DETECTING COMPLEX VULNERABILITIES IN REAL-WORLD CODE: LLM BENCHMARK FOR ENHANCED SOFTWARE SECURITY

Klaudia Kloc
San Francisco, USA
klaudia@vidocsecurity.com

Dawid Moczadlo
Gdansk, Poland
dawid@vidocsecurity.com

***Abstract* — Large Language Models (LLMs) are increasingly applied to software security tasks, yet their effectiveness in identifying complex vulnerabilities in real-world code remains poorly understood due to limitations in existing benchmark datasets. This research introduces a new benchmark specifically designed to evaluate LLMs in realistic security code reviews. It utilizes code snippets extracted from active open-source GitHub projects - capturing the complexity of modern software development - and includes a broad range of vulnerability types, such as Insecure Direct Object References (IDORs), broken authentication, and business logic flaws, which are often missed by traditional Static Application Security Testing (SAST) tools. Unlike benchmarks based on synthetic data, this dataset focuses on actual vulnerabilities observed in real projects and supports direct comparison between SAST tools and LLM-based approaches. The urgency of this problem is amplified by the growing risks in software supply chain security. As systems increasingly depend on third-party libraries and services, they not only inherit new vulnerabilities but also compound existing ones - a trend reflected in the rising number of reported security flaws [7]. In addition to technical evaluation, the research explores the economic implications of LLM-assisted vulnerability detection, including potential cost savings from early identification of security flaws and the downstream impact of false positives on development efficiency.**

Keywords (*code security, LLM)*

INTRODUCTION

The landscape of software security is rapidly changing, with Large Language Models (LLMs) emerging as powerful tools not only for code generation, but also vulnerability detection and remediation [1][2]. These models demonstrate significant potential due to their advanced capabilities in understanding both human and programming languages [1]. Despite this promise, a critical gap exists in comprehensively evaluating their efficacy, particularly in addressing **complex, real-world vulnerabilities.** Current evaluation methods often fall short, largely relying on **synthetic datasets** that may not accurately reflect the intricacies of modern software development.. Such synthetic datasets typically feature simpler vulnerabilities and struggle with issues like high data duplication and unrealistic class imbalance between different categories of vulnerabilities, vulnerable and non-vulnerable samples, which can lead models to learn irrelevant feature [3].

Traditional Static Application Security Testing (SAST) tools, while fundamental, are known to have their own limitations. These include producing an **overwhelming number of false positives**, having a **limited scope of analysis**, and being **resource-intensive** [4]. They often fail to identify vulnerabilities that are dependent on runtime behavior, access control mechanisms, or complex business logic flows. This highlights a pressing need for more sophisticated, adaptable, and context-aware vulnerability detection methods that can complement or even surpass existing tools. LLMs offer a novel avenue for addressing this challenge, having shown potential in understanding and generating code, and pinpointing and rectifying vulnerabilities [5].

PROPOSED BENCHMARK AND METHODOLOGY

To address the aforementioned challenges, this research introduces a **novel benchmark dataset** specifically tailored to evaluate LLMs in realistic security code reviews. This new benchmark distinguishes itself by utilizing **code snippets extracted directly from active open-source GitHub projects.** This approach is crucial for capturing the authentic complexity and nuances of modern software development environments.

The dataset is designed to encompass a diverse array of **complex vulnerability types**. These include, but are not limited to, Insecure Direct Object References (IDORs), various forms of broken authentication, and intricate business logic flaws. These specific types of vulnerabilities are particularly significant because they are frequently **overlooked by conventional SAST tools** due to their context-dependent nature. By focusing on actual vulnerabilities observed in real projects, the benchmark provides a more **veridical assessment** of LLM performance compared to synthetic datasets. We extracted 500 vulnerable code snippets, each mapped to a specific entry in the CVE database, and categorized them into 10 distinct vulnerability categories.

A key objective of this research is to enable a **direct and meaningful comparison** between the performance of traditional SAST tools and LLM-based approaches. Studies have shown that LLMs like GPT-4 can identify significantly more vulnerabilities (e.g., approximately **four times more than Snyk and Fortify**) and even propose viable fixes with a low false positive rate [5]. Similarly, GPT-3 has been observed to find **substantially more security issues** (e.g., 213

vulnerabilities in one codebase) than commercial tools, accurately detecting issues in 85 out of 129 files scanned [6]. Furthermore, integrating LLMs as assistants to SAST tools, for example by providing them with labels, has demonstrated improved outcomes, potentially **reducing both false positive and false negative rates** [7]. The *LLbezpeky* framework, for instance, showed that LLMs correctly flagged insecure Android applications in 91.67% of cases in the Ghera benchmark [1]. LLMs can successfully identify the programming language of a code snippet without explicit hints [5][10][11].

Mentioned evaluations have been conducted on a variety of code datasets, many of which were specifically constructed to target particular vulnerability categories or were limited to a single programming language or framework. Such conditions hinder meaningful comparisons across tools and methodologies due to the lack of consistency and balance in the datasets.

To address this limitation, the benchmark introduced in this study aims to provide a more comprehensive and balanced dataset, ensuring a comparable distribution of vulnerabilities across multiple categories and programming languages.

The benchmark consists of real-world code snippets that have been manually reviewed and classified into ten distinct vulnerability types. The dataset includes samples written in Java, Python, JavaScript, TypeScript, and Go. We evaluated a selection of static analysis tools and LLM-based approaches, measuring their performance in terms of accuracy, precision, and recall.

Economic Implications

Beyond the technical evaluation, this research delves into the **economic ramifications of adopting LLM-assisted vulnerability detection**. Early identification of security flaws is a critical factor in software development, commonly referred to as "shifting left" in the development lifecycle [8]. By catching bugs earlier, LLM-based analysis can significantly reduce the overall cost and time associated with remediation, preventing more expensive fixes later in the production cycle [8].

Another important economic consideration is the **downstream impact of false positives** on development efficiency. Traditional SAST tools are known for their high rates of false positives, which necessitate considerable developer time and effort to manually investigate non-existent issues, thereby **wasting resources and hindering productivity** [4]. There is evidence that LLMs, particularly when used as an assistant alongside SAST tools, have the potential to **reduce both false positive and false negative rates** [7].

Furthermore, recent user studies suggest a concerning trend: participants with access to AI assistants **consistently wrote less secure code** than those without [9]. This appears to be linked to users' over-reliance on and **misplaced trust in the AI to produce secure code**, even when their solutions were insecure. For instance, participants with AI access provided significantly less secure solutions for SQL vulnerabilities, with 36% being vulnerable to SQL injection compared to 7% in the control group [7]. This implies that while LLMs can generate correct code, they may often use insecure method. This underscores the critical need for developers to maintain their security knowledge and critically review LLM-generated code.

Future Work

Looking ahead, this research paves the way for several key future directions. A significant step will be to **open-source the newly developed dataset and its accompanying code**, which will foster reproducibility, encourage community contributions, and facilitate further advancements in the field. Future iterations of the benchmark will focus on scaling the dataset while maintaining a balanced distribution of vulnerability types and programming languages. An additional goal is to benchmark a wide range of static analysis and LLM-based tools against this dataset and to establish a public leaderboard that reflects their comparative performance.

Further research should also examine the security and functional correctness of code fixes proposed by LLMs.

1. N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh, “LLbezpeky: leveraging large language models for vulnerability detection,” CS858, Project Proposal, University of Waterloo
2. K. Shashwat, F. Hahn, X. Ou, D. Goldgof, L. Hall, J. Ligatti, S. R. Rajagopalan, and A. Z. Tabari, “A Preliminary Study on Using Large Language Models in Software Pentesting,” *arXiv preprint arXiv:2401.17459*, 2024.
3. S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, “Deep learning based vulnerability detection: are we there yet?” *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, Sept. 2022.
4. M. H. Buehler, “Detecting Insecure Code with LLMs: Prompt Experiments for Python Vulnerability Detection,” *Medium*, March 21, 2024. [https://medium.com/data-science/detecting-insecure-code-with-llms-8b8ad923dd98]
5. D. Noever, “Can Large Language Models Find And Fix Vulnerable Software?,” *arXiv preprint arXiv:2308.10345*, 2023.
6. C. Koch, “I Used GPT-3 to Find 213 Security Vulnerabilities in a Single Codebase,” *Medium*, 2023. [https://medium.com/better-programming/i-used-gpt-3-to-find-213-security-vulnerabilities-in-a-single-codebase-cc3870ba9411]
7. A. Bakhshandeh, A. Keramatfar, A. Norouzi, and M. M. Chekidehkhoun, “Using ChatGPT as a Static Application Security Testing Tool,” *ISeCure The ISC Int'l Journal of Information Security*, pp. 1–8, 2023
8. C. Koch, “Experimenting with GPT-3 for Detecting Security Vulnerabilities in Code,” *GitHub*, 2023. [https://github.com/chris-koch-penn/gpt3_security_vulnerability_scanner/blob/main/README.md]
9. N. Perry, M. Srivastava, D. Kumar, and D. Boneh, “Do users write more insecure code with AI assistants?” in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS ’23)*, Copenhagen, Denmark, Nov. 26–30, 2023, pp. 2785–2799.
10. H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing static analysis for practical bug detection: an LLM-integrated approach,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, Art. 111, Apr. 2024.
11. S. Sakaoglu, “Web application vulnerability hunting using large language models,” Master’s Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, June 2023.
12. M. A. Ferrag, M. Ndhlovu, N. Tihanyi, L. C. Cordeiro, M. Debbah, T. Lestable, and N. S. Thandi, “Generative AI and large language models for cyber security: all insights you need,” Survey, 2024.