

# Verificação Eficiente de Dados com Rust: Implementação e Análise da Estrutura da Árvore de Merkle

Raphael F. Queiroz<sup>1</sup>, Igor M. Coelho<sup>1</sup>

<sup>1</sup>Instituto de Computação - Universidade Federal Fluminense (UFF)

**Abstract.** *Verifying the authenticity of received data represents a contemporary challenge, something Ralph Merkle had already anticipated by highlighting the importance of ensuring the truthfulness of information. This article introduces a verifiable data structure, specifically the Merkle Tree, and illustrates its implementation in the Rust programming language, along with a detailed analysis of its performance in this language.*

**Resumo.** *Verificar a autenticidade de dados recebidos representa um desafio contemporâneo, algo que Ralph Merkle já antecipava ao destacar a importância de assegurar a veracidade das informações. Este artigo apresenta uma estrutura de dados verificável, especificamente a Merkle Tree, e ilustra sua implementação na linguagem de programação Rust, além de realizar uma análise detalhada de seu desempenho nessa linguagem.*

## 1. Introdução

Este estudo explora a implementação da estrutura de dados Merkle Tree utilizando a linguagem de programação Rust, destacando sua eficácia na verificação de integridade de dados em diversas aplicações, incluindo Blockchain [Pasco and Coelho 2021], certificados digitais, sistemas autônomos, sistemas de arquivos distribuídos e Machine Learning.

A Merkle Tree é uma estrutura binária eficiente para validações de dados, proposta por Ralph Merkle, e sua relevância é demonstrada através de uma prova de Merkle, que simplifica a verificação de dados.

A linguagem Rust é escolhida por suas características únicas, como segurança de memória e gerenciamento eficiente de concorrência, sendo crucial para a eficiência e segurança da implementação.

O artigo detalha a estrutura, métodos e performance da Merkle Tree implementada em Rust, demonstrando a escalabilidade e eficiência da estrutura em operações de verificação de dados. A análise de performance revela que a criação da árvore tem complexidade  $O(N)$ , enquanto a criação e validação de provas crescem de forma logarítmica. Em conclusão, a implementação em Rust da Merkle Tree não só valida sua eficiência e segurança, mas também reforça a adequação de Rust para estruturas de dados complexas e aplicações críticas em termos de segurança.

## 2. Estrutura de Dados Verificáveis

Em um contexto no qual dados guiam tomadas de decisões, a integridade dos dados utilizados é crucial para tomadas de decisões acertivas. Estrutura de dados verificáveis disponibiliza uma alternativa transparente e verificável para os dados nela contidos.

Essa classe de estrutura de dados pode ser aplicada para qualquer modelo no qual deseja-se verificar a integridade de dados. Sua aplicação mais notável na atualidade é *Blockchain* [Nakamoto 2009], possibilitando transações de moedas digitais de maneira segura e escalável. Suas aplicações vão para além da *Blockchain*, emissão de certificado digitais, sistemas autônomos, sistemas de arquivos distribuídos e *Machine Learning* são alguns exemplos do seu amplo campo de atuação. [Yu and Kumbier 2020]

### 3. Merkle Tree

No final da década de 70, Ralph Merkle propôs em seu artigo o conceito de Merkle Tree [Merkle 1989]. Ele antecipou a importância de assinaturas digitais, entendendo que esse era um ponto crucial para a segurança de operações digitais. Uma *Merkle Tree* é uma estrutura de dados utilizada para verificar um conjunto de dados.

Uma Merkle Tree é criada combinando dois nós de cada camada da estrutura, usando um algoritmo de hash nesses dados combinados e criando um nó pai, como demonstrado na equação 1.

$$tree[i][j] = hash(tree[i+1][2j].hash + tree[i+1][2j+1].hash) \quad (1)$$

A figura 1 mostra uma representação gráfica da Merkle Tree. Nota-se que essa estrutura é similar a uma estrutura de uma árvore binária. A característica da Merkle Tree de fazer validações de dados de maneira eficiente, está diretamente relacionada a sua estrutura.

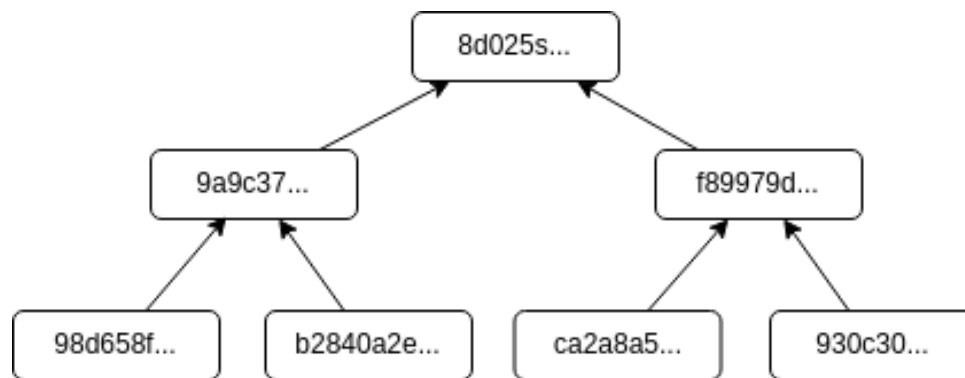


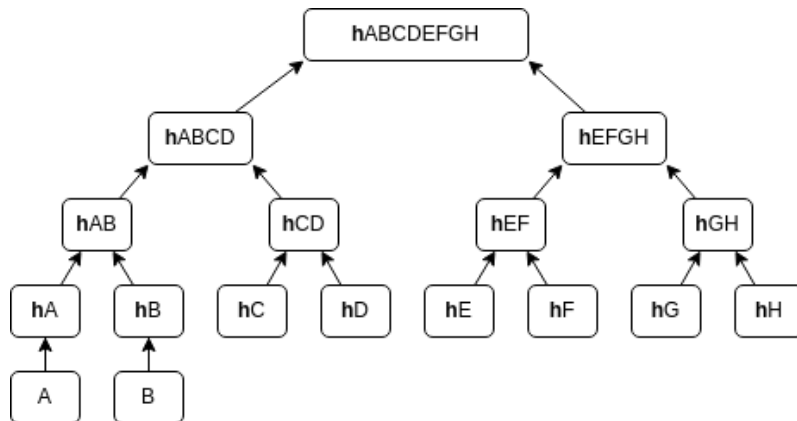
Figura 1. Essa figura representa a estrutura de uma Merkle Tree

#### 3.1. Prova de Merkle

Em uma estrutura de dados tradicional, para certificar-se de que os dados recebidos são válidos, deve-se analisar cada pacote recebido com o original, caso algum não seja coerente, conclui-se que o conjunto de dados é inválido. Essa abordagem pode ser simplificada através de uma prova de Merkle.

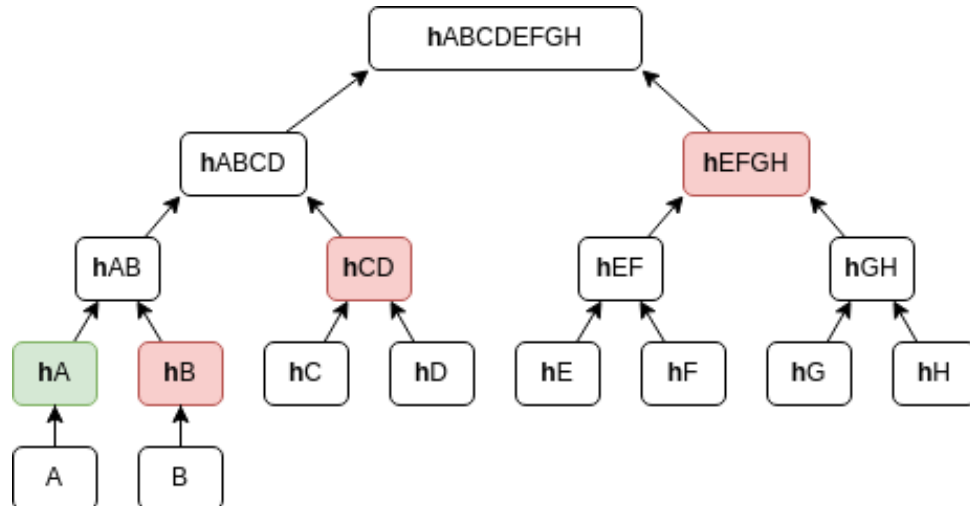
A prova de Merkle permite validar se um elemento, ou conjunto de elementos, faz parte da árvore, fornecendo apenas uma pequena quantidade de informação: os hashes que formam o caminho da folha desejada até a raiz. Dessa maneira, a quantidade de validações é reduzida. Digamos que existe um pacote A, no qual deseja-se saber se ele está de fato contido no grupo de dados. É necessário reconstruir o caminho da folha para

a raiz da árvore. Para essa reconstrução se faz necessário dados extras, onde é gerado um subconjunto de dados dentro da estrutura global da árvore, com apenas os dados necessários para a obtenção do hash da raiz.



**Figura 2. Estrutura Merkle Tree com representação de hash**

A figura 3 mostra que para reconstruir o hash da raiz, a partir do pacote A. Digamos que o objetivo seja reconstruir o hash da raiz (hABCDEFGH), como o valor do hA é dado, os dados de prova deve gerar hB, para reconstruir hAB, depois será necessário o hCD, para reconstruir hABCD e por fim, será necessário gerar o valor de hEFGH para reconstruir o valor da raiz.



**Figura 3. Validação pacotes na árvore de Merkle. Elemento marcado em verde será verificado, os elemento em vermelho serão necessários para a prova de integridade dos dados**

Essa abordagem simples, garante que caso seja possível reconstruir o hash da raiz, o dado em prova é possivelmente verdadeiro. Essa validação requer apenas um subconjunto dos dados da estrutura e possibilidade validar qualquer elemento da árvore.

#### 4. Aplicação Merkle Tree

Para desenvolvimento desse artigo, foi utilizada a linguagem de programação *Rust* para implementação da árvore de *Merkle*. O objetivo é construir a estrutura que receberia como

entrada um conjunto de dados, e a partir dessa estrutura gerar uma árvore de *Merkle*, gerando também uma prova, para validar se houve algum tipo de mudança nos dados.

#### 4.1. Linguagem programação *Rust*

A busca por linguagens que combinem eficiência, segurança e concorrência de forma equilibrada tem sido um desafio moderno. A linguagem de programação *Rust*, desenvolvida pela *Rust Project Developers* e documentada em "The Rust Programming Language"[Klabnik and Nichols 2022], emerge como uma resposta promissora a essas demandas.

*Rust* se diferencia por sua capacidade de garantir segurança de memória sem recorrer a um coletor de lixo, um marco significativo na programação de sistemas. Esta característica não apenas promove uma gestão de memória mais eficiente, mas também reduz o risco de erros comuns em linguagens de baixo nível. Além disso, *Rust* adota uma filosofia de "segurança como padrão", onde o compilador desempenha um papel crítico na prevenção de uma série de vulnerabilidades comuns, como condições de corrida em ambientes de concorrência.

#### 4.2. Implementação

Para esse artigo, foi definido a seguinte estrutura para a árvore de *Merkle*.

```
impl MerkleTree {
    fn new(
        data: Vec<u8>
    ) -> Self {}

    fn create_tree(
        &mut self,
        data: Vec<u8>
    ) {}

    fn generate_proof(
        &self,
        data_index: u8
    ) -> Vec<(Vec<u8>, bool)> {}

    fn verify_proof(
        &self,
        proof: Vec<(Vec<u8>, bool)>,
        data: u8,
        root_hash: Vec<u8>,
    ) -> bool {}
}
```

A implementação completa do código da árvore de *Merkle* em *Rust* pode ser encontrada no repositório GitHub. Para acessar o código, visite o seguinte link: [github.com/queirozrphl/rust\\_merkle\\_tree](https://github.com/queirozrphl/rust_merkle_tree).

A estrutura `MerkleTree` em Rust contém várias funções importantes para a manipulação e verificação de árvores de Merkle. As principais funções são descritas a seguir:

#### 4.2.1. `new`

Esta função cria uma nova instância da árvore de *Merkle*. Ela recebe um vetor de bytes como entrada e inicializa a árvore de Merkle com esses dados.

#### 4.2.2. `create_tree`

O método `create_tree` é utilizado para construir a árvore de Merkle a partir de um conjunto de dados fornecidos. Este método organiza os dados em uma estrutura de árvore e calcula os hashes dos nós.

#### 4.2.3. `generate_proof`

O método `generate_proof` gera a prova de inclusão para um determinado índice de dados na árvore. Retorna um vetor de pares, onde cada par contém um hash e um valor booleano indicando a posição relativa do nó. Como trata-se de uma estrutura de dados verificáveis, esse método é fundamental para a reconstrução do dado original, permitindo validar a subconjunto de dados em análise.

#### 4.2.4. `verify_proof`

O método `verify_proof` é usado para verificar se uma prova fornecida corresponde ao hash raiz da árvore para um determinado dado. Isso é essencial para validar a integridade e a autenticidade dos dados na árvore de Merkle.

### 4.3. Análise de Performance

Três operações foram avaliadas para analisar a performance do algoritmo, a criação da árvore de *Merkle*, o tempo de criação das provas e a validação das provas. Analisando o algoritmo o comportamento da criação da árvore possui complexidade  $O(N)$ , analisando a 1 é possível perceber um crescimento linear entre a quantidade de folhas na árvore e o tempo de criação da árvore.

A tabela 1 exibe o tempo médio de execução de diversas funções, com dados obtidos a partir da média aritmética de 1000 execuções individuais. Estes testes foram realizados em um computador de uso doméstico, com especificações padrão, visando replicar um ambiente computacional típico. O objetivo é fornecer uma avaliação realista do desempenho dessas funções em condições comuns de uso. Tal metodologia assegura a confiabilidade dos dados ao minimizar as variações de desempenho causadas por flutuações temporais ou especificidades do hardware.

Porém, a estrutura foi projetada para verificar dados de forma eficiente, e é possível perceber que o tempo de criação de provas (prova que será usada para validar

**Tabela 1. Análise de tempo de execução de criação da estrutura, criação de provas e tempo de validação das provas**

Folhas	Tempo Criação Árvore (ms)	Tempo Criação Provas ( $\mu$ s)	Tempo Validação Provas ( $\mu$ s)
100	1,72668	1,215	68,863
1000	15,14768	2,265	95,141
10000	209,8901	4,226	130,296
100000	1773,2686	5,166	158,705

um dado dentro da estrutura) e o tempo de validação das provas, dado um elemento, verificar se ele está correto, se existe dentro da árvore onde deveria existir, crescem de maneira logarítmica, possibilitando rápida verificação da árvore.

## 5. Conclusão

Foi apresentada a estrutura de dados Merkle Tree e uma implementação da sua estrutura utilizando a linguagem de programação Rust. Foi possível perceber que sua estrutura, apesar de simples, é verificável de maneira eficiente.

Em conclusão, a escolha do Rust como linguagem de programação para a implementação desta Merkle Tree revelou-se excepcionalmente eficaz, não apenas em termos de segurança de memória e desempenho, mas também na facilidade de gerenciamento de concorrência, demonstrando assim a adequação de Rust para estruturas de dados complexas e aplicações críticas em termos de segurança.

## Referências

- Klabnik, S. and Nichols, C. (2022). *The Rust Programming Language*. Rust Project Developers.
- Merkle, R. (1989). A certified digital signature. volume 435, pages 218–238.
- Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system.
- Pasco, C. and Coelho, I. (2021). Análise das estruturas de dados verificáveis nas blockchains ethereum e neo. In *Anais do IV Workshop em Blockchain: Teoria, Tecnologias e Aplicações*, pages 1–6, Porto Alegre, RS, Brasil. SBC.
- Yu, B. and Kumbier, K. (2020). Veridical data science. *Proceedings of the National Academy of Sciences*, 117(8):3920–3929.