do sistema RSA, quanto a assimetria dos pares de chaves, é baseado na intratabilidade do problema da fatorização (fatorização de um número inteiro em produto de dois números primos relativamente grandes); e a de criptossistemas posteriores derivados do esquema El-Gamal são baseados no problema do logaritmo discreto. Até o momento, não é conhecido um algoritmo não quântico que realize essa fatoração de forma eficiente, ou seja, em tempo polinomial, sendo o estado da arte atual dessa classe de algoritmos o algorítmo GNFS (*General Number Field Sieve*), que opera em tempo sub-exponencial.

A criptografia de curva elíptica (*Elliptic-curve cryptography*, ECC) foi descoberta de forma independente por Neal Koblitz e Victor S. Miller em 1985. Os esquemas criptográficos de curva elíptica proveem a mesma funcionalidade dos esquemas de criptografia assimétrica equivalentes definidos sobre anéis ou corpos de resíduos, porém com diversas vantagens a estes, uma vez que a segurança quanto a assimetria dos esquemas baseados em

curvas elípticas, provém da dureza de um problema diferente: o ECDLP (*Elliptic Curve Discrete Logarithm Problem*; Problema do Logaritmo Discreto em Curvas Elípticas). Os melhores algoritmos não quânticos conhecidos para a resolução do ECDLP operam em tempo inteiramente exponencial, em contraste com o tempo sub-exponencial do problema do logaritmo discreto (em anéis de resíduos). Dessa forma, dentro de sistemas de curva elíptica, é possível utilizar chaves menores do que aquelas utilizadas no RSA e ainda garantir a segurança da assimetria do sistema (é geralmente aceito que uma chave RSA de 1024 bits provê a mesma segurança que uma chave de curva elíptica de 160 bits), o que implica um menor espaço para o armazenamento das chaves, assim como mais velocidade e eficiência no processamento das chaves e da largura de banda da rede.[1]

## 3.2 Teoria de Grupos

Para iniciar o estudo das curvas elípticas, são necessárias algumas noções da teoria de grupos. Pela definição de [6]:
Seja $G$ um conjunto qualquer com uma operação $\circ$ qualquer, $(G, \bullet)$ é um grupo, se e somente se:

- A operação é associativa:

$$a \bullet (b \bullet c) = (a \bullet b) \bullet c, \forall a, b, c \in G$$

- Existe um elemento neutro:

$$\exists e \in G \text{ tal que } e \bullet a = a \bullet e = a, \forall a \in G$$

- Todo elemento possui um elemento inverso:

$$\forall a \in G, \exists b \in G, \text{ tal que } a \bullet b = b \bullet a = e$$

Um grupo $(G, \bullet)$ é um grupo abeliano (ou comutativo), se e somente se:

- A operação é comutativa:

$$\forall a, b \in G, a \bullet b = b \bullet a$$

Se $G$ é um conjunto finito e $(G, \bullet)$ é um grupo, então $(G, \bullet)$ é um grupo finito.

## 3.3 Corpos Finitos

Dado $\mathbb{F}$ um conjunto qualquer com duas operações, uma adição (denotada por $+$) e uma multiplicação (denotada por $\cdot$), temos que $(\mathbb{F}, +, \cdot)$ é um corpo se e somente se:[1]

- $(\mathbb{F}, +)$ é um grupo abeliano com identidade (aditiva) denotada por 0.
- $(\mathbb{F}\backslash\{0\}, \cdot)$ é um grupo abeliano com identidade (multiplicativa) denotada por 1. Onde $\mathbb{F}\backslash\{0\} = \mathbb{F} - \{0\}$.
- Vale a lei distributiva:

$$(a + b) \cdot c = a \cdot c + b \cdot c, \forall a, b, c \in \mathbb{F}.$$

Se $\mathbb{F}$ é um conjunto finito e $(\mathbb{F}, +, \cdot)$ é um corpo, dizemos que $(\mathbb{F}, +, \cdot)$ é um corpo finito.

Quando é implícito que $(\mathbb{F}, +, \cdot)$ é um corpo, as operações serão suprimidas e o corpo será indicado por apenas $\mathbb{F}$.

- A ordem $o(\mathbb{F})$ de um corpo finito é equivalente ao seu número de elementos.
- $\mathbb{F}$ é um corpo finito de ordem $q \iff q$ é potencia de primos, isto é, $q = p^m$, onde $p$ é um número primo e $m$ é um inteiro.
  - $p$ é chamado de característica de $\mathbb{F}$.
  - se $m = 1$, $\mathbb{F}$ é um corpo primo.
  - se $m \geq 2$, $\mathbb{F}$ é um corpo de extensão.
- Qualquer dois corpos de mesma ordem $q$ são **isomórficos** entre si (apenas o rótulo do conteúdo de cada corpo muda) e ambos são denotados $\mathbb{F}_q$.

## 3.4 Operações em corpos finitos

Como observado na subseção anterior, um corpo $(\mathbb{F}, +, \cdot)$ tem as operações de soma e multiplicação. A subtração é definida em termos da adição, ou seja: $\forall a, b \in \mathbb{F},\ a - b = a + (-b)$, tal que $b$ é um elemento único de $\mathbb{F}$ onde $b + (-b) = 0$ e $-b$ é chamado de oposto de $b$. De forma semelhante, a divisão é definida como $\forall a, b \in \mathbb{F},\ a/b = a \cdot (b)^{-1}$ tal que $b$ é um elemento único de $\mathbb{F}$ onde $b \cdot (b)^{-1} = 1$ e $b^{-1}$ é chamado de inverso de $b$.

## 3.5 Corpos Binários

- Corpos finitos de ordem $2^m$ são chamados de corpos binários ou corpos finitos de característica 2

- Representação em bases polinomiais:
  Elementos de $\mathbb{F}_{2^m}$ são representáveis por polinômios binários (polinômios cujos coeficientes estão no corpo $\mathbb{F}_2 = \{0, 1\}$) até um grau máximo $m - 1$:

$$\mathbb{F}_{2^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0, a_i \in \{0, 1\}\}$$

- A soma é feita via soma de polinômios *mod* 2.
- A multiplicação é feita via módulo $f(z)$, onde $f(z)$ é um polinômio redutor.
  - Polinômio binário de redução $f(z)$: irredutibilidade de $f(z)$ significa que $f(z)$ não pode ser fatorado como um produto de polinômios binários de grau menor que $m$. Para todo $m$ podemos encontrar $f(z)$ irredutível, operável como redutor para representação dos elementos do corpo de extensão.
  - Redução módulo $f(z)$: para qualquer polinômio binário $a(z)$, $a(z)$ *mod* $f(z)$ resulta em um resto único polinomial $r(z)$ com grau menor que $m$.

Exemplo: Sendo $\mathbb{F}_{2^4} = \{0, 1, z, z+1, z^2, z^2+1, z^2+z, \cdots, z^3+z^2+z+1\}$ um corpo com seu polinômio redutor $f(z) = z^4 + z + 1$

1. Adição: $(z^3 + z^2 + 1) + (z^2 + z + 1) = z^3 + z$
2. Subtração: $(z^3 + z^2 + 1) - (z^2 + z + 1) = z^3 + z$; note que, em $\mathbb{F}_2$, temos $-1 = 1$, logo, em $\mathbb{F}_{2^m}$, temos $-a = a, \forall a \in \mathbb{F}_{2^m}$
3. Multiplicação: $(z^3+z^2+1)\cdot(z^2+z+1) = z^5+z^4+z^3+z^4+z^2+z^2+z+1 = z^5+z+1 = z^2 + 1$ uma vez que, $z^5 + z + 1$ *mod* $f(z) = z^5 + z + 1$ *mod* $z^4 + z + 1 = z^2 + 1$
4. Inverso: $(z^3 + z^2 + 1)^{-1} = z^2$ uma vez que $(z^3 + z^2 + 1) \cdot z^2$ *mod* $z^4 + z + 1 = 1$

## 3.6 Corpos de extensão

As propriedades descritas acima para corpos binários podem ser generalizados para todos os corpos de extensão:

- Para um $p$ primo, onde $p \geq 2$.
- $\mathbb{F}_p[z]$ denota o conjunto de todos os polinômios de variável $z$ com coeficientes de $\mathbb{F}_p$.
- $f(z)$ é polinômio redutor de grau $m$ em $\mathbb{F}_p[z]$, $f(z)$ existe para todo p e m.
- Irredutível significa que $f(z)$ não pode ser fatorado como um produto de polinômios em $\mathbb{F}_p[z]$, cada um com grau menor que m.

Os elementos de $\mathbb{F}_{p^m}$ são polinômios em $\mathbb{F}_p[z]$ de grau até $m-1$:

$$\mathbb{F}_{p^m} = \{a_{m-1}z^{m-1} + a_{m-2}z^{m-2} + \cdots + a_1 z + a_0, a_i \in \mathbb{F}_{p^m}\}$$

A aritmética é feita de forma semelhante a em $\mathbb{F}_{2^m}$:
Exemplo: $p - 251, m = 5; \mathbb{F}_{p^m} = \mathbb{F}_{251^5}; a(z) = 123z^4 + 76z^2 + 7z + 4$.

## 3.7 Subcorpo de um corpo finito

Se $F$ é um corpo, $K$ é um subconjunto de $F$ e $K$ é um corpo, então $K$ é dito subcorpo de $F$ e $F$ é dito corpo de extensão de $K$.

Se $\mathbb{F}_{p^m}$ é um corpo e $p$ é primo, então $\mathbb{F}_{p^m}$ tem um subcorpo de ordem $p^l$ para cada $l$ onde $l$ é um divisor positivo de $m$.

## 3.8 Curvas Elípticas

Uma curva elíptica $E$ sobre um corpo $K$ é definida pela seguinte equação:

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \text{ (Equação de Weierstrass)}$$

Com $a_1, a_2, a_3, a_4, a_6 \in K$ e $\Delta \neq 0$, onde $\Delta$ é o discriminante de $E$, o qual é definido como:

$$\begin{cases} \Delta = -d_2^2 d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6 \\ d_2 = a_1^2 + 4a_2 \\ d_4 = 2a_4 + a_1a_3 \\ d_6 = a_3^2 + 4a_6 \\ d_8 = a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2 \end{cases}$$

Uma curva é dita "suave"quando $\Delta \neq 0$, isto é, não existem pontos nos quais a curva tem duas ou mais linhas tangentes distintas.
"$E$ definida sobre $K$"significa que os coeficientes $a_i$ da curva $E$ pertencem a $K$. Também é possível escrever $E/K$ para enfatizar que $E$ está definida sobre $K$ e $K$ é chamado de corpo subjacente (*underlying field*). Note também que se $E$ está definida sobre $K$, então $E$ também está definida sobre qualquer corpo de extensão de $K$.
Se $L$ é um corpo de extensão (qualquer) de $K$, então o conjunto de pontos $L$-racionais($L$-

*rational points*) em $E$ são:

$$E(L) = \{(x, y) \in L \times L : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\infty\}$$

Onde o ponto $\infty$ é chamado de ponto no infinito e corresponde ao elemento neutro da soma de pontos na curva elíptica (tal operação "soma de pontos"será definida adiante). Os pontos $L$-racionais ($L$-*rational points*) em $E$ são os pontos $(x, y)$ que satisfazem a equação da curva e cujas coordenadas $x$ e $y$ pertencem a L. O ponto no infinito é considerado um ponto $L$-racional para todos os corpos de extensão $L$ de $K$.

## 3.9 Equações simplificadas de Weierstrass

Considere duas curvas elípticas:

$$E1 : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

$$E2 : y^2 + a'_1xy + a'_3y = x^3 + a'_2x^2 + a'_4x + a'_6$$

$E1$ e $E2$ são isomórficos sobre o corpo $K$ se existe $u, r, s, t \in K$, $u \neq 0$, tal que a troca de variáveis:

$$(x, y) \to (u^2x + r, u^3y + u^2sx + t)$$

transforma a equação $E1$ na equação $E2$. Essa transformação é chamada de troca admissível de variáveis (*admissible change of variables*).

- Para curvas com característica diferente de 2 ou 3:

  $$(x, y) \to (\frac{x - 3a_1^2 - 12a_2}{36}, \frac{y - 3a_1x}{216} - \frac{a_1^3 + 4a_1a_2 - 12a_3}{24})$$

  transforma a curva $E$ (equação de Weiestrass) na curva:

  $$y^2 = x^3 + ax + b \text{ com } a, b \in K$$

  com discriminante: $\Delta = -16(4a^3 + 27b^2)$

- Para curvas com característica de $K$ igual a 2, temos dois casos:

  - Caso não super-singular $\implies$ $a_1 \neq 0$:

    $$(x, y) \to (a_1^2x + \frac{a_3}{a_1}, a_1^3y + \frac{a_1^2a_4 + a_3^2}{a_1^3})$$

Transforma a curva $E$ em:

$$y^2 + xy = x^3 + ax^2 + b \ , a, b \in K$$

E com discriminante $\Delta = b$

- Caso super-singular $\implies a_1 = 0$:

$$(x, y) \to (x + a_2, y)$$

Transforma a curva $E$ em:

$$y^2 + cy = x^3 + ax + b \ , a, b, c \in K$$

E com discriminante $\Delta = c^4$

- Para curvas com característica de K igual a 3:
  - Caso não super-singular $\implies a_1^2 \neq -a_2$:

$$(x, y) \to (x + \frac{d_4}{d_2}, y + a_1 x + a_1 \frac{d_4}{d_2} + a_3)$$

Transforma a curva $E$ em:

$$y^2 = x^3 + ax^2 + b \ , a, b \in K$$

E com discriminante $\Delta = -a^3 b$

  - Caso super-singular $\implies a_1^2 \neq -a_2$:

$$(x, y) \to (x, y + a_1 x + a_3)$$

Transforma a curva $E$ em:

$$y^2 = x^3 + ax + b \ , a, b \in K$$

E com discriminante $\Delta = -a^3$

## 3.10 Lei de grupos para curvas elípticas

O conjunto de pontos L-racionais numa curva elíptica forma um grupo quando munido de uma operação algébrica denominada "soma"(de pontos). Nesse grupo, onde a operação

soma tem "inspiração"geométrica, a iteração da soma de um ponto com ele mesmo é denominada "multiplicação por escalar".

A notação $E(\mathbf{F}_n)$ passa então a denotar tanto o conjunto de pontos $\mathbf{F}_n$-racionais sobre $E$, para a equação da curva elíptica dada ou indicada pelo contexto, quanto o correspondente grupo algébrico, formado por tais pontos sob tal operação.

Lembrando aqui das definições em 3.8, onde $E/K$ enfatiza que os coeficientes da implícita curva elíptica estão no subcorpo $K$ de $\mathbf{F}_n$.

A representação geométrica/gráfica da soma (Figura 3.1) e duplicação de pontos (multiplicação por escalar) (Figura 3.2) fornece uma visão mais intuitiva das operações:

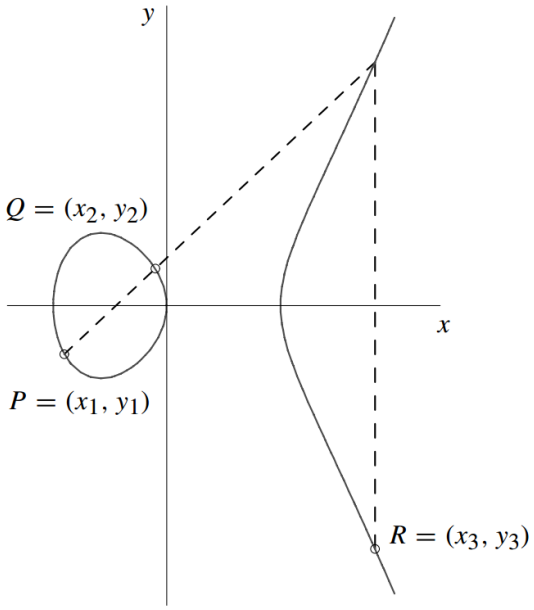


Figura 3.1: Soma: P+Q=R, retirado de [1]

- Lei de grupos para $E/K$: $y^2 = x^3 + ax + b$ com característica de K diferente de 2 e 3:

  1. Identidade: $P + \infty = \infty + P = P,\ \forall P \in E(K)$
  2. Negativos: $\forall P \in E(K), P = (x, y), (x, y) + (x, -y) = \infty \implies -P = (x, -y)$
  3. Adição de Pontos: $\forall P, Q \in E(K)$, com $P \neq \pm Q, P = (x_1, y_1), Q = (x_2, y_2)$ e $P + Q = (x_3, y_3)$ onde:

$$x_3 = (\frac{y_2 - y_1}{x_2 - x_1})^2 - x_1 - x_2 \text{ e } y_3 = (\frac{y_2 - y_1}{x_2 - x_1})(x_1 - x_3) - y_1$$

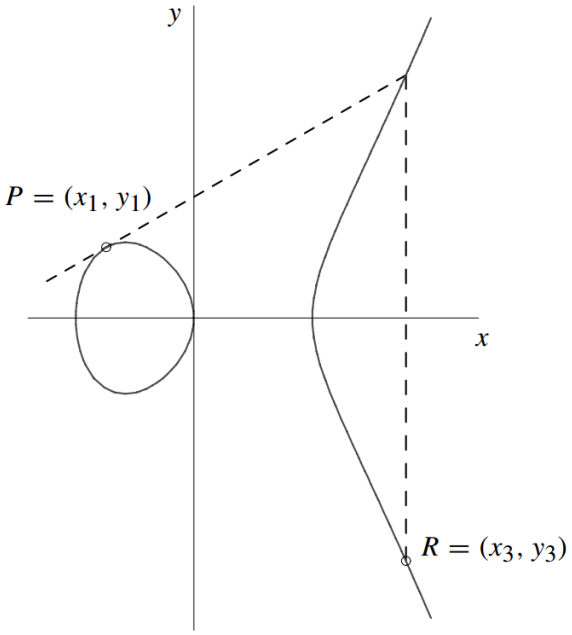


Figura 3.2: Duplicação: P+P=R, retirado de [1]

4. Duplicação de Pontos: $\forall P \in E(K)$, com $P = (x_1, y_1)$ e $P \neq -P$, então $2P = (x_3, y_3)$ onde:

$$x_3 = (\frac{3x_1^2 + a}{2y_1})^2 - 2x_1 \text{ e } y_3 = (\frac{3x_1^2 + a}{2y_1})(x_1 - x_3) - y_1$$

**Observação**: x,y e a são elementos de $K$, logo as suas operações são feitas com base nas operações do corpo $K$

- Lei de grupos para curvas com característica 2 não super-singular $E/\mathbb{F}_{2^m} : y^2 + xy = x^3 + ax^2 + b$:

1. Identidade: $P + \infty = \infty + P = P, \ \forall P \in E(\mathbb{F}_{2^m})$
2. Negativos: $\forall P \in E(\mathbb{F}_{2^m}), \exists Q \in E(\mathbb{F}_{2^m})$ tal que $P + Q = \infty$, onde, para $P = (x, y), Q = -P = (x, x + y)$
3. Adição de Pontos: $\forall P, Q \in E(\mathbb{F}_{2^m}), comP \neq \pm Q, P = (x_1, y_1), Q = (x_2, y_2) eP +$ $Q = (x_3, y_3)$ onde:

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \text{ e } y_3 = \lambda(x_1 + x_3) + x_3 + y_1$$

com $\lambda = \frac{y_1 + y_2}{x_1 + x_2}$

4. Duplicação de Pontos: $\forall P \in E(\mathbb{F}_{2^m})$, com $P = (x_1, y_1)$ e $P \neq -P$, então $2P = (x_3, y_3)$ onde:

$$x_3 = \lambda^2 + \lambda + a = x_1^2 + \frac{b}{x_1^2} \text{ e } y_3 = x_1^2 + \lambda x_3 + x_3$$

com $\lambda = \frac{x_1 + y_1}{x_1}$

**Observação**: as operações envolvendo elementos de $\mathbb{F}_{2^m}$ (ou seja, $x_i, y_i, a, b$) devem ser feitos com base nas operações de $\mathbb{F}_{2^m}$, logo, é necessário também, definir um polinômio redutor para as operações.

- Lei de grupos para curvas com característica 2 super-singular $E/\mathbb{F}_{2^m} : y^2 + cy = x^3 + ax + b$:

1. Identidade: $P + \infty = \infty + P = P, \ \forall P \in E(\mathbb{F}_{2^m})$
2. Negativos: $\forall P \in E(\mathbb{F}_{2^m}), \exists Q \in E(\mathbb{F}_{2^m})$ tal que $P + Q = \infty$, onde, para $P = (x, y), Q = -P = (x, y + c)$
3. Adição de Pontos: $\forall P, Q \in E(\mathbb{F}_{2^m}), com P \neq \pm Q, P = (x_1, y_1), Q = (x_2, y_2) e P + Q = (x_3, y_3)$ onde:

$$x_3 = (\frac{y_1 + y_2}{x_1 + x_2})^2 + x_1 + x_2 \text{ e } y_3 = (\frac{y_1 + y_2}{x_1 + x_2})(x_1 + x_3) + y_1 + c$$

4. Duplicação de Pontos: $\forall P \in E(\mathbb{F}_{2^m})$, com $P = (x_1, y_1)$ e $P \neq -P$, então $2P = (x_3, y_3)$ onde:

$$x_3 = (\frac{x_1^2 + a}{c})^2 \text{ e } y_3 = (\frac{x_1^2 + a}{c})(x_1 + x_3) + y_1 + c$$

## 3.11 Ordem do grupo

Considerando uma curva elíptica $E$, definida sobre um corpo $\mathbb{F}_q$ o número de pontos em $E(\mathbb{F}_q)$, denotada por $\#E(\mathbb{F}_q)$, é chamada de ordem de $E$ sobre $\mathbb{F}_q$

- **Teorema de Hasse**: $q + 1 - 2\sqrt{q} \leq \#E(\mathbb{F}_q) \leq q + 1 + 2\sqrt{q}$

  $[q + 1 - 2\sqrt{q}; q + 1 + 2\sqrt{q}]$ é chamado de **Intervalo de Hasse**

  **Expressão alternativa**: $\#E(\mathbb{F}_q) = q + 1 - t$, onde $|t| \leq 2\sqrt{q}$, onde $t$ é chamado de traço (trace) de $E$ sobre $\mathbb{F}_q$.

- Ordens admissíveis de curvas elípticas:

  Sendo $|t| \leq 2\sqrt{q}$ e $q = p^m$ para $p$ primo e $m$ inteiro

Existe uma curva elíptica $E$ definida sobre $\mathbb{F}_q$ com $\#E(\mathbb{F}_q) = q+1-t$, se e somente se, uma das condições a seguir é satisfeita:

1. $t \not\equiv 0(modp)$ e $t^2 \leq 4q$
2. $m$ é impar e:
   (a) $t = 0$, ou
   (b) $t^2 = 2q$ e $p = 2$, ou
   (c) $t^2 = 3q$ e $p = 3$
3. $m$ é par e:
   (a) $t^2 = 4q$, ou
   (b) $t^2 = q$ e $p \not\equiv 1(mod3)$, ou
   (c) $t = 0$ e $p \not\equiv 1(mod4)$

   Uma consequência do teorema acima é: para qualquer primo $p$ e inteiro $t$ que satisfaz $|t| \leq 2\sqrt{q}$, existe uma curva elíptica $E$ sobre $\mathbb{F}_q$ com $\#E(\mathbb{F}_q) = q+1-t$.

- **Super-singularidade**: seja $E$ uma curva sobre $\mathbb{F}_q$, com $q = p^m$ ( $p$ a característica de $\mathbb{F}_q$), e seja $t$ o traço de $E$. Se $p$ divide $t$, então $E$ é super-singular. Caso contrário, $E$ é não super-singular.[1]

- Seja $E$ uma cura elíptica definida sobre $\mathbb{F}_q$ e $\#E(\mathbb{F}_q) = q+1-t$. Então, $\#E(\mathbb{F}_{q^n}) = q^n + 1 - V_n$, para todo $n \geq 2$, onde $\{V_n\}$ é uma sequência definida por $V_0 = 2, V_1 = t, V_n = V_1V_{n-1} - qV_{n-2}$.

## 3.12 Estrutura de grupo de $E(\mathbf{F}_n)$

Sendo $\mathbb{Z}_n$ um grupo cíclico de ordem $n$, a estrutura de grupo de uma curva elíptica pode ser definida pelo seguinte teorema explicitado em [1]: seja $E$ uma curva elíptica definida sobre $\mathbb{F}_q$, então $E(\mathbb{F}_{||})$ é isomorfo a $\mathbb{Z}_{n_1} \oplus \mathbb{Z}_{n_2}$ , onde $n_1$ e $n_2$ são inteiros positivos unicamente determinados tal que $n_2|n_1$ e $n_2|(q-1)$ ($n_2$ divide $n_1$ e $q-1$).

## 3.13 ECDLP

- PLDCE/ECDLP - Problema do logaritmo discreto de curva elíptica (*Elliptic Curve Discrete Logarithm Problem*):

  Dado uma curva elíptica $E$ sobre um corpo finito $\mathbb{F}_q$, um ponto $P \in E(\mathbb{F}_q)$ de ordem $n$ e um ponto $Q \in \langle P \rangle$ (grupo cíclico gerado por $P$), ache um inteiro $l \in [0, n-1]$

tal que $Q = lP$. Tal inteiro l é chamado de logaritmo discreto de $Q$ na base $P$, denotado por $l = \log_P Q$

- Ataques ao ECDLP:

  Os parâmetros da curva elíptica utilizada em esquemas criptográficos devem ser cuidadosamente escolhidos de forma a resistir a todos os ataques conhecidos via solução eficiente do ECDLP.

  O método mais simples para se resolver o ECDLP é por meio da busca exaustiva, realizando o cálculo de $P, 2P, 3P, 4P, ...$ até que seja encontrado o ponto $Q$ desejado, essa busca opera em $O(n)$, onde $n$ é a ordem da curva. Logo deve ser escolhido uma curva com um $n$ suficientemente grande para resistir a esse ataque (por exemplo: $n \geq 2^{80}$).

  O melhor ataque de propósito geral conhecido ao ECDLP é uma combinação do algoritmo de Pohlig-Hellman e o algoritmo *rho* de Pollard, o qual roda em tempo inteiramente exponencial $O(\sqrt{p})$, onde $p$ é o maior divisor primo de $n$. Logo, para resistir a esse ataque, deve ser escolhida uma curva elíptica cujo $n$ é divisível por um primo $p$ suficientemente grande tal que $\sqrt{p}$ passos seja uma quantidade computacionalmente inviável (por exemplo: $p > 2^{160}$).[1]

  Ainda existem ataques direcionados a curvas elípticas com parâmetros específicos que devem ser verificados durante a escolha das curvas para esquemas criptográficos, entre eles: *index-calculus attacks* e ataques de isomorfismos (curvas anômalas de corpos primos, pareamento de Weil e Tate, Weil descent).

A dureza do ECLDP é uma condição necessária mas geralmente não suficiente para a segurança quanto a assimetria de um esquema criptográfico de curva elíptica. O ECDHP (enunciado a seguir) não é mais duro do que o ECDLP, no entanto não se sabe se o ECDHP é tão duro quanto o ECDLP, ou seja, não se sabe uma maneira de se resolver o ECDLP de forma eficiente dado um oráculo hipotético que resolve de forma eficiente o ECDHP.[1]

- (*Computational*) *Elliptic Curve Diffie-Hellman problem* (ECDHP):

  Dada uma curva elíptica $E$ definida sobre um corpo $\mathbb{F}_q$, um ponto $P \in E(\mathbb{F}_q)$ de ordem n, os pontos $A = aP, B = bP \in \langle P \rangle$, encontre o ponto $C = abP$

O ECDDHP anunciado a seguir é um problema utilizado para modelar a segurança quanto a assimetria de esquemas criptográficos onde, dados os pontos $aP$ e $bP$, o ponto $abP$ deve ser indistinguível de um ponto qualquer de $\langle P \rangle$, ou seja, nenhuma informação sobre $abP$ é disponibilizada a um adversário por meio do conhecimento de $aP$ e $bP$.

Logo, se o ECDHP em $\langle P \rangle$ pode ser resolvido de forma eficiente, então o ECDDHP em $\langle P \rangle$ também pode ser resolvido de forma eficiente, logo o ECDDHP não é mais duro do que o ECDHP e consequentemente não é mais duro do que o ECDLP.

- *Elliptic Curve Decision Diffie-Hellman Problem* (ECDDHP):

  Dada uma curva elíptica $E$ definida sobre um corpo $\mathbb{F}_q$, um ponto $P \in E(\mathbb{F}_q)$ de ordem $n$ e pontos $A = aP$, $B = bP$, e $C = cP \in \langle P \rangle$, determine se $C = abP$ ou de forma equivalente, se $c \equiv ab(modn)$. Isto é, determinar se $abP$ é distinguível de um ponto qualquer de $\langle P \rangle$. Alguns protocolos exigem que $abP$ seja indistinguível de um ponto qualquer de $\langle P \rangle$.

## 3.14 Parâmetros do domínio de uma curva $E(\mathbb{F}_q)$

Os parâmetros do domínio de uma curva $E(\mathbb{F}_q)$ são representados por $D = (q, FR, S, a, b, P, n, h)$, onde os parâmetros listados denotam, respectivamente:

- $q$: ordem do corpo $\mathbb{F}_q$
- $FR$ - *Field Representation*: indicação da representação utilizada para elementos de $\mathbb{F}_q$
- $S$ - *seed*: utilizado caso a curva elíptica for gerada aleatoriamente
- $a, b$: $a, b \in \mathbb{F}_q$ elementos que definem a equação da curva elíptica
- $P$: ponto $P = (x_p, y_p) \in E(\mathbb{F}_q)$, $x_p, y_p \in \mathbb{F}_q$. O ponto P é chamado de ponto base e tem ordem prima
- $n$: ordem de P
- $h$: cofator $h = \#E(\mathbb{F}_q)/n$

## 3.15 A curva secp256k1 - Curva do Bitcoin

A curva elíptica ***secp256k1*** é definida em Standards for Efficient Cryptography (SEC)[7] como $D = (q, a, b, P, n, h)$ com os seguintes parâmetros:

- $q$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F $= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
- A curva E: $y^2 = x^3 + ax + b$ sobre $\mathbb{F}_q$ é definida por:

  $a$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

  $b$ = 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000007

- O ponto base P em sua forma comprimida é dado por:

  $P =$ 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798

  E em sua forma não comprimida:

  $P =$ 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D959F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8

- A ordem n de P é dada por:

  $n$ = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141

- O cofator h é dado por:

  $h = 01$

### 3.15.1 Codificação de pontos de curvas elípticas

Um ponto $Q = (x, y)$ de uma curva elíptica pode ser representado das seguintes formas $B_0||x||y$ (forma completa) ou $B_0||X$ (forma compacta) onde:[8]

- $B_0 \in \{02, 03, 04\}$ tal que:
  - 02 e 03 são utilizados na representação compacta de $Q$ para identificar a paridade da coordenada $y$, tal que $B_0 = (y \ mod \ 2) + 2$ e convertendo esse valor para bytes tal que $B_0 \in \{02, 03\}$[9]
  - 04 indica que o ponto está sendo expresso na sua forma completa (incluindo a coordenada $y$).
- $x$: coordenada $x$ do ponto $Q$
- $y$: coordenada $y$ do ponto $Q$

A ideia geral para a compactação da representação do ponto de uma curva elíptica ocorre da seguinte forma:

- para dado ponto $Q = (x, y)$ a coordenada y pode ser derivada a partir de x resolvendo a equação da curva elíptica correspondente ($y^2 = C(x)$, o qual é a equação de Weierstrass apropriada que conecta $x$ a $y$)
- existem duas possibilidades de $y$ para um dado $x$
- uma das duas possibilidades está codificada de alguma forma na compressão

**Codificação**:

- dada a representação canônica de $Q = (x, y)$, retorne $B_0||x$ como sua representação compacta, tal que: $B_0 = (y \ mod \ 2) + 2$

**Decodificação**:

- dada uma representação compacta de $Q$, retorne $Q = (x, y)$ em sua forma canônica:
  - $y' = \sqrt{}(C(x))$, onde $y' > 0$
  - $y = min(y', p - y')$, onde $p$ é a ordem do corpo sobre o qual a curva elíptica está definida
  - $Q = (x, y)$ é a representação canônica do ponto

Tomando o ponto base $P$ da curva secp256k1 apresentado anteriormente como exemplo:

- $P =$ 02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798 $\implies$ $B_0 =$ 02, $x =$ 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
- $P =$ 04 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D959F2815B 16F81798 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8 $\implies$ $B_0 =$ 04, $x =$ 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D959F2815B 16F81798, $y =$ 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8.

# Capítulo 4

# Assinatura de Schnorr

## 4.1 Introdução

A assinatura de Schnorr é um esquema de assinatura digital de chave pública descrito por Claus-Peter Schnorr no artigo "Efficient Signature Generation by Smart Cards"[4] publicado em 1991. Simples e eficiente, o esquema é capaz de gerar assinaturas relativamente curtas e com uma verificação mais rápida quando comparado, por exemplo, ao ElGamal (comparação realizada no artigo), e sua segurança quanto à assimetria tem como base a dureza do problema do logaritmo discreto. A assinatura de Schnorr teve sua patente expirada em 2008 e é utilizada como a base do esquema de assinatura múltipla MuSig.

## 4.2 Descrição do esquema

O esquema de assinatura de Schnorr utiliza um grupo cíclico $\mathbb{G}$ de ordem prima $p$, um gerador $g$ de $\mathbb{G}$ e uma função de hash $H$.

Uma chave privada é representada por $x \in [0, \cdots, p-1]$ e a chave pública é representada por $X \in \mathbb{G}$, logo, o par de chaves privada/pública é representado pelo par $(x, X) \in [0, \cdots, p-1] \times \mathbb{G}$ onde $X = g^x$.

Para assinar a mensagem $m$, o signatário toma um número inteiro randômico $r \in \mathbb{Z}_p$, calcula $R = g^r$, $c = H(X, R, m)$ e $s = r + cx$.

A assinatura é o par $(R, s)$, e sua validade pode ser checada verificando se $g^s \stackrel{?}{=} RX^c$.[2]

## 4.3 Descrição do esquema para grupos de curva elíptica

O esquema de assinatura de Schnorr quando aplicado a grupos de curva elíptica tem as seguintes modificações:

- O grupo utilizado é o $E(\mathbb{F}_q)$, de ordem $\#E(\mathbb{F}_q)$.
- Ponto base $P \in E(\mathbb{F}_q)$ tal que $P$ é gerador de $\langle P \rangle$.
- O par de chaves privada/pública $(x, X) \in 1, \cdots, n-1 \times E(\mathbb{F}_q)$, onde n é a ordem de $\langle P \rangle$.

## 4.4 Descrição do processo de assinatura e verificação

**Notação**: sendo $S$ um conjunto não vazio, $s \xleftarrow{\$} S$ denota a operação de tomar um elemento de $S$ de maneira uniformemente aleatória e atribuí-lo a $s$; $n$ é a ordem de $\langle P \rangle$; $int(x)$ corresponde a conversão de $x$ a um inteiro correspondente.
**Assinatura**:

- **Entrada**: parâmetros da curva $(E(\mathbb{F}_q), n, P)$, função de hash $H$, mensagem $m$, par de chaves pública/privada $(X, x)$
- **Saída**: assinatura $(R, s)$
- $r \xleftarrow{\$} \{1, \cdots, n-1\}$
- $R \leftarrow rP$
- $c \leftarrow H(X, R, m)$
- $\bar{c} \leftarrow int(c)$
- Assinatura parcial: $s \leftarrow r + \bar{c}x \ mod \ n$
- A assinatura resultante é o par: $(R, s)$

**Verificação**:
A verificação da assinatura é realizada da seguinte forma:

$$sP \stackrel{?}{=} R + cX$$

Sendo $P$ o ponto base, obtido nos parâmetros da curva elíptica, $(R, s)$ a assinatura a ser verificada, $m$ a mensagem, e $c = H(X, R, m)$ calculado durante a verificação.

Analisando a corretude da verificação, é possível observar que, de fato:

$$sP = (r + cx)P = rP + c(xP) = R + cX$$

# Capítulo 5

# Naive Schnorr Multi-signature

## 5.1 Introdução

Com base na assinatura de Schnorr, é possível criar um esquema ingênuo de multi-assinatura que utiliza chaves e assinaturas agregadas da seguinte forma:

$$\tilde{s}P = (\sum_{i=1}^{n} s_i)P = (\sum_{i=1}^{n} r_i + cx_i)P = \sum_{i=1}^{n} r_iP + cx_iP = \sum_{i=1}^{n} R_i + cX_i = \sum_{i=1}^{n} R_i + c\sum_{i=1}^{n} X_i = \tilde{R} + c\tilde{X}$$

Uma vez que: $\tilde{s} = \sum_{i=1}^{n} s_i \ mod \ n = \sum_{i=1}^{n} r_i + cx_i \ mod \ n \in \{0, \cdots, n-1\}$, $\sum_{i=1}^{n} R_i \in \langle P \rangle$ e $\sum_{i=1}^{n} X_i \in \langle P \rangle$
Com base nessa observação, será descrito um esquema de multi-assinatura conhecido como *Naive Schnorr Multi-signature*, bem como as vulnerabilidades que este esquema apresenta.

## 5.2 Descrição do esquema

**Assinatura**:

- $n$ signatários desejam co-assinar a mensagem $m$, todos os signatários compartilham os mesmos parâmetros do domínio de uma curva elíptica $E$, de ordem $k$, ponto base $P$.
- cada signatário tem um par de chaves público/privada respectivo $(X_i, x_i)$, com $i \in 1, \cdots, n$ e $X_i = x_iP$
- $L = \{X_1, \cdots, X_n\}$ é o multiset[1] de todas as chaves públicas que participam do processo de assinatura

[1]Conjunto no qual são permitidos múltiplas instâncias de um mesmo elemento.

- cada signatário gera e compartilha seu respectivo $R_i \leftarrow r_i P$, sendo $r_i \xleftarrow{\$} \{1, \cdots, k-1\}$
- cada signatário calcula $R = \sum_{i=1}^{n} R_i = R_1 + R_2 + \cdots + R_n$ após o recebimentos dos demais $R_i$'s
- cada signatário calcula $\tilde{X} = \sum_{i=1}^{n} X_i = X_1 + X_2 + \cdots + X_n$ com base em $L$
- cada signatário calcula $c = H(\tilde{X}, R, m)$
- cada um calcula e compartilha a sua respectiva assinatura parcial $s_i = r_i + cx_i$
- as assinaturas parciais são combinadas em uma única assinatura $s = \sum_{i=1}^{n} s_i \ (mod\ n)$
- a assinatura resultante é $(R, s)$

**Verificação**: A verificação do *Naive Schnorr Multi-signature* é realizado da seguinte forma:

$$sP \stackrel{?}{=} \sum_{i=1}^{n} R_i + c \sum_{i=1}^{n} X_i = R + c\tilde{X}$$

É possível observar que, sendo transmitida junto à assinatura $(R, s)$ a chave agregada $\tilde{X}$ utilizada, a verificação é a mesma de uma assinatura simples de Schnorr.

## 5.3 *Rogue-Key Attack*

O *Naive Schnorr Multi-signature* está sujeito a um ataque conhecido como *Rogue-Key Attack*. Neste ataque a chave pública de um signatário corrompido é construída em função das chaves públicas dos seus co-signatários, sendo manipulado o resultado da chave agregada do grupo, impedindo que o verificador infira corretamente a identidade dos signatários e assim inviabilizando a utilização do Naive Schnorr como esquema criptográfico de assinatura.

Logo, por meio deste ataque, é possível que a chave agregada do grupo de signatários dependa apenas da chave privada do signatário, permitindo que o adversário (aquele que executa o ataque) produza assinaturas pelo grupo sem a necessidade da participação dos co-signatários.

O *Rogue-Key Attack* ocorre da seguinte forma (em notação multiplicativa, para grupos abstratos): considerando $X_1$ como a chave pública do signatário corrompido, este estabelece a sua chave pública forjada como

$$X_1 = g^{x_1} (\prod_{i=2}^{n} X_i)^{-1}$$

Seguindo com o cálculo da chave pública agregada temos:

$$\tilde{X} = \prod_{i=1}^{n} X_i = X_1 \prod_{i=2}^{n} X_i = g^{x_1}(\prod_{i=2}^{n} X_i)^{-1}(\prod_{i=2}^{n} X_i) = g^{x_1}$$

$$\implies \tilde{X} = g^{x_1}$$

Assim, é possível observar que, de fato, a chave pública agregada depende, na etapa de validação, apenas da chave privada do signatário corrompido. Além disso, como uma chave pública agregada é indistinguível das outras chaves públicas, não é possível apenas por meio de uma comparação direta entre as chaves, identificar se dentre as chaves públicas do grupo existe uma chave pública manipulada para a realização deste ataque, assim como não é possível verificar se a chave pública agregada resultante foi manipulada por algum dos signatários.

No âmbito das curvas elípticas, onde se usa a notação aditiva para as operações de grupo, o ataque se dá por:

$$X_1 = (x_1 P) - (\sum_{i=2}^{n} X_i),$$

onde $X_i = x_i P$ e $P$ é o ponto base da curva elíptica. Assim:

$$\tilde{X} = \sum_{i=1}^{n} X_i = X_1 + \sum_{i=2}^{n} X_i = x_1 P - (\sum_{i=2}^{n} X_i) + \sum_{i=2}^{n} X_i = x_1 P$$

$$\implies \tilde{X} = x_1 P$$

Como descrito anteriormente, o cálculo do negativo de um ponto de uma curva elíptica depende da característica e singularidade da curva:

- $E/K : y^2 = x^3 + ax + b$, $char(K) \neq 2, 3 \implies \forall P = (x, y) \in E/K, -P = (x, -y)$
- $E/\mathbb{F}_{2^m}$ não super-singular: $y^2 + xy = x^3 + ax^2 + b \implies \forall P = (x, y) \in E/\mathbb{F}_{2^m}, -P = (x, x + y)$
- $E/\mathbb{F}_{2^m}$ super-singular: $y^2 + cy = x^3 + ax^2 + b \implies \forall P = (x, y) \in E/\mathbb{F}_{2^m}, -P = (x, y + c)$

Assim, para a curva secp256k1, sendo ela uma curva elíptica de característica $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$ da forma $y^2 = x^3 + ax + b$ sobre $\mathbf{F}_p$ com $a = 0$ e $b = 7$, dado um ponto $Q$ pertencente a ela, o seu negativo é dado por $-Q = (x, -y)$.

Logo, o custo de realização desse ataque não é proibitivo dentro dos parâmetros das curvas comumente utilizadas, uma vez que o cálculo de $-Q$ consiste nos cálculos de $-y$,$(x + y)$ ou $y + c$ dentro dos seus respectivos corpos finitos.

# Capítulo 6

# Bellare-Neven

## 6.1 Introdução

Frente aos desafios impostos na criação de um esquema de multi-assinatura resistente ao *Rogue-key attack*, foram desenvolvidos diversos esquemas desse tipo que procuravam resistir a esse ataque, porém, até então, essa resistência era acompanhada de requisitos, configurações e nível de complexidade que os tornavam impraticáveis. Diante dessa situação, um esquema de multi-assinatura funcional no *plain public-key model*[1] foi desenvolvido por Mihir Bellare e Gregory Neven e publicada em 2006 no artigo "*Multi-Signatures in the Plain Public-Key Model and a General Forking Lemma*"[5]. O esquema Bellare-Neven (MS-BN) é o principal antecessor do esquema MuSig e no artigo no qual foi publicado é apresentado também o *General Forking Lemma*, parte essencial da prova de segurança do MuSig.

## 6.2 Descrição do esquema Bellare-Neven

**Notação**: nesta seção será descrito o esquema Bellare-Neven em notação multiplicativa, para grupos em abstrato. Cada signatário representa os seus elementos do protocolo $(x, X, r, R, c, s)$ em seu ambiente por meio do índice 1. **Assinatura**:[5]

- entrada local: chave privada do signatário $x_1$, multiset das chaves públicas de todos os co-signatários $L = \{X_1, ..., X_n\}$, mensagem $m$ a ser assinada.
- cada signatário gera de forma aleatória seu $r_1 \xleftarrow{\$} \mathbb{Z}_p$ e calcula $R_1 \leftarrow g^{r_1}$
- cada signatário envia o seu $R_1$ para seus co-signatários

[1] O *Plain Public-Key Model* é um modelo criptográfico no qual não é necessário que signatários apresentem uma prova de conhecimento da chave privada correspondente a sua chave pública aos outros signatários, ou a uma entidade certificadora.

- cada signatário recebe o $R_i$ de cada co-signatário e calcula $R \leftarrow \prod_{i=1}^{n} R_i$
- cada signatário computa seu $c_1 \leftarrow H(X_1||R||\langle L\rangle||m)$, sendo $\langle L\rangle$ uma codificação única de $L$ (exemplo: a sequência das chaves em ordem lexicográfica).
- cada signatário calcula seu $s_1 \leftarrow x_1 c_1 + r_1 \ mod \ p$ e envia $s_1$ para seus co-signatários
- cada signatário calcula $s \leftarrow \sum_{i=1}^{n} s_i \ mod \ p$
- saída local: assinatura $\sigma = (R, s)$

**Verificação**:

- dado $L = \{X_1, \cdots, X_n\}$, a mensagem $m$ e a assinatura $\sigma = (R, s)$
- o verificador calcula $c_1 \leftarrow H(X_1||R||\langle L\rangle||m)$ para todo $1 \leq i \leq n$, sendo $n$ o números de signatários
- $$g^s \stackrel{?}{=} R \prod_{i=1}^{n} X_i^{c_i}$$
$$\text{tal que } X_i^{c_i} = (g_i^x)^{c_i} = g^{x_i c_i}$$

## 6.3 Descrição do esquema Bellare-Neven para Curvas Elípticas

Assinatura:

- entrada local: chave privada do signatário $x_1$, multiset das chaves públicas de todos os co-signatários $L = \{X_1, ..., X_n\}$ (onde $X_i = x_i P$), mensagem $m$ a ser assinada.
- cada signatário gera de forma aleatória seu $r_1 \stackrel{\$}{\leftarrow} \{1, \cdots, k-1\}$, onde $k$ é a ordem de $\langle P\rangle$ e calcula $R_1 \leftarrow r_1 P$
- cada signatário envia o seu $R_1$ para seus co-signatários
- cada signatário recebe o $R_i$ de cada co-signatário e calcula $R \leftarrow \sum_{i=1}^{n} R_i$
- cada signatário computa $c_1 \leftarrow H(X_1||R||\langle L\rangle||m)$, sendo $\langle L\rangle$ uma codificação única de $L$ (exemplo: a sequência das chaves públicas em ordem lexicográfica).
- cada signatário calcula seu $s_1 \leftarrow x_1 c_1 + r_1 \ mod \ k$ e envia $s_1$ para seus co-signatários
- cada signatário calcula $s \leftarrow \sum_{i=1}^{n} s_i \ mod \ k$
- saída local: assinatura $\sigma = (R, s)$

Verificação:

- dado $L = \{X_1, \cdots, X_n\}$, a mensagem $m$ e a assinatura $\sigma = (R, s)$
- o verificador calcula $c_i \leftarrow H(X_i||R||\langle L\rangle||m)$ para todo $1 \leq i \leq n$, sendo $n$ o números de signatários
- $$sP \stackrel{?}{=} R + \sum_{i=1}^{n} c_i X_i$$
$$\text{tal que } c_i X_i = c_i(x_i P) = (c_i x_i P)$$

## 6.4 Prova de corretude

Analisando a corretude da verificação, é possível observar que, de fato:

$$g^s = g^{(\sum_{i=1}^{n} s_i)} = g^{(\sum_{i=1}^{n} x_i c_i + r_i)} = g^{(\sum_{i=1}^{n} x_i c_i)} \cdot g^{(\sum_{i=1}^{n} r_i)}$$

$$= \prod_{i=1}^{n} (g^{x_i})^{c_i} + \prod_{i=1}^{n} g^{r_i} = \prod_{i=1}^{n} X_i^{c_i} + \prod_{i=1}^{n} R_i = \prod_{i=1}^{n} X_i^{c_i} + R$$

$$\implies g^s = R + \prod_{i=1}^{n} X_i^{c_i}$$

E para as curvas elípticas:

$$sP = (\sum_{i=1}^{n} s_i)P = (\sum_{i=1}^{n} x_i c_i + r_i)P = \sum_{i=1}^{n} c_i x_i P + r_i P$$

$$= \sum_{i=1}^{n} c_i x_i P + \sum_{i=1}^{n} r_i P = \sum_{i=1}^{n} c_i X_i + \sum_{i=1}^{n} R_i = R + \sum_{i=1}^{n} c_i X_i$$

$$\implies sP = R + \sum_{i=1}^{n} c_i X_i$$

## 6.5 Prova de segurança

O esquema Bellare-Neven é baseado no esquema de assinatura de Schnorr e sua segurança é comprovável no modelo de oráculo randômico[2] e assume a dureza do problema do

[2]O oráculo randômico funciona como uma *black-box* teórica onde cada requisição única (*input*) que é feita a ele junto a moedas aleatórias $\rho$, tem como resposta um resultado verdadeiramente aleatório (*output*) tomado do domínio de saída. Toda requisição repetida (mesma *input*) tem sempre o mesmo resultado para as mesmas moedas $\rho$. Devido a esse comportamento, o oráculo randômico é utilizado para gerar provas de segurança mais fortes, substituindo as funções de hash nos esquemas criptográficos,

logaritmo discreto[5]. Uma parte fundamental da prova de segurança é o *General Forking Lemma*, que será enunciado e discorrido na seção referente ao MuSig, uma vez que este lema também faz parte de sua prova de segurança.
O seguinte teorema implica que o esquema MS-BN atende às definições de segurança (descritas no artigo) no *plain public-key model* e a prova desse teorema pode ser encontrada com mais detalhes em [5]:

**Teorema 1** *Se existe um* $(t, q_S, q_H, N, \epsilon)$*-falsificador(2)* $\boldsymbol{F}$ *no modelo de oráculo-randômico contra o esquema de multi-assinatura* ***MS-BN*** *descrito acima, então existe um algoritmo* $\boldsymbol{B}$ *tal que* $(t', \epsilon')$*-quebra(1) o problema do logaritmo discreto em* $\mathbb{G}$*, onde*

$$\epsilon' \geq \frac{\epsilon^2}{q_H + q_S} - \frac{2q_H + 16N^2 q_s}{2^{l_0}} - \frac{8Nq_s}{2^k} - \frac{1}{2^{l_1}}$$

$$t' = 2t + q_S t_{exp} + O((q_S + q_H)(1 + q_h + Nq_S))$$

*e* $t_{exp}$ *é o tempo necessário para realizar uma exponenciação em* $\mathbb{G}$*.*

## 6.6 Limitações do esquema Bellare-Neven

É possível observar que o esquema Bellare-Neven é resistente ao *Rogue-key attack* por meio de uma operação que atrela cada chave pública $X_i$ com um hash respectivo $c_i = H(X_i||R||\langle L\rangle||m)$, logo não é mais possível manipular a chave pública $X_i$ de forma a obter um resultado previsível em $\prod_{i=1}^{n} X_i^{c_i}$, uma vez que o resultado de $c_i$ está atrelado aos declarados participantes da assinatura (lista de chaves públicas $L$, as próprias chaves públicas $X_i$ e a mensagem $m$) além de um fator que só é conhecido durante o processo de assinatura ($R = \prod_{i=1}^{n} R_i$, na notação para grupos em abstrato).
Assim, o esquema Bellare-Neven apresenta um esquema de multi-assinatura funcional no *plain public-key model* (principal contribuição frente aos esquemas de multi-assinaturas até a data de sua publicação), diferentemente dos esquemas anteriores onde a resistência a *Rogue-key attacks* era obtida ao custo de aumento significativo da complexidade ou utilizando suposições irrealistas e onerosas na infraestrutura de chaves públicas (como o processo dedicado de geração de chaves no esquema MS-MOR e a suposição do modelo *knowledge of secret key* (KOSK) nos esquemas MS-Bo e MS-LOSSW[5].
No entanto, apesar do tamanho constante das assinaturas geradas pelo o esquema Bellare-Neven, ele não permite gerar uma chave agregada correspondente a um multiset de chaves públicas $L = \{X_1, \cdots, X_n\}$, logo, para realizar a verificação da assinatura $\sigma = (R, s)$, é

nesse caso, caso a segurança do esquema for comprovada utilizando oráculos randômicos, é dito que o a segurança do esquema é comprovável no modelo de oráculo randômico.

necessário que a lista de chaves públicas seja transmitida para o verificador e processada por ele. Dessa forma, a falta da possibilidade de gerar chaves agregadas para verificação impõe um custo adicional de transmissão, armazenamento e processamento que é crítico para aplicações em *blockchains* de criptomoedas descentralizadas.

# Capítulo 7

# MuSig

## 7.1 Introdução

Diante das limitações impostas pela falta de agregação de chaves públicas para verificação de assinaturas no esquema Bellare-Neven (e suas variantes: Bagherzandi et al. (ACM-CCS 2008) e Ma et al. (Des. Codes Cryptogr., 2010)), o esquema **MuSig** foi desenvolvido com o intuito de melhorar esses esquemas de multi-assinatura, que eram estado da arte até então.

O esquema **MuSig** foi publicado em 2018 por Gregory Maxwell, Andrew Poelstra, Yannick Seurin e Pieter Wuille no artigo "*Simple Schnorr Multi-Signatures with Applications to Bitcoin*". Trata-se de um esquema de multi-assinaturas com segurança comprovável no *plain public-key model* que permite a produção de assinaturas conjuntas e de chaves públicas agregadas, as quais mantêm o mesmo tamanho de assinaturas e chaves públicas individuais, respectivamente. Com isso, é possível realizar a verificação de uma assinatura conjunta por meio de um processo de verificação similar a de uma assinatura de Schnorr individual, utilizando-se da chave pública agregada.

São também apresentadas melhorias em relação a privacidade dos signatários, uma vez que, ao se utilizar uma chave pública agregada para verificar uma assinatura respectiva, as chaves públicas utilizadas para gerar a chave pública agregada e a sua contagem, são informações conhecidas apenas pelos signatários que geraram a assinatura.

## 7.2 Diferenças entre MuSig e Bellare-Neven

- O desafio $c_i$ do esquema Bellare-Neven é modificado para: $c_i = H_{agg}(\langle L \rangle, X_i) \cdot H_{sig}(\tilde{X}, R, m)$, onde $\langle L \rangle$ é uma codificação única de $L$ e $\tilde{X}$ é a chave pública agregada correspondente ao conjunto de chaves públicas $L = \{X_1, \cdots, X_n\}$, definida como

(em notação multiplicativa, para grupos em abstrato):

$$\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}, \text{ onde } a_i = H_{agg}(\langle L \rangle, X_i)$$

- Note que os $a_i$'s dependem apenas das chaves públicas dos signatários
- A verificação é modificada para:

$$g^s \stackrel{?}{=} R \prod_{i=1}^{n} X_i^{a_i c} = R\tilde{X}^c, \text{ onde } c = H_{sig}(\tilde{X}, R, m)$$

Para grupos de curvas elípticas:

- $$c_i = H_{agg}(\langle L \rangle, X_i) \cdot H_{sig}(\tilde{X}, R, m)$$
- $$\tilde{X} = \sum_{i=1}^{n} a_i X_i, \text{ onde } a_i = H_{agg}(\langle L \rangle, X_i) \text{ e } X_i = x_i P$$
- Verificação:

$$sP = R + \sum_{i=1}^{n} a_i c X_i = R + c\tilde{X}, \text{ onde } c = H_{sig}(\tilde{X}, R, m)$$

## 7.3 Descrição do esquema MuSig

**Notação**:

- Dado um conjunto $S$ não vazio, $s \stackrel{\$}{\leftarrow} S$ denota a operação de tomar um elemento de $S$ de forma aleatória e uniforme e então o atribuir a variável $s$
- A notação $\langle L \rangle$ é substituída por somente $L$, ao ser assumido que o conjunto das chaves públicas $L = \{X_1, \cdots, X_n\}$ dado como entrada está em uma codificação única (ex: ordem lexicográfica).
- Três funções de hash (não necessariamente distintas) $H_{com}, H_{agg}, H_{sig}$, de $\{0,1\}^*$ para $\{0,1\}^l$.

- $H_{com}$ é utilizada na fase de de comprometimento (*commitment*)[1], $H_{agg}$ é utilizada para realizar o cálculo do hash da chave agregada e $H_{sig}$ é utilizada para calcular o hash da assinatura. Essas funções de hash podem ser construídas a partir de apenas uma única função de hash utilizando a separação de domínio apropriada.
- A tripla $(\mathbb{G}, p, g)$ com $\mathbb{G}$ sendo o grupo cíclico de ordem $p$ e $g$ é um gerador de $\mathbb{G}$

**Geração de chaves**:

- chave privada aleatória $x \xleftarrow{\$} \mathbb{Z}_p$
- chave pública $X = g^x$

**Assinatura**:

- $X_1$ e $x_1$ são respectivamente a chave pública e a chave privada do signatário
- $m$ é a mensagem a ser assinada
- $X_2, \cdots, X_n$ são as chaves públicas dos co-signatários
- $L = \{X_1, \cdots, X_n\}$ é o conjunto de todas as chaves públicas envolvidas no processo de assinatura
- para cada $i \in \{1, \cdots, n\}$, cada signatário calcula: $a_i = H_{agg}(L, X_i)$
- a chave pública "agregada"é calculada:

$$\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$$

- cada signatário gera seu $r_1$ randômico, tal que $r_1 \xleftarrow{\$} \mathbb{Z}_p$ e calcula $R_1 = g_1^r$
- cada signatário calcula seu compromisso $t_1 = H_{com}(R_1)$ e envia $t_1$ para os seus co-signatários
- uma vez recebidos os compromissos $t_2, \cdots, t_n$, o signatário envia $R_1$

[1]Esta fase de comprometimento, se refere ao processo de "*bit commitment*"(comprometimento de bits). Neste processo, dois pares da comunicação, chamados aqui de Alice e Bob, desejam cada um transmitir entre si, uma informação respectiva que não é de conhecimento mútuo, denotadas por $x_A$ e $x_B$. Logo, para que Alice possa garantir que o $x_B$ de Bob não seja modificado no momento que Bob recebe o $x_A$ de Alice (para garantir, por exemplo, que Bob não construa seu $x_B$ em função do $x_A$ de Alice para realizar um ataque), Alice exige um compromisso de $x_B$ antes de enviar o seu $x_A$ a Bob. Uma forma de gerar este compromisso é calculando o valor de $H(x_B)$, onde $H$ é uma função de hash criptográfica, dessa forma, Bob pode gerar um compromisso de $x_B$ sem revelar (idealmente) informações do próprio $x_B$. Por meio desse hash, Alice, ao receber $x'_B$ de Bob, pode verificar se $H(x'_B) = H(x_B)$ e confirmar se houve modificações ao $x_B$ de Bob ou não.

- uma vez recebidos $R_2, \cdots, R_n$ dos seus co-signatários, é checado se $t_i \stackrel{?}{=} H_{com}(R_i)$ para cada $t_i$ e seu respectivo $R_i$
- se forem válidos, é computado:
  - $$R = \prod_{i=1}^{n} R_i$$
  - $$c = H_{sig}(\tilde{X}, R, m)$$
  - $$s_1 = r_1 + c \cdot a_1 \cdot x_1 \ mod \ p$$
- cada signatário envia seu $s_1$ para seus co-signatários e após receber $s_2, \cdots, s_n$ é computado:
$$s = \sum_{i=1}^{n} s_i \ mod \ p$$
- A assinatura é:
$$\sigma = (R, s)$$

**Verificação**:

- dado $L = X_1, \cdots, X_n$, a mensagem $m$ e a assinatura $\sigma = (R, s)$
- o verificador calcula:

$$a_i = H_{agg}(L, X_i) \text{ para cada } i \in \{1, \cdots, n\}$$

$$\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$$

$$c = H_{sig}(\tilde{X}, R, m)$$

- o verificador checa se:
$$g^s \stackrel{?}{=} R \prod_{i=1}^{n} X_i^{a_i c} = R\tilde{X}^c$$

## 7.4 Descrição do esquema MuSig para grupos de curvas elípticas

Considerando agora a tripla $(E(\mathbb{F}_q), k, P)$ com $\mathbb{F}_q$ sendo o curva elíptica $E$ descrita sobre um corpo $\mathbb{F}_q$, sendo $\langle P \rangle$ o grupo cíclico de ordem $k$ gerado a partir do ponto base $P \in$

$E(\mathbb{F}_q)$, temos:

**Geração de chaves**:

- chave privada aleatória $x \xleftarrow{\$} \{1, \cdots, k-1\}$
- chave pública $X = xP$

**Assinatura**:

- $X_1$ e $x_1$ são respectivamente a chave pública e a chave privada do signatário
- $m$ é a mensagem a ser assinada
- $X_2, \cdots, X_n$ são as chaves públicas dos co-signatários
- $L = \{X_1, \cdots, X_n\}$ é o conjunto de todas as chaves públicas envolvidas no processo de assinatura
- para cada $i \in \{1, \cdots, n\}$, cada signatário calcula: $a_i = H_{agg}(L, X_i)$
- a chave pública "agregada"é calculada:

$$\tilde{X} = \sum_{i=1}^{n} a_i X_i$$

- cada signatário gera seu $r_1$ randômico, tal que $r_1 \xleftarrow{\$} \{1, \cdots, k-1\})$, e calcula $R_1 = r_1 P$
- cada signatário calcula seu compromisso $t_1 = H_{com}(R_1)$ e envia $t_1$ para os seus co-signatários
- uma vez recebidos os compromissos $t_2, \cdots, t_n$, o signatário envia seu $R_1$
- uma vez recebidos $R_2, \cdots, R_n$ dos seus co-signatários, é checado $t_i \stackrel{?}{=} H_{com}(R_i)$ para cada $t_i$ e seu respectivo $R_i$
- se forem válidos, é computado:
  - $$R = \sum_{i=1}^{n} R_i$$
  - $$c = H_{sig}(\tilde{X}, R, m)$$
  - $$s_1 = r_1 + c \cdot a_1 \cdot x_1 \ mod \ k$$

- cada signatário envia seu $s_1$ para os co-signatários e após receber $s_2, \cdots, s_n$ é computado:

$$s = \sum_{i=1}^{n} s_i \ mod \ k$$

- A assinatura é:

$$\sigma = (R, s)$$

**Verificação**:

- dado $L = \{X_1, \cdots, X_n\}$, a mensagem $m$ e a assinatura $\sigma = (R, s)$
- o verificador calcula:

$$a_i = H_{agg}(L, X_i) \text{ para cada } i \in \{1, \cdots, n\}$$

$$\tilde{X} = \sum_{i=1}^{n} a_i X_i$$

$$c = H_{sig}(\tilde{X}, R, m)$$

- o verificador checa se:

$$sP \stackrel{?}{=} R + \sum_{i=1}^{n} a_i c X_i = R + c\tilde{X}$$

Observe que, pela equação acima, também é possível realizar a verificação utilizando apenas $(R, s)$, $m$ e $\tilde{X}$ transmitidos para o verificador. Neste caso o verificador apenas calcula $c = H_{sig}(\tilde{X}, R, m)$ e verifica se $sP \stackrel{?}{=} R + c\tilde{X}$, logo, não é necessário a transmissão do multiset $L$, assim como não é necessário o cálculo de $\tilde{X}$ a partir deste $L$.

## 7.5 Prova de Corretude

Analisando a corretude da verificação, é possível observar que, de fato:

$$g^s = g^{(\sum_{i=1}^{n} s_i)} = g^{(\sum_{i=1}^{n} (r_i + c a_i x_i))}$$

$$= g^{(\sum_{i=1}^{n} r_i)} \cdot g^{(\sum_{i=1}^{n} c a_i x_i)} = \prod_{i=1}^{n} g^{r_i} \cdot \prod_{i=1}^{n} (g^{x_i})^{c a_i}$$

$$= \prod_{i=1}^{n} R_i \cdot \prod_{i=1}^{n} (X_i^{a_i})^c = R \cdot (\prod_{i=1}^{n} X_i^{a_i})^c = R \cdot \tilde{X}^c$$

$$\implies g^s = R \cdot (\prod_{i=1}^{n} X_i^{a_i})^c = R \cdot \tilde{X}^c$$

E para as curvas elípticas:

$$sP = (\sum_{i=1}^{n} s_i)P = (\sum_{i=1}^{n}(r_i + ca_i x_i))P$$

$$= (\sum_{i=1}^{n} r_i + \sum_{i=1}^{n} ca_i x_i)P = \sum_{i=1}^{n} r_i P + \sum_{i=1}^{n} ca_i x_i P$$

$$= \sum_{i=1}^{n} R_i + \sum_{i=1}^{n} ca_i X_i = R + c\sum_{i=1}^{n} a_i X_i = R + c\tilde{X}$$

$$\implies sP = R + c\sum_{i=1}^{n} a_i X_i = R + c\tilde{X}$$

## 7.6 Prova de Segurança

**Notação:**

- Seja **A** um algoritmo randomizado, tal que $y \leftarrow \mathbf{A}(x_1, \cdots; \rho)$ denote a operação de executar **A** utilizando as entradas $x_1, \cdots$ e moedas aleatórias $\rho$ e atribuir a sua saída a $y$.
- Quando as moedas $\rho$ são escolhidas de forma uniforme e aleatória, a operação é denotada por $y \xleftarrow{\$} \mathbf{A}(x_1, \cdots)$.
- Sejam $(\mathbb{G}, p, g)$ os parâmetros do grupo, onde $\mathbb{G}$ é um grupo cíclico de ordem $p$, $p$ é um inteiro de $k$-bits e $g$ é um gerador de $\mathbb{G}$. Serão adotados $(\mathbb{G}, p, g)$ como fixos, mas o comprimento $k$ de $p$ pode ser considerado como um parâmetro de segurança se necessário.[2]
- Um algoritmo de multi-assinatura $\Pi$ consiste de três algoritmos (**KeyGen**, **Sign**,**Ver**).
- **KeyGen** é o algoritmo de geração aleatória de chaves, que produz o par de chaves privada/público $(\mathbf{sk}, \mathbf{pk}) \xleftarrow{\$} \mathbf{KeyGen}$.
- **Sign** é o algoritmo de assinatura executado por cada signatário, tomando como entrada o par de chaves $(\mathbf{sk}, \mathbf{pk})$, o multiset de chaves públicas $L = \{\mathbf{pk}_1, \cdots, \mathbf{pk}_n\}$ contendo pelo menos uma instância de $pk$ e a mensagem $m$, retornando ao final da execução uma assinatura $\sigma$ para $L$ e $m$ (sendo produzido o mesmo $\sigma$ para todos os signatários caso não haja desvio do protocolo).
- **Ver** é um algoritmo determinístico de verificação que toma o multiset $L = \{\mathbf{pk}_1, \cdots, \mathbf{pk}_n\}$, a mensagem $m$ e a assinatura $\sigma$ e retorna 1 se $\sigma$ for válido para $L$ e $m$, caso contrário, ele retorna 0.

Será enunciado a seguir o problema do logaritmo discreto aplicado ao modelo de segurança proposto:

**Definição 1** [Problema do Logaritmo Discreto] Seja $(\mathbb{G}, p, g)$ o parâmetro do grupo. Um algoritmo **A** é dito que $(t, \varepsilon)$-soluciona o problema do logaritmo discreto com relação a $(\mathbb{G}, p, g)$ se a partir de uma entrada de um elemento aleatório $X$ do grupo, ele opera em um tempo de até $t$ e retorna $x \in \{0, \cdots, p-1\}$ tal que $X = g^x$ com probabilidade de no mínimo $\varepsilon$, tal que a probabilidade é tomada sobre um sorteio aleatório de $X$ e sobre as moedas aleatórias de **A**.[2]

E, de forma semelhante, é enunciado o *General Forking Lemma*:

**Lema 1 (Generalized Forking Lemma)** *Afixe os inteiros $q$ e $l$. Seja $\boldsymbol{A}$ um algoritmo randomizado que toma como entrada uma entrada principal **inp** e strings de $l$-bits $h_1, \cdots, h_q$ que retorna ou um símbolo diferenciado de falha $\perp$ ou o par $(i, \boldsymbol{out})$, onde $i \in \{1, \cdots, q\}$ e **out** é alguma saída lateral. A probabilidade de aceitação (accepting probability) de $\boldsymbol{A}$, denotada por $\boldsymbol{acc}(\boldsymbol{A})$, é definida como a probabilidade sobre um sorteio aleatório de **inp**(de acordo com uma distribuição bem conhecida), $h_1, \cdots, h_q \xleftarrow{\$} \{0,1\}^l$, e as moedas aleatórias de $\boldsymbol{A}$, tal que $\boldsymbol{A}$ retorna uma saída diferente de $\perp$. Considere o algoritmo $\text{Fork}^{\boldsymbol{A}}$, tomando **inp** como entrada assim como em no algoritmo 1. Seja **frk** a probabilidade (sobre um sorteio de **inp** e as moedas aleatórias de $\text{Fork}^{\boldsymbol{A}}$) tal que $\text{Fork}^{\boldsymbol{A}}$ retorna uma saída diferente de $\perp$. Então:*

$$\boldsymbol{frk} \geq \boldsymbol{acc}(\boldsymbol{A})\left(\frac{\boldsymbol{acc}(\boldsymbol{A})}{q} - \frac{1}{2^l}\right)$$

Será descrito em seguida o funcionamento do "**jogo de segurança**"no qual se baseia a prova.

O modelo de segurança utilizado pelo MuSig necessita que seja inviável forjar multi-assinaturas envolvendo no mínimo um signatário honesto. Será assumido, sem perda de generalidade, que existe apenas um signatário honesto e todos os outros foram corrompidos pelo adversário, assim, este adversário tem controle de todas as outras chaves públicas envolvidas na assinatura e as escolhe de forma arbitrária (possivelmente produzindo-as em função da chave pública do signatário honesto). O adversário pode participar em qualquer número de protocolos de assinatura com o signatário honesto antes de retornar a sua tentativa de falsificação[2].

Assim, o "jogo de segurança"envolvendo um adversário (falsificador) **F** procede da seguinte forma[2]:

**Algorithm 1** $\text{Fork}^{\mathbf{A}}$

**Entrada: inp**
**Saída:** $\perp$ ou $(i, \mathbf{out}, \mathbf{out}')$

sorteie moedas aleatórias $\rho$ para $\mathbf{A}$
$h_1, \cdots, h_q \xleftarrow{\$} \{0,1\}^l$
$\alpha \leftarrow \mathbf{A}(\mathbf{inp}, h_1, \cdots, h_q; \rho)$
**if** $\alpha = \perp$ **then**
  **return** $\perp$
**else**
  processe $\alpha$ como $(i, \mathbf{out})$
**end if**
$h'_1, \cdots, h'_q \xleftarrow{\$} \{0,1\}^l$
$\alpha' \leftarrow \mathbf{A}(\mathbf{inp}, h_1, \cdots, h_{i-1}, h'_i, \cdots, h'_q; \rho)$
**if** $\alpha' = \perp$ **then**
  **return** $\perp$
**else**
  processe $\alpha'$ como $(i', \mathbf{out}')$
**end if**
**if** $(i = i'$ e $h_i \neq h'_i)$ **then**
  **return** $(i, \mathbf{out}, \mathbf{out}')$
**else**
  **return** $\perp$
**end if**

- É gerado um par de chaves aleatório $(\mathbf{sk*}, \mathbf{pk*}) \xleftarrow{\$} \mathbf{KeyGen}$ para o signatário honesto e **pk*** é fornecido como entrada para **F**.
- O adversário tem acesso a um oráculo de assinatura que toma $L = \{\mathbf{pk}_1, \cdots, \mathbf{pk}_n\}$, no qual **pk*** ocorre no mínimo uma vez, e a mensagem $m$. O oráculo de assinatura implementa o algoritmo de assinatura correspondente a chave secreta **sk*** do signatário honesto (efetivamente operando como o signatário honesto), enquanto o falsificador toma o papel de todos os outros signatários em $L$ (possivelmente desviando do protocolo). Note que a chave pública **pk*** do signatário honesto pode aparecer $k \geq 2$ vezes em $L$, assim, o adversário faz o papel de $k-1$ instâncias de **pk***, porém como o algoritmo depende apenas no multiset $L$, não há necessidade de especificar quais instâncias de **pk*** estão associadas com o oráculo honesto e o adversário. O falsificador pode interagir de forma concorrente com quantas instâncias do oráculo de assinatura honesto ele desejar.
- Ao fim da execução, o falsificador retorna um multiset de chaves públicas $L = \{\mathbf{pk}_1, \cdots, \mathbf{pk}_n\}$, a mensagem $m$, e uma assinatura $\sigma$. O falsificador vence se $\mathbf{pk*} \in L$, a falsificação é válida (ou seja $\mathbf{Ver}(L, m, \sigma) = 1$) e **F** nunca iniciou uma instância do protocolo de assinatura com o signatário honesto, para o multiset $L$ e a mensagem $m$, que produzisse $\sigma$.

Adicionalmente, se for utilizado o modelo de Oráculo Randômico, o adversário pode fazer solicitações arbitrárias e aleatórias ao oráculo em qualquer momento do jogo de segurança.[2]

Assim, a segurança é definida da seguinte forma:

**Definição 2** Seja Π = (KeyGen,Sign,Ver) um esquema de multi-assinatura. Dizemos que um adversário **F** é um $(t, q_s, q_h, N, \varepsilon)$-falsificador (*forger*) no modelo de oráculo randômico contra um esquema de multi-assinatura Π se ele opera em até um tempo $t$, inicia até $q_s$ instâncias do protocolo de assinatura com o signatário honesto, realiza até $q_h$ consultas randômicas ao oráculo e vence o jogo de segurança descrito acima com probabilidade de no mínimo $\varepsilon$, sendo o tamanho de $L$ no máximo $N$ em qualquer consulta de assinatura e na falsificação resultante. [2]

Outro ponto necessário para a produção da prova de segurança do **MuSig** é explicitar a técnica chave utilizada no artigo.

***Técnica do Double-Forking***[2]: com base no jogo de segurança descrito anteriormente (7.6), a interação de um falsificador **F** com o oráculo de assinatura honesto se dá da seguinte forma:

- seja $(x^*, X^*)$ o par de chaves privada/pública do signatário honesto
- $X^*$ é fornecida como entrada ao falsificador **F**
- $L$ é o multiset de chaves públicas, tal que $X^* \in L$ e o falsificador **F** simula todos os signatários exceto uma única instância de $X^*$
- sendo $m$ a mensagem a ser assinada, **F** envia $L$ e $m$ ao oráculo de assinatura, iniciando o protocolo de assinatura
- o oráculo de assinatura processa $L$ como $L = \{X_1 = X^*, X_2, \cdots, X_n\}$, sorteia aleatoriamente $r_1 \xleftarrow{\$} \mathbb{Z}_p$, calcula $R_1 = g^{r_1}$ e envia $t_1 = H_{com}(R_1)$ a **F**
- **F** então envia $t_2, \cdots, t_n$ ao oráculo
- o oráculo de assinatura envia $R_1$ a **F**, que responde com $R_2, \cdots, R_n$
- se $\forall i \in \{2, \cdots, n\},\ t_i = H_{com}(R_i)$, então:

$$\begin{aligned} R =& \prod_{i=2}^{n} R_i \\ c =& H_{sig}(\tilde{X}, R, m) \\ s_1 =& r_1 + ca_1x^* \ mod\ p, \end{aligned}$$

  e envia $s_1$ para **F**.
- os passos seguintes do protocolo podem ser omitidos, uma vez que o comportamento do oráculo de assinatura não depende de $x^*$ e pode ser perfeitamente simulado por **F**.

O ponto crucial da prova está em como extrair o logaritmo discreto $x^*$ da chave pública $X^*$ apresentada como o desafio. A técnica padrão para tal seria a de bifurcar (*fork*) a execução de **F** (formando assim duas execuções concorrentes de **F** a partir de um determinado ponto no protocolo) de forma a obter duas falsificações **válidas** $(R, \sigma)$ e $(R', \sigma')$ para o mesmo multiset $L = \{X_1, \cdots, X_n\}$ (com $X^* \in L$) e mesma mensagem $m$ (i.e. $\mathbf{Ver}(L, m, \sigma) = 1$ e $\mathbf{Ver}(L, m, \sigma') = 1$), tal que $R = R'$, $H_{sig}(\tilde{X}, R, m)$ foi programada para o mesmo valor de $h_1$ em ambas as execuções, $H_{agg}(L, X_i)$ foi programada para o mesmo valor de $a_i$ para cada $i$ tal que $X_i \neq X^*$, e, por fim, $H_{agg}(L, X^*)$ foi programada para valores distintos $h_0$

e $h'_0$ para as suas respectivas execuções, implicando que:

$$\begin{aligned} g^s &= R(X^*)^{n^* h_0 h_1} \prod_{i \in \{1,\cdots,n\}, X_i \neq X*} X_i^{a_i h_1} \\ g^{s'} &= R(X^*)^{n^* h'_0 h_1} \prod_{i \in \{1,\cdots,n\}, X_i \neq X*} X_i^{a_i h_1}, \end{aligned}$$

onde $n^*$ é o número instâncias de $X^*$ em $L$. Assim, é possível obter o valor do logaritmo discreto de $X^*$ dividindo as duas equações acima:

$$\frac{g^s}{g^{s'}} = \frac{R(X^*)^{n^* h_0 h_1} \prod X_i^{a_i h_1}}{R(X^*)^{n^* h'_0 h_1} \prod X_i^{a_i h_1}} \tag{7.1}$$

$$\implies \frac{g^s}{g^{s'}} = \frac{(X^*)^{n^* h_0 h_1}}{(X^*)^{n^* h'_0 h_1}} \tag{7.2}$$

$$\implies \frac{g^s}{g^{s'}} = \frac{(X^*)^{(n^* h_1) h_0}}{(X^*)^{(n^* h_1) h'_0}} \tag{7.3}$$

$$\implies g^{(s-s')} = (X^*)^{(n^* h_1) h_0 - (n^* h_1) h'_0} \tag{7.4}$$

$$\implies g^{(s-s')} = (X^*)^{(n^* h_1)(h_0 - h'_0)} \tag{7.5}$$

$$\implies g^{(s-s')} = (g^{x^*})^{(n^* h_1)(h_0 - h'_0)} \tag{7.6}$$

$$\implies s - s' \equiv x^* (n^* h_1)(h_0 - h'_0) \bmod p \tag{7.7}$$

$$\implies x^* \equiv (s - s')[(n^* h_1)(h_0 - h'_0)]^{-1} \bmod p \tag{7.8}$$

No entanto, simplesmente realizar a bifurcação da execução em relação a resposta da consulta $H_{agg}(L, X^*)$ não funciona[2]: observando a sequência de eventos durante o protocolo de assinatura, é possível notar que não há garantia de que a consulta $H_{sig}(\tilde{X}, R, m)$ já foi realizada por **F** antes da bifurcação, de forma que não é possível garantir se essa consulta será realizada novamente na segunda execução, situação em que não se terá garantias de que a falsificação produzida na segunda execução será em relação ao mesmo valor $h_1 = H_{sig}(\tilde{X}, R, m)$.

Logo, para remediar a situação acima, são realizadas bifurcações em dois momentos da execução de **F**:

- uma vez em relação a resposta de $H_{sig}(\tilde{X}, R, m)$, que permite a obtenção do logaritmo discreto $\tilde{x}$ da chave agregada $\tilde{X}$, em relação a qual **F** retorna uma falsificação,
- em seguida, uma bifurcação em relação a resposta de $H_{agg}(L, X^*)$, a qual permite a obtenção do logaritmo discreto $x^*$ da chave pública $X^*$.

Será apresentado a seguir as abstrações dos algoritmos utilizados para executar o falsificador, realizar as bifurcações necessárias e assim serem analisados os parâmetros de segurança do MuSig.

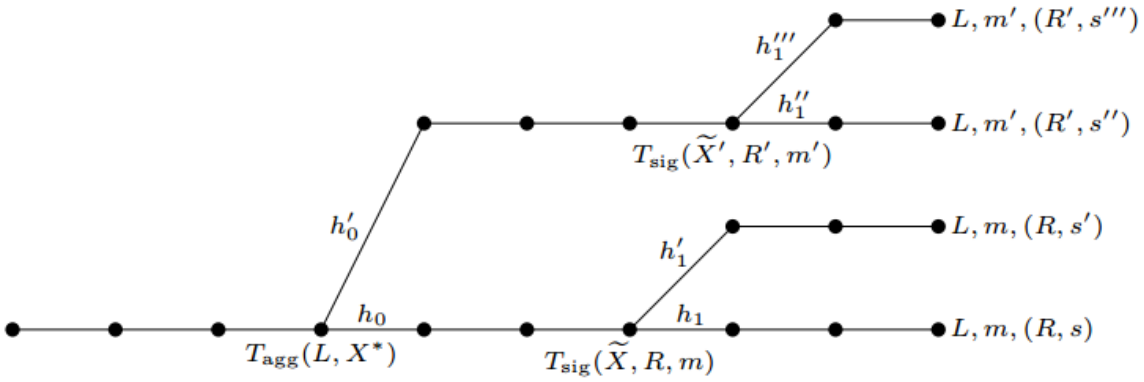


Figura 7.1: Exemplo de uma possível execução do algoritmo **C**. Cada caminho desde a raiz mais a esquerda até as folhas mais a direita representa uma execução do falsificador **F**. Cada vértice simboliza uma atribuição às tabelas $T_{agg}$ e $T_{sig}$, usadas para armazenar valores programados para $H_{agg}$ e $H_{sig}$ respectivamente, e as arestas que originam desses vértices simbolizam o valor utilizado na atribuição (por exemplo: a execução de $H_{agg}$ provoca uma bifurcação tal que, em uma execução é retornado e utilizado o valor $h_0$ enquanto na outra é retornado e utilizado o valor $h_0'$). As folhas simbolizam a falsificação resultante retornada pelo falsificador. Apenas são rotulados vértices e arestas relevantes à falsificação[2].

Dentro das descrições que se seguem, considera-se que o falsificador **F** nunca repete uma mesma consulta e suas consultas são sempre "bem-formadas", isto é, $X^* \in L$ e $X \in L$ para qualquer consulta a $H_{agg}$. Também é assumido que são realizadas exatamente $q_h$ consultas a cada oráculo randômico e são feitas exatamente $q_s$ consultas de assinatura.

**Algoritmo A**: funciona como um algoritmo "invólucro"da execução do falsificador **F**, simulando a execução de $H_{com}, H_{agg}, H_{sig}$ por meio do acesso uniforme e aleatório de valores programados nas tabelas $T_{com}, T_{agg}, T_{sig}$ e simulando o oráculo de assinatura programando $H_{sig}$. De forma geral, uma consulta de **hash** a um oráculo randômico $H$, realizada por meio de $H(p_1, \cdots, p_n)$ (sendo $p_1, \cdots, p_n$, o parâmetros utilizados em $H$), é verificada em sua respectiva tabela $T$ e é retornado $T(p_1, \cdots, p_n)$ se possível, e caso $T(p_1, \cdots, p_n)$ seja indefinido, um novo valor é atribuído a $T(p_1, \cdots, p_n)$ e este valor é então retornado como resposta da consulta.

Dois contadores $\mathbf{ctr}_0$ e $\mathbf{ctr}_1$ (inicialmente zero) são utilizados para identificar, respectivamente, uma nova inclusão da forma $T_{agg}(\cdot, X^*)$ e cada nova inclusão em $T_{sig}$.

Se uma consulta $H_{agg}(L, X)$ (com $X^*, X \in L$) é realizada e $T_{agg}(L, X)$ é indefinido, então **A** incrementa $\mathbf{ctr}_0$, atribui $T_{agg}(L, X^*) := h_{0,\mathbf{ctr}_0}$, atribui aleatoriamente $\forall X \in L$ $X^*, T_{agg}(L, X) \xleftarrow{\$} \{0,1\}^l$ e retorna $T_{agg}(L, X)$ como resposta da consulta.

Se uma consulta $H_{sig}(\tilde{X}, R, m)$ é realizada e $T_{sig}(\tilde{X}, R, m)$ é indefinido, então **A** incrementa $\mathbf{ctr}_1$, atribui $T_{sig}(\tilde{X}, R, m) := h_{1,\mathbf{ctr}_1}$ e retorna $T_{sig}(\tilde{X}, R, m)$ como resposta da

consulta.

O algoritmo **A** toma como entrada as moedas aleatórias $\rho_A$, a chave pública $X^*$ do signatário honesto e as strings de $l$-bits $h_{0,1}, \cdots, h_{0,q}$ e $h_{1,1}, \cdots, h_{1,q}$, utilizadas para programar as tabelas $T_{agg}$ e $T_{sig}$ respectivamente. Sua execução se inicia tomando moedas aleatórias $\rho_F$ e executando o falsificador **F** com $X^*$ como sua entrada. Durante a execução, o algoritmo responde a consultas a $H_{com}, H_{agg}, H_{sig}$ de **F**, monitorando a ordem de execução das consultas (por exemplo a consulta $H_{sig}(\tilde{X}, R, m)$ não deve ocorrer antes da consulta a $H_{agg}(L, X^*)$) e ativando *flags* que indicam se um "evento ruim"ocorreu (como a execução fora da ordem desejada).

Se alguma *flag* de "eventos ruins"foi ativada, **F** retornou $\perp$ ou a falsificação produzida por **F** é inválida, **A** retorna $\perp$. Caso contrário, **A** retorna $(i_0, i_1, L, R, s, \mathbf{a})$, com $i_0$ sendo o índice tal que $T_{agg}(L, X^*) = h_{0,i_0}$, $i_1$ o índice tal que $T_{sig}(\tilde{X}, R, m) = h_{1,i_1}$, $L$ o conjunto de chaves públicas utilizado para gerar $\tilde{X}$, $(R, s)$ a falsificação válida produzida e $\mathbf{a} = (a_1, \cdots, a_n)$, com $a_i = h_{0,i_0}$ para $X_i = X^*$.

Calculando a probabilidade de aceitação de **A**, é obtido o seguinte lema:

**Lema 2 (Probabilidade de aceitação mínima de A)** *Assuma a existência de um $(t, q_S, q_H, N, \varepsilon)$-falsificador **F** no modelo de oráculo randômico contra o esquema de multi-assinatura **MuSig** com parâmetros de grupo $(\mathbb{G}, p, g)$ e seja $q = q_h + q_s + 1$. Então, existe um algoritmo **A** que toma como entrada um elemento do grupo uniformemente aleatório $X^*$ e strings uniformemente aleatórias de $l$-bits $h_{0,1}, \cdots, h_{0,q}$ e $h_{1,1}, \cdots, h_{1,q}$ e com probabilidade de aceitação (assim como definido em 1) de no mínimo*

$$\varepsilon - \frac{4q_s(q_h + Nq_s)}{2^k} - \frac{4(q_h + Nq_s)^2}{2^l},$$

*saídas $(i_0, i_1, L, R, s, \boldsymbol{a})$ onde $i_0, i_1 \in \{1, \cdots, q\}$, $L = \{X_1, \cdots, X_n\}$ é um multiset de chaves públicas tal que $X^* \in L$, $\boldsymbol{a} = (a_1, \cdots, a_n)$ é uma tupla de valores com $l$-bits tal que $a_i = h_{0,i_0}$ para qualquer $i$ tal que $X_i = X^*$ e*

$$g^s = R \prod_{i=1}^{n} X_i^{a_i h_{1,i_1}}$$

**Algoritmo B**: é construído a partir do algoritmo **A** e executa $\mathbf{Fork}^{\mathbf{A}}$, essencialmente realizando a bifurcação de **A** com relação a resposta da consulta a $H_{sig}$ (mais especificamente, $H_{sig}(\tilde{X}, R, m)$).

O algoritmo **B** toma como entrada moedas aleatórias $\rho_A$, a chave pública $X^*$ do signatário honesto e string de $l$-bits uniformemente aleatórias $h_{0,1}, \cdots, h_{0,q}$.

Sua execução se inicia tomando moedas aleatórias $\rho_A$, tomando as strings de $l$-bits $h_{1,1}, \cdots, h_{1,q}$, de forma uniforme e aleatória, e executando o algoritmo $\mathbf{A}$ com $\rho_A$, $X^*$, $h_{0,1}, \cdots, h_{0,q}$ e $h_{1,1}, \cdots, h_{1,q}$ como suas entradas.

Se $\mathbf{A}$ retorna $\perp$ então $\mathbf{B}$ retorna $\perp$. Caso contrário $\mathbf{A}$ retorna $(i_0, i_1, L, R, s, \mathbf{a})$, tal que $L = X_1, \cdots, X_n$ e $\mathbf{a} = (a_1, \cdots, a_n)$, e $\mathbf{B}$ executa $\mathbf{A}$ novamente com $\rho_A$, $X^*$, $h_{0,1}, \cdots, h_{0,q}$ (os mesmos para a primeira execução de $\mathbf{A}$) e $h_{1,1}, \cdots, h_{1,i_1-1}, h'_{1,i_1}, \cdots, h'_{1,q}$, tal que $h'_{1,i_1}, \cdots, h'_{1,q}$são strings de $l$-bits tomadas de forma uniforme e aleatória.

Se $\mathbf{A}$ retorna $\perp$ nesta segunda execução, então $\mathbf{B}$ retorna $\perp$. Caso contrário $\mathbf{A}$ retorna $(i'_0, i'_1, L', R', s', \mathbf{a}')$, com $L' = \{X'_1, \cdots, X'_n\}$ e $\mathbf{a}' = (a'_1, \cdots, a'_n)$.

Se $i_1 \neq i'_1$ ou $i_1 = i'_1$ e $h_{1,i_1} = h'_{1,i_1}$, então $\mathbf{B}$ retorna $\perp$.

Caso contrário, $i_1 = i'_1$ e $h_{1,i_1} \neq h'_{1,i_1}$, então $i_0 = i'_0, L = L', R = R', \mathbf{a} = \mathbf{a}'$ e $\tilde{X}' = \tilde{X}^{\prime\prime}$, pois ambas as execuções de $\mathbf{A}$ são idênticas até $T_{sig}(\tilde{X}, R, m) := h_{1,i_1}$ e $T_{sig}(\tilde{X}^{\prime\prime}, R', m') := h'_{1,i_1}$, logo, os parâmetros de $T_{sig}$ devem ser iguais.

Dessa forma, de acordo com o Lema 2, as saídas produzidas por ambas as execuções de $\mathbf{A}$ seguem:

$$g^s = R\tilde{X}^h_{1,i_1} \text{ e } g^{s'} = R'\tilde{X}'^{h'_{1,i_1}} = R\tilde{X}^{h'_{1,i_1}}$$

Logo, dividindo a primeira equação pela segunda, é obtido:

$$\frac{g^s}{g^{s'}} = \frac{R\tilde{X}^{h_{1,i_1}}}{R\tilde{X}^{h'_{1,i_1}}} \tag{7.9}$$

$$\implies g^{s-s'} = \frac{g^{\tilde{x}h_{1,i_1}}}{g^{\tilde{x}h'_{1,i_1}}} \tag{7.10}$$

$$\implies g^{s-s'} = g^{(\tilde{x}h_{1,i_1})-(\tilde{x}h'_{1,i_1})} \tag{7.11}$$

$$\implies s - s' = (\tilde{x}(h_{1,i_1} - h'_{1,i_1}) \bmod p \tag{7.12}$$

$$\implies \tilde{x} = (s - s')(h_{1,i_1} - h'_{1,i_1})^{-1} \bmod p \tag{7.13}$$

tal que, $\tilde{x}$ é o logaritmo discreto da chave agregada $\tilde{X}$.

O algoritmo $\mathbf{B}$ então retorna $(i_0, L, \mathbf{a}, \tilde{x})$.

Calculando a probabilidade de aceitação de $\mathbf{B}$, é obtido o seguinte lema:

**Lema 3 (Probabilidade de aceitação mínima de B)** *Assuma a existência de um $(t, q_S, q_H, N, \varepsilon)$-falsificador $\boldsymbol{F}$ no modelo de oráculo randômico contra o esquema de multi-assinatura **MuSig** com parâmetros de grupo $(\mathbb{G}, p, g)$ e seja $q = q_h + q_s + 1$. Então, existe um algoritmo $\boldsymbol{B}$ que toma como entrada um elemento do grupo uniformemente aleatório $X^*$ e strings uniformemente aleatórias de $l$-bits $h_{0,1}, \cdots, h_{0,q}$ e com probabilidade de aceitação (assim como definido em 1) de no mínimo*

$$\frac{\varepsilon^2}{q_h + q_s + 1} - \frac{8q_s(q_h + Nq_s)}{2^k} - \frac{8(q_h + Nq_s)^2 + 1}{2^l},$$

*saídas* $(i_0, L, \boldsymbol{a}, \tilde{x})$ *onde* $i_0 \in 1, \cdots, q$, $L = \{X_1, \cdots, X_n\}$ *é um multiset de chaves públicas tal que* $X^* \in L$, $\boldsymbol{a} = (a_1, \cdots, a_n)$ *é uma tupla de valores com l-bits tal que* $a_i = h_{0,i_0}$ *para qualquer i tal que* $X_i = X^*$ *e* $\tilde{x}$ *é o logaritmo discreto de* $\tilde{X} = \prod_{i=1}^{n} X_i^{a_i}$ *na base g.*

**Algoritmo C**: é construído a partir do algoritmo **B** e executa $\mathbf{Fork}^{\mathbf{B}}$, essencialmente realizando a bifurcação de **B** com relação a resposta da consulta a $H_{agg}$ (mais especificamente, $H_{agg}(L, X^*)$).

O algoritmo **C** toma como entrada a chave pública do signatário honesto $X^*$. A partir desta entrada, **C** toma moedas aleatórias $\rho_B$ e executa o algoritmo **B** com entradas $\rho_B$, $X^*$ e strings de $l$-bits uniformemente aleatórias $h_{0,1}, \cdots, h_{0,q}$.

Se **B** retorna $\perp$ então **C** retorna $\perp$. Caso contrário, **B** retorna $(i_0, L, \mathbf{a}, \tilde{x})$ e **C** executa novamente **B** nas entradas $\rho_B$, $X^*$ (as mesmas da primeira execução) e

$$h_{0,1}, \cdots, h_{0,i_0-1}, h'_{0,i_0}, \cdots, h'_{0,q},$$

onde $h'_{0,i_0}, \cdots, h'_{0,q}$ são tomadas de forma uniformemente aleatória.

De forma semelhante, se a segunda execução de **B** retorna $\perp$ então **C** retorna $\perp$. Caso contrário, a segunda execução de **B** retorna $(i'_0, L', \mathbf{a}', \tilde{x}')$.

Sejam $L = \{X_1, \cdots, X_n\}$, $\mathbf{a} = (a_1, \cdots, a_n)$, $L' = \{X_1, \cdots, X'_n\}$, $\mathbf{a'} = (a'_1, \cdots, a'_n)$ e $n^*$ o número de instâncias de $X^*$ em $L$. Se $i_0 \neq i'_0$ ou $i_0 = i'_0$ e $h_{0,i_0} = h'_{0,i_0}$, **C** retorna $\perp$.

Se $i_0 = i'_0$ e $h_{0,i_0} \neq h'_{0,i_0}$, então $L = L'$ e $a_i = a'_i$, para cada $i$ tal que $X_i \neq X^*$. Como as quatro execuções de **A** (duas execuções produzidas pela primeira execução de **B** e outras duas na segunda execução de **B**) são idênticas até as atribuições de $T_{agg}(L, X^*) := h_{0,i_0}$ e $T_{agg}(L', X^*)' := h'_{0,i_0}$, logo, o parâmetro dessas atribuições são os mesmos, logo $L = L'$ . E como todos os valores de de $T_{agg}(L, X)$, com $X \neq X^*$, são escolhidos de forma uniformemente aleatória para as mesmas moedas $\rho_A$ em todas as execuções de **A**, $a_i = a'_i$ para cada $i$ tal que $X_i \neq X^*$.[2]

Pelo resultado do lema 3:

$$g^{\tilde{x}} = \prod_{i=1}^{n} X_i^{a_i} = (X^*)^{n^* h_{0,i_0}} \prod_{i \in \{1,\cdots,n\}, X_i \neq X^*}^{n} X_i^{a_i}, \tag{7.14}$$

$$g^{\tilde{x}'} = \prod_{i=1}^{n} X_i^{a'_i} = (X^*)^{n^* h'_{0,i_0}} \prod_{i \in \{1,\cdots,n\}, X_i \neq X^*}^{n} X_i^{a_i} \tag{7.15}$$

Dividindo a primeira equação pela segunda:

$$\frac{g^{\tilde{x}}}{g^{\tilde{x}'}} = x\frac{(X^*)^{n^*h_{0,i_0}}\prod_{i\in\{1,\cdots,n\},X_i\neq X^*}^{n} X_i^{a_i}}{(X^*)^{n^*h'_{0,i_0}}\prod_{i\in\{1,\cdots,n\},X_i\neq X^*}^{n} X_i^{a_i}} \tag{7.16}$$

$$\implies g^{\tilde{x}-\tilde{x}'} = \frac{(X^*)^{n^*h_{0,i_0}}}{(X^*)^{n^*h'_{0,i_0}}} \tag{7.17}$$

$$\implies g^{\tilde{x}-\tilde{x}'} = \frac{(g^{x^*})^{n^*h_{0,i_0}}}{(g^{x^*})^{n^*h'_{0,i_0}}} \tag{7.18}$$

$$\implies g^{\tilde{x}-\tilde{x}'} = (g)^{((x^*)n^*h_{0,i_0})-((x^*)n^*h'_{0,i_0})} \tag{7.19}$$

$$\implies \tilde{x}-\tilde{x}' = ((x^*n^*(h_{0,i_0}-h'_{0,i_0}) \bmod p \tag{7.20}$$

$$\implies x^* = (\tilde{x}-\tilde{x}')(n^*)^{-1}(h_{0,i_0}-h'_{0,i_0})^{-1} \bmod p \tag{7.21}$$

Dessa forma, o logaritmo discreto $x^*$ de $X^*$ é obtido e retornado por **C**.

Analisando o algoritmo **C** e calculando a sua probabilidade de aceitação, é obtido o seguinte teorema:

**Teorema 2 (Segurança do esquema MuSig)** *Assuma a existência de um $(t, q_S, q_H, N, \varepsilon)$-falsificador $\boldsymbol{F}$ contra o esquema de multi-assinatura **MuSig** com parâmetros de grupo $(\mathbb{G}, p, g)$, onde $p$ tem $k$-bits de comprimento, e as funções de hash $H_{com}, H_{agg}, H_{sig} : \{0,1\}^* \to \{0,1\}^l$ modelados como oráculos randômicos. Então, existe um algoritmo $\boldsymbol{C}$ que $(t', \varepsilon')$-resolve o problema do logaritmo discreto para $(\mathbb{G}, p, g)$, com $t' = 4t + 4Nt_{exp} + O(N(q_h + q_s + 1))$ onde $t_{exp}$ é o tempo de uma exponenciação em $\mathbb{G}$ e*

$$\varepsilon' \geq \frac{\varepsilon^4}{(q_h+q_s+1)^3} - \frac{16q_s(q_h+Nq_s)}{2^k} - \frac{16(q_h+Nq_s)^2+3}{2^l}$$

Assim, é obtida a relação entre os parâmetros do **MuSig** e a probabilidade mínima que um adversário possui de recuperar a chave privada de um signatário honesto a partir das assinaturas geradas junto a ele, e dessa forma, produzir assinaturas sem a participação deste signatário.

## 7.7 Vantagens comparativas do esquema MuSig

Como evidenciado ao longo dessa seção, o esquema **MuSig** permite a agregação de chaves, sendo o primeiro esquema de multi-assinatura com segurança comprovável no *plain public-key model* com essa propriedade[2].

A agregação de chaves permite que uma assinatura conjunta $\sigma = (R, s)$ seja verificada utilizando somente chave pública agregada $\tilde{X}$ correspondente, de forma semelhante a verificação de uma assinatura de Schnorr simples (substituindo a chave pública do signatário

pela chave pública agregada e o desafio $c = H_{sig}(X, R, m)$ por $c = H_{sig}(\tilde{X}, R, m)$). Logo, diferentemente do esquema Bellare-Neven, não é necessário que a lista contendo as chaves públicas dos signatários $L = \{X_1, \cdots, X_n\}$ seja utilizada na fase de verificação. Dessa forma, são obtidos ganhos em relação a banda utilizada durante a transmissão, privacidade para os signatários (realizando apenas a transmissão da chave pública agregada sem revelar quais são as chaves públicas utilizadas na sua produção, sendo uma chave pública agregada não diferenciável de uma chave pública individual) e também no custo de validação das multi-assinaturas.

Como o **MuSig** é funcional no *plain public-key model*, não é necessário que haja uma etapa prévia inicial que garanta o conhecimento da chave privada relativa a cada chave pública em $L$, etapa que acaba atrelando o inicio de um processo de assinatura às suas entradas previamente analisadas. Logo, o **MuSig** permite que multi-assinaturas sejam feitas em múltiplas possíveis entradas de um processo de assinatura onde a escolha dos signatários não pode ser comprometida de forma prévia. Aumentando assim, o número de situações nas quais multi-assinaturas podem ser utilizadas.[2]

# Capítulo 8

# Árvore de Merkle

## 8.1 Introdução

Até agora o esquema **MuSig** foi descrito utilizando o cenário $n$-of-$n$, onde dado um multiset $L = \{X_1, \cdots, X_n\}$, tal que $X_i$, com $i \in \{1, \cdots, n\}$, é um chave pública, a chave agregada é formada utilizando todos as chaves públicas em $L$. Porém, ao ser considerado o cenário $m$-of-$n$ (com $2 \leq m \leq n$) onde são utilizadas apenas $m$ chaves específicas do multiset $L$ para gerar a chave pública agregada ( consequentemente realizando o processo de assinatura com apenas $m$ signatários específicos), é necessário garantir que a chave pública agregada gerada é de fato uma combinação válida dentre todas as combinações possíveis das chaves públicas em $L$. Será visto mais a diante que uma forma eficiente de obter essa garantia é através das **árvores de Merkle**.

O conceito de árvores de Merkle (também conhecidas por árvores de hash) foi publicado em 1987 por Ralph C. Merkle no artigo "*A Digital Signature Based on a Conventional Encryption Function*". As árvores de Merkle são utilizadas para (mas não somente) garantir a integridade dos dados em sistemas descentralizados e redes P2P. No Bitcoin, por exemplo, as árvores de Merkle são utilizadas para garantir que os blocos de transações não sofreram danos nem adulterações durante a transmissão dos mesmos.

## 8.2 Árvore de Merkle

Uma árvore de Merkle é uma árvore binária formada inteiramente por valores de saída de um hash criptográfico, tal que cada folha da árvore é rotulada com o hash de um bloco de dados correspondente e cada nó que não seja uma folha é rotulado com o hash de seus nós filhos. É possível perceber que qualquer alteração em algum dos blocos de dados, produz um valor diferente do valor do hash esperado de um bloco íntegro e, devido a sucessão de cálculos de hash que se seguem a partir desse primeiro hash, todos os níveis superiores

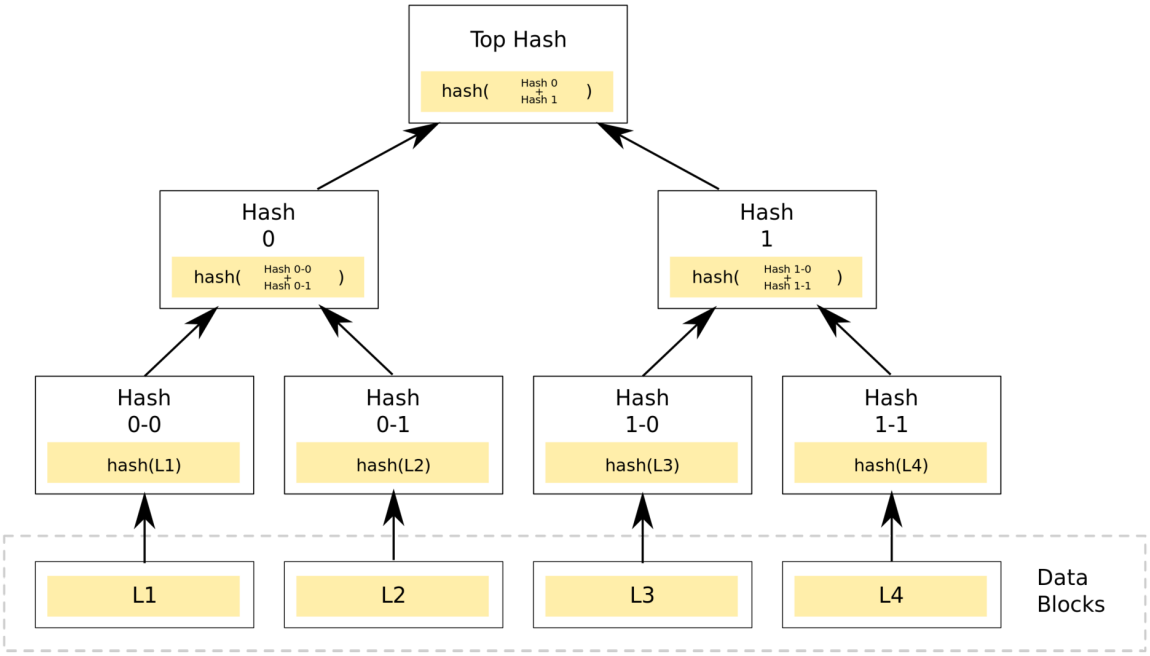


Figura 8.1: Exemplo de uma árvore de Merkle formada a partir dos blocos de dados L1, L2, L3 e L4

seriam afetados e produziriam um valor diferente, consequentemente uma raiz diferente e assim uma árvore de Merkle diferente.

Para realizar a verificação de um bloco de dados, o par não confiável da comunicação deve produzir uma prova a partir de uma árvore de Merkle construída por ele utilizando os blocos de dados que ele deseja enviar, sendo esta prova um caminho na árvore de hash que vai desde o hash do bloco que se deseja verificar até a raiz desta árvore. Esta prova deverá conter todos os filhos necessários para calcular o hash de nós pai presentes no caminho construído. Já o verificador deve ter consigo apenas a raiz de uma árvore de Merkle relativa aos blocos de dados esperados sem adulterações(deve-se garantir que essa raiz é válida para os parâmetros de construção da árvore desejados e que ela não sofreu adulterações durante a sua transmissão). Iniciando o processo de verificação, após o verificador receber o bloco de dados e sua respectiva prova, o verificador monta uma árvore parcial que contém apenas os nós filhos fornecidos na prova até ser possível calcular a raiz dessa árvore. O valor dessa raiz é comparado com o valor da raiz que o verificador obteve com integridade. Se elas forem iguais, então o bloco é válido, caso contrário o bloco sofreu alguma modificação.

Tomando a figura 8.1 como exemplo. Suponha que se deseja enviar o bloco L4 para o respectivo par da comunicação. Para isso, a árvore de Merkle é construída utilizando os

blocos L1, L2, L3 e L4 e então é produzida uma prova para o bloco L4. Esta prova contêm o bloco **L4**, o **Hash 1-0** e o **Hash 0**, uma vez que o caminho de **L4** até a raiz passa pelo **Hash 1-1**, **Hash 1** e finalmente chega-se a raiz **Top Hash**. O par da comunicação que receberá o bloco L4, tem consigo uma cópia íntegra de **Top Hash** (calculada a partir de L1 até L4 íntegros e obtida de um terceiro confiável). Após o par receber o bloco L4 junto a sua respectiva prova, o verificador calcula o hash de **L4** e utilizando **Hash 1-0** da prova, calcula **Hash 1** e em posse desse hash, utiliza **Hash 0** da prova para obter **Top Hash***, o qual será comparado com **Top Hash** e caso sejam iguais, o bloco **L4** é considerado válido, caso contrário ele é descartado.

Manter a estrutura de uma árvore binária cheia é desejável para aumentar a eficiência do percorrimento da árvore. Dessa forma, para $n$ entradas, onde $n$ não é uma potência de dois, a última entrada é repetida $k$ vezes, tal que $n+k$ é a potência de dois mais próxima de $n$.

Ao ser mantida a estrutura de uma árvore binária cheia, é possível representar essa árvore por uma lista, onde o elemento do índice 0 é a raiz da árvore, para cada nó de índice $i$ que não seja uma folha, seus respectivos nós filhos estão localizados nos índices $2i+1$ e $2i+2$ e para cada nó de índice $j$ que não seja a raiz, seu nó pai está no índice $\left\lfloor \frac{j-1}{2} \right\rfloor$. Logo, é possível calcular de forma eficiente uma prova para um determinado bloco de dados encontrando o índice desse bloco na lista (folhas estão sempre no final da lista), percorrendo a árvore em direção a raiz, identificando os nós pais que fazem parte deste caminho e, a partir desses nós, seus respectivos filhos são identificados e adicionados a prova.

## 8.3 Árvore de Merkle aplicada a verificação de chaves agregadas

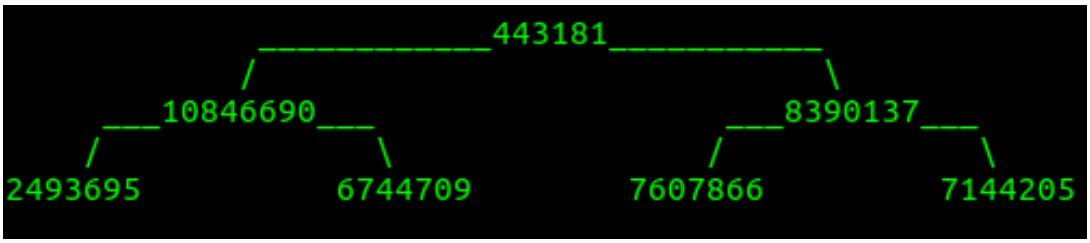


Figura 8.2: Árvore de Merkle construída com todas as possíveis combinações de $L = \{X_1, X_2, X_3\}$. São exibidas apenas uma parte inicial do valores de hash para facilitar a visualização.

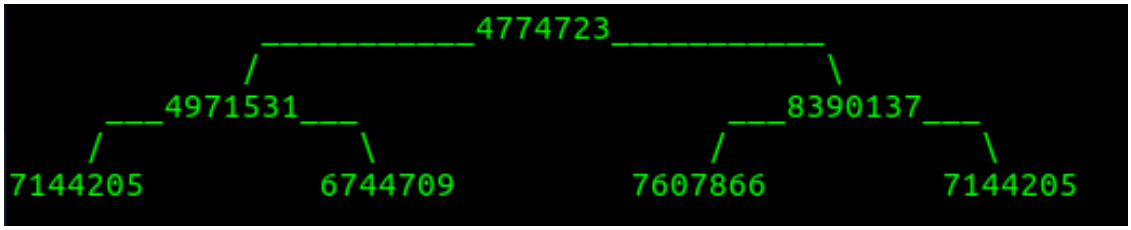


Figura 8.3: Árvore de Merkle construída com todas as possíveis combinações de $L = \{X_1, X_2, X_3\}$ e uma restrição na combinação de $X_2$ com $X_3$

Substituindo os blocos de dados da descrição anterior por combinações válidas das chaves em $L = \{X_1, \cdots, X_n\}$, uma árvore de Merkle pode ser utilizada para verificar se uma chave pública agregada $\tilde{X}$ é uma combinação válida de $m$ $(2 \leq m \leq n)$ chaves de $L$.

Por exemplo, seja $L = \{X_1, X_2, X_3\}$ e $a_i = H_{agg}(L, X_i)$, as possíveis chaves agregadas são: $\tilde{X}_1 = X_1^{a_1} \cdot X_2^{a_2}$, $\tilde{X}_2 = X_1^{a_1} \cdot X_3^{a_3}$, $\tilde{X}_3 = X_2^{a_2} \cdot X_3^{a_3}$, $\tilde{X}_4 = X_1^{a_1} \cdot X_2^{a_2} \cdot X_3^{a_3}$. Logo, as folhas da árvore de Merkle a ser construida são $h_{0,1} = H_{tree}(\tilde{X}_1)$, $h_{0,2} = H_{tree}(\tilde{X}_2)$, $h_{0,3} = H_{tree}(\tilde{X}_3)$, $h_{0,4} = H_{tree}(\tilde{X}_4)$, sendo $H_{tree}$ a função de hash utilizado no cálculo de hash da arvore de Merkle. O nível superior é calculado por $h_{1,0} = H_{tree}(h_{0,1}, h_{0,2})$, $h_{1,1} = H_{tree}(h_{0,3}, h_{0,4})$ e, por fim, a raiz da árvore é calculada por $h_{2,0} = H_{tree}(h_{0,1}, h_{0,2})$. A produção da prova e a verificação da mesma ocorre da mesma forma descrita anteriormente.

Aplicando ao contexto geral do **MuSig** no cenário $m$-de-$n$, após uma assinatura $\sigma = (R, s)$ relativa a $\tilde{X}$ e $L$ ser gerada, é necessário montar uma árvore de Merkle a partir de $L$ (aplicando as restrições necessárias, caso existam) e produzir uma prova $\mathbf{P}$ para $\tilde{X}$ a partir dessa árvore com raiz **TR***. Assim, o signatário deverá enviar para o verificador o conjunto $(\sigma, \tilde{X}, \mathbf{P})$. O verificador, por sua vez, deverá obter a raiz integra **TR** (obtido através de um terceiro confiável ou construído pelo próprio verificador a partir de $L$ com as restrições necessárias caso existam) e realizar um passo de verificação de chaves antes de verificar a assinatura $\sigma$. Neste processo de verificação será calculada a raiz **TR*** a partir da prova $\mathbf{P}$ e $\tilde{X}$, se **TR*** = **TR**, então $\tilde{X}$ é uma combinação válida de $L$ e o verificador prossegue com a análise da assinatura $\sigma$, caso contrário o processo é abortado produzindo um erro indicando que a chave agregada apresentada não é válida (e consequentemente $\sigma$ não será verificado).

Dessa forma, o total de combinações $\mathbf{C}$ permitidas para um conjunto $L = \{X_1, \cdots, X_n\}$ é:

$$\mathbf{C}(L, \mathbf{rest}) = \left( \sum_{i=2}^{n} \frac{n!}{i! \cdot (n-i)!} \right) - |\mathbf{rest}| \ ,$$

onde **rest** é o conjunto contendo combinações proibidas de chaves em $L$ e $|\mathbf{rest}|$ indica a quantidade de elementos em **rest**.

De forma semelhante a seção anterior, se o total de combinações $\mathbf{C}(L, \mathbf{rest})$ não for uma potência de dois, a última combinação válida é repetida $k$ vezes até que $\mathbf{C}(L, \mathbf{rest}) + k = 2^j$, sendo $j$ o menor inteiro positivo não nulo que satisfaz a equação, sendo assim mantida a estrutura de uma árvore binária cheia.

Analisando as figuras 8.2 e 8.3, é possível observar que, sendo adicionada uma restrição de combinação e montada a árvore de hash, a raiz das duas árvores resultantes são claramente distintas. É possível notar também que ao ser adicionada a restrição no exemplo 8.3, existem apenas **três** combinações de chaves permitidas, logo não seria possível manter a estrutura da árvore binária cheia com apenas esses valores. A solução para essa situação é a de, durante a construção da árvore, ao ser notado que uma combinação proibida seria calculada e adicionada a árvore, seu valor é substituído pelo resultado do hash da combinação válida anterior (caso a combinação proibida seja a primeira, seu valor é substituído pelo hash da última combinação válida).

# Capítulo 9

# Aplicações ao Bitcoin

## 9.1 Introdução

O Bitcoin [10] é uma moeda digital descentralizada que utiliza da tecnologia do *blockchain* para montar um livro razão contento todas as transações realizadas até então na rede *peer-to-peer* do Bitcoin, transações estas que podem ser visualizadas e verificadas publicamente.

O *blockchain* é uma lista crescente de blocos (registros) encadeados, isto é, cada novo bloco adicionado ao *blockchain* contém o hash do bloco anterior, que por sua vez contém o hash do bloco anterior a ele e assim por diante. Dessa forma, cada novo bloco está atrelado a todos os outros blocos precedidos por ele. O *blockchain* fornece o caráter de imutabilidade das transações realizadas na rede Bitcoin, uma vez que uma alteração em um bloco intermediário do *blockchain* (de forma a falsificar uma transação já ocorrida por exemplo) resultaria em uma inconsistência em todo o restante do *blockchain* até então, sendo assim, facilmente identificada.

O estado final do *blockchain* é o conjunto de moedas não gastas (conjunto UTXO (*Unspent Transcation Output*)), onde cada moeda não gasta tem um valor associado (expressado como o um múltiplo de $10^{-8}$ bitcoin) associado a uma chave pública programável do dono dessa moeda. Cada transação consome um ou mais moedas de um dos pares dessa transação (mediante uma assinatura respectiva que a autorize) e cria mais uma ou mais moedas para o(s) destinatário(s), moedas cujo valor total não deve exceder o valor de moedas consumidas.[2]

Implantada em 2009, o Bitcoin foi a primeira criptomoeda descentralizada criada e é a que apresenta o maior uso atualmente. Seu desenvolvimento e sucesso propulsionou a criação de diversas outras criptomoedas (*altcoins*) que procuravam melhorar pontos específicos do Bitcoin e adicionar funcionalidades. Como exemplo: o **Monero** apresenta um foco em privacidade e irrastreabilidade; o **Litecoin** oferece transações mais rápidas e utiliza uma função de hash mais adequada aos seus propósitos; e o **Ethereum** utiliza a

tecnologia do *blockchain* não somente para transações, mas também para criar também uma plataforma onde é possível desenvolver e executar aplicações descentralizadas (*smart contracts*).

## 9.2 Assinatura de Schnorr no Bitcoin

Tradicionalmente o Bitcoin utiliza assinaturas do esquema **ECDSA** (Elliptic Curve Digital Signature Algorithm) sobre a curva **secp256k1** para autenticar as sua transações. Comparativamente, a esquema de assinatura de Schnorr apresenta diversas vantagens em relação ao ECDSA:[3]

- **Prova de Segurança**: a assinatura de Schnorr tem segurança comprovável no modelo de oráculo randômico (assumindo a dureza do problema do logaritmo discreto), enquanto tal prova não existe para o ECDSA.
- **Imaleabilidade**: as assinaturas ECDSA estão sujeitas a ataques de maleabilidade, nos quais é possível produzir uma assinatura válida para uma determinada mensagem e chave pública a partir de uma assinatura válida já existente para os mesmos parâmetros sem a utilização de sua respectiva chave privada. As assinaturas de Schnorr por sua vez, não são maleáveis.
- **Linearidade**: como visto nas seções anteriores, é possível utilizar a assinatura de Schnorr em esquemas de multi-assinatura onde a assinatura conjunta resultante tem o mesmo tamanho de uma assinatura simples. Logo, o tamanho da assinatura não muda em função do número de signatários que participam do protocolo, ou seja, a assinatura de Schnorr apresenta um comportamento linear em relação ao tamanho da assinatura resultante.

Utilizando assinaturas de Schnorr, também é possível realizar verificações de assinaturas em lote, uma vez que, para um lote de $n$ assinaturas $(R_1, s_1), \cdots, (R_n, s_n)$, suas respectivas chaves públicas $X_1, \cdots, X_n$ e compromissos $c_1, \cdots, c_n$, tal que $\forall i \in \{1, \cdots, n\}, c = H(X_i, R_i, m_i)$, sendo $H$ uma função de hash, $m_i$ a respectiva mensagem do signatário $X_i$ e $P$ o ponto base da curva elíptica utilizada:

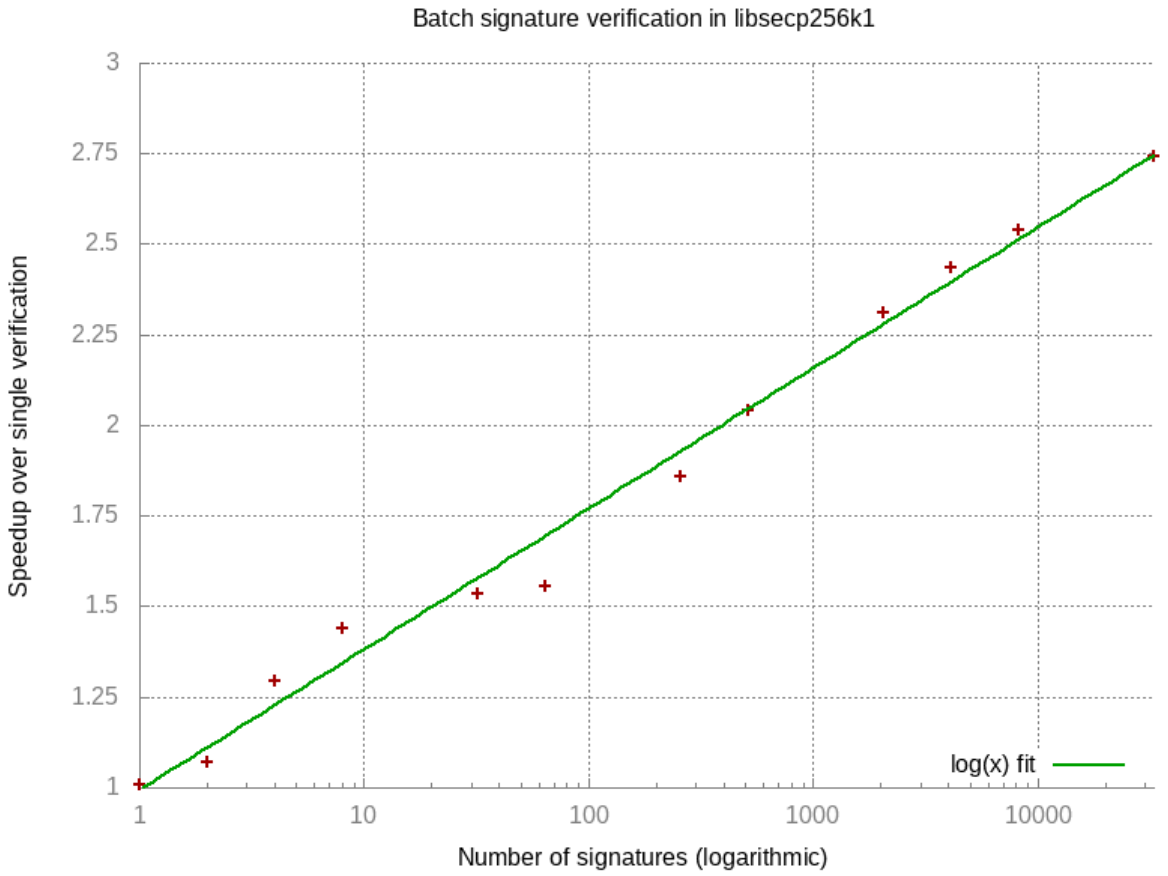


Figura 9.1: Relação entre o tempo levado para verificar $n$ assinaturas individualmente e o tempo levador para verificar $n$ assinaturas em lote. O tempo total para verificar $n$ assinaturas cresce com complexidade $O(n/\log n)$. Retirado de [3].

$$
\begin{aligned}
&(s_1 + s_2 + \cdots + s_n)P \\
=&[(r_1 + c_1x_1) + (r_2 + c_2x_2) + \cdots + (r_n + c_nx_n)]P \\
=&(r_1 + c_1x_1)P + (r_2 + c_2x_2)P + \cdots + (r_n + c_nx_n)P \\
=&(r_1P + c_1x_1P) + (r_2P + c_2x_2P) + \cdots + (r_nP + c_nx_nP) \\
=&(R_1 + c_1X_1) + (R_2 + c_2X_2) + \cdots + (R_n + c_nX_n) \\
=&(R_1 + R_2 + \cdots + R_n) + (c_1X_1 + c_2X_2 + \cdots + c_nX_n)
\end{aligned}
$$

$$
\implies (\sum_{i=1}^{n} s_i)P = \sum_{i=1}^{n} R_i + \sum_{i=1}^{n} c_iX_i \tag{9.1}
$$

Logo, por meio de 9.1, um lote de $n$ assinaturas pode ser verificado de forma mais

eficiente quando comparado a verificação de cada assinatura individualmente, conforme evidenciado na figura 9.1.

Outro ponto relevante é a possibilidade de utilizar a curva *secp256k1* em assinaturas de Schnorr junto às mesmas chaves públicas e privadas utilizadas atualmente no ECDSA. Tal situação é um grande facilitador de uma possível transição do ECDSA para o esquema de assinatura de Schnorr.

## 9.3 Aplicações do MuSig no Bitcoin

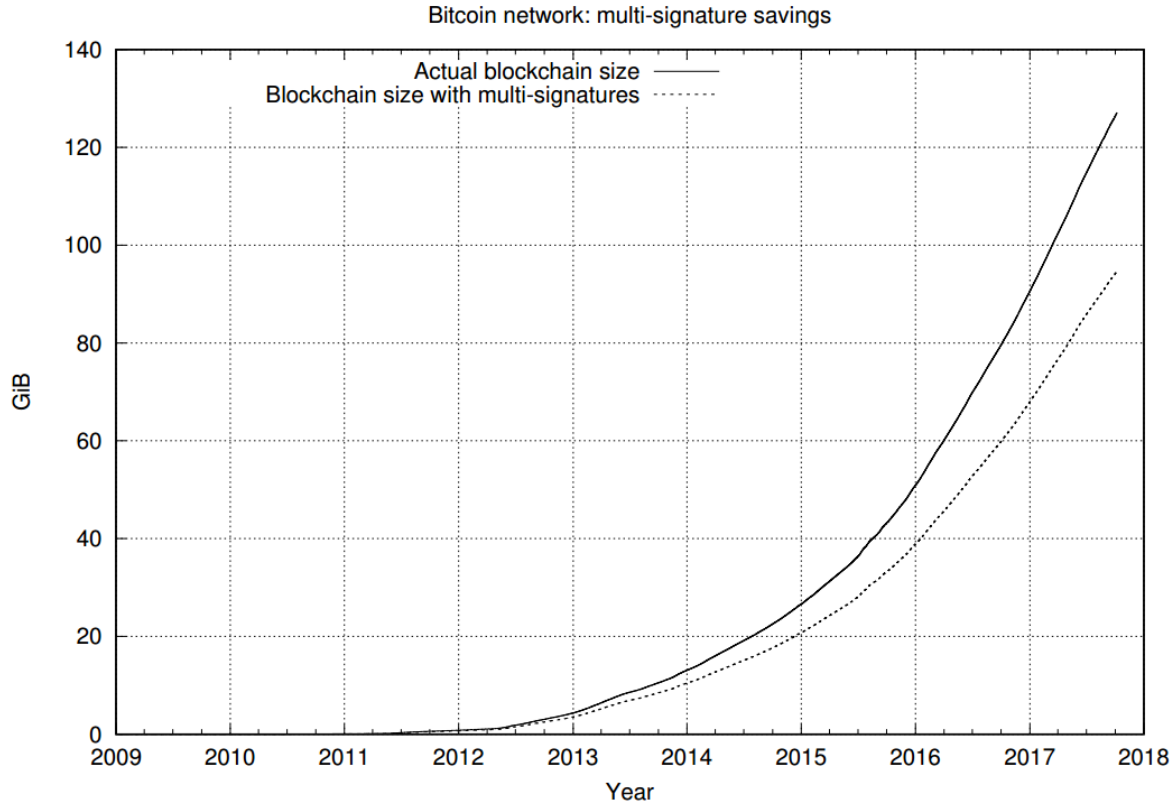


Figura 9.2: Comparativo entre o tamanho atual do *blockchain* do Bitcoin e o tamanho estimado do *blockchain* utilizando multi-assinaturas (não é contabilizado a economia que seria gerada com o uso de agregação de chaves). Retirado de [2].

No contexto do Bitcoin, uma "multi-assinatura"se refere a qualquer política de gasto de moedas que necessitam assinaturas de $m$-de-$n$ chaves públicas. Até então, as multi-assinaturas no Bitcoin são implementadas por meio de uma política de limite (*threshold*), onde a possibilidade de se gastar um conjunto de moedas, neste caso, é obtida mediante múltiplas assinaturas individuais verificáveis, cada uma, por uma chave pública respectiva. Apesar dessa aplicação *naive* de multi-assinaturas ser flexível (implementação sim-

ples do modelo $m$-de-$n$), ela é ineficiente em termos de tamanho, custo computacional e privacidade[2].

Utilizando o MuSig para multi-assinaturas, o custo de processamento e armazenamento são reduzidos (evidenciado na figura 9.2), uma vez que, a autorização para se gastar um conjunto de moedas pode ser obtida mediante verificação de uma única multi-assinatura com relação a uma única chave pública agregada, produzida a partir de $m$-de-$n$ chaves públicas do grupo de signatários respectivo.

A utilização de chaves públicas agregadas também representa uma melhora na privacidade das multi-assinaturas, visto que as chaves públicas dos participantes e sua contagem [1] permanecem acessíveis somente aos signatários.

Como visto anteriormente, a verificação da validade de uma chave agregada no cenário $m$-de-$n$ pode ser realizada de forma eficiente por meio de árvores de Merkle quando o total de possibilidades de combinações de chaves é tratável. Esta solução ainda permite o uso de chaves agregadas com uma boa privacidade, uma vez que a política exata de produção da chave agregada não é visível a partir da prova que a acompanha. Como somente a raiz da árvore é necessária para verificar a validade da prova, é possível manter em sigilo informações adicionais sobre a estrutura dessa árvore e, consequentemente, informações sobre as políticas utilizadas.

[1] Número de instâncias de uma chave pública $X$ em $L = \{X_1, \cdots, X_n\}$, tal que $X \in L$.

# Capítulo 10

# Resultados

## 10.1 Introdução

Como parte deste trabalho foi construído uma implementação do esquema **MuSig** no cenário $m$-de-$n$. Dessa forma, procurou-se desenvolver implementações prova de conceito dos temas aqui discutidos utilizando a linguagem de programação ***Python*** e curvas elípticas, em particular a curva ***secp256k1***. Nesta seção, estas implementações serão descritas de forma aprofundada.

## 10.2 Principais bibliotecas

A biblioteca ***fastecdsa*** (disponível em [11]) desenvolvida por Anton Kueltz, tem como objetivo a implementação eficiente de utilidades de criptografia de curva elíptica. Neste trabalho são utilizados a implementação da curva ***secp256k1***, consequentemente as implementações das operações da curva e do seu corpo subjacente e as implementações de importação e exportação de chaves privadas/públicas para/de arquivos ***pem*** de acordo com RFC5480 e RFC5915.

Como os parâmetros criptográficos de segurança dos esquemas exigem a utilização de números (e sua aritmética) que ultrapassam a representação padrão de 64-bits (chaves privadas obtidas utilizando a curva secp256k1, por exemplo, têm 256 bits de comprimento), foi utilizada a biblioteca ***gmpy2*** (disponível em [12]) que dá suporte a aritmética precisão múltipla. A biblioteca ***gmpy2*** é um módulo de extensão do Python escrito em **C** que dá suporte a utilização (mas não somente) das bibliotecas **GMP** (*GNU Multiple Precision Arithmetic Library*) e **MPIR** (*Multiple Precision Integers and Rationals*), originalmente desenvolvidas para o **C**.

Outras importantes bibliotecas que fazem parte da implementação são:

- ***hashlib***: biblioteca que contém implementações de funções de hash criptográficos, entre elas SHA-2, SHA-3 e BLAKE, com destaque ao **SHA256**, função de hash utilizada no Bitcoin e nesta implementação.
- ***threading***: esta biblioteca fornece as ferramentas necessárias para implementar paralelismo baseado em *threads*, o qual é utilizado para melhorar a eficiência das comunicações da rede P2P de signatários e dos cálculos de hash quando estes são paralelizáveis.
- ***itertools***: este módulo apresenta ferramentas para a construção de iteradores inspirados por construções provindas do *APL*, *Haskell* e *SML*. Entre as funções disponíveis, o principal destaque para a implementação é a função **combinations**(), que permite a obtenção de todas as possíveis combinações de elementos de uma lista.
- ***tkinter***: biblioteca que apresenta diversas ferramentas para a construção de interfaces gráficas e sobre a qual foi construída a interface de usuário da implementação.
- ***socket***: biblioteca que provê acesso a interface de *socket* do BSD com abstrações adequadas ao Python e utilizada como base do módulo de rede da implementação.

## 10.3 *Schnorr Signature*

O esquema de assinatura (individual) de Schnorr implementado (**simple_schnorr.py**), tem como objetivo produzir e verificar uma assinatura de Schnorr baseado no esquema apresentado em 4.4.

O processo de **assinatura** se inicia tomando como entrada o par de chaves privada/público $(x, X)$, a mensagem $m$, uma curva elíptica $ec$ (por padrão, a curva **secp256k1** da biblioteca **fastecdsa**) e uma função de hash (por padrão, a função de hash **SHA256**). Sorteia-se então um $r$ randômico diferente de zero e dentro da ordem da curva (representado por $ec.q$, o atributo $q$ da curva $ec$) utilizando a função **mpz_random()** da biblioteca **gmpy2**. O ponto $R$ é então calculado multiplicando $r$ pelo ponto base $G$ da curva $ec$. Essa ação é escrita como uma multiplicação padrão $R = r * ec.G$, uma vez que a biblioteca **gmpy2** já proporciona um *overload* da função de multiplicação quando estão envolvidos um inteiro e um ponto da curva $ec$. Após isso, é calculado $c \leftarrow H(X, R, m)$ convertendo as coordenadas $x$ e $y$ dos pontos $X$ e $R$ para string, concatenando as strings das coordenadas junto a mensagem $m$ e fornecendo o resultado dessa concatenação como entrada da função de hash. O resultado da função de hash é então convertido de hexadecimal para um inteiro que é utilizado para o cálculo da assinatura parcial $s = r + c * x \ mod \ ec.q$. Assim, é obtida a assinatura $(R, s)$ para a mensagem $m$ e o par de chaves $(x, X)$.

A verificação se inicia com os parâmetros de entrada $X$ (a chave pública), $m$ (mensagem), a assinatura $(R, s)$ , a curva $ec$ e uma função de hash. Prosseguindo, $c \leftarrow H(X, R, m)$ é calculado e convertido da mesma forma descrita no processo de assinatura. São então calculados os dois lados da equação $sG \stackrel{?}{=} R + cX$, tal que $left = s * ec.G$ e $right = R + c * X$. Os pontos $left$ e $right$ são comparados, e caso ambas as suas coordenadas $x$ e $y$ sejam iguais, a assinatura $(R, s)$ é valida para $X$ e $m$. Caso contrário, $(R, s)$ é considerada inválida para $X$ e $m$.

## 10.4 *Naive Schnorr Multi-signature*

A implementação do esquema *Naive Schnorr* (**naive_schnorr_multisig.py**) teve como base a descrição do esquema apresentada em 5.2 e procura simular um protocolo de assinatura baseado nesse esquema. Nesta simulação o usuário desempenha localmente o papel de todos os signatários envolvidos no processo de assinatura, tendo em sua posse todas as chaves públicas e privadas envolvidas neste processo.

Para iniciar a assinatura, é necessário uma lista contendo todas as chaves públicas e privadas que serão utilizadas, a mensagem $m$ que será assinada, uma curva elíptica e uma função de hash. Por padrão são utilizadas a curva **secp256k1** e a função de hash **SHA256**.

Para cada par de chaves na lista (representando um signatário), é obtido um $r_i$ aleatório por meio da função **mpz_random()** (diferente de zero e dentro da ordem da curva) e calculado o seu respectivo ponto $R_i = r * ec.G$, onde $ec.G$ é o ponto base da curva $ec$ utilizada. Cada $r_i$ e $R_i$ são gravados em listas respectivas.

Em seguida, é utilizada a lista de $R_i$'s para calcular $R = \sum_{i=1}^{n} R_i$, tal que $n$ corresponde ao número de signatários (obtida pelo número de elementos na lista de pares de chaves). De forma semelhante, a chave pública agregada $\tilde{X} = \sum_{i=1}^{n} X_i$, é calculada.

Em posse de $R$ e $\tilde{X}$, $c = H(\tilde{X}, R, m)$ é calculado convertendo as coordenadas $x$ e $y$ de $\tilde{X}$ e $R$ para strings, concatenando o resultado à mensagem $m$ e fornecendo este último resultado para a função de hash.

Em seguida, são calculados $s_i = r_i + c * x_i$ para cada signatário e os resultados são gravados em uma lista respectiva. Esta lista é utilizada para o cálculo de $s = \sum_{i=1}^{n} s_i \bmod p$.

O processo de assinatura finaliza retornando a assinatura $(R, s)$ e a chave pública agregada $\tilde{X}$.

Para o processo de verificação é utilizado como entrada uma assinatura $(R, s)$, uma mensagem $m$, uma curva elíptica $ec$ e uma função de hash $H$ (as mesmas do processo de assinatura). Também é necessário a chave pública agregada gerada no processo de assinatura ou uma lista de chaves públicas para que a chave agregada possa ser calculada.

A partir da chave agregada (calculada ou fornecida como entrada) é possível realizar o cálculo de $c = H(\tilde{X}, R, m)$ de forma idêntica àquela realizada na assinatura. É então necessário verificar se $sG \stackrel{?}{=} R + c\tilde{X}$. Para tal, são calculados $left = sG$ e $right = R + c\tilde{X}$. Os pontos $left$ e $right$ calculados são comparados coordenada a coordenada. Caso ambas sejam iguais, $(R, s)$ é uma assinatura válida para $m$ e $X$, caso contrário, a assinatura fornecida é inválida.

## 10.5 *Rogue-key attack*

Esta parte da implementação (**rogue-key_attack.py**) procura simular a principal parte do *Rogue-key attack*, produzindo a chave pública do adversário (aquele que executa o ataque) em função das chaves públicas dos co-signatários honestos e assim forjar a chave pública agregada resultante.

Para efetuar o ataque, é tomado como entrada as chaves públicas dos co-signatários honestos ($\{X_2, \cdots, X_n\}$) e a chave pública do adversário ($X_1$). Então, é calculado a soma das chaves públicas dos co-signatários ($\tilde{X}_{\text{partial}} = \sum_{i=2}^{n} X_i$) e, em seguida, é calculado o inverso (em relação a soma de pontos) do ponto resultante da soma anterior ($-(\tilde{X}_{\text{partial}})$). Devido ao *overload* das operações envolvendo pontos da curva elíptica pela classe de curvas elípticas da biblioteca **fastecdsa**, o inverso é calculado realizando uma operação da seguinte forma pub_keys_sum_inv = −pub_keys_sum.

A partir deste ponto, é possível calcular a *rogue-key* que será fornecida durante o processo de assinatura como se fosse a verdadeira chave pública do adversário. Para tal, a chave pública legítima do adversário (da forma $X = xP$) é somada com o inverso da soma das chaves públicas dos co-signatários ($(X_{\text{rogue\_key}} = X_1 + (-(\tilde{X}_{\text{partial}}))$).

Uma vez em posse da *rogue-key* é possível calcular qual seria a chave pública agregada resultante da soma de todas as chaves públicas dos signatários, realizando a soma da *rogue-key* obtida com a soma das chaves públicas dos co-signatários realizada anteriormente, obtendo assim a chave pública agregada resultante $\tilde{X} = X_{\text{rogue\_key}} + \tilde{X}_{\text{partial}}$.

Para testar o sucesso do ataque, é verificado se $\tilde{X} \stackrel{?}{=} X_1$, que, em caso positivo, indica que a chave pública agregada resultante depende apenas da chave privada do adversário, sendo assim possível que o adversário produza assinaturas pelo grupo sem necessidade de participação dos outros signatários.

## 10.6 Ordenação de pontos da curva elíptica

Devido a necessidade de produzir uma codificação única $\langle L \rangle$ do multiset de chaves públicas $L = \{X_1, \cdots, X_n\}$, foi criado o módulo **pointsort.py**.

Neste módulo, é realizada a ordenação das chaves de $L$ na ordem lexicográfica. Para tal, como as chaves pública são pontos da curva elíptica, é necessário estabelecer uma relação de ordem entre pontos.

Considerando dois pontos $P_1, P_2$ pertencentes a mesma curva elíptica, $P.x, P.y$ as coordenadas $x$ e $y$ de um ponto $P$ respectivamente, a ordenação baseada na ordem lexicográfica se dá por:

1. $P_1 > P_2 \iff (P_1.x > P_2.x) \vee (P_1.x = P_2.x \wedge P_1.y > P_2.y)$
2. $P_1 = P_2 \iff (P_1.x = P_2.x) \wedge (P_1.y = P_2.y)$

Dessa forma, foi construída uma função de comparação de pontos (**compare_points()**), que recebe como entrada dois pontos $P_1$ e $P_2$ e retorna um inteiro correspondente ao resultado da comparação da seguinte forma:

- $P_1 > P_2 \implies$ **return** 0
- $P_2 > P_1 \implies$ **return** 1
- $P_1 = P_2 \implies$ **return** 2

Essa comparação de pontos é então utilizada para montar uma implementação de propósito específico do algoritmo de ordenação ***Merge Sort***.

A função de ordenação implementada (**sort()**), toma como entrada o multiset $L$. As chaves passam a ser ordenadas por meio do *Merge Sort*, estabelecendo uma relação de ordem entre as chaves comparadas por meio da função **compare_points()**. A saída produzida é o multiset $\langle L \rangle$, contendo todas as chaves públicas de $L$ ordenadas lexicograficamente.

## 10.7 Bellare-Neven

Esta parte da implementação (**bellare-neven.py**) tem o objetivo de simular o esquema de multi-assinatura Bellare-Neven (assim como descrito em 6.3), onde o usuário faz localmente o papel de todos os signatários.

Para iniciar o processo de assinatura, é necessário uma mensagem $m$ que será assinada em relação ao multiset de chaves públicas $L = \{X_1, \cdots, X_n\}$. Como o usuário faz o papel de todos os signatários, deverão ser fornecidas as chaves privadas relativas a cada chave pública em $L$.

É então construído um dicionário da forma **"chave_pública":*chave_privada***, para que a partir de uma chave pública seja possível encontrar a sua chave privada respectiva.

O multiset $\langle L \rangle$ deve apresentar uma codificação única de $L$ comum a todos os signatários, para isso, é utilizada a função **sort()** com $L$ como entrada para realizar a sua ordenação lexicográfica e obter a lista $\langle L \rangle$.

Em seguida, deve ser obtido um $r_i$ randômico (utilizando a função **mpz_random()**) para cada signatário (chave pública) e calculado $R_i = r_i G$ (sendo $G$ o ponto base da curva $ec$). Estes valores são salvos em listas com uma ordenação relativa a $\langle L \rangle$, isto é, dada uma chave pública em $\langle L \rangle$ com índice $i$ nesta lista, os respectivos $r_i$ e $R_i$ devem ser encontrados no mesmo índice $i$ em suas respectivas listas. É então realizado o cálculo de $R = \sum_{i=1}^{n} R_i$ somando todos os pontos presentes na lista de pontos R.

Cada $c_i = H(X_i, R, \langle L \rangle, m)$ e $s_i = r_i + c_i \cdot x_i$ são calculados no passo seguinte. Para tal, a lista $\langle L \rangle$ é iterada e para cada entrada $X_i$, as coordenadas dos pontos $X_i$, $R$ e a lista $\langle L \rangle$ são convertidas para strings e concatenadas com a mensagem $m$ para o cálculo de hash de $c_i$. Utilizando o $c_i$ recém calculado e o obtendo os respectivos $r_i$ e $x_i$ a partir de $X_i$ (utilizando o seu índice e consulta ao dicionário respectivamente), é possível calcular o respectivo $s_i$

Os valores de $s_i$ são somados para formar a assinatura parcial $s = \sum_{i=1}^{n} s_i \bmod q$ e assim a assinatura $(R, s)$ é obtida.

A verificação toma como entrada uma assinatura $(R, s)$, uma mensagem $m$, uma lista de chaves públicas $L = \{X_1, \cdots, X_n\}$, uma curva elíptica $ec$ e uma função de hash $H$. Deseja-se verificar se $(R, s)$ é uma assinatura válida para $m$ e $L$ utilizando o esquema de multi-assinatura Bellare-Neven.

Iniciando o processo de verificação, $\langle L \rangle$ é calculado ordenando as chaves públicas pela ordem lexicográfica (por meio da função **sort()** com $L$ como entrada). O multiset $\langle L \rangle$ é então utilizado para o cálculo de cada $c_i = H(X_i, R, \langle L \rangle, m)$.

Os dois lados da equação $sP \stackrel{?}{=} R + \sum_{i=1}^{n} c_i X_i$ são calculados realizando $left = s \cdot ec.G$ e $right = R + \sum_{i=1}^{n} c_i X_i$. Os pontos da curva $left$ e $right$ são então comparados coordenada a coordenada. Caso os pontos sejam iguais, $(R, s)$ é um assinatura válida para $m$ e $L$, caso contrário, a assinatura é considerada inválida.

## 10.8 MuSig $n$-de-$n$ (protótipo)

Nesta parte da implementação (**musig.py**), é implementado uma simulação do esquema **MuSig** (baseada em 7.4) $n$-de-$n$, na qual o usuário faz localmente o papel de todos os signatários que participam do esquema.

O processo de assinatura se inicia com o fornecimento de uma mensagem $m$, uma lista de chaves privadas e públicas $key\_lst = [(x_1, X_1), \cdots, (x_n, X_n)]$, sendo $x_i$ e $X_i$ a chave privada e pública (respectivamente) de um signatário e três funções de hash $H_{com}$,

$H_{agg}$, $H_{sig}$, utilizadas respectivamente na fase de comprometimento, no cálculo da chave agregada e no cálculo da assinatura.

A lista de chaves privada/pública é utilizada para gerar uma lista $L = [X_1, \cdots, X_n]$ das chaves públicas dos signatários e um dicionário da forma "*chave_pública*":*chave_privada*, sendo assim possível acessar as chaves privadas respectivas a cada chave pública.

Em seguida, é utilizado $L$ para computar $\langle L \rangle$ da seguinte forma: $L$ é ordenado de acordo com a ordem lexicográfica das chaves públicas utilizando a função **sort()** gerando $L'$, $\langle L \rangle$ é então calculado realizando o hash de $L'$, sendo assim obtido uma codificação única e comum de $L$ a todos os signatários.

Diferentemente da implementação do esquema Bellare-Neven, optou-se por utilizar $\langle L \rangle$ como o hash do multiset $L$ ordenado e não o próprio $L$ ordenado. Com o esquema **MuSig** é possível verificar a multi-assinatura resultante como uma assinatura simples de Schnorr, utilizando a chave pública agregada. Porém, como o cálculo de $c = H_{sig}(\tilde{X}, R, m)$ pelo verificador envolve $\langle L \rangle$, é necessário enviá-lo para o verificador junto a assinatura. Assim, o tamanho de $L$ ordenado como codificação única, é $n \cdot |X|$, sendo $n$ o número de signatários e $|X|$ o tamanho de uma chave pública utilizada no esquema, enquanto o tamanho de $\langle L \rangle$ como a codificação única obtida pelo hash de $L$ ordenado é o tamanho da saída da função de hash utilizada. Dessa forma, o $\langle L \rangle$ implementado gera uma economia significativa no custo de transmissão de dados para o verificador.

Em seguida é calculado $a_i = H_{agg}(\langle L \rangle, X_i)$, para cada chave pública $X_i$ em $L'$. Cada $a_i$ é gravado em uma lista com ordenação relativa a $L'$, isto é, se uma chave pública se encontra no índice $i$ da lista $L'$, então o seu respectivo $a$ se encontra no índice $i$ de sua respectiva lista.

A chave agregada $\tilde{X} = \sum_{i=1}^{n} a_i \cdot X_i$ é então calculada utilizando os $a_i$'s do passo anterior.

Para cada $X_i$ de $L'$, é obtido um $r_i$ aleatório, que é então utilizado para calcular o ponto $R_i = r_i G$, sendo $G$ o ponto base da curva elíptica *ec*.

São então calculados $t_i = H_{com}(R_i)$, para cada $R_i$. Cada $t_i$ é então verificado para seu respectivo $R_i$. Nesta simulação, este passo é redundante, porém, ele é mantido para preservar o fluxo de execução do protocolo de assinatura.

Os $R_i$'s são então utilizados para calcular $R = \sum_{i=1}^{n} R_i$.

O desafio $c = H_{sig}(\tilde{X}, R, m)$ é calculado convertendo $\tilde{X}$ e $R$ para strings, concatenando-as junto a mensagem $m$ e fornecendo o resultado para a função de hash $H_{sig}$.

Para cada $X_i \in L'$ é calculada a assinatura parcial $s_i = r_i + c \cdot a_i \cdot x_i \bmod q$, tal que $q$ é a ordem do corpo subjacente da curva *ec*. Em seguida, é calculada a assinatura parcial do grupo $s = \sum_{i=1}^{n} s_i$.

Dessa forma, é obtida a assinatura $(R, s)$ para a mensagem $m$ e chaves públicas $L$ produzida a partir do esquema **MuSig**.

São retornados a assinatura $(R, s)$, a chave agregada $\tilde{X}$ e a codificação única $\langle L \rangle$ de $L$.

Para a verificação de uma assinatura, são tomados como entrada a assinatura $(R, s)$, uma mensagem $m$, uma curva elíptica $ec$, duas funções de hash $H_{agg}$ e $H_{sig}$ e uma das duas opções a seguir:

1. um conjunto de chaves públicas $L = \{X_1, \cdots, X_n\}$
2. uma chave pública agregada $\tilde{X}$, referente a um conjunto de chaves públicas $L$, e $\langle L \rangle$, a codificação única de $L$.

No primeiro caso, para verificar a assinatura, ainda é necessário calcular $\langle L \rangle$, cada $a_i$ e a chave pública agregada $\tilde{X}$, de forma semelhante àquela realizada no processo de assinatura. Enquanto no segundo caso, é possível pular esses passos (uma vez que $\tilde{X}$ e $\langle L \rangle$ já são fornecidos como entrada) e prosseguir de forma semelhante a verificação de uma assinatura simples de Schnorr.

Prosseguindo com a verificação, é calculado $c = H_{sig}(\tilde{X}, R, m)$. São então calculados $left = s \cdot G$ e $right = R + c \cdot \tilde{X}$. Os pontos $left$ e $right$ resultantes são comparados coordenada a coordenada a fim de avaliar a equação $sG \stackrel{?}{=} R + \tilde{X}$. Se os dois pontos são iguais, então $(R, s)$ é uma assinatura válida para $m$, $\tilde{X}$ e $L$. Caso contrário, a assinatura é considerada inválida.

## 10.9 Módulo de Rede P2P

O módulo de rede (**p2pnetwork.py**) foi construído para organizar a comunicação *peer-to-peer* dos signatários que participam do processo de assinatura. A comunicação entre os signatários é necessária em três situações, sendo elas o envio respectivos $t_i = H_{com}(R_i)$, $R_i = r_i P$ e $s_i = r_i + c \cdot a_i \cdot x_i \ mod \ p$ dos signatários. Como processo de comunicação é o mesmo para o envio de $t_i$, $R_i$ e $s_i$, durante a descrição eles serão abstraídos como o envio de um **pacote**.

É utilizado um dicionário que relaciona as chaves públicas dos signatários com os seus endereços para a comunicação com entradas na forma **"chave-pública":(ip,porta**). Esse dicionário e o multiset $L$ em uma codificação única $\langle L \rangle$ comum a todos os signatários, são utilizados para construir um lista ordenada comum a todos os signatários das tupla de endereços (ip,porta). Essa lista ordenada de endereços é utilizada para formar uma fila comum a todos para definir qual dos signatários irá enviar o seu pacote e quais irão

recebe-los. Caso o endereço na frente da fila corresponda ao endereço do signatário, ele envia seu pacote, enquanto todos os seus co-signatários o aguardam.

Nesta implementação, o signatário que irá enviar o seu pacote atua como o **servidor** enquanto os co-signatários que aguardam o pacote, atuam como **clientes**, ambos implementados em suas respectivas classes **Server** e **Client** utilizando principalmente a biblioteca **socket**.

O **servidor** utiliza como parâmetros de inicialização uma lista de pares da comunicação, o pacote a ser enviado e o seu endereço. Após a criação do *socket* do servidor, são criadas *threads* que esperam por conexões dos co-signatários. Uma vez efetuada a conexão com os clientes, o servidor espera a requisição do cliente pelo pacote do servidor (o pacote é o mesmo para todos os clientes), sendo esta requisição o código "⟨SEND_PKG⟩". Após receber a requisição, o servidor envia o pacote para o cliente e, após o seu recebimento, o cliente deve responder com "⟨PKG_OK⟩". Quando o servidor receber todos as confirmações de recebimento, ele envia "⟨CONTINUE⟩"aos clientes para que eles possam prosseguir com sua execução. A conexão com os clientes é finalizada e assim, o servidor termina a sua execução.

Já o **cliente**, utiliza como parâmetro de inicialização apenas o endereço (ip e porta) do servidor com o qual ele realizará a comunicação, e retorna o pacote distribuído pelo servidor. Durante a sua execução, o cliente cria um socket de conexão e realiza tentativas de conexão com o servidor. Após uma conexão bem sucedida, o cliente envia a requisição pelo pacote "⟨SEND_PKG⟩"ao servidor e o aguarda. Uma vez em posse do pacote, é enviado ⟨PKG_OK⟩"ao servidor. O cliente então passa a aguardar a permissão do servidor para que ele prossiga com a sua execução. Após o recebimento de "⟨CONTINUE⟩"o cliente encerra sua execução.

Após um signatário finalizar a sua atuação como cliente ou servidor, o líder da fila de endereços é retirado da lista e o novo líder é analisado para identificar novamente se o signatário deve atuar como servidor ou cliente. As comunicações se encerram quando não há mais elementos na lista.

## 10.10 Árvores de Merkle

O módulo relacionado às árvores de Merkle da implementação (**merkle.py**), tem como objetivos principais:

1. Construir uma árvore de hash a partir de um conjunto de chaves agregadas;
2. Produzir uma prova que indique que uma chave agregada pertence a determinada árvore de hash;

3. Verificar se uma prova é válida para determinada chave agregada e árvore de hash

Para a construção da árvore de Merkle, é necessário primeiramente definir o padrão de árvore a ser utilizada e a sua representação. Dessa forma, foram escolhidas árvores binárias completas representadas por meio de uma lista onde o valor de índice 0 é ocupado pela raiz da árvore. Dado um nó de índice $i$, caso não seja uma folha, seus nós filhos podem ser encontrados nos índices $2i + 1$ e $2i + 2$ e, caso não seja a raiz, seu nó pai pode ser encontrado no índice $\left\lfloor \frac{i-1}{2} \right\rfloor$.

A construção da árvore é feita a partir de um multiset $L = \{X_1, \cdots, X_n\}$ e um conjunto **rest** de restrições de combinações de chaves onde $\textbf{rest} = \{(X_i, \cdots, X_j), \cdots, (X_l, \cdots, X_k)\}$ e $(X_i, \cdots, X_j)$ representa uma tupla com chaves públicas arbitrárias de $L$ (a ordem que as chaves aparecem não importa), considerando que pode não haver restrições quanto às combinações permitidas, em cujo caso, **rest** é um conjunto vazio, representado por "***None***".

É necessário produzir uma codificação única $\langle L \rangle$ do multiset $L$ comum a todos aqueles que irão construir a mesma árvore de Merkle. Para isso, a lista completa de chaves públicas é ordenada lexicograficamente utilizando a função **sort()**, convertida para string e então utilizada como parâmetro da função de hash escolhida para a árvore de Merkle. O resultado desse hash é convertido para string e, dessa forma, é obtido $\langle L \rangle$.

Para o cálculo de todas as combinações possíveis de chaves agregadas do conjunto $L$, é utilizada a função **combinations** presente na biblioteca **itertools**. Fornecendo a lista $L$ como parâmetro de entrada da função, obtemos todas as combinações possíveis de um a $n$ elementos de $L$ dentro de uma lista maior onde cada elemento desta corresponde a uma tupla de uma possível combinação, na forma $[(X_1), (X_2), \cdots, (X_1, X_2), \cdots, (X_1, X_2, \cdots, X_n)]$.

Em posse das possíveis combinações, é então realizado o cálculo das chaves públicas agregadas a partir das combinações obtidas, excluindo aquelas combinações com apenas um elemento e substituindo o resultado das combinações presentes em **rest** pelo valor "**None**"do Python. Dessa forma é possível identificar onde estariam, nas respectivas posições da árvore binária de combinações $m$-de-$n$, as combinações proibidas. Ao final dos cálculos, cada combinação proibida é substituída pela combinação válida mais recente respectiva a ela.

O cálculo da chave pública agregada é feito de acordo com o esquema MuSig. Logo, $\tilde{X}_j = \sum_{i=1}^{k} a_i X_i$, tal que $X_i$ pertence a uma tupla de combinação, $a_i = H_{agg}(\langle L \rangle, X_i)$ e $\tilde{X}_j$ representa a chave pública agregada produzida pela $j$-ésima combinação na lista de combinações válidas. Essas chaves agregadas são então incluídas em uma lista de saída que corresponde a todas as chaves públicas agregadas permitidas.

Em seguida, é realizado o cálculo de hash de cada uma das chaves presentes nessa lista. Cada um destes cálculos é paralelizável, então é criada uma *thread* para cada uma

das chaves que serão calculadas. Ao fim de todas as *threads*, os resultados são adicionados a uma lista correspondente às folhas da árvore de Merkle a ser construída. Um ponto importante é a ordem dos elementos dessa lista de folhas, uma vez que a ordem das folhas influencia no resultado final da árvore de Merkle. Como, para um mesmo conjunto $L$ com as mesmas restrições de combinações, a árvore de Merkle produzida deve ser a mesma, é necessário estabelecer um padrão de construção. No caso desta implementação, o padrão utilizado é obtido a partir da ordenação das chaves presentes em $L$ pela ordem lexicográfica e esta ordenação se mantém durante o cálculo das combinações possíveis e após os cálculos de hash.

Em posse das folhas, é necessário verificar se o número de folhas é uma potência de 2, caso contrário não seria possível montar uma árvore binária cheia. Se o número de folhas for de fato uma potência de 2, prossegue-se com a construção da árvore, caso contrário, a última folha é copiada e adicionada a lista de folhas até que o número de elementos seja uma potência de 2.

Por fim, em posse das folhas, a árvore é construída recursivamente, sendo montado um nível da árvore em cada recursão. Inicialmente, a lista de folhas é percorrida (a partir do índice 0 e a cada novo cálculo, $i = i + 2$) e o nó pai de cada dois elementos $H(\text{folhas}[i], \text{folhas}[i+1])$ é calculado. Os nós calculados são adicionados a uma lista que será utilizada no cálculo do próximo nível, realizando uma nova chamada da função com essa nova lista de nós como entrada, repetindo o processo tratando esses nós como novas folhas. O processo se repete até ser obtida uma lista com apenas um elemento, neste caso, a raiz da árvore foi atingida (note que o número de nós na lista de entrada diminui pela metade em cada recursão). Essa raiz é adicionada a uma lista de saída que corresponde a árvore binária cheia estruturada em forma de lista, e a função retorna. Após esse retorno, as listas de nós calculadas nas recursões anteriores vão sendo progressivamente adicionadas a lista na ordem inversa de chamada das recursões. Por fim, é obtida a árvore de Merkle de combinações permitidas das chaves públicas de $L$.

Para produzir uma prova de pertencimento de uma determinada chave agregada a sua respectiva árvore de hash, é calculado o hash da chave agregada e então o índice desse hash é identificado na lista da árvore. A busca por esse índice é realizada do final até o início da lista, uma vez que as folhas da árvore se concentram no final da lista. A partir desse índice $i$ é possível começar a construção da prova percorrendo a árvore recursivamente. Utilizando $i$, o seu nó pai é identificado por $\left\lfloor \frac{i-1}{2} \right\rfloor$ e seus filhos são adicionados a prova $\mathbf{P}$. É então realizada uma nova chamada da função recursiva, desta vez utilizando o índice do nó pai identificado anteriormente como o novo índice $i$, sendo calculado o seu nó pai e os seus nós filhos adicionados a prova. A recursão finaliza quando é fornecido o nó raiz como entrada para a função e a prova $\mathbf{P}$ contendo os nós filhos necessários para o cálculo

da raiz é retornada.

E por fim, para realizar a verificação de uma prova **P** relativa a uma chave agregada $\tilde{X}$, é utilizado, além de **P** e $\tilde{X}$, uma raiz integra de uma árvore de Merkle relativa a ao multiset de chaves públicas $L$ e suas respectivas restrições[1]. São utilizados os valores de hash da prova dois a dois para calcular o próximo nível até que seja calculado a raiz. Essa raiz obtida da prova é comparada com a raiz integra fornecida. Se elas forem iguais, $\tilde{X}$ é uma combinação permitida das chaves públicas em $L$, caso contrário, $\tilde{X}$ é considerada inválida.

## 10.11 MuSig $m$-de-$n$

Nesta seção será descrita a implementação do esquema **MuSig** no cenário $m$-de-$n$. Diferentemente dos esquemas implementados até então, cada signatário executa o protocolo de forma independente, realizando as comunicações necessárias com os co-signatários por meio do módulo de rede.

No esquema **MuSig** no cenário $m$-de-$n$, dado o multiset de $n$ chaves públicas do grupo $L = \{X_1, \cdots, X_n\}$, são utilizas apenas $m$ chaves de $L$, tal que $2 \leq m \leq n$, para formar a chave agregada $\tilde{X}$. Dessa forma, durante o processo de verificação da assinatura, é necessário verificar também se $\tilde{X}$ é uma combinação válida (e permitida) das chaves de $L$.

Estabelecendo a notação a ser utilizada nesta seção: "signatário"será utilizado para se referir ao usuário local do protocolo de assinatura; "co-signatários"será utilizado para se referir aos outros membros do grupo que participam do protocolo; elementos com o índice 1 se referem aos elementos pertencentes ao signatário, enquanto os demais índices são utilizados para se referir aos elementos dos co-signatários. Por exemplo, $X_1$ se refere a chave pública do signatário, enquanto $R_2$ se refere ao ponto $R_i$ de um co-signatário. A ordem dos índices podem não ser as mesmas localmente para cada participante do protocolo, porém, tal situação não apresenta impactos no processo de assinatura.

Os seguintes parâmetros são utilizados como entrada para implementação da assinatura:

- Uma mensagem $m$ a ser assinada.
- O par de chaves privada/pública $(x_1, X_1)$ do signatário.
- Um multiset $L^* = \{X'_1, X'_2, \cdots, X'_n\}$ contendo todas as chaves públicas do grupo.

[1]Para aplicação em protocolos de criptomoedas, esta raiz integra estaria associada ao valor de uma moeda ou *token*, em uma transação ainda não gasta, e seria armazenada em algum bloco da rede encadeada (*blockchain*) respectiva

- Um multiset $L = \{X_1, X_2, \cdots, X_m\}$, tal que $L \subseteq L^*$, contendo todas as chaves públicas que participarão do processo de assinatura (e formarão a chave pública agregada).
- O hostname/ip e porta utilizados pelo signatário para a comunicação no protocolo.
- Um dicionário de endereços dos co-signatários, relacionando cada chave pública de $L$ a ao seu respectivo endereço (ip, porta) utilizado para a comunicação.
- Uma curva elíptica *ec*.
- Uma função de hash $H_{com}$ utilizada na fase de comprometimento.
- Uma função de hash $H_{agg}$ utilizada no cálculo da chave pública agregada.
- Uma função de hash $H_{sig}$ utilizado na fase de assinatura.
- Uma função de hash $H_{tree}$ utilizada na construção da árvore de Merkle.
- Uma lista contendo as restrições de combinações de chaves públicas de $L$, na forma $[(\text{pub\_key}_1, \cdots, (\text{pub\_key}_k), \cdots, ((\text{pub\_key}_i, \cdots, (\text{pub\_key}_j))]$, onde cada tupla representa uma combinação específica restrita (não importando a ordem) e é composta pelas próprias chaves públicas[2].

Iniciando o protocolo de assinatura, $L^*$ é ordenado lexicograficamente e o resultado obtido é convertido para uma string que é fornecida para uma função de hash. O resultado deste hash é convertido para uma string e assim é obtida a codificação única $\langle L^* \rangle$ de $L^*$ comum a todos os signatários.

Em posse de $\langle L^* \rangle$ é realizado o cálculo de $a_i = H_{agg}(\langle L^* \rangle, X_i)$ para cada $X_i \in L$. Cada $a_i$ é obtido concatenando $\langle L^* \rangle$ com a conversão de $X_i$ para uma string, fornecendo essa concatenação como entrada de $H_{agg}$ e convertendo o resultado para um inteiro módulo a ordem da curva elíptica . Cada $a_i$ é relacionado com sua respectiva chave pública $X_i$ em um dicionário para facilitar o seu acesso nos passos que se seguem.

A chave pública agregada $\tilde{X}$ é então calculada de acordo com a equação $\tilde{X} = \sum_{i=1}^{m} a_i X_i$.

O signatário gera aleatoriamente $r_1 \xleftarrow{\$} \{1, \cdots, q-1\}$, sendo $q$ a ordem da curva elíptica *ec*, e calcula $R_1 = r_1 Q$, sendo $Q$ o ponto base da curva elíptica.

É então calculado o comprometimento $t_1 = H_{com}(R_1)$, convertendo $R_1$ para uma string e o fornecendo como parâmetro de $H_{com}$.

A partir do compromisso $t_1$ calculado, é preparado um pacote para o envio na rede, sendo gerada uma string a partir da concatenação da chave pública, um símbolo separador padronizado entre os signatários (neste caso é utilizado '|') e o pacote $t_1$.

[2]Para aplicações em *blockchains* que permitem a execução de contratos inteligentes, as regras para formação dessa lista pode estar registradas em um contrato armazenado na respectiva rede.

O pacote é então enviado para os co-signatários por meio da função **p2p_get()** do módulo de rede. Esta função é o ponto de entrada do protocolo de troca de pacotes descrito em 10.9.

Ao fim da execução de **p2p_get()**, é obtida uma lista contendo todos os pacotes enviados na rede. A lista obtida seguinte formatação

$$['\text{pub\_key}_1|\text{pacote}_1', '\text{pub\_key}_2|\text{pacote}_2', \cdots, '\text{pub\_key}_m|\text{pacote}_m']$$

. Como neste passo o pacote é o compromisso $t$, a lista obtida é

$$['\text{pub\_key}_1|t_1', '\text{pub\_key}_2|t_2', \cdots, '\text{pub\_key}_m|t_m']$$

O ponto $R_1$ calculado é então enviado para os co-signatários, que por sua vez enviam seu respectivo $R_i$. O preparo do pacote, envio de $R_1$ e recebimento da lista de $R_i$'s ocorre de forma semelhante ao passo anterior.

A lista de $R_i$'s é então convertida para um dicionário que relaciona cada $R_i$ com sua respectiva chave pública.

Essa conversão é realizada da seguinte forma:

1. Cada item da lista é uma string no formato'$\text{pub\_key}_i|R_i$'
2. É realizado um split com relação ao separador '|', sendo assim obtidas duas strings, '$\text{pub\_key}_i$' e '$R_i$'
3. A string '$R_i$' é então convertida para um ponto da curva elíptica da seguinte maneira:
   (a) A string de um ponto da curva elíptica tem a seguinte formatação:
   X: 'valor da coordenada x em hexadecimal'\nY: 'valor da coordenada y em hexadecimal'\n(On curve <'nome da curva'>)[3]
   (b) São então utilizadas as seguintes expressões regulares (regex) para identificar respectivamente a coordenada x, coordenada y e a curva utilizada
      - 'X:(.*)\\nY:'
      - 'Y:(.*)\\n\\('
      - '<(.*)>'
   (c) Os valores obtidos são formatados para retirar espaços
   (d) As coordenadas são então convertidas de hexadecimal para inteiros

[3]Exemplo de uma string de um ponto da curva: X: 0x19a2a2ed86d2907e62042f14b3726563c94d677ff4751b5eb039eb5a49e7222f\nY: 0x41760d468136baa830e6be48a0eaf7561f8ff63b28e068f39d9acc20b7317ad5\n(On curve <secp256k1>

(e) O nome da curva é utilizado para obter o respectivo objeto da curva utilizando a classe ***Curve*** da biblioteca ***fastecdsa***

(f) São utilizados os valores das coordenadas $x$ e $y$ e o objeto representando a curva para inicializar um objeto da classe ***Point***

(g) O ponto obtido é retornado.

4. O ponto $R_i$ é então associado a string da chave pública 'pub_key$_i$' por meio de um dicionário.

5. O processo se repete para cada elemento da lista de $R_i$'s.

Para cada ponto $R_i$ recebido, calcula-se $t'_i = H_{com}(R_i)$ e verifica-se $t'_i \stackrel{?}{=} t_i$ para garantir que não houve adulteração dos pontos $R_i$. Caso alguma checagem indique que $t'_i \neq t_i$, o protocolo é abortado, caso não haja irregularidades, prossegue-se com o protocolo.

Os pontos $R_i$ são então somados para formar $R = \sum_{i=1}^{m} R_i$.

Uma vez em posse do $R$ e da chave agregada $\tilde{X}$, é possível calcular o desafio $c = H_{sig}(\tilde{X}, R, m)$. Para tal, $\tilde{X}$ e $R$ são convertidos para strings e concatenados, este resultado é concatenado com a mensagem $m$ e a string resultante é fornecida para a função $H_{sig}$. O resultado do hash é convertido de bytes para um inteiro módulo a ordem do grupo da curva e assim é obtido $c$ comum a todos os signatários.

Em seguida, o signatário calcula a sua assinatura parcial $s_1 = r_1 + c \cdot a_1 \cdot x_1 \bmod q$, sendo $x_1$ a chave privada do signatário e $q$ a ordem da curva *ec*.

A partir da assinatura parcial $s_i$ é preparado um pacote ('$str(X_1)|str(s_1)$') que será enviado para os co-signatários por meio de **p2p_get()** e também por meio desta são recebidos os $s_i$ dos co-signatários. De forma semelhante ao recebimento dos pontos $R_i$, a comunicação retorna uma lista de pacotes de $s_i$ em uma formatação padrão. Esta lista é então convertida para um dicionário correlacionando as chaves públicas $X_i$ com seus respectivos $s_i$.

Em posse dos $s_i$'s de todos os signatários, é calculado a assinatura parcial do grupo $s = \sum s_i \bmod q$. Dessa forma, obtemos a multi-assinatura $(R, s)$.

Como a implementação ocorre no cenário $m$-de-$n$ é necessário produzir uma prova de validez da chave agregada $\tilde{X}$. Para isso, é utilizada a função **build_merkle_tree()** (pertencente ao módulo **merkle.py**) que irá construir uma árvore de Merkle (como descrito em 10.10) com base no multiset $L^*$ contendo todas as chaves públicas do grupo e a lista de restrições, fornecidas como entrada da função (considerando que o número de combinações possíveis de chaves de $L^*$ é tratável).

Em seguida, é utilizada a função **produce_proof()** (pertencente ao módulo **merkle.py**), que constrói uma prova **P** a partir da chave agregada  e da árvore de Merkle

construída previamente. A prova é produzida de acordo com a descrição apresentada em 10.10.

O processo de assinatura se encerra retornando a assinatura $(R, s)$, a chave agregada $\tilde{X}$ e a prova $\mathbf{P}$.

Diferentemente do esquema **MuSig** no cenário $n$-de-$n$, a verificação de uma assinatura $(R, s)$ no cenário $m$-de-$n$ envolve dois passos: a verificação da validade da chave agregada $\tilde{X}$ utilizada na assinatura e, subsequentemente, a verificação da validade da assinatura $(R, s)$ para $m$ e $\tilde{X}$.

Os seguintes parâmetros são utilizados como entrada para a verificação da assinatura:

- Uma assinatura $(R, s)$
- Uma mensagem $m$
- Uma chave agregada $\tilde{X}$
- Uma prova $\mathbf{P}$
- Uma raiz integra *root* da árvore de Merkle construída a partir de combinações válidas das chaves públicas agregadas do grupo de signatários
- Uma curva elíptica *ec*.
- Uma função de hash $H_{sig}$ utilizado na fase de assinatura.
- Uma função de hash $H_{tree}$ utilizada na construção da árvore de Merkle.

A verificação da validade de $\tilde{X}$ ocorre fornecendo $\tilde{X}$,$\mathbf{P}$ e a raiz integra como parâmetros da função **verify()** do módulo **merkle.py**. O processo de verificação ocorre assim como descrito em 10.10. Caso $\tilde{X}$ não seja válido para $\mathbf{P}$ e a raiz, **verify()** retorna *False*, a verificação é abortada e a assinatura é considerada inválida, caso contrário **verify()** retorna *True* e prossegue-se com a verificação de $(R, s)$.

A verificação da assinatura $(R, s)$ ocorre de forma similar a verificação de uma assinatura (simples) de Schnorr. É calculado $c = H_{sig}(\tilde{X}, R, m)$ assim como no processo de assinatura, em seguida são calculados os dois lados da equação $sG \stackrel{?}{=} R + c\tilde{X}$, calculando os pontos $left \leftarrow sG$ e $right \leftarrow R + c\tilde{X}$. Os pontos $left$ e $right$ são então comparados coordenada a coordenada, e caso sejam iguais, $(R, s)$ é uma multi-assinatura válida para a mensagem $m$ e chave agregada $\tilde{X}$, e a verificação retorna *True*. Caso contrário, $(R, s)$ é considerada inválida e a verificação retorna *False*.

É possível realizar a verificação também por meio da lista completa de chaves públicas $L^*$, a chave agregada $L$ (ou uma lista informando quais são as chaves de $L^*$ serão utilizadas na assinatura) e uma lista de restrições de combinações $rest$ sem utilizar diretamente $\tilde{X}$, $\mathbf{P}$ e a raiz integra $root$.

Neste caso, é verificado se $L \in rest$ e se $L \subseteq L^*$, pois se $L \notin rest$, então a chave agregada gerada por $L$ é uma combinação proibida e, se $L \not\subset L^*$, então a chave agregada gerada por $L$ não é uma combinação de chaves de $L^*$.

Uma vez verificado $L$, $L^*$ é utilizada para gerar $\langle L^* \rangle$. A partir de $\langle L^* \rangle$ e $L$ são calculados os $a_i$ e subsequentemente é calculado $\tilde{X}$. A partir deste ponto, a verificação da assinatura ocorre como descrito anteriormente.

Esta segunda possibilidade de verificação não foi implementada pois, em geral, ela apresenta um custo significativamente maior para o verificador do que a primeira. Em cenários reais de assinatura digital, especialmente em protocolos de criptomoedas, o verificador processa (verifica) uma quantidade muito maior de assinaturas quando comparada ao número de assinaturas (ou multi-assinaturas) produzidas por um signatário (ou um grupo de signatários). Logo, a maior parte do custo do esquema deve ser direcionado para o signatário, diminuindo assim, sempre que possível, o custo sobre o verificador.

Dessa forma, na primeira verificação apresentada, o custo de processamento é maior para o verificador, uma vez que ele deve construir uma árvore de Merkle a partir de $L^*$ e produzir uma prova para a sua chave agregada a partir dela. Enquanto o verificador deve apenas analisar a prova produzida e verificar a assinatura como se fosse uma assinatura padrão de Schnorr com relação a chave pública agregada. O custo de transmissão no segundo caso também é significativamente maior, uma vez que é necessário o envio de $L^*$ e $L$ para o verificador.

Tomando por exemplo o cenário atual do Bitcoin, que utiliza a curva *secp256k1* e a função de hash *SHA-256*, uma chave pública tem 512+8 bits na sua forma não comprimida ($256 + 8$ bits na forma comprimida) e o resultado de um hash tem 256 bits. Dessa forma:

- $R$ é um ponto da curva então ele ocupa 264 bits em sua forma comprimida
- $s$ é um inteiro dentro da ordem da curva, logo ele ocupa 256 bits
- a mensagem $m$ tem tamanho variado, representado por $|m|$
- $\tilde{X}$ é um ponto da curva logo ele ocupa 264 bits em sua forma comprimida
- $\mathbf{P}$ é composto de valores de hash com 256 bits cada e tem o custo total de[4]:

$$256 \cdot \lceil \log_2(\sum_{i=2}^{n} \frac{n!}{i!(n-i)!}) \rceil \text{ bits}$$

[4]É possível verificar o número de folhas de uma árvore de Merkle produzida a partir de um multiset $L^*$ com $n$ entrada, tem $k = \sum_{i=2}^{n} \frac{n!}{i!(n-i)!}$ combinações, tal árvore tem altura $\lceil \log_2(k) \rceil$, uma vez que o total de folhas será a potência de 2 mais próxima maior ou igual a k e considerando a altura da raiz como 0. Logo, serão necessários no mínimo $\lceil \log_2(k) \rceil$ valores de hash e o hash da chave agregada para calcular cada nível da árvore até a raiz. Assim, temos que $\mathbf{P}$ tem $\lceil \log_2(k) \rceil$ elementos.

- $L^* = \{X'_1, \cdots, X'_n\}$ tem custo de $n \cdot 256$ bits
- $L = \{X_1, \cdots, X_k\}$ tem custo de $k \cdot 256$ bits

Dessa forma, na primeira verificação, tem um custo aproximado de

$$264 + 256 + |m| + 264 + 256 \cdot \lceil \log_2(\sum_{i=2}^{n} \frac{n!}{i!(n-i)!}) \rceil \text{ bits}$$

enquanto a segunda verificação tem um custo aproximado de

$$264 + 256 + |m| + 264(n + k) \text{ bits.}$$

Assim, é possível notar que nos casos em que o número total de combinações das chaves públicas de $\langle L^* \rangle$ é tratável, o custo de transmissão no primeiro caso é menor que no segundo. Este custo de transmissão somado ao custo de processamento do verificador, torna a verificação utilizando $\tilde{X}$, $\mathbf{P}$ e *root* a escolha preferencial de verificação sempre que for possível utilizá-la.

## 10.12 Interface Gráfica

A construção da interface de usuário tem como base as bibliotecas ***json*** e ***tkinter***, que são utilizadas na construção dos dois principais componentes deste módulo, respectivamente: analisador de entradas (**parser.py**) e interface da aplicação (**gui.py**).

O analisador de entradas (**parser.py**) é utilizado para o tratamento e manipulação das entradas obtidas por meio da interface gráfica e subsequentemente utilizadas nos processos de assinatura e verificação.

Para tal, são lidos arquivos **json** contendo os todos os parâmetros de entrada necessários para as funções de assinatura e verificação do **MuSig** $m$-de-$n$.

Devido a necessidade de parâmetros diferentes para cada função, os arquivos são divididos em três categorias, identificados pela chave ***"data_type"***:

- Assinatura (tipo 0): mensagem a ser assinada, endereço de comunicação do signatário, nome do arquivo contendo o par de chaves do signatário, chaves públicas e endereços de comunicação respectivos dos co-signatários, lista completa de chaves públicas dos signatários, restrições de combinações, curva elíptica utilizada e funções de hash utilizadas.
- Verificação (tipo 1): ponto $R$ da assinatura, assinatura parcial $s$, mensagem respectiva, chave agregada, prova de validade da chave agregada, curva elíptica utilizada e funções de hash.

- Informações da raiz de uma árvore de Merkle (tipo 2): raiz de uma árvore de Merkle ou lista completa de chaves públicas dos signatários e suas respectivas restrições de combinações.

As informações são codificadas de acordo com os padrões de arquivos **json**.

Os objetos da classe ***Info*** (de **parser.py**) são utilizados para armazenar as informações presentes nos arquivos de entrada e formata-las de acordo com os padrões de entrada da implementação do MuSig $m$-de-$n$ (10.11).

Para a construção da interface gráfica foi criada a classe ***Application*** com herança da classe ***Tk*** do **tkinter**. Esta classe serve como a janela principal da aplicação e como a base da mesma. Nela, é implementada a função de troca de *frames* da aplicação, cada *frame* funciona como uma nova janela que é aberta sobre a aplicação principal e são implementadas como objetos da classe ***Frame*** (do próprio **tkinter**) atrelados a aplicação principal. Cada troca de *frame* é realizada utilizando primeiramente o método *destroy()* da classe ***Frame*** e substituindo o *frame* atual da aplicação pelo desejado.

Cada nova página/janela da aplicação é construída a partir da classe ***PageBase*** com herança da classe ***Frame***. ***PageBase*** serve como a base das páginas da aplicação, onde são implementados atributos de uso comum entre as páginas.

Foram criadas então as classes ***PageMuSig*** e ***PageMuSigVer***, ambas com herança de ***PageBase***. A classe ***PageMuSig*** implementa a interface da assinatura do MuSig $m$-de-$n$, onde é possível fornecer o endereço do arquivo de configuração da assinatura em formato **json**, informar o endereço de arquivo no qual será criado o arquivo **json** contendo as informações da assinatura (assinatura $(R, s)$, chave agregada e prova de validade da chave) e verificar o resultado da assinatura também por meio de uma caixa de exibição de saída textual da aplicação.

De forma semelhante, a classe ***PageMuSigVer***, apresenta a interface de verificação de assinatura, onde deverão ser apresentados em campos respectivos os endereços dos arquivos **json** contendo as informações para a verificação (dados tipo 1) e informações da raiz da árvore de Merkle (obtidas com integridade, dados tipo 2). O resultado da verificação são exibidos por meio de texto pela interface em uma caixa de exibição semelhante a do processo de assinatura.

A interface é executada por meio da inicialização de um objeto da classe ***Application*** e a execução do seu método ***mainloop()***.

# Capítulo 11

# Considerações Finais

Assim como descrito ao longo deste trabalho, a substituição do esquema **ECDSA** pelo **esquema de assinatura de Schnorr** traz melhorias em termos de eficiência de processamento das assinaturas, menor custo de armazenamento e suporte nativo disponível para a utilização do esquema de multi-assinatura **MuSig**. O **MuSig**, por sua vez, traz melhoras significativas para o cenário de multi-assinaturas do **Bitcoin**, uma vez que não é necessário a verificação individual das assinaturas produzidas por cada membro do grupo de signatários. Ao se utilizar uma assinatura conjunta verificável a partir de uma única chave pública agregada produzida a partir das chaves públicas dos signatários, o custo de armazenamento e verificação é significativamente reduzido, com um bônus em aumento da privacidade.

Diante desse cenário, foi proposto neste trabalho a construção de uma implementação do esquema **MuSig** no cenário $m$-de-$n$. A implementação obtida foi satisfatória, permitindo a geração de multi-assinaturas baseadas no esquema **MuSig** por um grupo de signatários de forma descentralizada, isto é, sem o uso de um terceiro (confiável) que realize a mediação da comunicação, assim como a verificação dessas multi-assinaturas utilizando chaves públicas agregadas e a validação dessas chaves agregadas por meio de árvores de Merkle.

Foi também possível realizar a implementação satisfatória de provas de conceito dos outros tópicos aqui apresentados, sendo eles: esquema de assinatura de Schnorr, esquema *Naive Schnorr Multi-signature*, *Rogue-key attack* e esquema Bellare-Neven.

Em relação às limitações da implementação do **MuSig** $m$**-de-**$n$, percebe-se que esta ainda pode agregar as seguintes melhorias: transmissão de pontos em sua forma compacta (reduzindo o custo de armazenamento e transmissão) e melhorar a aleatoriedade das funções pseudo-randômicas utilizadas, possivelmente por meio da utilização dos parâmetros da assinatura como parte da *seed*. É necessário também um estudo de melhores possibilidades de realizar a comunicação entre os signatários, uma vez que esta transmissão se

apresenta como um dos gargalos de eficiência da implementação.

Como trabalho futuro é indicado a possibilidade de utilização do esquema **MuSig** como base para um esquema IAS (*Interactive Aggregate Signatures*), no qual cada signatário pode utilizar uma mensagem diferente das demais e a assinatura conjunta produzida pode ser utilizada para identificar se cada mensagem $m_i$ foi assinada pelo seu respectivo $i$-ésimo signatário. Outra indicação é a implementação de uma prova de conceito do Bitcoin utilizando a assinatura de Schnorr como seu esquema de assinatura e o esquema **MuSig** como esquema seu esquema para multi-assinaturas.

# Referências


[1] Hankerson, Darrel, Alfred J. Menezes e Scott Vanstone: *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Berlin, Heidelberg, 2003, ISBN 038795273X. x, 3, 6, 7, 8, 13, 14, 16, 17

[2] Gregory Maxwell, Andrew Poelstra, Yannick Seurin e Pieter Wuille: *Simple schnorr multi-signatures with applications to bitcoin*. Designs, Codes and Cryptography, 87, 2019. x, 3, 4, 21, 38, 39, 41, 43, 44, 47, 48, 49, 55, 58, 59

[3] Wuille, Pieter: *"bitcoin improvement proposal - schnorr signatures for secp256k1", https://github.com/sipa/bips/blob/bip-schnorr/bip-schnorr.mediawiki*. x, 4, 56, 57

[4] Schnorr, C. P.: *Efficient signature generation by smart cards*. J. Cryptol., 4(3):161–174, janeiro 1991, ISSN 0933-2790. `http://dx.doi.org/10.1007/BF00196725`. 3, 21

[5] Bellare, Mihir e Gregory Neven: *Multi-signatures in the plain public-key model and a general forking lemma*. Em *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, páginas 390–399, New York, NY, USA, 2006. ACM, ISBN 1-59593-518-5. `http://doi.acm.org/10.1145/1180405.1180453`. 3, 27, 30

[6] Arnaldo Leite Pinto Garcia, Yves Albert Emile Lequain: *Elementos de Álgebra*. IMPA, 2018, ISBN 9788524404504. 7

[7] Research, Certicom: *Sec 2: Recommended elliptic curve domain parameters*. `http://www.secg.org/sec2-v2.pdf`. 18

[8] Jivsov, A.: *Compact representation of an elliptic curve point*. `https://tools.ietf.org/id/draft-jivsov-ecc-compact-05.html`. 19

[9] Joppe W. Bos1, J. Alex Halderman, Nadia Heninger Jonathan Moore Michael Naehrig1 e Eric Wustrow: *Elliptic curve cryptography in practice*. `https://eprint.iacr.org/2013/734.pdf`. 19

[10] Nakamoto, Satoshi: *Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf*. 55

[11] Kueltz, Anton: *"fastecdsa", https://pypi.org/project/fastecdsa/*. 60

[12] *"gmpy2", https://gmpy2.readthedocs.io/en/latest/*. 60