# Análise de Código para Detecção de Vulnerabilidades: Comparação entre Abordagens Baseadas em Aprendizado de Máquina e em Algoritmos Determinísticos

**Ana Carolina Caldas de Mello**
**Gustavo Menezes Barbosa**
**João Pedro Queiroz Rocha**
**Maria Eduarda Gonçalves de Souza Ferreira**
**Pedro Henrique Dias Camara**
**Wanessa Dias Costa**

[1]Pontifícia Universidade Católica de Minas Gerais
Engenharia de Software
Campus Lourdes

ana.caldas@sga.pucminas.br, gustavo.barbosa.1386677@sga.pucminas.br

joao.rocha.1439661@sga.pucminas.br, 1380383@sga.pucminas.br

pcamara@sga.pucminas.br, wanessa.costa@sga.pucminas.br

***Abstract.*** *This work investigates the effectiveness of code analysis approaches based on machine learning (ML) and deterministic algorithms in detecting vulnerabilities in open-source projects. Four tools are compared: two deterministic (Codeql and Snyk) and two ML-based (AutoVAS and VulCNN), using the Software Assurance Reference Dataset (SARD) from the National Institute of Standards and Technology (NIST). Accuracy, coverage by Common Weakness Enumeration (CWE) categories, and efficiency (time, Central Processing Unit — CPU, and memory) are evaluated, presenting limitations and directions for future work.*

***Resumo.*** *Neste trabalho, é investigada a eficácia de abordagens de análise de código baseadas em machine learning (ML) e em algoritmos determinísticos na detecção de vulnerabilidades em projetos open-source. São comparados quatro ferramentas: duas determinísticas (Codeql e Snyk) e duas de ML (AutoVAS e VulCNN), utilizando a base Software Assurance Reference Dataset (SARD) do National Institute of Standards and Technology (NIST). Os critérios de avaliação são: precisão, cobertura por categorias Common Weakness Enumeration (CWE), e eficiência (tempo, Central Processing Unit — CPU e memória), a fim de apresentar limitações e direções para trabalhos futuros.*

## 1. Introdução

Vulnerabilidades de software podem ser entendidas como falhas no projeto, na implementação, na operação ou na gestão que permitem a violação da política de segurança do mesmo [Shirey 2007]. Esses defeitos variam desde problemas simples, como o uso de variáveis não inicializadas, até erros críticos, como injeção de código e estouro de inteiros [Arusoaie et al. 2017]. Essas falhas no sistema, se não identificadas e

tratadas corretamente, conseguem causar perdas inestimáveis, e a literatura tem como registro a missão *Mars Polar Lander* (MPL) e *Deep Space 2* (DS2), consequentes de um erro de software [Albee et al. 2000], e os acidentes do Therac-25, que expuseram pacientes a doses excessivas de radiação [Levenson and Turner 1993].

Dessa forma, é possível afirmar que as vulnerabilidades de software não devem ser negligenciadas. No entanto, métodos tradicionais de teste podem demandar tempo e recursos escassos [Abdullahi et al. 2020] [Garousi et al. 2020]. Como consequência, vulnerabilidades críticas seguem recorrentes em diferentes contextos [Sánchez et al. 2020], o que reforça a necessidade de técnicas automáticas e robustas. Nesse cenário, são utilizadas ferramentas de análise estática (*Static Application Testing* — SAT) e dinâmica (*Dynamic Application Security Testing* — DAST), além de abordagens com *machine learning* (ML) e *deep learning* (DL).

Apesar de estudos anteriores avaliarem ferramentas SAT e DAST, além de explorarem ML para detecção de vulnerabilidades [Qadir et al. 2025] [Russell et al. 2018] [Wu et al. 2017] [Steenhoek et al. 2023], ainda há uma lacuna quanto à comparação direta entre ferramentas baseadas em ML e algoritmos determinísticos. Essa ausência de evidência formal dificulta a escolha técnica embasada.

## 2. Objetivo

Investigar a eficácia de abordagens de análise de código baseadas em ML e em algoritmos determinísticos na detecção de vulnerabilidades em projetos C/C++.

## 3. Perguntas de Pesquisa

**P1.** Qual a precisão de ferramentas de análise de código que utilizam ML em comparação com algoritmos determinísticos?

- Taxa de vulnerabilidades encontradas [Jeon and Kim 2021] [Arusoaie et al. 2017] [Maskur and Asnar 2019];
- Taxa de falsos positivos e negativos [Jeon and Kim 2021] [Arusoaie et al. 2017] [Emanuelsson and Nilsson 2008] [Russell et al. 2018];
- Detecções únicas por ferramenta [Arusoaie et al. 2017].

**P2.** Quais tipos de vulnerabilidades são mais detectadas e negligenciadas por cada abordagem?

- Cobertura por categoria *Common Weakness Enumeration* (CWE): *buffer overflow*, uso de variáveis não inicializadas, injeções, entre outros [MITRE 2024] [Arusoaie et al. 2017] [Russell et al. 2018] [Steenhoek et al. 2023];
- Vulnerabilidades não detectadas por categoria de ferramentas(pontos cegos) [Russell et al. 2018];
- Vulnerabilidades mais frequentes por ferramenta [Russell et al. 2018] .

**P3.** Como as ferramentas de análise se comportam em ambientes reais?

- Quantidade de vulnerabilidades encontradas;
- Tempo gasto de análise;
- Utilização média dos recursos computacionais (CPU, *Central Processing Unit*, e RAM, *Random Access Memory*);

## 4. Trabalhos Relacionados

### 4.1. *Comparative Evaluation of Approaches & Tools for Effective Security Testing of Web Applications*

Este estudo avalia comparativamente ferramentas e abordagens de testes de segurança de aplicações *web*, utilizando simultaneamente métodos de análise estática (SAST) e análise dinâmica (DAST). Foram testadas 75 aplicações reais com nove ferramentas abertas e gratuitas, mapeando as vulnerabilidades detectadas às listas OWASP Top 10:2021 e CWE Top 25:2023. Os resultados indicam que as ferramentas DAST são mais eficazes em categorias como Broken Access Control e Security Misconfiguration, enquanto ferramentas SAST se destacam na detecção de falhas de alta severidade [Qadir et al. 2025].

Esse artigo serve como base metodológica e comparativa, demonstrando o valor de combinar múltiplas abordagens automatizadas para ampliar a cobertura e precisão na detecção de vulnerabilidades. Ele proporciona uma visão de como ferramentas SAST se comportam em sistemas reais.

### 4.2. *A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code*

O artigo compara diversas ferramentas SAST *open source* aplicadas a código C/C++, com foco na eficácia na detecção de vulnerabilidades. Foram avaliadas ferramentas como Cppcheck, Flawfinder e RATS, considerando métricas de precisão, taxa de falsos positivos e tipos de vulnerabilidade identificados. Os resultados mostram que cada ferramenta tem especializações distintas [Arusoaie et al. 2017].

Esse estudo é relevante para compreender-se como analisar o desempenho de *softwares* SATS. Ele fornece uma visão prática da eficiência e limitações de ferramentas estáticas.

### 4.3. *Automated Vulnerability Detection in Source Code Using Deep Representation Learning*

Este trabalho propõe um sistema automatizado de detecção de vulnerabilidades em código C/C++ baseado em aprendizado de máquina de representações. O modelo utiliza milhões de funções extraídas de bases *open source* (como GitHub e Debian) e aplica técnicas de *deep representation learning* sobre código, combinando uma rede neural convolucional com um classificador Random Forest. O sistema atinge bom desempenho na detecção de falhas associadas a categorias CWE, como Buffer Overflow e NULL Pointer Dereference [Russell et al. 2018].

O artigo é relevante para compreender-se como as ferramentas de aprendizado de máquina são criadas e utilizadas em cenários de detecção de vulnerabilidades. Ademais, ele proporciona uma visão geral de como avaliar esse tipo de sistema.

### 4.4. *Vulnerability Detection with Deep Learning*

O artigo propõe o uso de modelos de *deep learning* para detectar vulnerabilidades em programas binários, com base em análise dinâmica de sequências de chamadas de funções. Foram coletadas 9.872 sequências de execução e os modelos alcançaram acurácia de até 83,6%, superando os métodos tradicionais avaliados. O estudo destaca a capacidade das redes profundas em capturar padrões relacionados a vulnerabilidades [Wu et al. 2017].

Esse estudo explora redes neurais aplicadas à segurança de software, reforçando a importância de técnicas híbridas (estáticas e dinâmicas) e o uso de *deep learning* para aprimorar a precisão e generalização na detecção automática de falhas. Portanto, além de proporcionar uma visão de como ferramentas de ML funcionam nesse contexto, ele demostrar como comparar essa abordagem com algoritmos determinísticos.

### 4.5. *An Empirical Study of Deep Learning Models for Vulnerability Detection*

Este estudo realiza uma análise empírica comparativa de nove modelos de aprendizado profundo para detecção de vulnerabilidades, incluindo arquiteturas GNNs, RNNs, Transformers e CNNs, aplicadas a conjuntos de dados reais (Devign e MSR). Os autores investigam variabilidade entre execuções, impacto do tamanho e da composição dos dados de treino e interpretabilidade dos modelos. Constatam que existe uma grande variação entre eles e que o desempenho depende fortemente do tipo de vulnerabilidade e das informações proporcionadas [Steenhoek et al. 2023].

O artigo contribui para esse trabalho ao oferecer uma visão sobre a robustez, reprodutibilidade e explicabilidade das ferramentas de ML, enfatizando a necessidade de combinar técnicas e ajustar modelos conforme o tipo de falha e contexto do código analisado. Portanto, ele é de externa importância para compreender o estado da arte do uso de aprendizado de máquina na detecção de vulnerabilidades.

## 5. Metodologia

A pesquisa utiliza abordagem comparativa entre ferramentas de análise de código baseadas em algoritmos determinísticos e ferramentas baseadas em aprendizado de máquina (*machine learning* — ML). O método consiste em cinco etapas principais, como ilustrado na Figura 1: seleção das ferramentas, definição do conjunto de dados, mineração e análise de repositórios, normalização dos resultados e comparação quantitativa e qualitativa.

### 5.1. Questões de Pesquisa e Hipóteses

Nessa seção, será explorado as hipóteses sobre as perguntas descritas na Seção 3. Ela será organizadas em subseções, cada uma contendo uma questão.

#### 5.1.1. Qual a precisão de ferramentas de análise de código que utilizam ML em comparação com algoritmos determinísticos?

- **Hipótese nula:** Não há diferença estatisticamente significativa entre o número médio de vulnerabilidades detectadas por ferramentas determinísticas e por ferramentas de aprendizado de máquina.
- **Hipótese alternativa:** Ferramentas de aprendizado de máquina detectam um número significativamente maior de vulnerabilidades que as determinísticas, porém apresentam uma taxa superior de falsos positivos.

#### 5.1.2. Quais tipos de vulnerabilidades são mais detectadas e negligenciadas por cada abordagem?

- **Hipótese nula:** Não há diferença significativa entre as abordagens determinísticas e as de aprendizado de máquina em relação à complexidade das vulnerabilidades detectadas.
- **Hipótese alternativa:** As abordagens determinísticas são mais precisas na detecção de vulnerabilidades simples e estruturais, enquanto as de aprendizado de máquina obtêm melhores resultados na detecção de vulnerabilidades contextuais e dependentes de fluxo, embora com maior incidência de falsos positivos.

#### 5.1.3. Como as ferramentas de análise se comportam em ambientes reais?

- **Hipótese nula:** Ferramentas baseadas em aprendizado de máquina tendem a encontrar menos vulnerabilidades e utilizam muitos mais recursos computacionais.
- **Hipótese alternativa:** Ferramentas determinísticas tendem a encontrar menos vulnerabilidades e utilizam muitos mais recursos computacionais.

### 5.2. Seleção das ferramentas

Selecionaram-se duas ferramentas determinísticas e duas ferramentas baseadas em ML. As determinísticas são Codeql [GitHub 2021] e Snyk [Snyk ], amplamente utilizadas na indústria de software para análise estática (*Static Application Testing* — SAT) e análise de dependências. As ferramentas de ML escolhidas foram o AutoVAS [Jeon and Kim 2021] e o VulCNN [Wu et al. 2022], ambas fundamentadas em técnicas de *deep learning* (*DL*) para detecção automática de vulnerabilidades. As quatro ferramentas foram selecionadas por possuírem documentação pública, código-fonte acessível e relevância em estudos recentes [Arusoaie et al. 2017] [Russell et al. 2018] [Steenhoek et al. 2023].

### 5.3. Base de dados

A avaliação é conduzida utilizando a *Software Assurance Reference Dataset* (SARD), do *National Institute of Standards and Technology* [NIST ], que contém projetos *open-source* escritos em C e C++, com vulnerabilidades catalogadas conforme o padrão *Common Weakness Enumeration* (CWE). Essa base foi escolhida por permitir a comparação entre diferentes abordagens de análise de código em condições controladas e reproduzíveis.

Para enriquecer os resultados desse estudo, foram utilizados, também, repositórios C/C++ do GitHub [GitHub, Inc. ]. Eles foram utilizados para comparar o desempenho de cada ferramenta em ambientes reais.

### 5.4. Procedimentos de mineração e análise

Foram minerados mil casos de teste do SARD [NIST ], todos em linguagens C/C++. Cada repositório foi submetido à análise pelas quatro ferramentas selecionadas. Em seguida, coletaram-se as métricas de desempenho de cada abordagem: (i) taxa de vulnerabilidades detectadas, (ii) taxas de falsos positivos e falsos negativos, (iii) tempo médio de execução, (iv) utilização média de *Central Processing Unit* (CPU, em português Unidade Central de Processamento — UCP) e (v) consumo médio de memória.

Após isso, foram extraídos os 140 repositórios mais populares de C/C++ no GitHub [GitHub, Inc. ] que foram submetidos a mesma análise realizada anteriormente.

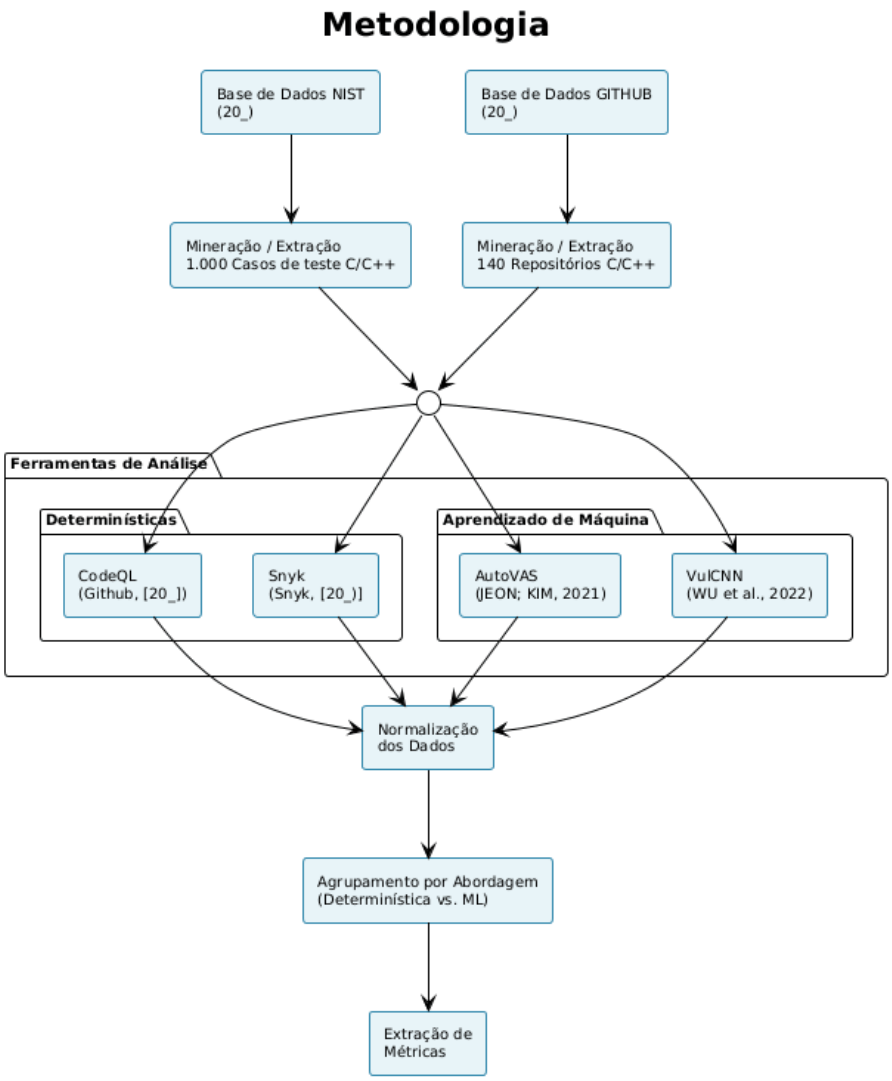


**Figure 1. Metodologia**

### 5.5. Normalização e agrupamento dos dados

A normalização dos dados foi feita através de *scripts* Python que fazem a comparação com as vulnerabilidades esperadas, realizam a sintetizam para análise e, por fim, criam um relatório final em Markdown. Após isso, os dados são transferidos para uma planilha e, através do Power BI [Microsoft Corporation ], eles são agrupados e a visualização dos resultados é montada.

### 5.6. *Setup* Experimental

O experimento foi conduzido de forma controlada, aplicando as quatro ferramentas selecionadas sobre dois conjuntos de dados: a base SARD/NIST [NIST ], com mil casos de teste rotulados, e 140 repositórios reais do GitHub [GitHub, Inc. ] em C/C++.

### 5.6.1. Variáveis independentes

- Tipo de abordagem (determinística vs. aprendizado de máquina)
- Tipo de vulnerabilidade (categoria CWE)
- Tamanho e origem do código (SARD vs. GitHub)

### 5.6.2. Variáveis dependentes

- Número de vulnerabilidades detectadas
- Taxa de falsos positivos
- Taxa de falsos negativos
- Precisão e cobertura obtidos por cada abordagem/ferramenta
- Consumo médio de CPU e RAM

## 5.7. Síntese metodológica

A Figura 2 apresenta uma visão geral da estrutura metodológica e das fontes bibliográficas que sustentam cada grupo de ferramentas e conceitos. Nela, observa-se que as referências bibliográficas foram organizadas em três blocos conceituais: vulnerabilidades (em vermelho), ferramentas determinísticas (em verde) e ferramentas baseadas em aprendizado de máquina (em azul). Essa estrutura orientou a coleta e análise dos dados, garantindo que cada grupo de ferramentas fosse contextualizado com sua respectiva base teórica.

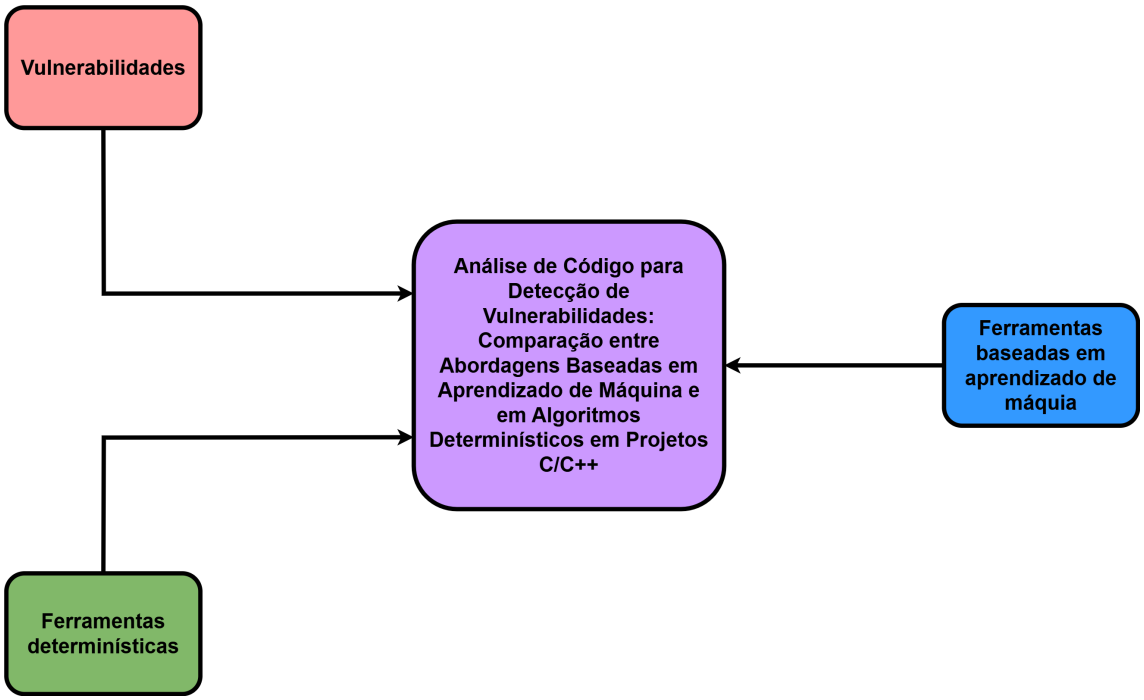


**Figure 2. Síntese metodológica do estudo: relação entre vulnerabilidades, ferramentas determinísticas e ferramentas baseadas em aprendizado de máquina.**

# 6. Resultados

Nessa seção, são relatados os resultados encontrados durante a execução dos experimentos. Ela foi organizada de modo a responder às perguntas estabelecidas no modelo GQM,

descritas na Seção 3. Cada subseção apresenta as métricas e evidências que fundamentam a análise comparativa entre as abordagens determinísticas (CodeQL e Snyk) e baseadas em aprendizado de máquina (AutoVAS e VulCNN).

### 6.1. Qual a precisão de ferramentas de análise de código que utilizam ML em comparação com algoritmos determinísticos?

Essa pergunta foca no desempenho geral de cada uma das abordagens estudadas, a fim de compreender a confiabilidade e cobertura de cada uma delas. Para determinar isso, foram utilizados os dados das análises realizadas nos códigos do SARD [NIST ].

Na Figuras 3 é possível visualizar a taxa detecção de cada tipo de ferramenta, sendo notável a superioridade dos sistemas de ML nisso quesito. No entanto, como mostrado nas Figuras 9, 10 e 11, eles gastam muito mais recursos computacionais.

Apesar de diversos modelos de ML alucinarem algumas vezes [Ji et al. 2023], nesse estudo foi evidenciado que as ferramentas dessa categoria encontraram menos falsos positivos que as abordagens determinísticas, como é possível visualizar na Figura 4. Ademais, os *softwares* baseados em aprendizado de máquina tiveram uma taxa de falsos negativos menor que suas contra parte determinísticas, como mostrado na Figura 5.

Por fim, nas Figuras 6 e 7 são listadas as detecções únicas de cada ferramenta e abordagem, então sendo possível evidenciar, novamente, que os sistemas de ML se sobressaíram nessa métrica. Assim, é possível concluir que eles tendem a balancear bem confiabilidade e cobertura, como demonstrado na Figura 8.

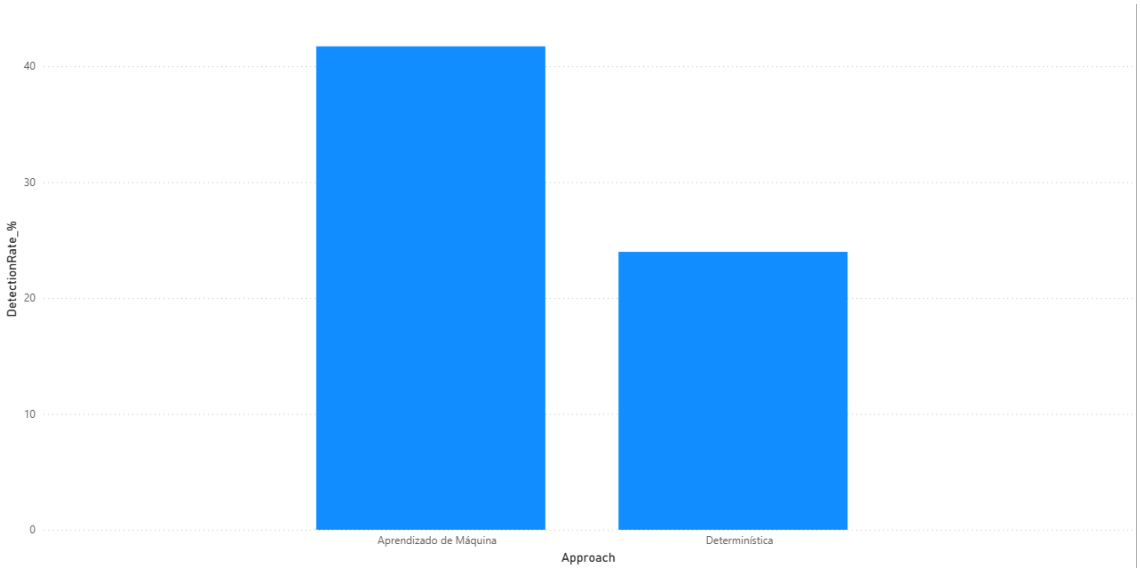


**Figure 3. Taxa de vulnerabilidades encontradas por ferramenta**

### 6.2. Quais tipos de vulnerabilidades são mais detectadas e negligenciadas por cada abordagem?

As Figuras 12 e 13 fazem uma síntese para essa questão, sendo possível identificar que os pontos fortes e fracos de cada ferramenta e abordagem. Os *softwares* determinísticas são mais precisas nas detecções de vulnerabilidades como CWE416, CWE415 e CWE78,

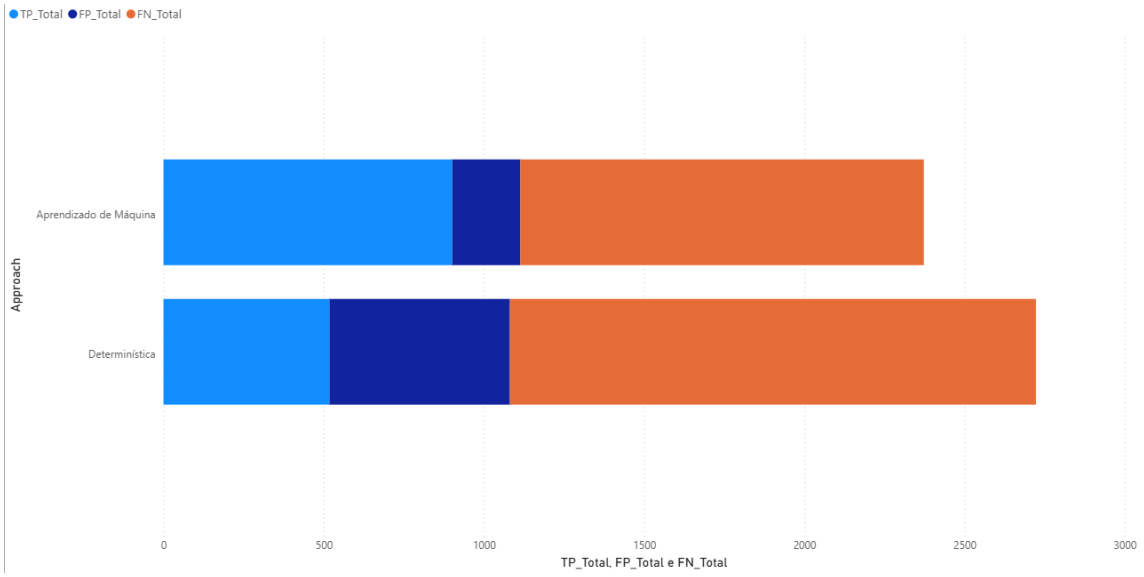


**Figure 4. Taxa de falsos positivos/negativos por ferramenta**

como evidenciado pela Figura 15. Enquanto isso, as abordagens baseadas em aprendizado de máquina tendem a encontrar mais problemas como CWE122 e CWE369.

Apesar dos pontos fortes, todas as ferramentas também possuem fraquezas, como exemplificado pela Figura 14 e 13. As abordagens determinísticas tem um baixo desempenho em falhas como CWE369, enquanto os *sistemas* de ML tem dificuldade em encontrar vulnerabilidades como CWE476 e CWE775.

Portanto, apesar das ferramenta baseadas em aprendizado de máquina possuírem mais capacidade de detectar certos problemas, elas pecam muito é várias outras categorias. Enquanto as abordagens determinísticas tendem a ter um desempenho similar na maioria das vulnerabilidades CWE.

### 6.3. Como as ferramentas de análise se comportam em ambientes reais?

Para responder essa questão, foram utilizados os 140 repositórios C/C++ do Github [GitHub, Inc. ] como base para a geração dos resultados. Todas as ferramentas selecionadas analisaram cada um deles, a fim de compreender seu desempenho em casos reais, mas sem determinar sua eficácia.

Na Figura 16 é possível observar que as ferramentas baseadas em aprendizado de máquina encontraram mais vulnerabilidades e levando menos tempo para executar, como visto nas Figuras 17 e 18. No entanto, elas utilizaram mais poder computacional, assim como mostra a Figura 19, mesmo que o pico de uso da CPU tenha sido parecido para as duas abordagens analisadas. Por fim, os sistemas determinísticos necessitaram mais RAM para executar, como mostrando na Figura 20, demonstrando que eles possuem algoritmos mais pesados, embora sejam mais consolidados no mercado.

## 7. Limitações do Estudo

As principais limitações do estudo está relacionada à quantidade e ao escopo das ferramentas analisadas. Foram avaliadas apenas quatro ferramentas, sendo elas duas determinísticas e duas baseadas em aprendizado de máquina, o que não representa a totalidade

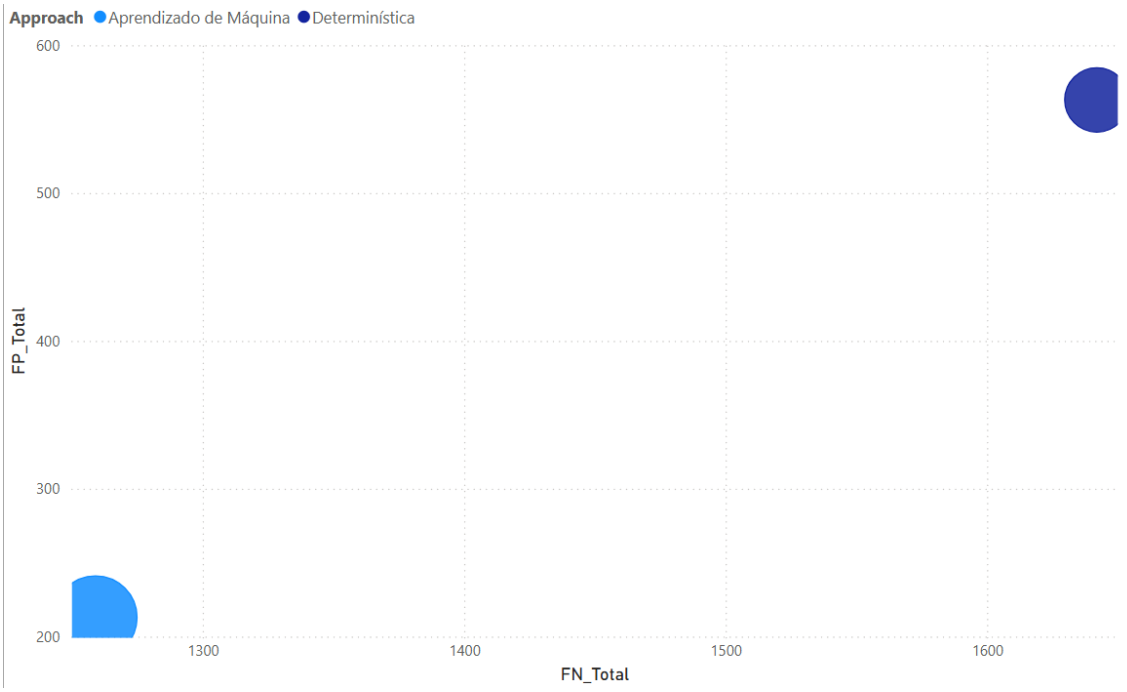


**Figure 5. Taxa de falsos positivos/negativos por ferramenta**

das soluções existentes no mercado. No entanto, existem ferramentas emergentes, tanto comerciais quanto de código aberto, que podem apresentar resultados distintos em termos de precisão, desempenho e cobertura de vulnerabilidades.

Outra limitação refere-se à linguagem de programação adotada na base experimental. A pesquisa foi centralizada em repositórios desenvolvidos em C e C++, o que restringe a generalização dos resultados para outras linguagens. Sendo assim, abordagens aplicadas a contextos como Java, Python ou JavaScript podem apresentar comportamentos diferentes devido a características estruturais e de tipagem próprias de cada linguagem.

Além disso, apesar a análise foi conduzida de forma controlada, utilizando a base *Software Assurance Reference Dataset* (SARD), o que garante reprodutibilidade, mas não reflete completamente o ambiente de desenvolvimento de sistemas reais. Apesar de serem utilizados repositórios reais para a análise de desempenho, a precisão da análise de cada ferramenta neles não foi medida. Portanto, os resultados obtidos devem ser interpretados considerando essas limitações metodológicas e contextuais.

## 8. Trabalhos Futuros

Como continuação desta pesquisa, propõe-se ampliar o escopo de ferramentas e linguagens de programação analisadas. A inclusão de novas abordagens determinísticas e modelos de aprendizado profundo (*deep learning*) pode fornecer uma visão mais abrangente sobre a eficácia e os limites de cada técnica de detecção de vulnerabilidades.

Outra possibilidade consiste em aplicar o método proposto em projetos de *software* reais, avaliando a capacidade das ferramentas de identificar falhas em cenários dinâmicos, com bases de código em constante evolução. Essa etapa permitiria verificar, na prática, se o uso combinado de ferramentas determinísticas e de aprendizado de máquina

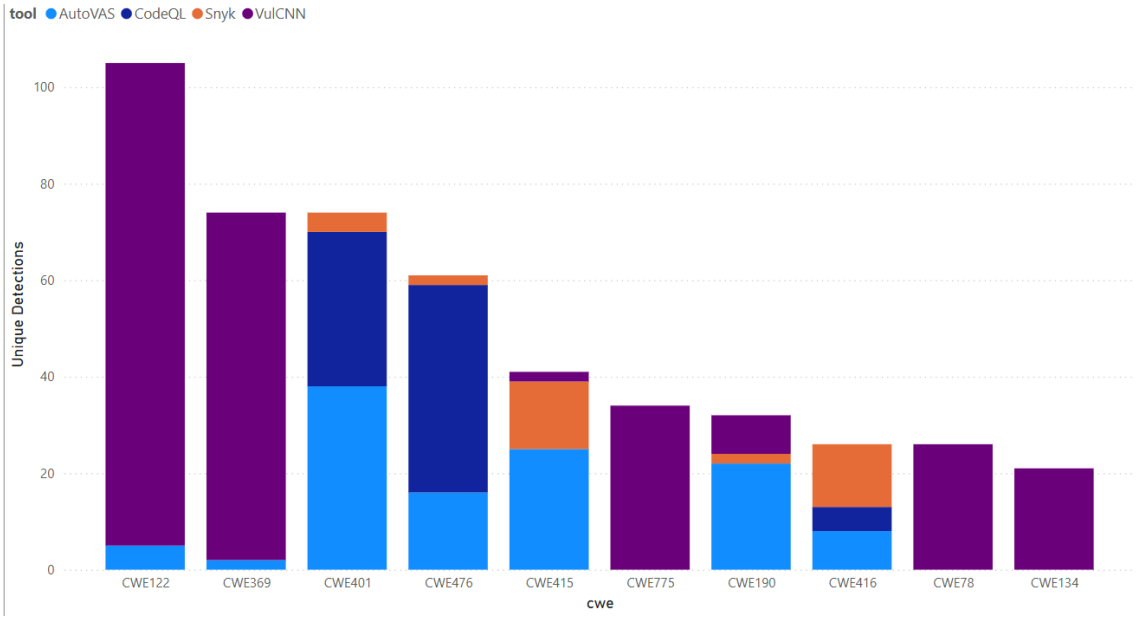


**Figure 6. Cobertura comparada com a confiabilidade**

resulta em redução efetiva de vulnerabilidades durante o ciclo de desenvolvimento.

Também, é sugerido explorar a integração entre análise estática e dinâmica, a fim de desenvolver uma estrutura híbrida de detecção automatizada que una precisão, cobertura e eficiência. Por fim, recomenda-se a realização de estudos comparativos longitudinais, a fim de observar a evolução das ferramentas e o impacto das atualizações de modelos de aprendizado sobre sua eficácia.

## 9. Conclusão

## 10. Referências

## References

Abdullahi, S. et al. (2020). Software testing: review on tools, techniques and challenges. *International Journal of Advanced Research in Technology and Innovation*, 2(2):11–18.

Albee, A. et al. (2000). Report on the loss of the mars polar lander and deep space 2 missions. Technical report.

Arusoaie, A. et al. (2017). A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *Proceedings of the 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 161–168, Timisoara. IEEE.

Emanuelsson, P. and Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21.

Garousi, V. et al. (2020). Exploring the industry's challenges in software testing: an empirical study. *Journal of Software: Evolution and Process*, 32(8):e2251.

GitHub (2021). Codeql — descubra vulnerabilidades em toda a base de código. Acesso em: 22 out. 2025.

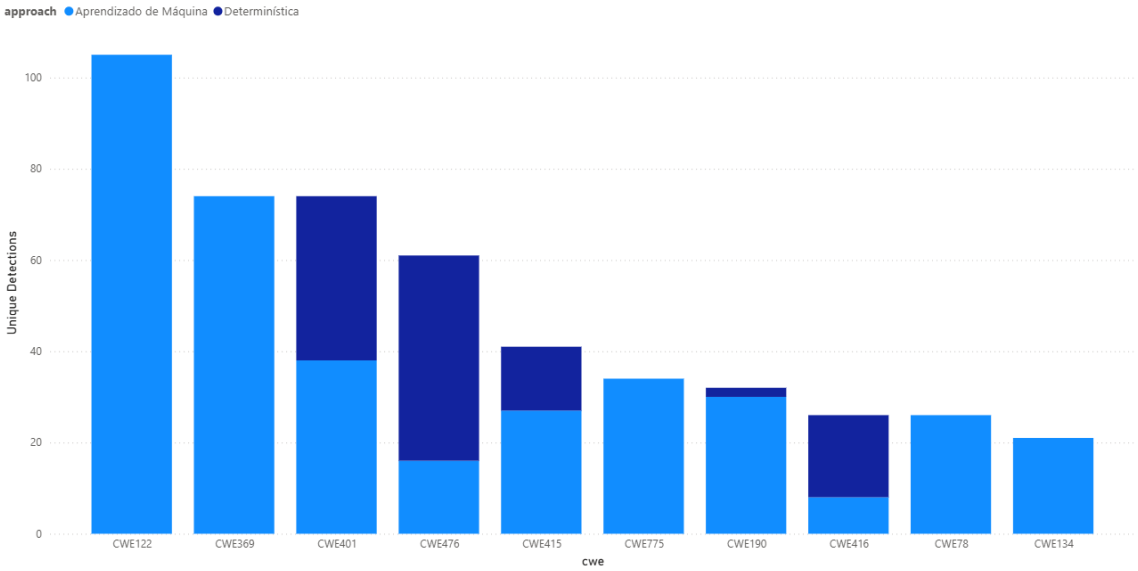


**Figure 7. Cobertura comparada com a confiabilidade**


GitHub, Inc. GitHub: Where the world builds software. `https://github.com/`. acessado em 11 nov 2025.

Jeon, S. and Kim, H. (2021). Autovas: An automated vulnerability analysis system with a deep learning approach. *Computers & Security*, 106.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Comput. Surv.*, 55(12).

Levenson, N. and Turner, C. (1993). An investigation of the therac-25 accidents. *Computer*, 26(7):18–41.

Maskur, A. and Asnar, Y. (2019). Static code analysis tools with the taint analysis method for detecting web application vulnerability. In *Proceedings of the International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, Pontianak. IEEE.

Microsoft Corporation. Power BI – Visualização de Dados — Microsoft Power Platform. `https://www.microsoft.com/pt-br/power-platform/products/power-bi`. acessado em 11 nov 2025.

MITRE (2024). Cwe—2024 cwe top 25 most dangerous software weaknesses. Acesso em: 8 out. 2025.

NIST. Software assurance reference dataset (sard). Acesso em: 8 out. 2025.

Qadir, S. et al. (2025). Comparative evaluation of approaches & tools for effective security testing of web applications. *PeerJ Computer Science*, 11:e2821.

Russell, R. et al. (2018). Automated vulnerability detection in source code using deep representation learning. In *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, Orlando. IEEE.

Shirey, R. (2007). Rfc 4949: Internet security glossary, version 2. Technical report, IETF, Fremont. Acesso em: 8 out. 2025.

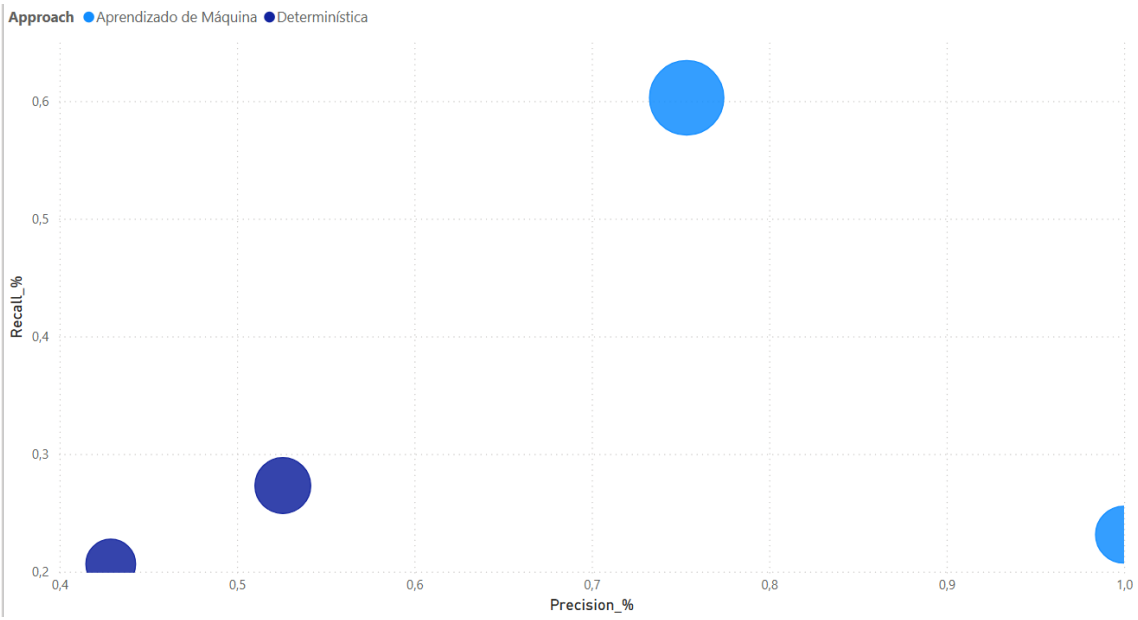


**Figure 8. Cobertura comparada com a confiabilidade**

Snyk. Snyk — segurança de código, dependências, contêineres e infraestrutura. Acesso em: 8 out. 2025.

Steenhoek, B. et al. (2023). An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 2237–2248, Melbourne. IEEE.

Sánchez, M. et al. (2020). Software vulnerabilities overview: A descriptive study. *Tsinghua Science and Technology*, 25(2):270–280.

Wu, F. et al. (2017). Vulnerability detection with deep learning. In *Proceedings of the 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 1298–1302, Chengdu. IEEE.

Wu, Y., Zou, D., Dou, S., Yang, W., Xu, D., and Jin, H. (2022). Vulcnn: an image-inspired scalable vulnerability detection system. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2365–2376, New York, NY, USA. Association for Computing Machinery.

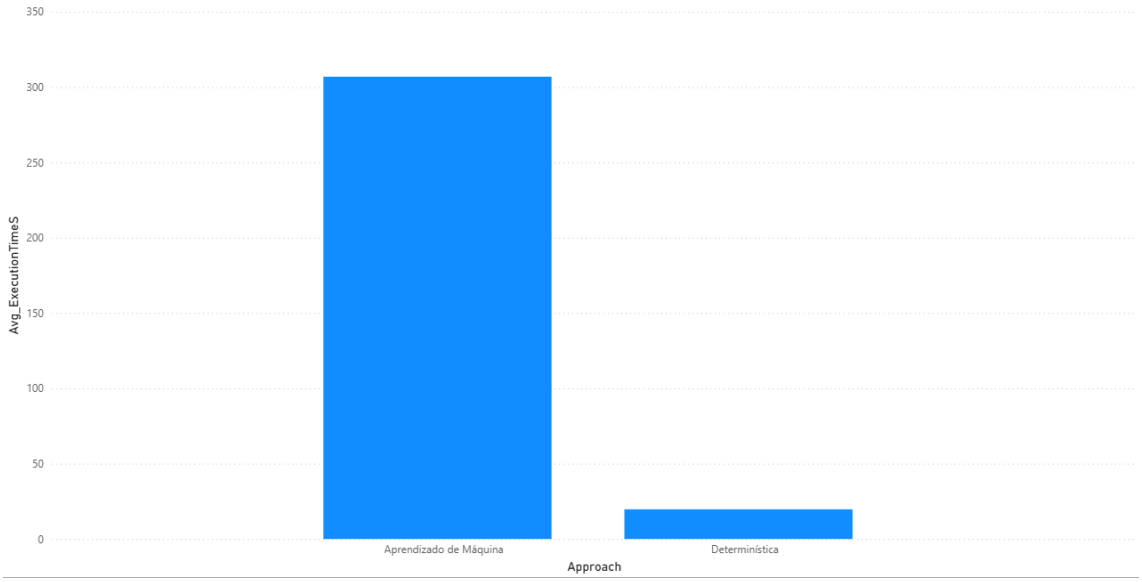


**Figure 9. SARD - Tempo de execução de cada ferramenta**

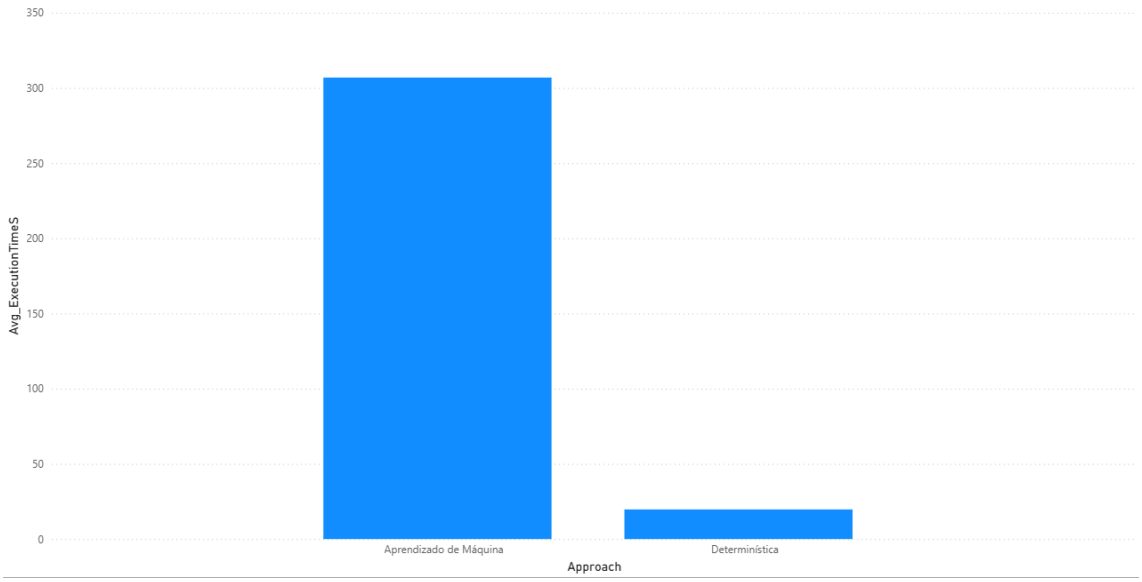


**Figure 10. SARD - Uso médio de CPU de cada ferramenta**

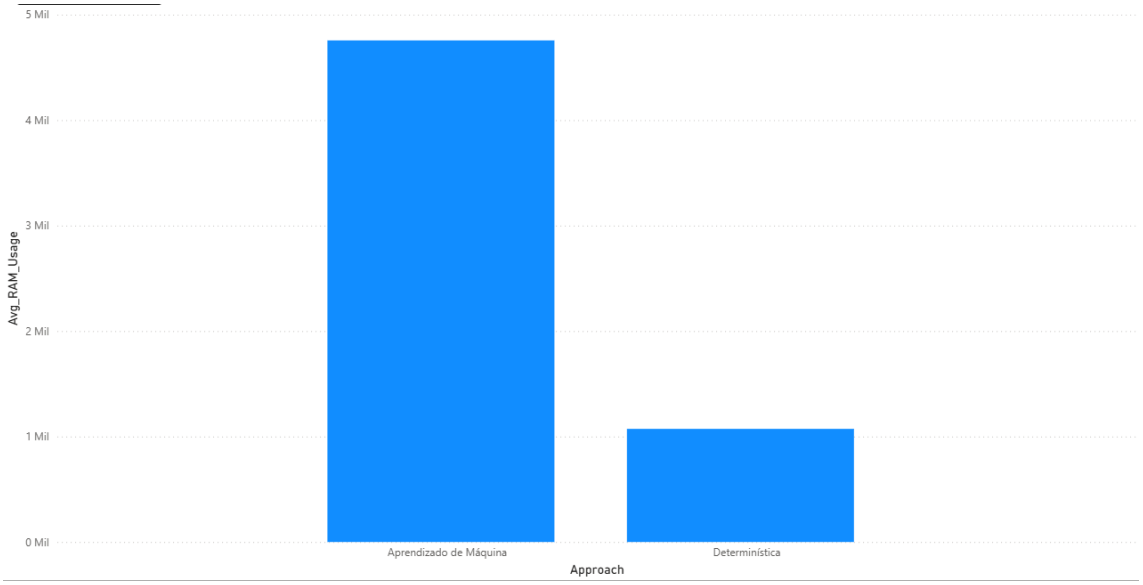


**Figure 11. SARD - Uso médio de RAM de cada ferramenta**

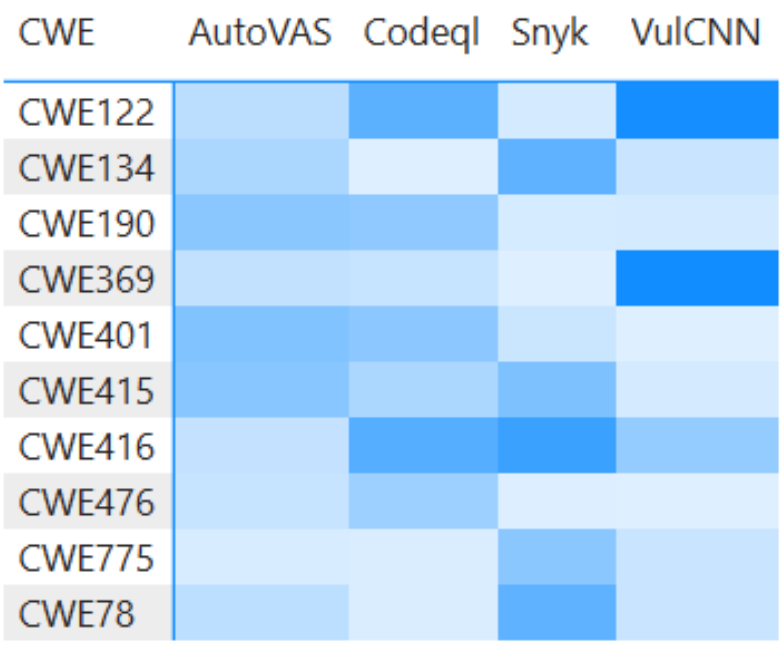


**Figure 12. Heatmap de Categoria CWE por ferramenta**

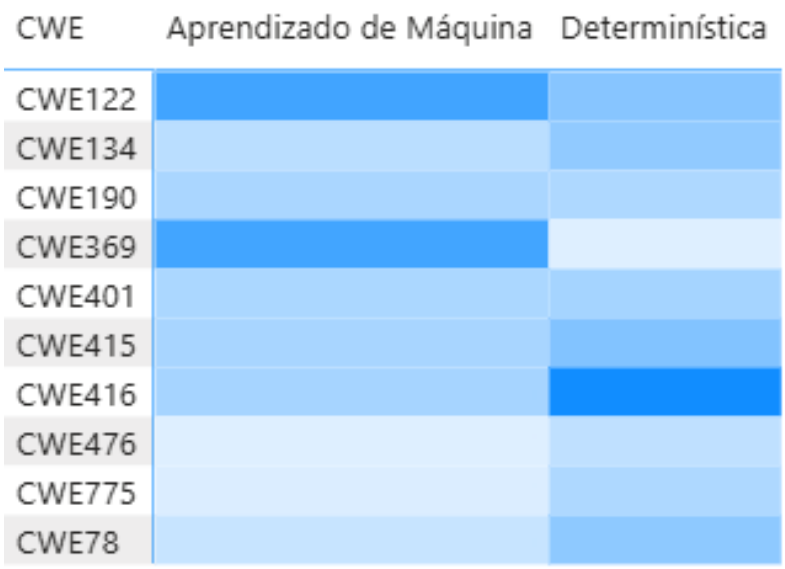


**Figure 13. Heatmap de Categoria CWE por abordagem**

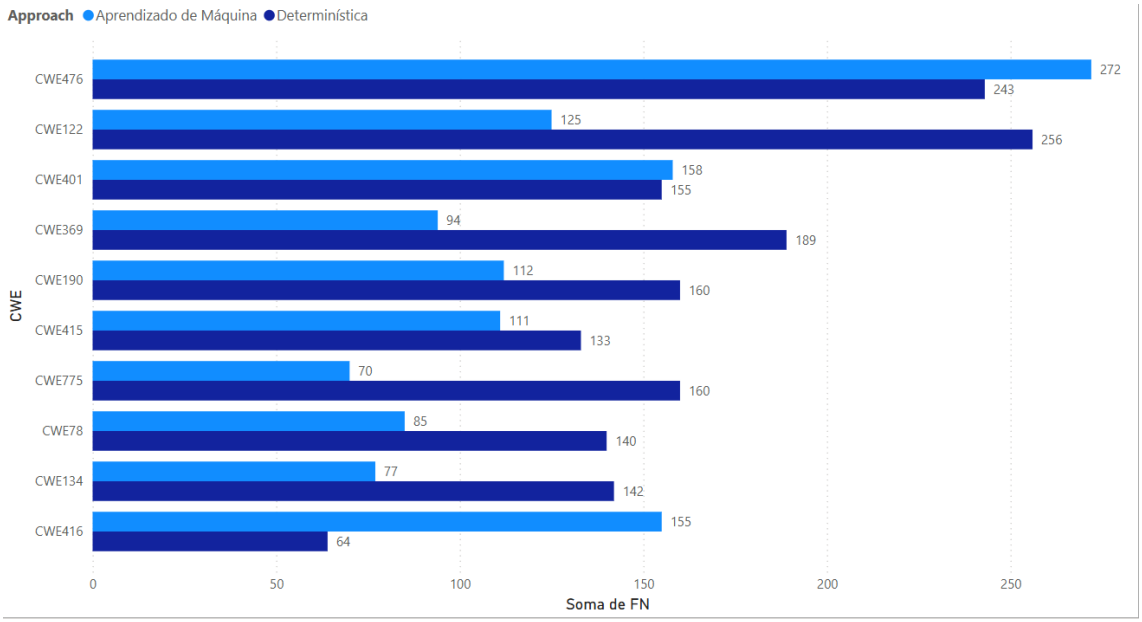


**Figure 14. Vulnerabilidades não encontradas por cada ferramenta**

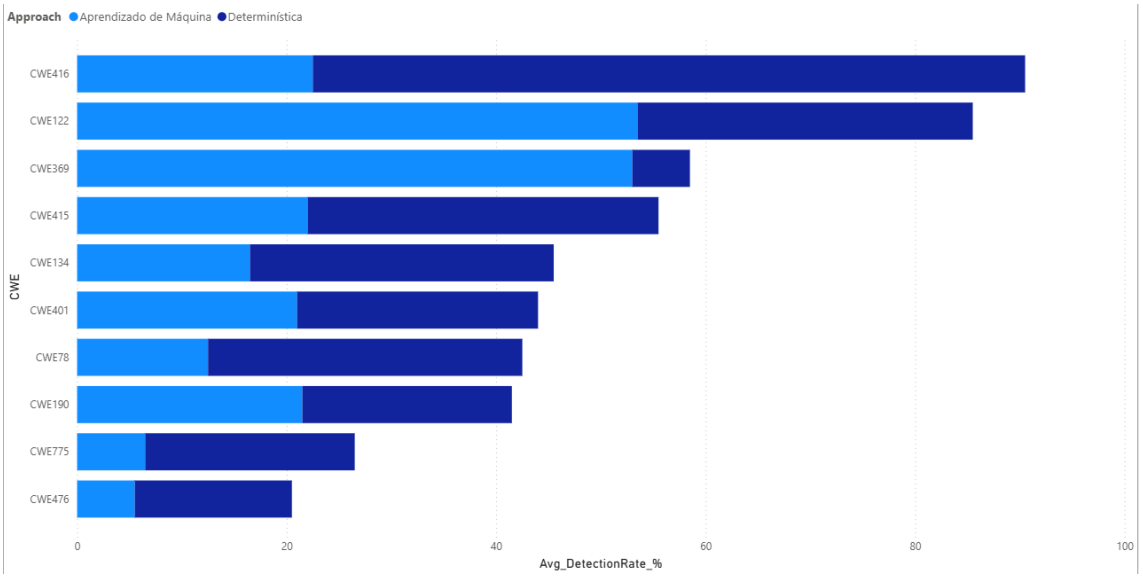


**Figure 15. Detecções únicas de cada abordagem por categoria CWE**

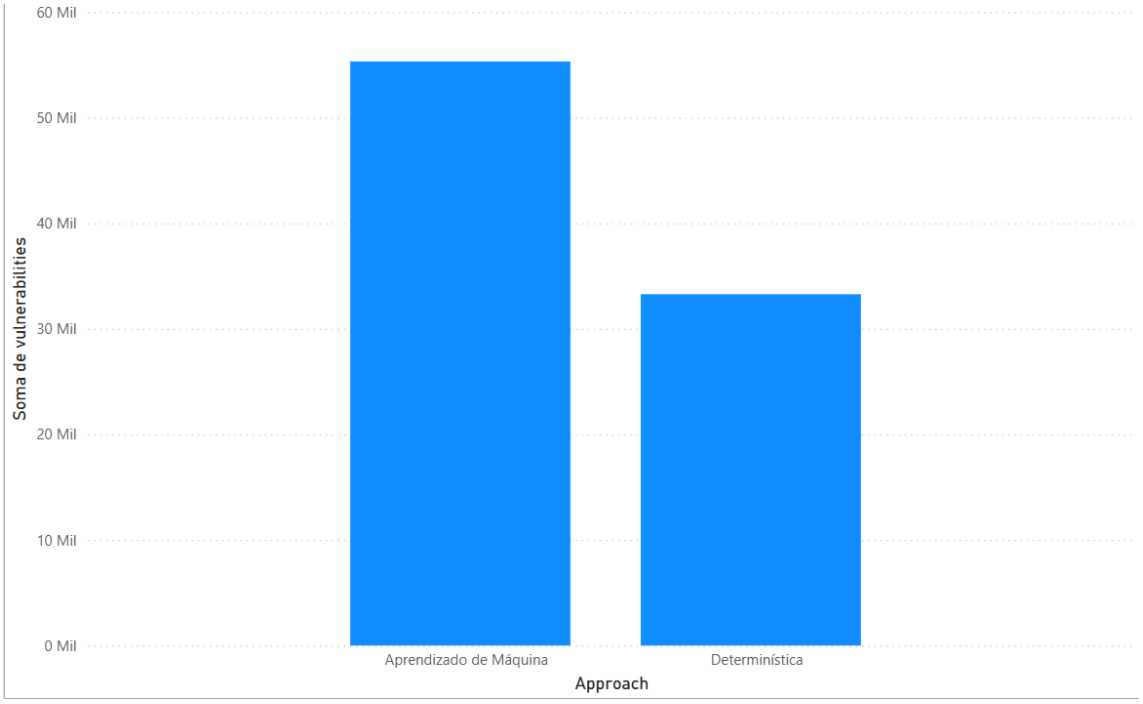


**Figure 16. Quantidade de vulnerabilidades encontradas por abordagem**

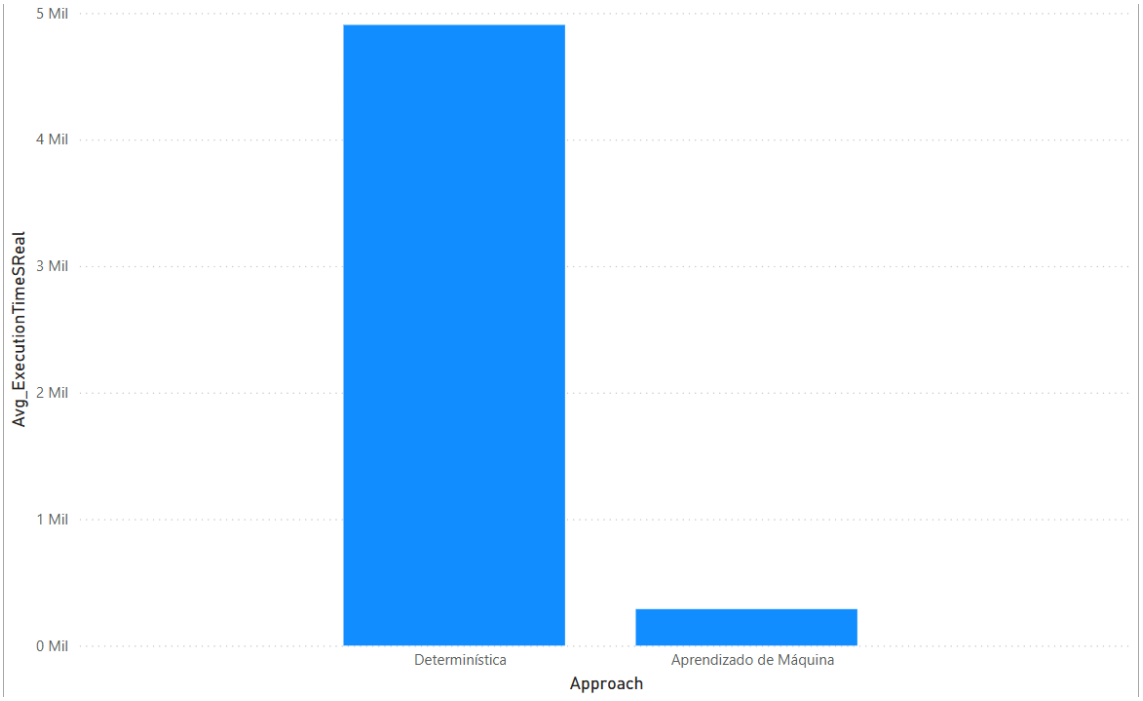


**Figure 17. Tempo de execução de cada análise**

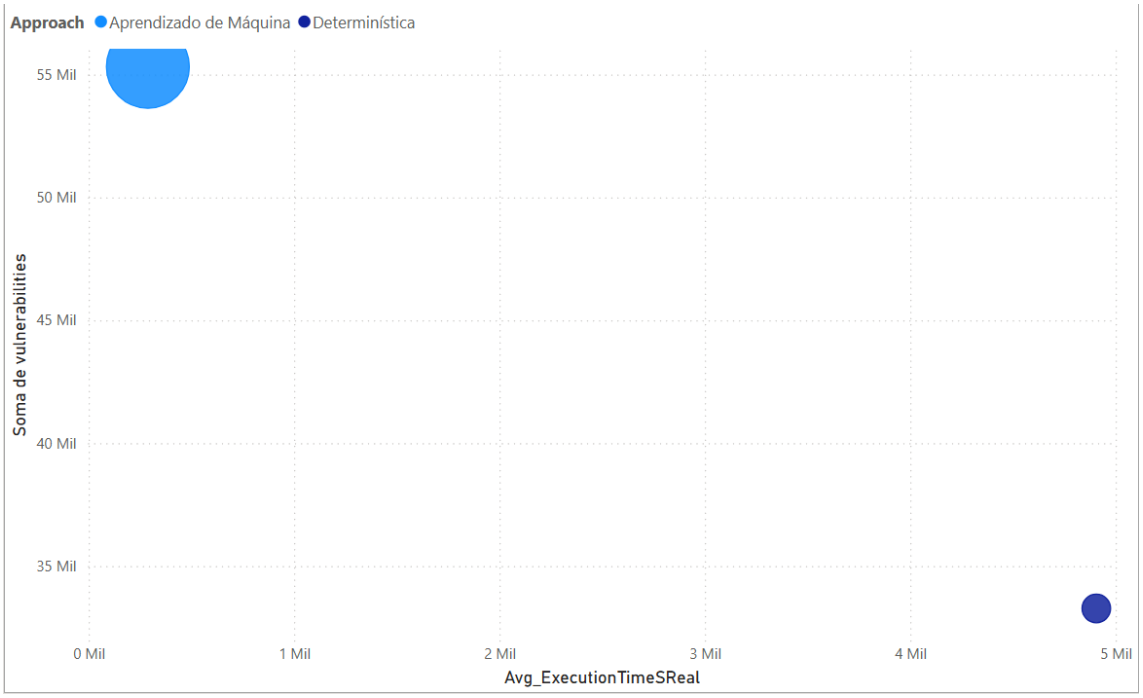


**Figure 18. Vulnerabilidades encontradas x tempo de execução x uso médio de cpu**

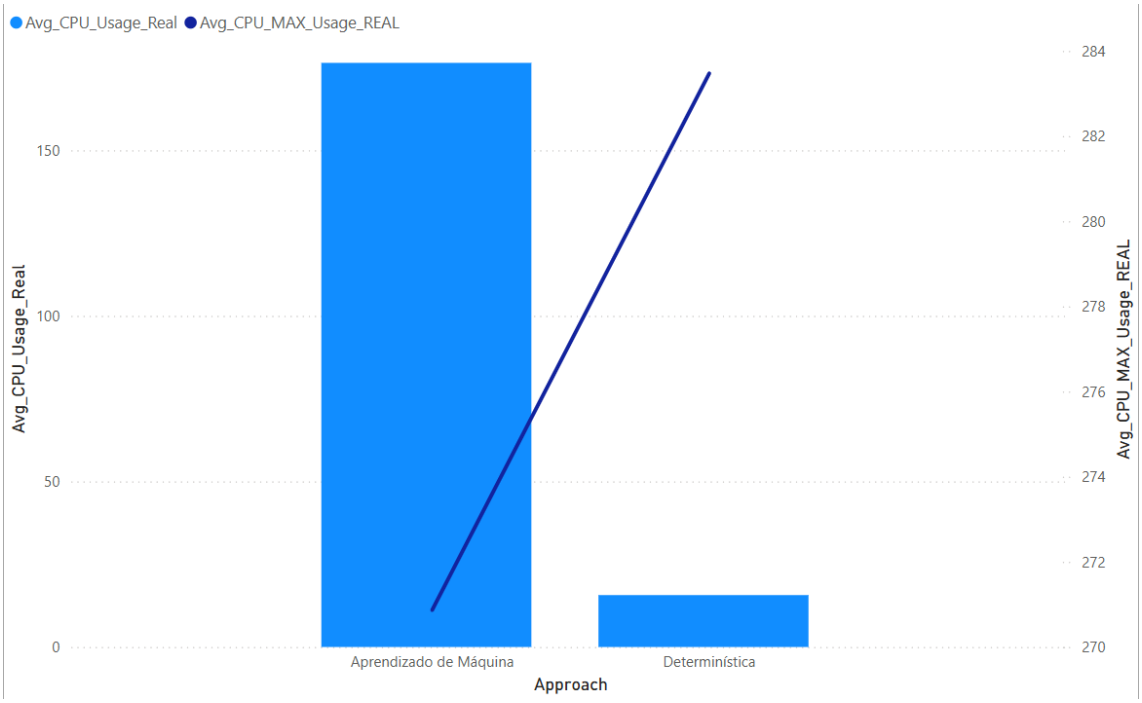


**Figure 19. Uso médio de CPU por abordagem**

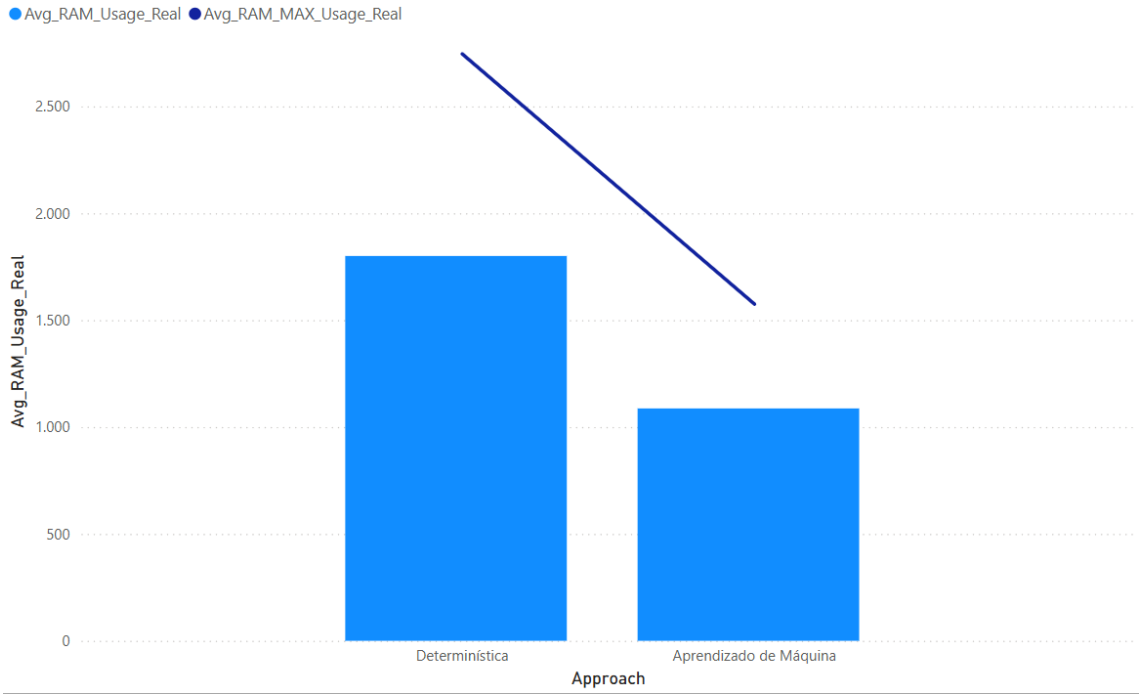


**Figure 20. Uso médio de RAM por abordagem**