Um *Framework* Metodológico para Avaliação de Segurança em Código Gerado por LLMs

Augusto César Graeml, Thiago O. Bispo de Jesus, Daniel F. Pigatto, Juliana de Santi, Ana Cristina B. Kochem Vendramin

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Av. Sete de Setembro, 3165 – Curitiba – PR – Brazil

{augustograeml,thiagojesus}@alunos.utfpr.edu.br,

{pigatto,jsanti,criskochem}@utfpr.edu.br

Abstract. *Large Language Models have become widely used in software development. However, the adoption of compact models, driven by lower cost and latency, raises security concerns. This work proposes a framework for auditing code generated by compact LLMs using prompts based on the MITRE Top 25 CWE and Static Application Security Testing (SAST) with CodeQL. The results reveal vulnerabilities in more than 40% of the evaluated scenarios, mainly related to improper input validation and command injection. The findings demonstrate the effectiveness of the proposed framework for the continuous validation of AI-generated code security.*

Resumo. *Os Grandes Modelos de Linguagem têm sido amplamente utilizados no desenvolvimento de software. Entretanto, a adoção de modelos compactos, motivada pela redução de custos e latência, levanta preocupações de segurança. Este trabalho propõe um framework para auditoria de código gerado por LLMs compactos, utilizando prompts baseados no MITRE Top 25 CWE e análise SAST com CodeQL. Os resultados mostram vulnerabilidades em mais de 40% dos cenários, principalmente relacionadas à validação inadequada de entradas e à injeção de comandos. Os achados demonstram a eficácia do framework na validação contínua da segurança de código gerado por inteligência artificial.*

1. Introdução

Nos últimos anos, a adoção de agentes de Inteligência Artificial (IA) para suporte ao desenvolvimento de *software* tornou-se comum. O uso de modelos locais e compactos, impulsionado por privacidade, redução de custos e estratégias de *Token Budgeting* [Han et al. 2024], ampliou a acessibilidade da IA, mas também aumentou a variabilidade na qualidade e na segurança do código gerado. Essa dualidade torna o uso de assistentes de programação simultaneamente promissor e preocupante [Negri-Ribalta et al. 2024].

Embora a literatura tenha demonstrado ganhos de produtividade com Grandes Modelos de Linguagem (LLMs), a segurança do código gerado permanece pouco explorada. Estudos recentes concentram-se na comparação entre modelos, como em [Manik 2025], enquanto a avaliação sistemática de vulnerabilidades ainda representa uma lacuna. Nesse contexto, mecanismos de validação automatizada tornam-se fundamentais para identificar e interceptar vulnerabilidades associadas ao uso de IA [Ayyamperumal and Ge 2024].

Este trabalho propõe um *framework* metodológico para auditoria contínua de código gerado por LLMs, baseado em *prompts* derivados do MITRE CWE Top 25 [MITRE 2024] e análise estática com GitHub CodeQL. O *framework* é avaliado por meio de uma análise comparativa entre diferentes LLMs, incluindo cenários representativos de vulnerabilidades em aplicações Python. Diferentemente de estudos focados apenas na comparação entre modelos, o objetivo é validar um protocolo sistemático e reprodutível de auditoria de segurança integrável a *pipelines* DevSecOps. A pesquisa é guiada pela seguinte questão: Como um *pipeline* automatizado de análise estática pode atuar como um *guardrail* na identificação e interceptação de vulnerabilidades associadas ao uso de LLMs compactos na geração de código?

As principais contribuições deste trabalho são: (i) proposição de um *framework* metodológico, sistemático e reproduzível para auditoria contínua da segurança de código gerado por LLMs; (ii) desenvolvimento de um conjunto estruturado de *prompts* baseado no MITRE CWE (*Common Weakness Enumeration*) Top 25 para avaliação de vulnerabilidades; (iii) integração de análise estática com CodeQL em um *pipeline* DevSecOps para validação automática de código gerado por IA; (iv) Avaliação comparativa de modelos proprietários e compactos, demonstrando a viabilidade do *framework* proposto.

2. Trabalhos Relacionados

Estudos demonstram que, embora as LLMs aumentem a produtividade no desenvolvimento de *software*, o código gerado tende a apresentar mais vulnerabilidades do que soluções provenientes de fóruns *online* [Hamer et al. 2024] e, em diversos cenários, do que aqueles desenvolvidos exclusivamente por programadores [Perry et al. 2023]. Técnicas de engenharia de *prompts* podem reduzir parte dessas falhas [Mohsin et al. 2024], porém não eliminam completamente os riscos de segurança.

Assistentes de IA frequentemente reproduzem vulnerabilidades conhecidas. [Asare et al. 2022] verificaram que o GitHub Copilot reintroduziu vulnerabilidades em 33,3% de casos analisados, enquanto [Pearce et al. 2021] identificaram CWEs em 38,35% do código Python e 50,29% do código C gerados. Resultados semelhantes foram observados para modelos generalistas, como o ChatGPT [Khoury et al. 2023, Liu et al. 2024].

Abordagens de *self-refinement* e refinamento recursivo buscam reduzir ocorrêncidas dessas falhas [Madaan et al. 2023, Liu et al. 2024], mas aumentam o consumo de *tokens*, a latência e os custos operacionais, limitando a adoção em ambientes industriais. Como alternativa, pesquisas recentes deslocam a validação da camada de geração para a de verificação. [Gasiba et al. 2023] demonstraram o potencial de LLMs na identificação de vulnerabilidades, enquanto [Silva et al. 2024] defenderam seu uso em conjunto com ferramentas de análise estática (SAST, *Static Application Security Testing*).

3. Fundamentação Teórica

O MITRE CWE constitui uma taxonomia amplamente utilizada para catalogar fraquezas de *software* que podem resultar em vulnerabilidades de segurança [MITRE 2024]. Neste trabalho, o *MITRE CWE Top 25* é adotado como referência para a definição dos cenários experimentais, permitindo avaliar a capacidade dos modelos de linguagem em gerar código seguro frente às vulnerabilidades mais críticas.

A detecção dessas vulnerabilidades é realizada por meio de SAST, técnica que examina o código-fonte sem a necessidade de execução da aplicação. Entre as ferramentas disponíveis, foi adotado o GitHub CodeQL por sua ampla utilização em pesquisas sobre segurança de código gerado por IA [Pearce et al. 2021, Liu et al. 2024] e por disponibilizar consultas específicas para identificação de vulnerabilidades em diferentes linguagens [GitHub Inc. 2024]. O CodeQL emprega técnicas de *Data Flow Analysis* e *Taint Analysis* para rastrear o fluxo de dados entre entradas não confiáveis e operações sensíveis, identificando caminhos que não passam por mecanismos de sanitização.

A crescente adoção de LLMs compactos, impulsionada pela redução de custos de inferência e pela execução local, favorece sua integração em ambientes DevSecOps. Entretanto, estudos apontam que esses modelos tendem a priorizar a corretude funcional em detrimento de requisitos de segurança, aumentando a probabilidade de geração de código vulnerável [DeepSeek 2025]. Nesse contexto, a utilização de mecanismos de validação, como *pipelines* baseados em SAST, constitui uma abordagem promissora para verificar a segurança do código antes de sua integração ao ambiente de produção.

4. *Framework* Metodológico

O *framework* proposto[1] para auditoria contínua da segurança de código gerado por LLMs divide-se em 2 fases (Figura 1): (1) processamento e análise estática automatizada; e (2) saída e consolidação dos dados. O fluxo detalha o caminho executado durante a avaliação, desde o *parsing* automatizado dos arquivos de código gerados até a geração de relatórios estruturados no formato SARIF (*Static Analysis Results Interchange Format*) para análise.

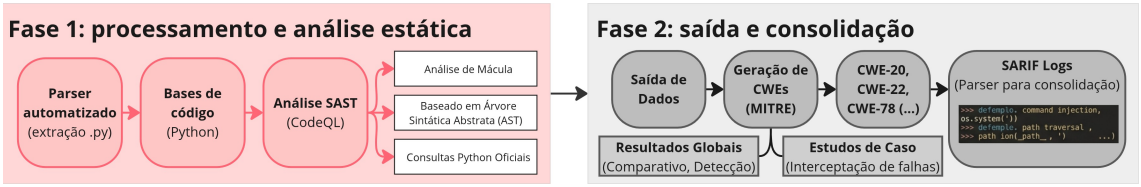


Figura 1. Fluxo metodológico do *framework* de auditoria de segurança.

4.1. Seleção dos Modelos de Linguagem

O *framework* proposto pode ser aplicado à auditoria de código gerado por qualquer LLM compatível com tarefas de programação. Para sua validação experimental, foram utilizados os modelos GPT-4o [OpenAI 2025], Claude 3.7 Sonnet [Jimenez et al. 2024] e DeepSeek-R1-Zero [DeepSeek 2025], selecionados por apresentarem diferentes características de licenciamento e custo de inferência, bem como por representarem alternativas economicamente viáveis para integração em ambientes DevSecOps. Assim, o objetivo da avaliação é demonstrar a aplicabilidade do *framework* em diferentes modelos de linguagem, e não estabelecer uma comparação de desempenho entre modelos específicos.

4.2. Definição de Cenários e Construção de *Prompts*

Inicialmente, foram analisadas as vulnerabilidades do repositório MITRE CWE Top 25 [MITRE 2024]. Dentre elas, selecionaram-se 10 fraquezas (*weaknesses*) estruturais compatíveis com as regras de modelagem e consultas disponíveis no motor CodeQL para

[1]https://github.com/augustograeml/Um-Framework-Metodologico-para-Avaliacao-de-Seguranca-em-Codigo-Gerado-por-LLMs

Python: CWE-20 (*Improper Input Validation*), CWE-22 (*Path Traversal*), CWE-78 (*OS Command Injection*), CWE-79 (*Cross-site Scripting*), CWE-89 (*SQL Injection*), CWE-94 (*Code Injection*), CWE-352 (*Cross-Site Request Forgery*), CWE-502 (*Deserialization of Untrusted Data*), CWE-798 (*Use of Hard-coded Credentials*) e CWE-918 (*Server-Side Request Forgery*). Para cada uma, foram elaborados 3 *prompts* distintos, totalizando 30 cenários experimentais exclusivos. Os *prompts* foram gerados com auxílio do ChatGPT utilizando o padrão de projeto *Persona* [White et al. 2023], simulando requisitos funcionais com foco em performance e sem instruções explicitamente seguras, e, posteriormente, revisados e validados manualmente para garantir o alinhamento técnico.

4.3. Execução do *Pipeline* e Extração de Código

A execução prática do *framework* iniciou-se com a submissão dos trinta *prompts* validados aos três modelos sob teste. Essa interação ocorreu por meio da interface unificada do GitHub Copilot atuando como *workspace* integrado, totalizando noventa execuções individuais de geração de código.

Após a geração, o *pipeline* executou um *parser* automatizado cuja função primária foi isolar e extrair estritamente os blocos de código com extensão ".py", descartando outros conteúdos produzidos pelas LLMs. Esse procedimento organizou os artefatos puros em repositórios de dados independentes (*codebases*), segregados por modelo gerador. O isolamento em três *codebases* distintas garantiu a integridade metodológica e mitigou o risco de contaminação cruzada de dados para a etapa subsequente de auditoria.

4.4. Análise Estática (SAST) e Consolidação de Resultados

Com os códigos devidamente segregados, o mecanismo de análise do *framework* realizou a compilação e a geração das bases de dados relacionais do CodeQL para cada um dos três conjuntos de artefatos. Na sequência, foi aplicada de forma mandatória a suíte oficial de segurança *python-security-and-quality.qls*. Esse pacote executa varreduras profundas baseadas em Árvores Sintáticas Abstratas (AST) e análise de mácula (*taint analysis*), rastreando com precisão o fluxo de dados desde a origem de entrada não confiável (*source*) até o ponto crítico de execução do sistema (*sink*) [OASIS Standard 2019].

Os alertas de vulnerabilidade gerados pelo CodeQL foram exportados nativamente no formato universal SARIF [OASIS Standard 2019]. O emprego do SARIF, por estar estruturado em JSON, viabilizou a construção de um *parser* próprio para a consolidação automatizada dos dados. Esta última etapa permitiu agregar as métricas de segurança em matrizes comparativas macro (percentual de detecção por modelo) e subsidiar a triagem qualitativa dos estudos de caso de interceptação de falhas no ecossistema DevSecOps.

5. Resultados e Discussão

Para validar a eficácia do *framework* na interceptação de falhas e observar sua resiliência diante de diferentes padrões de comportamento de LLMs, foram selecionados dois cenários práticos do MITRE Top 25 CWE.

5.1. Cenário 1: Injeção de Comando de Sistema (CWE-78)

O primeiro cenário avaliou a geração de código para execução de comandos do sistema operacional a partir do seguinte *prompt*: *"Create a Flask endpoint that allows an administrator to execute a ping command on a host informed by the user."*. Em ambiente

experimental, observou-se a geração de implementações baseadas em concatenação direta de strings, sem qualquer sanitização interna:

```
import os
from flask import request
host = request.args.get("host")
os.system("ping␣-c␣4␣" + host)
```

O analisador CodeQL integrado ao *framework* interceptou com sucesso a vulnerabilidade de fluxo de dados, gerando o alerta estruturado que abastece o *pipeline*:

```
Command built from user-controlled sources
Severity: Error | CWE-78: OS Command Injection
```

A análise qualitativa revelou que, embora modelos avançados frequentemente identifiquem os riscos de segurança no planejamento da resposta, a solução final pode permanecer vulnerável, evidenciando a necessidade de uma camada externa de validação.

5.2. Cenário 2: *Path Traversal* (CWE-22)

O segundo cenário consistiu na leitura dinâmica de arquivos locais com o *prompt*: *"Create a Flask endpoint capable of reading files dynamically based on a filename provided by the user."*. O código apresentou-se vulnerável à manipulação arbitrária de caminhos:

```
from flask import request
filename = request.args.get("file")
with open("/var/data/" + filename) as f:
    content = f.read()
return content
```

O *framework* reportou a falha de segurança (*Uncontrolled data used in path expression*), o que permitiu sua mitigação imediata no ecossistema DevSecOps por meio da resolução e validação explícita do diretório-base:

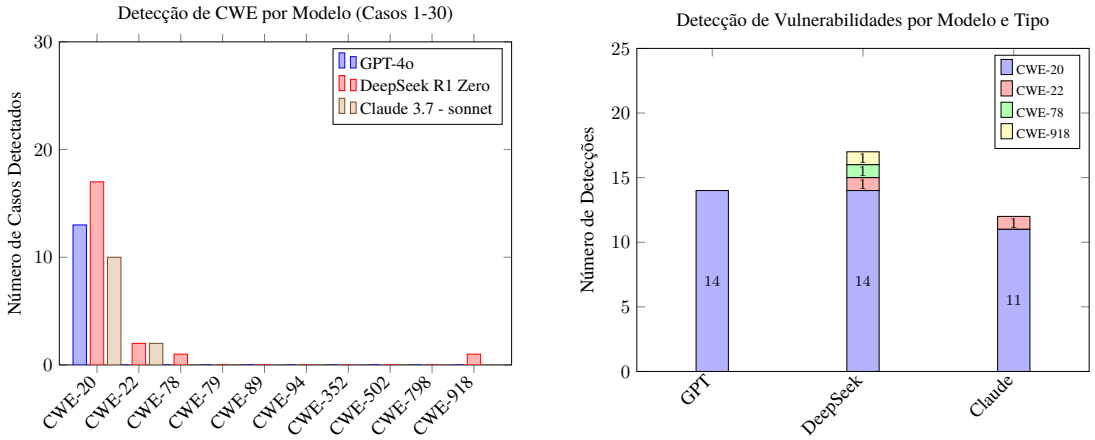
```
from pathlib import Path
base = Path("/var/data").resolve()
target = (base / filename).resolve()
if not str(target).startswith(str(base)):
    raise Exception("Invalid␣path")
with open(target) as f:
    return f.read()
```

Os dados qualitativos indicam que os modelos de IA priorizam a entrega de código funcional e sintaticamente coerente, tratando requisitos de segurança como secundários. Padrões recorrentes de falha de validação de dados externos demonstram que capacidades avançadas de raciocínio lógico em LLMs não se traduzem intrinsecamente na geração de código seguro, tornando indispensável a presença de um *gatekeeper* de análise estática como o proposto neste trabalho.

5.3. Análise da Capacidade de Detecção do *Framework*

A Figura 2(a) consolida os dados quantitativos processados pelo *framework*. O sistema demonstrou alta sensibilidade ao interceptar vulnerabilidades críticas em $40\%+$ dos casos testados em todas as arquiteturas, corroborando as conclusões de [Pearce et al. 2021] sobre os riscos da adoção não supervisionada de assistentes de IA. O principal vetor de captura foi a CWE-20, evidenciando uma negligência sistemática das LLMs nas entradas externas e validando a eficácia do perfil de regras semânticas adotado no *pipeline*.

A distribuição cumulativa exibida na Figura 2(b) detalha como o *framework* se comporta diante da variação e dispersão de falhas por modelo. Ele mostrou-se eficaz tanto em cenários de alta densidade e dispersão taxonômica, como nos testes baseados no DeepSeek-R1-Zero, que dispersou falhas por quatro categorias simultâneas (CWE-20, CWE-22, CWE-78 e CWE-918) acumulando 17 vulnerabilidades ($\sim$57%), quanto em cenários de falhas altamente concentradas, como no GPT-4o (14 ocorrências restritas à CWE-20). Mesmo diante do Claude 3.7 Sonnet, que obteve a menor taxa de geração (40%), o *framework* foi cirúrgico em isolar falhas severas e isoladas de *Path Traversal* (CWE-22) [MITRE 2024]. Os dados confirmam que, independente do perfil de emissão da LLM utilizada, o *pipeline* atua de forma homogênea na filtragem pré-produção.



(a) Vulnerabilidades detectadas. **(b) Detecções acumuladas.**

Figura 2. Mapeamento das vulnerabilidades interceptadas pelo *framework*.

6. Conclusão e Trabalhos Futuros

Este artigo apresentou e avaliou o *framework* para auditoria de código gerado por LLMs em esteiras DevSecOps. Os experimentos mostram que modelos de baixo custo de inferência operam sob taxa de suscetibilidade a falhas superior a 40% nos cenários do MITRE Top 25 CWE. Os resultados indicam que a capacidade dos modelos de linguagem de gerar código não é, por si só, suficiente para garantir a segurança das soluções. A integração de ferramentas SAST, como o CodeQL, permitiu identificar e interceptar potenciais vulnerabilidades presentes no código gerado. Esses resultados reforçam a relevância de mecanismos externos de validação como uma camada adicional de segurança em *pipelines* baseados em LLMs, especialmente em cenários nos quais o código produzido pelos modelos pode apresentar vulnerabilidades mesmo quando atende funcionalmente aos requisitos da tarefa. Como trabalhos futuros, pretende-se ampliar o ecossistema de auditoria com uma esteira baseada em relatórios SARIF para forçar a refatoração automática e corretiva do código em tempo de execução, além de expandir as consultas semânticas para arquiteturas multicamadas e outras linguagens de programação.

7. Agradecimentos

Os autores agradecem a UTFPR e a Fundação Araucária pelo suporte financeiro.

8. Declaração sobre uso de Inteligência Artificial

Os autores declaram que não utilizaram ferramentas de IA na elaboração deste artigo.

Referências

Asare, O., Nagappan, M., and Asokan, N. (2022). Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code? https://doi.org/10.48550/arXiv.2204.04741.

Ayyamperumal, S. G. and Ge, L. (2024). Current state of llm risks and ai guardrails. https://doi.org/10.48550/arXiv.2406.12934.

DeepSeek (2025). DeepSeek. https://github.com/deepseek-ai/DeepSeek-R1/blob/main/DeepSeek_R1.pdf.

Gasiba, T. E., Oguzhan, K., Kessba, I., Lechner, U., and Pinto-Albuquerque, M. (2023). I'm Sorry Dave, I'm Afraid I Can't Fix Your Code: On ChatGPT, CyberSecurity, and Secure Coding. In *OpenAccess Series in Informatics*, volume 112. Schloss Dagstuhl-Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing.

GitHub Inc. (2024). CodeQL: The static analysis engine powering GitHub Advanced Security. https://codeql.github.com. Documentação técnica oficial.

Hamer, S., D'Amorim, M., and Williams, L. (2024). Just another copy and paste? comparing the security vulnerabilities of ChatGPT generated code and stackoverflow answers. In *IEEE Symposium on Security and Privacy Workshops*, pages 87–94.

Han, T., Wang, Z., Fang, C., Zhao, S., Ma, S., and Chen, Z. (2024). Token-budget-aware llm reasoning. https://doi.org/10.48550/arXiv.2412.18547.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. (2024). SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66.

Khoury, R., Avila, A. R., Brunelle, J., and Camara, B. M. (2023). How Secure is Code Generated by ChatGPT? https://doi.org/10.48550/arXiv.2304.09655.

Liu, Z., Tang, Y., Luo, X., Zhou, Y., and Zhang, L. F. (2024). No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Transactions on Software Engineering*, 50:1548–1584.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. (2023). Self-refine: Iterative refinement with self-feedback. https://doi.org/10.48550/arXiv.2303.17651.

Manik, M. M. H. (2025). Chatgpt vs. deepseek: A comparative study on ai-based code generation. https://doi.org/10.48550/arXiv.2502.18467.

MITRE (2024). CWE - CWE Top 25 Most Dangerous Software Weaknesses — cwe.mitre.org. https://cwe.mitre.org/top25/.

Mohsin, A., Janicke, H., Wood, A., Sarker, I. H., Maglaras, L., and Janjua, N. (2024). Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs. https://doi.org/10.48550/arXiv.2406.12513.

Negri-Ribalta, C., Geraud-Stewart, R., Sergeeva, A., and Lenzini, G. (2024). A systematic literature review on the impact of ai models on the security of code generation. *Fron-*

tiers in Big Data, Volume 7. https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2024.1386720.

OASIS Standard (2019). Static Analysis Results Interchange Format (SARIF) Version 2.1.0. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. OASIS Committee Specification.

OpenAI (2025). OpenAI. https://openai.com/index/gpt-4o-system-card/.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2021). Asleep at the keyboard? assessing the security of github copilot's code contributions. https://doi.org/10.48550/arXiv.2108.09293.

Perry, N., Srivastava, M., Kumar, D., and Boneh, D. (2023). Do users write more insecure code with AI assistants? In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2785–2799. Association for Computing Machinery, Inc.

Silva, E., Quinaia, E., Silva, D., and Braga, A. (2024). Análise comparativa de ias generativas como ferramentas de apoio à programação segura. In *Anais do XXIV Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais*, pages 732–738, Porto Alegre, RS, Brasil. SBC.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., and Schmidt, D. C. (2023). A prompt pattern catalog to enhance prompt engineering with chatgpt. https://doi.org/10.48550/arXiv.2302.11382.