Análise de Código para Detecção de Vulnerabilidades: Comparação entre Abordagens Baseadas em Aprendizado de Máquina e em Algoritmos Determinísticos em Códigos C/C++

Ana Carolina Caldas de Mello[1]
Gustavo Menezes Barbosa[1]
João Pedro Queiroz Rocha[1]
Maria Eduarda Gonçalves de Souza Ferreira[1]
Pedro Henrique Dias Camara[1]
Wanessa Dias Costa[1]

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

{ana.caldas[1], gustavo.barbosa.1386677[1],
joao.rocha.1439661[1], 1380383[1], pcamara[1],
wanessa.costa[1]}@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. The results showed that machine learning-based approaches outperformed deterministic ones in terms of coverage and balance between precision and recall. The machine learning approachs achieved the best performance, reaching 41,71% detection and an F1-score of 0.55. Deterministic tools, such as CodeQL and Snyk, exhibited lower sensitivity but greater stability and integration within development pipelines. It is concluded that combining deterministic and machine learning techniques is the most effective strategy for vulnerability detection, achieving an optimal balance between reliability, coverage, and computational efficiency.*

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. Os resultados demonstraram que as abordagens baseadas em aprendizado de máquina superaram as determinísticas em cobertura e equilíbrio entre precisão e recall. As abordagens de aprendizado de máquia obteveram desempenho superior, alcançando 41,71% de detecção e F1-score de 0,55. Já as ferramentas determinísticas, CodeQL e Snyk, mostraram menor sensibilidade, mas maior estabilidade e integração com pipelines de desenvolvimento. Conclui-se que a combinação de técnicas determinísticas e de aprendizado de máquina representa a estratégia mais eficaz para detecção de vulnerabilidades, equilibrando

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. Organização do texto

A Seção 4 apresenta as perguntas de pesquisa, enquanto a Seção 5 apresenta 5 trabalhos fortemente relacionados com essa pesquisa. A Seção 6 descreve a metodologia e a Seção 7 revela os resultados obtidos. As Seções 8 e 9 se referem às limitações e aos trabalhos futuros, e a Seção 10 revela a conclusão.

3. 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++.

4. 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*);

5. Trabalhos Relacionados

5.1. *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.

5.2. *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.

5.3. *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.

5.4. *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.

5.5. *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.

6. 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 foi estruturado de forma a garantir reprodutibilidade, equilíbrio entre precisão e profundidade de análise, e consistência na coleta de métricas. Ele 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.

6.1. Questões de Pesquisa e Hipóteses

Nessa seção, será explorado as hipóteses sobre as perguntas descritas na Seção 4. Ela será organizadas em subseções, cada uma contendo uma questão.

6.1.1. 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.

6.1.2. 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.

6.1.3. 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.

6.2. 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.

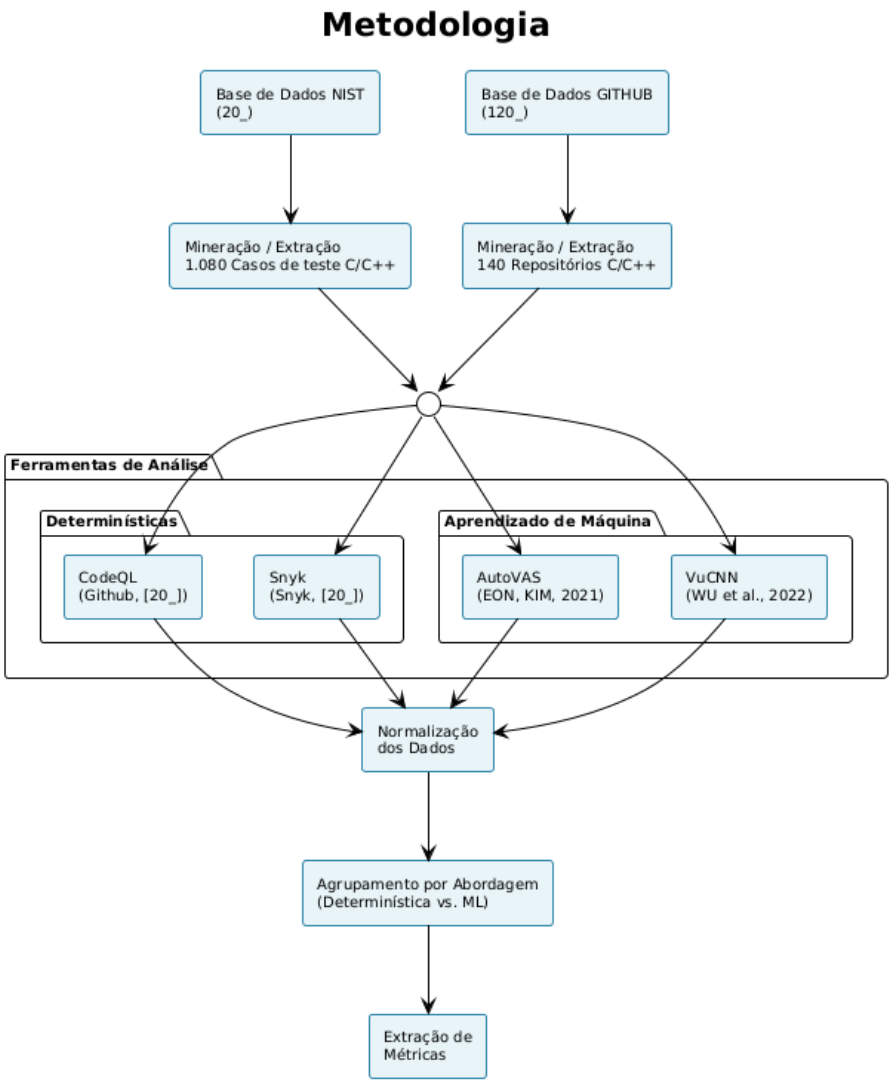

Figure 1. Metodologia

As ferramentas foram selecionadas considerando critérios de disponibilidade pública, documentação técnica, reprodutibilidade, relevância em pesquisas recentes e diversidade metodológica [Arusoaie et al. 2017] [Russell et al. 2018] [Steenhoek et al. 2023].. Dessa forma, buscou-se abranger tanto soluções amplamente adotadas pela indústria quanto modelos de pesquisa que representam o estado da arte em detecção de vulnerabilidades.

6.3. 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). Cada amostra do *dataset* associa um fragmento de código a uma categoria de vulnerabilidade específica, o que permite avaliar a precisão e a cobertura das ferramentas sob condições controladas.

O uso do SARD se justifica pela sua ampla adoção na literatura, pela reprodutibil-

idade experimental e pela possibilidade de mensurar métricas objetivas de desempenho. Entretanto, o conjunto também apresenta desafios, como a presença de vulnerabilidades sintéticas e variação de complexidade entre exemplos.

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.

6.4. Procedimentos de mineração e análise

Foram minerados 1.080 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:

1. Quantidade de vulnerabilidades detectadas corretamente (verdadeiros positivos — TP);
2. Quantidade de vulnerabilidades não detectadas (falsos negativos — FN);
3. Quantidade de vulnerabilidades incorretamente sinalizadas (falsos positivos — FP);
4. Tempo médio de execução (em segundos);
5. Utilização média da unidade central de processamento (CPU);
6. Consumo médio de memória (RAM).

As detecções foram posteriormente agrupadas por categoria CWE, possibilitando a comparação entre os tipos de vulnerabilidade que cada ferramenta identificou com maior frequência ou deixou de detectar.

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

6.5. Normalização e agrupamento dos dados

Após a coleta, os resultados foram submetidos a um processo de normalização para garantir a comparabilidade entre ferramentas. As métricas de desempenho (Precisão, Cobertura e F1-score) foram calculadas conforme as fórmulas:

$$Precisão = \frac{VerdadeirosPositivos}{VerdadeirosPositivos + FalsosPositivos} \tag{1}$$

$$Cobertura = \frac{VerdadeirosPositivos}{VerdadeirosPositivos + FalsosNegativos} \tag{2}$$

$$F1 = 2 \times \frac{Precisão \times Cobertura}{Precisão + Cobertura} \tag{3}$$

Os valores foram normalizados em relação ao total de vulnerabilidades esperadas para cada CWE, conforme o *ground truth* fornecido pela base SARD. Em seguida, foram calculadas médias ponderadas por categoria, levando em consideração o número de amostras de cada tipo de vulnerabilidade. Esse processo permitiu identificar tanto o desempenho geral das ferramentas quanto a especialização de cada uma em determinadas classes de falhas.

As métricas finais foram agrupadas em dois níveis de comparação: (i) por ferramenta individual, para avaliar desempenho isolado, e (ii) por abordagem (determinística versus aprendizado de máquina), permitindo identificar padrões comportamentais entre os dois paradigmas de análise.

Por fim, foram utilizados *scripts* Python para fazer a comparação com as vulnerabilidades esperadas, realizar a sintetizam para análise e criar um relatório final em Markdown. Após isso, os dados foram transferidos para uma planilha e, através do Power BI [Microsoft Corporation], eles foram agrupados e a visualização dos resultados foi montada.

6.6. *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++.

6.6.1. Variáveis independentes

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

6.6.2. 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

6.7. 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.

7. 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 4. 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).

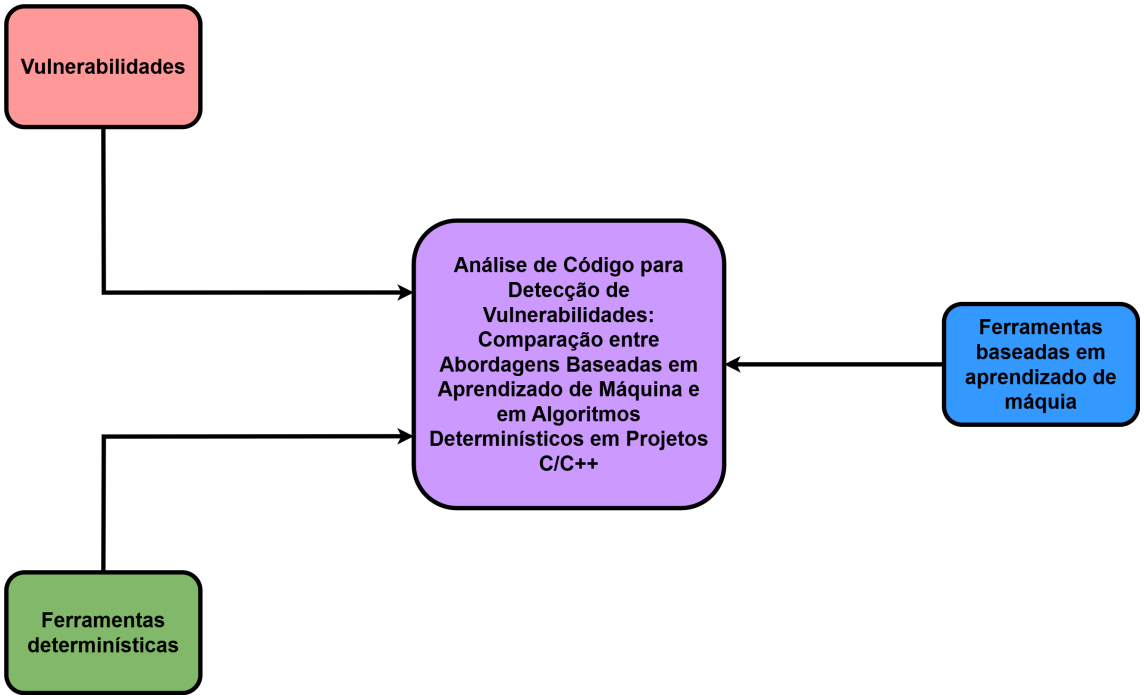

Figure 2. Síntese metodológica do estudo: relação entre vulnerabilidades, ferramentas determinísticas e ferramentas baseadas em aprendizado de máquina.

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

Esta questão busca comparar o desempenho geral de cada abordagem, observando a precisão (*precision*) e a capacidade de cobertura (*recall*) das abordagens. Todas as métricas foram calculadas a partir das análises realizadas sobre o conjunto SARD [NIST], composto por 1080 vulnerabilidades conhecidas.

A Figura 3 demonstra a taxa de detecção de cada ferramenta. Observa-se que as abordagens de aprendizado de máquina apresentaram o melhor desempenho, com taxa de detecção de 41,71% e *F1-score* de 0,55, seguido pelas outras ferramentas (23,98%). Essa diferença reflete a capacidade dos modelos baseados em *deep learning* de identificar padrões complexos de vulnerabilidade a partir de contextos semânticos do código.

Em contrapartida, a Figura 4 evidencia que, apesar da menor cobertura, o AutoVAS atingiu precisão de 100%, não gerando nenhum falso positivo, um resultado superior às demais ferramentas, que apresentaram quantidades expressivas de detecções incorretas (CodeQL: 297; Snyk: 266; VulCNN: 386). Além disso, a Figura 5 demonstra que os modelos de aprendizado de máquina apresentaram menor taxa de falsos negativos do que as abordagens determinísticas, indicando que tendem a capturar um número maior de vulnerabilidades reais. A Figura 6 reforça essa tendência, mostrando que as ferramentas de ML também apresentaram mais detecções únicas, ou seja, encontraram falhas não detectadas pelas ferramentas determinísticas.

Por fim, a Figura 7 apresenta o equilíbrio entre precisão e cobertura, destacando o melhor balanceamento obtido pelas abordagens de aprendizado de máquina. Apesar disso, as Figuras 8, 9 e 10 revelam que o custo computacional dos modelos de ML foi

substancialmente maior, exigindo mais tempo, CPU e memória para executar as análises.

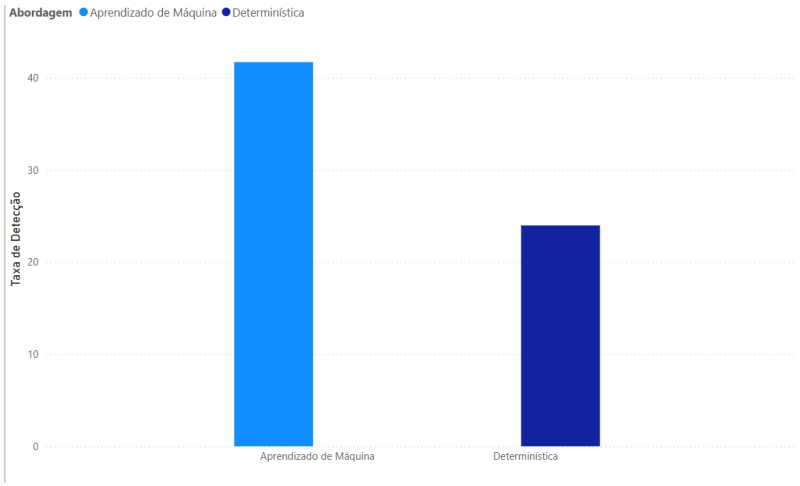

Figure 3. Taxa de vulnerabilidades encontradas por ferramenta

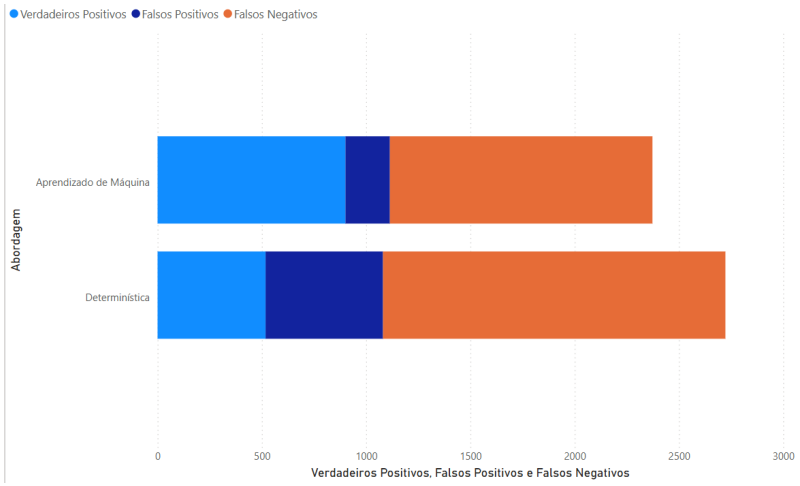

Figure 4. Taxa de falsos positivos/negativos por abordagem

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

A Figura 11 apresenta um *heatmap* da cobertura por categoria CWE, permitindo identificar os tipos de vulnerabilidades em que cada abordagem apresentou melhor desempenho. Observa-se que as abordagens determinísticas, obtiveram desempenho mais consistente em vulnerabilidades como CWE416 (*Use After Free*), CWE415 (*Double Free*) e CWE78 (*Command Injection*). Já as ferramentas baseadas em aprendizado de máquina tiveram destaque em CWE122 (*Heap-based Buffer Overflow*) e CWE369 (*Divide by Zero*).

A Figura 13 consolida essas observações por abordagem, mostrando que os sistemas de ML possuem maior variabilidade de desempenho entre diferentes tipos de falhas, apresentando taxas próximas de 100% em algumas categorias e 0% em outras. Isso indica

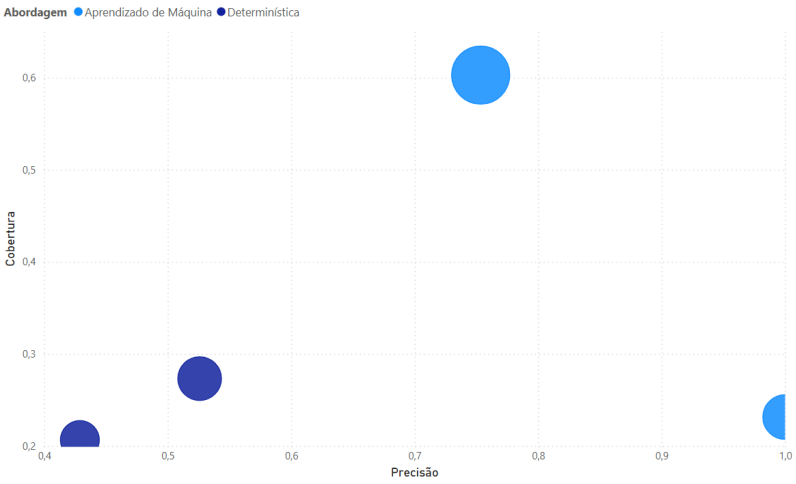

Figure 5. Taxa de falsos positivos/negativos por ferramenta

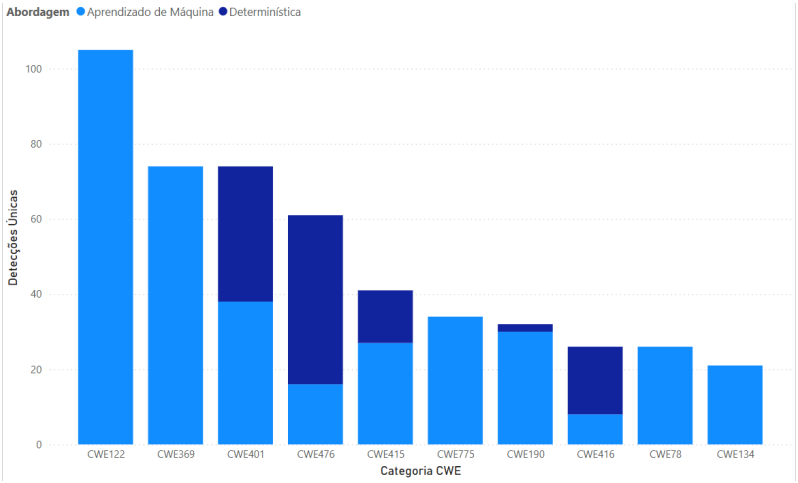

Figure 6. Detecções únicas de cada abor

que o aprendizado do modelo é fortemente influenciado pela representatividade dos dados de treinamento em cada classe de vulnerabilidade.

A Figura 12 ilustra as vulnerabilidades não detectadas por cada ferramenta. Nota-se que as abordagens de aprendizado de máquina falharam em categorias como CWE401 (*Memory Leak*) e CWE476: (*NULL Pointer Dereference*), enquanto os outros sistemas não detectaram muitras vulnerabilidade do tipo CWE122, CWE476 e CWE369.

Por outro lado, conforme mostrado na Figura 13, as detecções mais proeminentes de cada abordagem confirmam que os modelos de aprendizado de máquina têm maior capacidade de identificar falhas distintas, complementando as ferramentas determinísticas. Assim, conclui-se que, embora os sistemas de ML sejam mais sensíveis a certas vulnerabilidades, os métodos determinísticos continuam úteis pela consistência e previsibilidade de comportamento em múltiplas categorias CWE.

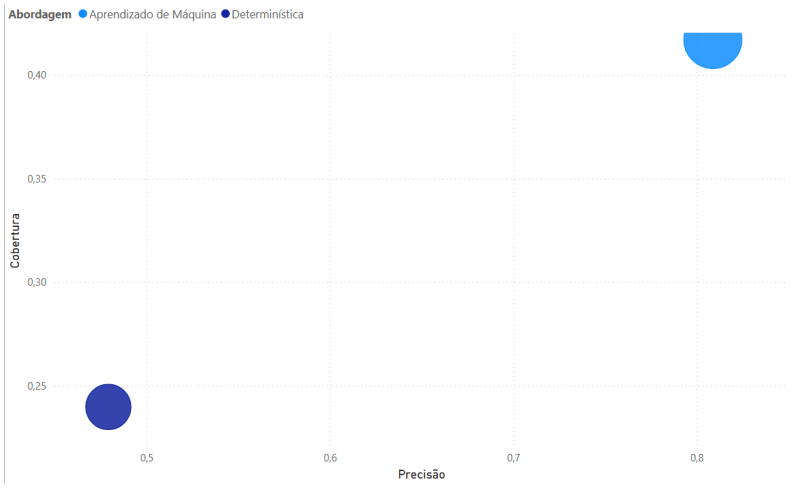

Figure 7. Cobertura comparada com a confiabilidade

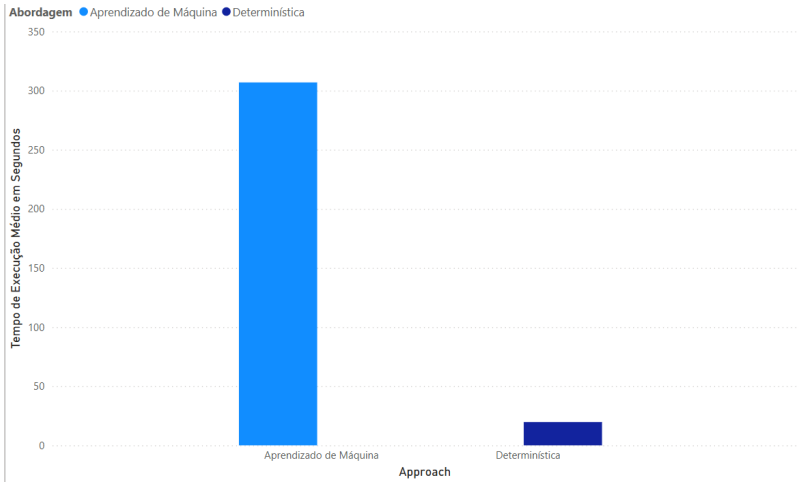

Figure 8. SARD - Tempo de execução de cada ferramenta

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

Para avaliar o desempenho em contextos mais próximos do ambiente industrial, foram analisados 140 repositórios C/C++ do GitHub [GitHub, Inc.]. Todas as ferramentas processaram os mesmos projetos, e os resultados foram comparados quanto ao número de vulnerabilidades identificadas, tempo de execução e consumo de recursos computacionais.

Na Figura 14, verifica-se que as ferramentas baseadas em aprendizado de máquina encontraram um número maior de vulnerabilidades, corroborando o comportamento observado no conjunto SARD. Entretanto, conforme as Figuras 15 e 16, essa detecção ampliada implicou em maior tempo de execução médio e maior uso de CPU.

A Figura 17 mostra que as abordagens de ML apresentaram picos de processamento mais altos, embora o uso médio de CPU tenha sido comparável entre os dois grupos de ferramentas. Curiosamente, as abordagens determinísticas, apresentaram maior consumo médio de memória (Figura 18), o que sugere algoritmos mais pesados e dependentes de análise sintática profunda.

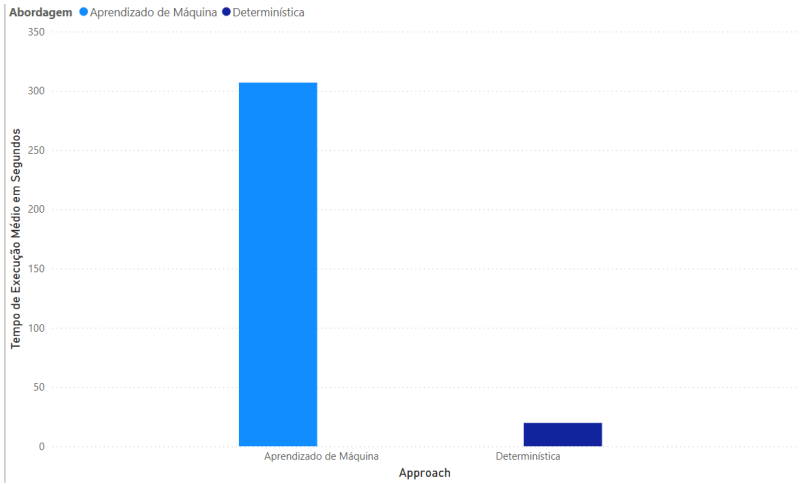

Figure 9. SARD - Uso médio de CPU de cada ferramenta

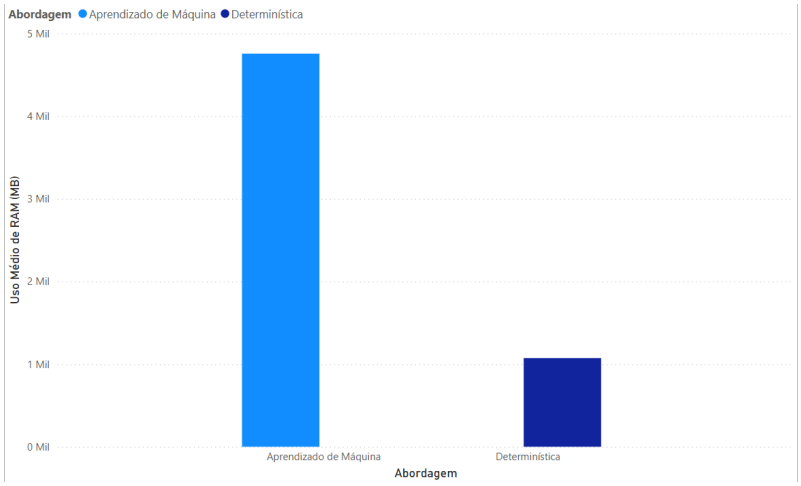

Figure 10. SARD - Uso médio de RAM de cada ferramenta

Esses resultados indicam que, em ambientes reais, as soluções de ML tendem a ser mais eficazes em termos de detecção e tempo, mas ainda demandam otimização de recursos para adoção em larga escala. Já as ferramentas determinísticas, mesmo com menor sensibilidade, oferecem maior estabilidade e maturidade de integração com pipelines CI/CD, o que as torna complementares às abordagens baseadas em aprendizado de máquina.

8. Limitações do Estudo

Embora o trabalho tenha adotado uma metodologia estruturada e reprodutível, algumas limitações devem ser reconhecidas para a interpretação adequada dos resultados. A primeira diz respeito à quantidade e diversidade das ferramentas analisadas. Foram avaliadas apenas quatro soluções, sendo duas delas determinísticas (CodeQL [GitHub 2021] e Snyk [Snyk]) e duas baseadas em aprendizado de máquina (AutoVAS [Jeon and Kim 2021] e VulCNN [Wu et al. 2022]), o que representa apenas uma fração do ecossistema existente de ferramentas de análise de código. É possível que outras soluções, especialmente comerciais ou proprietárias, apresentem resultados distintos em termos de

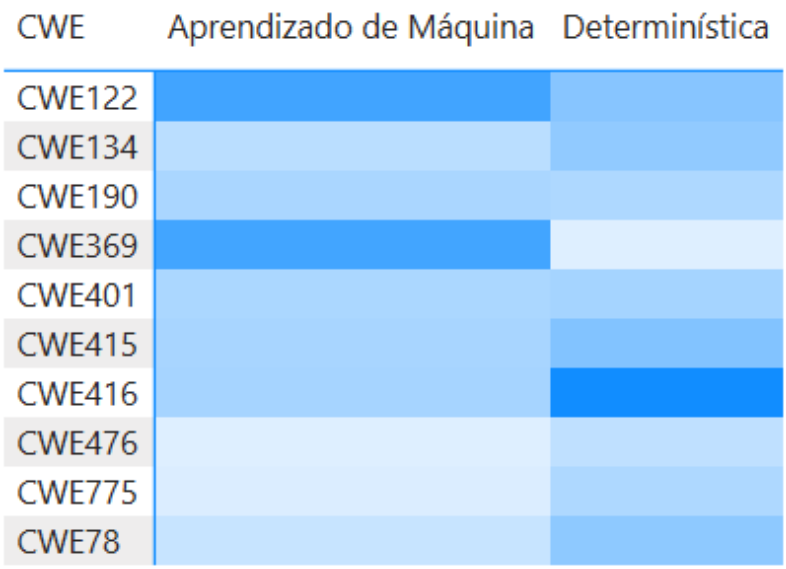

Figure 11. Heatmap de Categoria CWE por abordagem

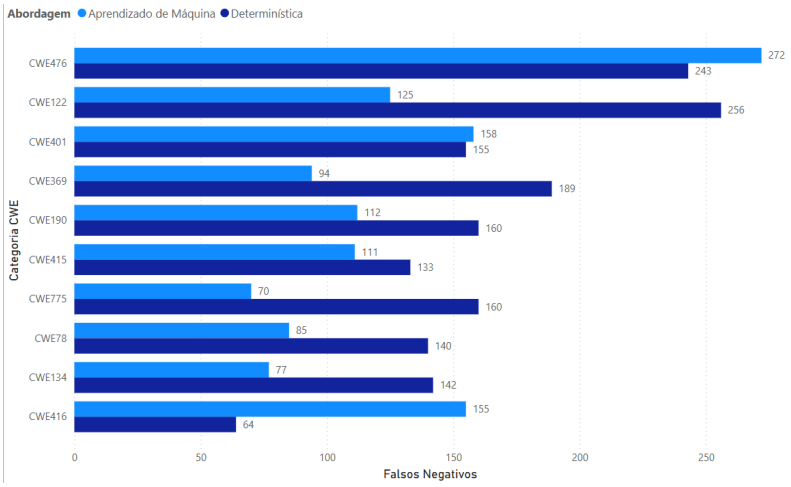

Figure 12. Vulnerabilidades não encontradas por cada ferramenta

precisão, desempenho e cobertura de vulnerabilidades.

Outra limitação importante refere-se ao conjunto de dados utilizado. A pesquisa foi conduzida com base na *Software Assurance Reference Dataset* (SARD) [NIST], composta por códigos em C e C++. Embora essa base seja amplamente reconhecida pela comunidade científica, suas vulnerabilidades são inseridas de forma controlada e artificial. Assim, os resultados obtidos podem não refletir totalmente o comportamento das ferramentas em contextos reais, onde as falhas são mais complexas, distribuídas e frequentemente combinadas com más práticas de desenvolvimento. Além disso, a concentração em apenas duas linguagens limita a generalização dos achados para outras, como Java, Python ou JavaScript, que possuem estruturas e paradigmas diferentes.

No que diz respeito às abordagens de aprendizado de máquina, é necessário considerar que o desempenho dos modelos depende fortemente da qualidade e da representatividade dos dados de treinamento. O modelo VulCNN, por exemplo, obteve excelente

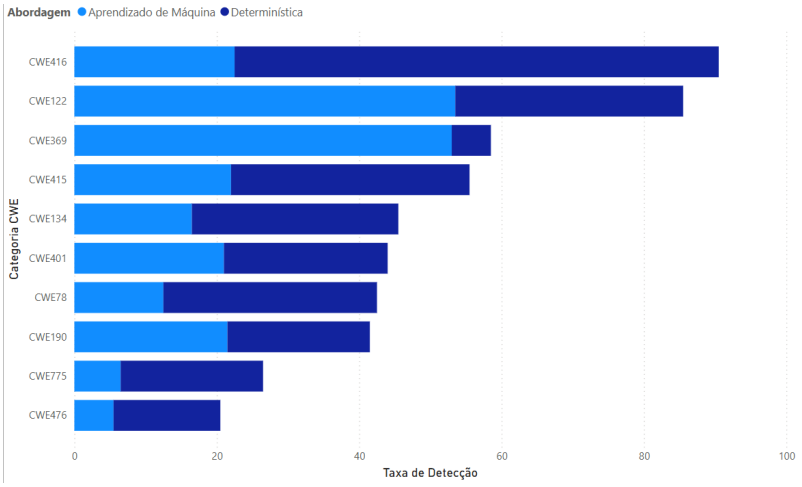

Figure 13. Detecções únicas de cada abordagem por categoria CWE

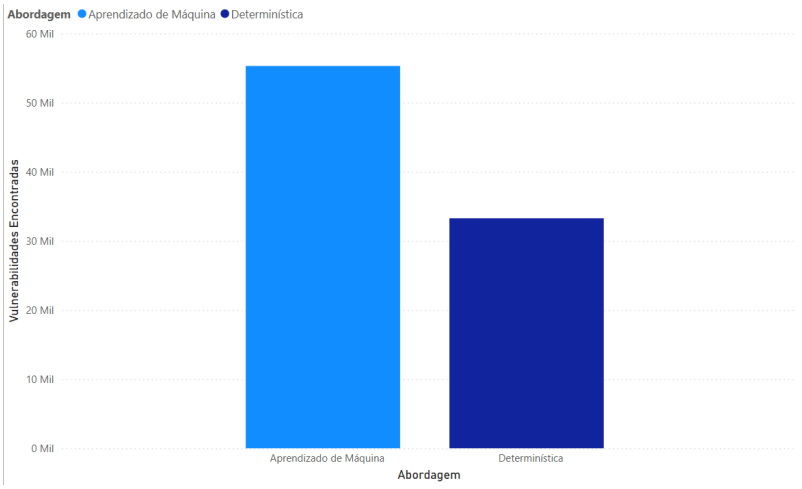

Figure 14. Quantidade de vulnerabilidades encontradas por abordagem

desempenho em várias categorias CWE, mas falhou completamente em *Memory Leak (CWE401)*, possivelmente devido à ausência de amostras suficientes dessa classe no conjunto de treino. Essa limitação evidencia a importância de ampliar e balancear os *datasets* utilizados no treinamento para garantir maior cobertura e generalização.

Adicionalmente, o estudo não considerou métricas de eficiência energética nem variações de ambiente de execução. O consumo de CPU e memória foi avaliado de forma média, sem controle de fatores externos, como paralelização, cache e limitações de hardware. Assim, pequenas variações no ambiente de teste podem ter influenciado os resultados de desempenho.

Por fim, ressalta-se que a avaliação foi conduzida em ambiente controlado, com foco na reprodutibilidade dos experimentos. Em cenários industriais, as ferramentas estão sujeitas a repositórios de grande porte, dependências externas, múltiplas linguagens e integração contínua (CI/CD), fatores que podem alterar significativamente o comportamento observado. Dessa forma, as conclusões apresentadas devem ser interpretadas considerando essas restrições contextuais e metodológicas.

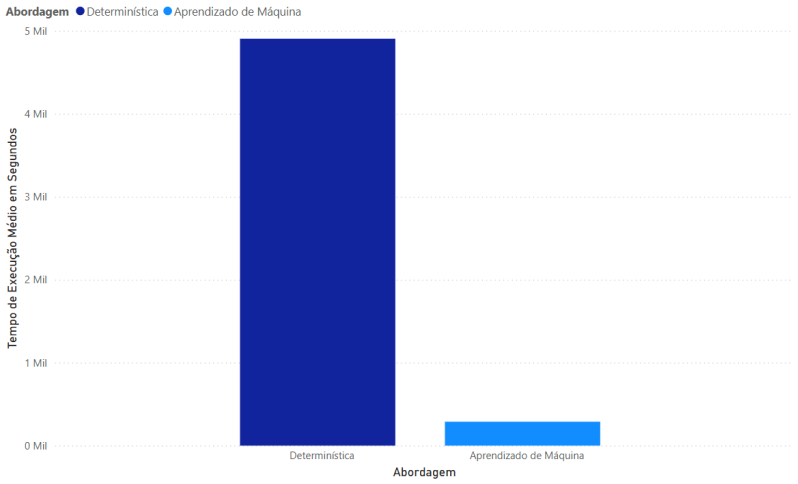

Figure 15. Tempo de execução de cada análise

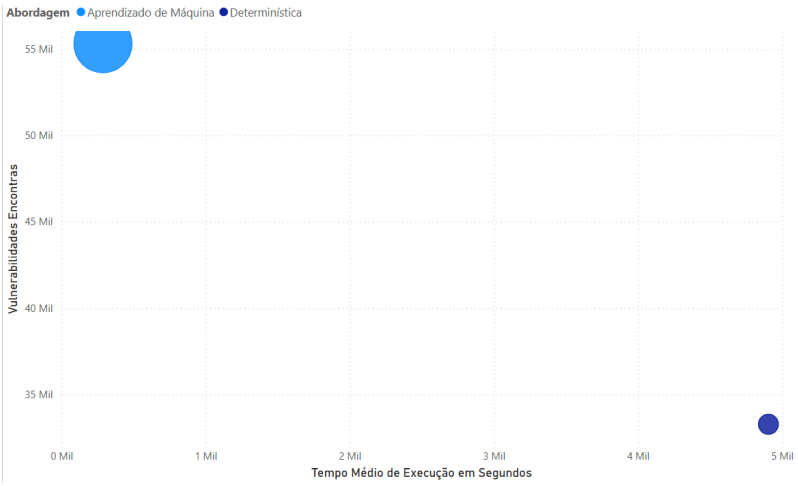

Figure 16. Vulnerabilidades encontradas x tempo de execução x uso médio de cpu

9. Trabalhos Futuros

A partir das análises e resultados obtidos neste estudo, diversas possibilidades de continuidade podem ser exploradas para aprofundar e expandir a compreensão sobre a eficácia de ferramentas de análise de código baseadas em aprendizado de máquina e em algoritmos determinísticos.

Uma das principais direções para trabalhos futuros consiste em ampliar o escopo experimental, incluindo novas ferramentas e abordagens híbridas. A comparação com outros modelos de aprendizado profundo, como redes neurais baseadas em grafos (Graph Neural Networks), e soluções emergentes de análise estática poderá fornecer uma visão mais abrangente sobre as potencialidades e limitações de cada técnica. Além disso, a incorporação de ferramentas comerciais ou proprietárias permitiria avaliar a maturidade das soluções utilizadas no mercado frente às de código aberto analisadas neste trabalho.

Outra linha promissora envolve a expansão da base de dados utilizada. Apesar da relevância da Software Assurance Reference Dataset (SARD), o uso de bases com-

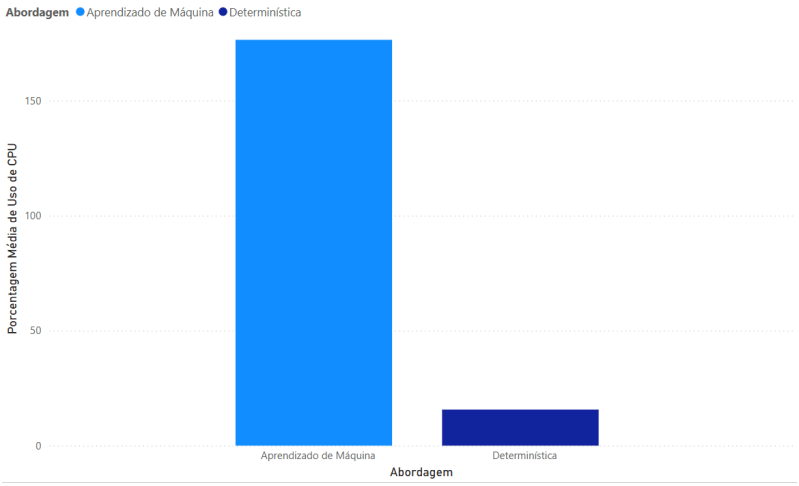

Figure 17. Uso médio de CPU por abordagem

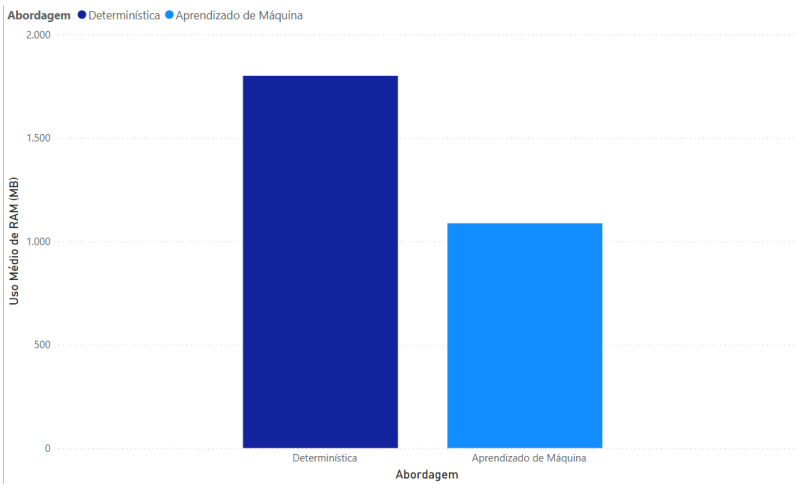

Figure 18. Uso médio de RAM por abordagem

plementares compostas por vulnerabilidades extraídas de repositórios reais do GitHub ou de bancos públicos, como CVE e NVD, poderia melhorar a diversidade dos exemplos e aumentar a robustez dos modelos de aprendizado de máquina. Também seria relevante realizar o balanceamento das amostras por categoria CWE, de forma a mitigar vieses de detecção, como o observado em Memory Leak (CWE401).

No campo do aprendizado de máquina, recomenda-se explorar técnicas de explicabilidade e interpretabilidade de modelos (Explainable AI), com o objetivo de compreender melhor as decisões dos classificadores e aumentar a confiança dos desenvolvedores em sua adoção. Abordagens como atenção visual, mapeamento de gradiente e extração de padrões sintáticos podem auxiliar na identificação dos fatores que levam o modelo a classificar determinado trecho de código como vulnerável.

Além disso, a integração entre análise estática e dinâmica representa um caminho promissor. O uso combinado de ferramentas SAT (Static Application Testing) e DAST (Dynamic Application Security Testing) poderia unir a precisão das abordagens determinísticas com a ampla cobertura dos modelos de aprendizado, promovendo uma análise

mais completa e confiável. Essa integração também é compatível com práticas modernas de desenvolvimento seguro, como DevSecOps, permitindo automação contínua da detecção de vulnerabilidades durante todo o ciclo de vida do software.

Por fim, sugere-se a realização de estudos longitudinais para avaliar a evolução do desempenho das ferramentas ao longo do tempo, considerando atualizações de modelo, novas versões de *datasets* e mudanças nas linguagens de programação. Tais estudos poderiam identificar tendências, gargalos e oportunidades de padronização para futuras gerações de ferramentas de análise automática de vulnerabilidades.

10. Conclusão

A análise comparativa realizada neste trabalho demonstrou diferenças significativas entre as abordagens determinísticas e aquelas baseadas em aprendizado de máquina para detecção automática de vulnerabilidades em código-fonte. De forma geral, as técnicas de *machine learning* apresentaram desempenho superior em termos de cobertura e equilíbrio entre precisão e *recall*, enquanto as ferramentas determinísticas mostraram maior previsibilidade e menor custo de execução.

As abordagens de aprendizado de máquina destacaram-se como o mais eficaz, alcançando 41,71% de detecção, precisão de 81% e F1-score de 0,55, evidenciando o potencial das redes neurais convolucionais em compreender padrões complexos de vulnerabilidade. Apesar disso, essas ferramentas ainda estão no campo de estudo acadêmico, sendo difíceis de configurar e utilizar. Já o Snyk e o CodeQL exibiram desempenho intermediário, mas ainda relevantes pela estabilidade e ampla adoção em ecossistemas de desenvolvimento.

Esses resultados evidenciam que não há uma solução única que supere todas as demais em todos os critérios. As ferramentas baseadas em aprendizado de máquina são mais sensíveis e abrangentes, mas demandam mais recursos e apresentam variação de desempenho entre categorias CWE. As ferramentas determinísticas, embora mais limitadas, garantem consistência e interpretabilidade, sendo ideais para uso em etapas finais de verificação.

Portanto, conclui-se que a estratégia mais promissora consiste em combinar ambas as abordagens. Um modelo híbrido, integrando aprendizado de máquina e regras determinísticas, poderia unir a cobertura ampliada dos modelos de ML à precisão e confiabilidade dos analisadores estáticos, promovendo uma detecção mais robusta e eficiente.

Além disso, o estudo reforça a importância de expandir o treinamento de modelos de ML para categorias menos representadas, como *Memory Leaks*, e de otimizar o uso de recursos computacionais para viabilizar sua aplicação em escala industrial. Assim, o trabalho contribui para o avanço da engenharia de software segura, oferecendo evidências quantitativas que orientam a escolha e combinação de ferramentas de análise de vulnerabilidades.

Assim, este estudo reforça o papel crescente da inteligência artificial como ferramenta essencial na engenharia de software segura, ao mesmo tempo em que demonstra a relevância de abordagens híbridas para garantir equilíbrio entre precisão, desempenho e confiabilidade.

11. 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.

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.

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.

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.