LLMVet: An LLM-based Vetting Tool for the npm Ecosystem

Md Atiqur Rahman, Nazia Afreen, Muhammad Alahmadi
mrahma22@ncsu.edu, nafreen@ncsu.edu, mjalahma@ncsu.edu

Abstract

The npm ecosystem lacks a robust vetting system to ensure the security of published code, leaving users exposed to vulnerabilities often discovered only through manual analysis or limited research. Addressing this gap, we propose LLMVet, an LLM-based vetting tool designed to detect vulnerable or malicious code in npm packages. Leveraging large language models (LLMs) fine-tuned for vulnerability detection, our approach evaluates package releases for security risks before public availability. Trusted maintainers will initially validate the system's findings, enabling iterative improvement as the model learns from additional data. To enhance analysis, we incorporate contributor metadata alongside the code. Methodologically, we fine-tune existing LLMs, perform ablation studies to evaluate performance, and explore integrating semi-automatic processes for scalability. Our findings are expected to demonstrate that LLMVet outperforms existing crowd-sourced or manual vetting approaches in efficiency and effectiveness, offering a feasible path to improving the security of npm's extensive ecosystem. This work contributes to software supply chain security by introducing a scalable and intelligent vetting mechanism capable of adapting to the growing demands of modern package management systems.

1 Introduction

Modern software development heavily relies on third-party packages, particularly in ecosystems like npm, which provide reusable modules to accelerate development. The proliferation of open-source software has driven innovation across industries, but it has also introduced significant challenges in maintaining secure and reliable codebases. Security vulnerabilities, often subtle and difficult to detect, remain a persistent threat to modern software systems [7]. Left unchecked, these vulnerabilities can lead to severe consequences, including data breaches, financial losses, and reputational damage [26]. For example, the Log4Shell vulnerability [18, 1] was a critical remote code execution (RCE) flaw in the Apache Log4j 2 library, affecting millions of applications worldwide.

Identifying and addressing the vulnerabilities often relies on reactive measures, such as code review, community reports or manual analysis after a version gets released. However, these approaches suffer from certain limitations. While code reviews are a standard part of software development practices, they too often focus on functionality and correctness rather than proactive vulnerability assessment. The process of gathering, analyzing, and addressing vulnerabilities through community contributions are generally reliable in detecting and resolving a particular vulnerability. Nevertheless this takes time and creates a critical gap where vulnerabilities can persist undetected until exploitation occurs, leaving users exposed during this window of risk. [14]

In this context, advances in machine learning (ML) [15, 17] and natural language processing (NLP), particularly the rise of large language models (LLMs), offer transformative potential in addressing the challenges. Large Language Models are extensively pretrained on massive datasets that include diverse programming languages, software repositories, and technical documentation. This exposure equips LLMs with a deep contextual understanding of code patterns as a structured natural language, common programming paradigms, and the nuances of software vulnerabilities. Furthermore, code-specific Transformer-based LLMs, such as CodeBERT, CodeT5, and OpenAI's Codex, have been specifically trained on code-focused datasets, enhancing their ability to analyze, generate, and infer meaning from source code. These models can be further fine-tuned with domain-specific or task-specific data, such as vulnerability patches, security advisories, and structured metadata, to improve their identifying risky changes in code, explaining the nature of vulnerabilities, and suggesting potential mitigations.

However, applying LLMs to vulnerability detection is not without its challenges. Real-world datasets on vulnerabilities, which are essential for fine-tuning and evaluating these models, are often noisy, incomplete, and lack standardization. This inconsistency arises because vulnerabilities may be reported in various formats—some with detailed descriptions and patches, others with vague or insufficient information. Such variability complicates the training process, as the model might struggle to general-

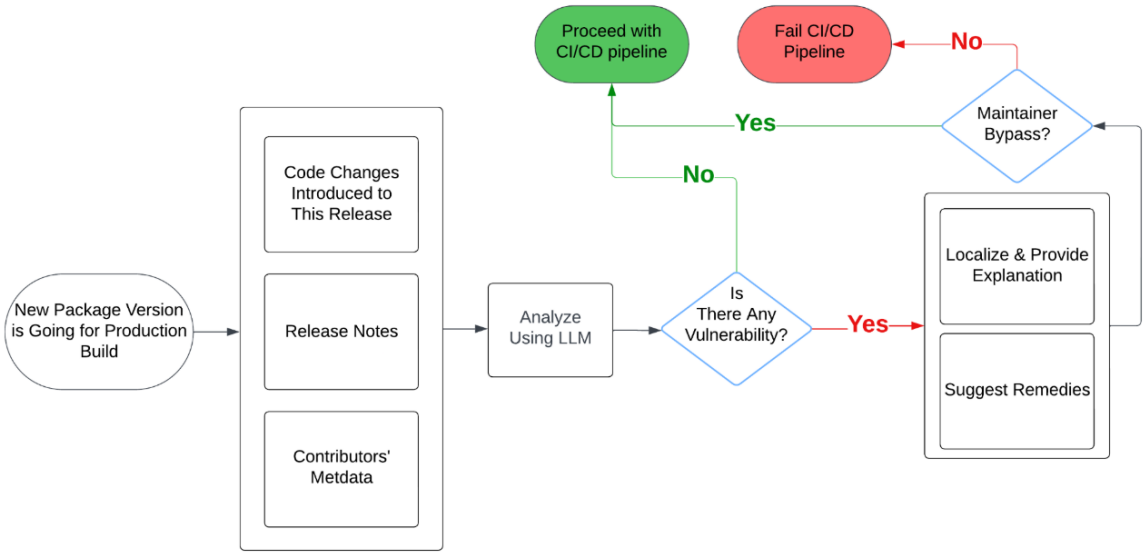

Figure 1: A high-level architecture of LLM-Vet in CI/CD Pipeline of npm Packages

ize across different types of data inputs. Additionally, the high computational cost of fine-tuning large models often imposes practical limitations, particularly for resource-constrained environments.

This study is motivated by the need to address these challenges while leveraging state-of-the-art large language models to improve the effectiveness of vulnerability detection in npm Ecosystem. We introduce **LLMVet**, an intelligent vetting framework that identifies vulnerabilities or malicious code in npm packages before they are released. By leveraging LLMs fine-tuned specifically for this purpose, the framework performs an in-depth evaluation of package updates, analyzing code alongside contributor metadata to provide a holistic view of potential risks. Integration into CI/CD pipelines allows LLMVet to seamlessly evaluate packages as part of the development workflow, reducing latency in vulnerability detection. Trusted maintainers verify the system's insights, enabling iterative improvements as the model is exposed to diverse datasets. Semi-automated processes and ablation studies ensure scalability and robust performance. LLMVet surpasses traditional manual or crowd-sourced methods in both efficiency and effectiveness, offering an adaptive solution to enhance the security of npm's extensive and rapidly evolving package ecosystem. This innovation significantly contributes to securing modern software supply chains by embedding intelligent, automated vetting directly into the development lifecycle.

In this work, we present a study evaluating the performance of various models, including Qwen2.5-Coder-3B-Instruct, CodeBERT, and CodeT5-Large, on a curated dataset of npm packages vulnerabilities. Our contributions are threefold:

- First, we curate and preprocess a dataset of vulnerabilities sourced from the GitHub Advisory Database, addressing the noise and inconsistencies that plague such data. We also utilize summarization techniques to generate structured and concise descriptions of vulnerabilities, improving data quality and model compatibility.

- Second, we fine-tune and evaluate the performance of state-of-the-art LLMs and transformers on the vulnerability detection task, highlighting their strengths and limitations.

- Finally, we provide an in-depth analysis of the results, identifying key factors that impact model performance and discussing the practical challenges of deploying these models in real-world scenarios.

2 Overview

This section outlines the core aspects of our study, including research goals, our key approach, and the research questions guiding our investigation. Together, these elements establish the foundation for understanding the significance and direction of our work.

2.1 Research Goal

The npm ecosystem currently lacks a comprehensive vetting system to ensure package security. Vulnerabilities are often discovered and reported by the community, but this reactive approach leaves many packages already infected and exposes users to significant risks. To address this, we aim to develop an automatic and scalable security vetting system that reviews code updates in new package releases to detect vulnerabilities proactively.

We aim to leverage a large language model (LLM) to analyze code updates in newly released npm package versions. By integrating metadata such as contributor details and prior version history, the LLM will assess whether the code contains vulnerabilities and provide clear, actionable explanations. This approach identifies risks and empowers maintainers to understand and address security concerns effectively.

2.2 Key Approach

We propose incorporating LLM-Vet into the CI/CD pipeline of npm package releases. When a new version is submitted, LLM-Vet will first analyze the changes introduced. If the system detects any vulnerabilities, it will flag the maintainer with an explanation and suggest potential fixes. This enables maintainers to halt the pipeline, address the issue, and release a more secure package. Figure 1 illustrates the overall pipeline design.

To accommodate real-world complexities, we plan to include mechanisms to handle false positives. Maintainers will have an override option to bypass the LLM-Vet's assessment and publish the release if they determine the flagged changes are safe. A feedback loop will also be implemented to continuously refine LLM-Vet by training it on real-time data, improving its accuracy and adaptability to evolving security patterns.

2.3 Research Questions

To explore the feasibility and effectiveness of LLM-based vetting in npm package releases, we address the following research questions:

RQ1: *How accurately can the LLM detect vulnerable code changes?*

RQ2: *How effective is the metadata in improving vulnerability detection?*

RQ3: *How will maintainers interact with and override the system when necessary?*

These research questions form the basis of our investigation and will guide our efforts to evaluate the impact of LLM-Vet on npm package security.

3 Methodology

We first focus on collecting and processing data to address our research questions. Given the absence of an existing dataset for this custom task, we built our dataset from scratch with meticulous attention to detail. Following data collection and preprocessing, we trained the LLMs on the curated dataset and evaluated their performance. This section outlines the dataset, data preprocessing methods, and the training and fine-tuning process of the LLMs.

3.1 Data Used for the Study

To source vulnerabilities within the npm ecosystem, we relied on the GitHub Advisory Database [10]. Using their REST API, we crawled advisory data, capturing detailed information about identified vulnerabilities. Each advisory entry was enriched by mapping its CVE IDs to corresponding records in the National Vulnerability Database (NVD) [21] using their REST API. This allowed us to incorporate additional information, such as CVE descriptions, into our dataset.

Each advisory includes a vulnerable version range and the first patched version. To construct meaningful data for our study, we focused on the last vulnerable version before a patch was applied. By querying the GitHub REST API, we extracted the code changes introduced between the last vulnerable version and the first patched version. These changes represent the transition from vulnerable to non-vulnerable states and serve as the cornerstone of our analysis. The combined dataset, integrating advisory information, CVE details, and associated code changes, forms the baseline for our study. Table 1 provides an overview of the columns included in this dataset.

3.2 Data Preprocessing

To ensure the reliability and relevance of the data used for this study, a comprehensive preprocessing pipeline was employed. Given the diversity and inconsistency of the data, several steps were undertaken to refine the dataset and prepare it for use in vulnerability detection tasks.

The textual descriptions of vulnerabilities were often found poorly documented. Through manual analysis of several packages, we determined that elements such as hyperlinks, Proof of Concept (POC) references, and contributor or Fixed by acknowledgments do not contribute to the core understanding of vulnerabilities and can be omitted. To address this, these elements were removed to improve focus and coherence. Additionally, any inline code blocks within descriptions were excluded, and excessive newlines or redundant whitespace were cleaned, resulting in streamlined and standardized descriptions. To ensure consistency and improve model performance, vul-

Table 1: Sample Columns in the Dataset

Column Name	Type	Description
`Package Name`	String	The name of the npm package associated with the advisory.
`Severity`	String	The severity level of the vulnerability (e.g. Critical, High, Medium, Low).
`GHSA Summary`	Text	Summary of the advisory from the GitHub Advisory Database.
`GHSA Description`	Text	Detailed description of the advisory from the GitHub Advisory Database.
`CVE ID`	String	Corresponding CVE identifier mapped from the National Vulnerability Database.
`CVE Description`	Text	Detailed description of the vulnerability from the National Vulnerability Database.
`Vulnerable Version`	String	The specific version extracted from the range of versions known to be vulnerable.
`First Patched Version`	String	The first version where the vulnerability was patched.
`Patches`	Array(Text)	Extracted code changes between the last vulnerable version and the first patched version.

nerability descriptions were summarized using the BART model. The summarization process consolidated information from various fields within the dataset into a standardized format.

Since this study focuses exclusively on npm packages, data from other ecosystems was filtered out. Further, to reduce noise, non-functional files and directories were excluded from the dataset. This included files with extensions such as .md, .json, .yaml, .ipynb, and .html, as well as directories like .github/, test/, and docs/ . Configuration and changelog files such as setup , Pipfile , and CHANGELOG were also discarded. This ensured that the dataset consisted only of files contributing directly to the functionality of the npm packages.

The commit patches were then processed to isolate meaningful changes. Each patch was divided into smaller units called hunks based on standard diff formatting $(@@ - X, Y + X, Y @@)$. The hunks were further classified into three categories: added lines $(+)$, removed lines $(-)$, and context lines. Consecutive blocks of similar statements were grouped, and redundant elements such as empty lines and excessive whitespace were removed. Comments within the code were retained to preserve contextual information.

To facilitate analysis, the content of each patch was organized into two primary categories: added statements with context and removed statements with context. This structure provides a clear representation of code changes, enabling models to focus on the functional impact of the patches.

Finally, to augment the dataset and simulate diverse scenarios, the patches were reversed to create non-vulnerable-to-vulnerable transitions. This approach effectively doubled the dataset, resulting in approximately 4,000 data points containing both positive and negative examples. These preprocessing steps ensured that the dataset was both high quality and sufficiently diverse, providing a robust foundation for training and evaluating large language models for vulnerability detection in npm ecosystems.

3.3 Training and Fine-tuning LLMs

To test the idea of our work, we evaluated a mixture of large-language models and transformer-based models to assess their suitability for vulnerability detection. The models included:

- **Qwen2.5-Coder-3B-Instruct[13]:** A language model pre-trained in code and fine-tuned with instructional data. Both the base variant and the fine-tuned variants of the 3B model were tested, as memory constraints prevented the use of larger variants such as the 7B model.
- **CodeBERT[9]:** A transformer model optimized for code understanding tasks.
- **CodeT5-Large[32]:** Another transformer model designed specifically for code-related tasks.

Table 2: Dataset Preparation Statistics

Stage	Count	Description
Total Crawled Advisories	6,217	Advisories initially collected from the GitHub Advisory Database across various severity categories.
Discarded (No Patched Version)	931	Advisories without a patched version were removed.
Retained (Missing/Invalid CVE IDs)	Approx. 1,800	Advisories missing valid CVE IDs; these entries were retained but lacked CVE descriptions.
Discarded (No Repository URL)	1,987	Advisories without a source repository URL were removed since patches could not be retrieved.
Discarded (Invalid Tags)	Approx. 1,000	Advisories with invalid or missing repository tags were excluded as patch processing was not feasible.
Final Samples	Approx. 2,000	Curated samples representing transitions from vulnerable to non-vulnerable code.
Augmented Samples	Approx. 2,000	Reverse transitions created by flipping code changes and labeling as non-vulnerable to vulnerable transitions.
Total Dataset Size	Approx. 4,000	Combined dataset used for training and evaluation, split into Approx. 3,700 training samples and 300 testing samples.

For finetuning CodeBERT and CodeT5-Large, we prepare the input sequence to be the patches data and the generated summary description. Then, this input text was provided to the default tokenizer for each model which has a limit of 512 tokens. This limit is smaller than many of number of the tokens in many of collected data which leads to lower performance. However, we observed around 5% increase in the performance of the finetuned models compared to the base moodels in both CodeBERT and CodeT5-Large. Due to the limitations of these transformers-based models, we conduct a classification evaluation only as the models not generate meaningful explanation.

For LLMs experiments, we first pass the input into data preparation process which will generate an instruction chat format for the model. After that it will be passed to the tokenizer of the LLM which have context length of 32,768 tokens. Although this is larger than the transformers-based model, some of the data points have larger size than the limit tokens. Then we fine-tuned the LLM using the `unsloth` and `RoLA`[12] libraries, with the primary goal of training them to classify JavaScript code snippets as either vulnerable or non-vulnerable and generate an explanation of the vulnerability if there is one. However, the process presented several challenges that impacted the efficiency and effectiveness of the training phase. One significant limitation was the constrained memory capacity, which restricted the ability to experiment with larger models, such as higher-parameter versions of Qwen2.5-Coder. Furthermore, we have finetuned and run several models other than Qwen2.5-Coder-3B-Instruct, but they exhibited undesirable behaviors during training, such as echoing input prompts or generating outputs that lacked meaningful content, undermining their potential performance. Additionally, some input samples exceeded the token length limitations of the models, necessitating truncation or alternative preprocessing strategies to ensure compatibility with the architecture. These challenges underscored the complexity of adapting large-scale models for highly specialized tasks like vulnerability detection. To evaluate the accuracy of the generated explanation by the LLMs, we computed the BLEU score of each generated response with the description generated by the BART model. BLEU is a standard method of calculating semantic similarity by measuring the correspondence between a two texts by calculating the precision of n-grams while penalizing short output.

4 Evaluation

We begin by presenting key statistics about the dataset we curated, followed by an evaluation of the performance of LLMs and Transformer-based models on this dataset.

Table 3: Model Performance on Vulnerability Detection Task

Model	Accuracy	Precision	Recall	F1-Score	BLEU Score
Qwen2.5-Coder-3B-Instruct (Base)	0.41	0.41	0.41	0.41	0.00048
Qwen2.5-Coder-3B-Instruct (Fine-tuned)	0.34	0.33	0.34	0.33	0.00038
CodeBERT (Fine-tuned)	0.56	0.54	0.86	0.67	-
CodeT5-Large (Fine-tuned)	0.55	0.55	0.67	0.60	-

4.1 Data Statistics

Initially, we collected a total of 6,217 advisories from the GitHub Advisory Database [10], ensuring an equal sampling of advisories across different severity categories. Of these, 931 advisories were excluded because they lacked any patched version information. Among the remaining advisories, approximately one-third had either missing or invalid CVE IDs. These entries were retained but were not enriched with CVE descriptions due to the incomplete data.

Additionally, 1,987 advisories were discarded because they lacked source repository URLs, making it impossible to retrieve the associated code patches. Around 1,000 advisories were excluded due to invalid tags or improper tagging in their respective repositories, which hindered patch processing. After rigorous filtering, we curated a final dataset of approximately 2,000 samples.

All these samples represented transitions from vulnerable to non-vulnerable code. However, obtaining the opposite—transitions from non-vulnerable to vulnerable code—was infeasible. To address this, we reversed the code changes in the existing samples and labeled them as transitions from non-vulnerable to vulnerable code. This augmentation doubled the dataset size to 4,000 samples, with 3,700 used for training and 300 reserved for testing.

Table 2 summarizes the statistics and key decisions involved in curating the dataset for our study, providing an overview of how the initial data was processed and augmented to create a robust training and evaluation set.

4.2 Models Performance

The performance of the models was evaluated using metrics such as accuracy, precision, recall, F1 score, and BLEU score. The results are summarized in Table 3.

5 Discussion

In our analysis, the observed underperformance of the fine-tuned LLM compared to the base model can be attributed to several factors. First, the small size and limited diversity of the fine-tuning dataset, consisting of only 4000 samples, likely hindered the model's ability to generalize effectively. This lack of diverse examples may have caused the model to overfit to the training data, impairing its ability to perform well on unseen instances. Additionally, data quality issues, such as incomplete or ambiguous vulnerability descriptions, could have introduced noise into the fine-tuning process. Poor data quality often leads to the model learning spurious or irrelevant patterns, which compromises its performance. Another potential explanation is catastrophic forgetting, wherein fine-tuning on a narrow dataset causes the model to lose the general knowledge acquired during pretraining. This phenomenon can result in a degradation of overall performance, as the model becomes overly specialized in the fine-tuning task and less capable of leveraging its pre-existing knowledge. Finally, the fine-tuning process itself might not have been fully optimized, and the hyperparameters or training procedure used may not have been ideal for this particular task, further contributing to the suboptimal results. These challenges underscore the importance of addressing data limitations and refining the fine-tuning process to improve model performance in specialized tasks like vulnerability detection.

This study underscores the potential and limitations of using LLMs and transformer-based models for vulnerability detection. Although transformer models demonstrated better recall, their dependence on high-quality data highlights the need for robust data curation processes. However, we believe that larger LLms will outperform the transformers models and overcome the need for a costly data collection process. Although larger LLm models are promising, they were constrained by computational resources.

The challenges associated with data collection were significant. The GitHub Advisory Database, while a valuable resource, lacks standardization and completeness. Many advisories miss key information, such as repository URLs or proper versioning, making it difficult to extract meaningful data. Improved data collection and cleaning methods are essential to maximize the effectiveness of machine learning models in this domain.

6 Related Work

In this section we discuss the closest related works across four key areas: *works on the npm ecosystem*, *reliable data for studying the ecosystem*, *Vulnerability Detection from*

Git Commits, and *LLM in Vulnerability Detection*. While some prior work has explored the npm ecosystem, to the best of our knowledge, we are the first to propose a framework that utilizes code patches to detect vulnerability before the release of a version.

Works on the npm Ecosystem: Research on security within the npm ecosystem has gained significant attention recently. Zimmerman et al. [38] explore security risks, focusing on dependencies, maintainers, and vulnerabilities, emphasizing the dangers of single points of failure due to a few maintainers or unmaintained packages. Zahan et al. [36] identify security signals in 1.63 million npm packages, advocating for proactive notifications to prevent supply chain attacks. However, both studies rely only on npm registry metadata, which was called into question by the 2023 *Manifest Confusion* security issue [5], where registry data may not reflect the actual dependencies. Similarly, Tolhurst [31] exposes discrepancies between GitHub code and npm packages, demonstrating vulnerabilities in the open-source supply chain. Other studies, such as those by Alfadel et al. [2], Raula et al. [16], and Wittern et al. [33], focus on specific aspects like undisclosed vulnerabilities, dependency chains, and ecosystem trends.

Reliable Dataset for the npm Ecosystem: Several studies have focused on creating datasets to aid research in the npm ecosystem. Zahan et al. [34] introduce MalwareBench, a benchmark dataset designed for evaluating malware detection tools, which includes both npm and PyPI packages. By systematically gathering malicious and neutral packages, it fills a critical gap in resources for researchers and developers. Pinckney et al. [24] examine semantic versioning (semver) in npm by building a dataset of all package versions, augmented by a time-traveling dependency resolver. They later expand this work with *npm-follower* [25], a dataset and crawling architecture that archives both metadata and code for npm packages, including versions that are deleted, which is essential for security research. While these datasets are valuable, they do not reflect the rapid evolution of the npm ecosystem. To ensure our analysis is current, we use the BigQuery data from Google's Open Source Insights [3], which provides real-time updates of npm packages.

Vulnerability Detection from Git Commits: Several studies have explored the detection of vulnerabilities through git commits. For instance, [4] presents a system to detect exploitable bugs in glue code between high-level languages and low-level components, while [6, 28] investigate ReDoS vulnerabilities in live websites. Staicu et al. [29] identify injection vulnerabilities in Node.js, and [20] introduces MiDas, a deep-learning model for identifying vulnerability-fixing commits at various granularities. Additionally, [23, 8] and [37] explore using NLP and deep learning to detect vulnerability fixes, including silent fixes made before public disclosure. Despite these advancements, vulnerability detection in npm packages remains in its infancy, with no systematic vetting process in place, leaving much of the work to individual reporting rather than comprehensive security assessments.

LLM in Vulnerability Detection: Mathews et al. [19] investigate using GPT-4 to detect Android vulnerabilities with the Ghera benchmark dataset, exploring prompting techniques such as vulnerability summaries and file access requests to enhance code analysis. Similarly, Zahan et al. [35] introduce SecurityAI, which leverages ChatGPT with Iterative Self-Refinement and Zero-Shot-Role-Play-CoT prompting to detect malicious npm packages. Tested on the MalwareBench dataset, this approach shows how LLMs can complement tools like CodeQL in identifying diverse attack types. Steenhoek et al. [30] analyze LLM error responses using the SVEN dataset, comparing 11 models with prompting techniques like contrastive pairs and static analysis reports, and categorize errors to guide future improvements. Guo et al. [11] evaluate fine-tuned and general-purpose LLMs across a new dataset and five benchmarks, highlighting the benefits and limitations of fine-tuning for smaller models and the impact of dataset mislabeling on performance. Sejfia et al. [27] present Amalfi, a system combining classifiers, a reproducer, and a clone detector to detect malicious npm packages using features like sensitive information access and suspicious network activity. Lastly, Omar et al. [22] propose a fine-tuned GPT-2 model for classifying vulnerabilities in C/C++ and Java source code, outperforming traditional systems like SySeVR and VulDeBERT on the SARD and SeVC datasets.

7 Conclusion

In conclusion, future research should explore distributed training techniques and model compression to enable the use of larger, more complex architectures. This study highlighted memory constraints, which limit the ability to work with sophisticated models. Distributed training could improve scalability, allowing for more powerful models to be used without being hindered by hardware limitations. Additionally, model compression techniques, such as pruning or knowledge distillation, could reduce the memory footprint of these large models while maintaining or even enhancing their performance. These strategies, in combination with better data collection and improved data quality, would optimize the fine-tuning process and allow for the development of more efficient and scalable vulnerability detection systems.

Furthermore, integrating fine-tuned models into CI/CD pipelines holds substantial potential for real-time vulnerability detection in JavaScript code. By automating the

identification of vulnerabilities as they are introduced into the codebase, this approach can provide a proactive first line of defense in production environments. Continuous monitoring and immediate feedback would significantly reduce the time between vulnerability introduction and detection, thus enhancing the security of the software development lifecycle. Given the increasing adoption of AI-driven tools in modern software development practices, the integration of machine learning models into CI/CD workflows is a feasible and promising solution. This approach can transform how vulnerabilities are addressed, ensuring that potential security issues are detected and mitigated before deployment, ultimately strengthening the overall integrity of production systems.

References

[1] Another apache log4j vulnerability is actively exploited in the wild (cve-2021-44228) (updated).

[2] M. Alfadel, D. E. Costa, E. Shihab, and B. Adams. On the discoverability of npm vulnerabilities in node.js projects. *ACM Trans. Softw. Eng. Methodol.*, 32(4), May 2023.

[3] Bigquery dataset — open source insights, 2024.

[4] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578, 2017.

[5] D. Clarke. The massive bug at the heart of the npm ecosystem, June 27 2023.

[6] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee. The impact of regular expression denial of service (redos) in practice: an empirical study at the ecosystem scale. ESEC/FSE 2018, page 246–256, New York, NY, USA, 2018. Association for Computing Machinery.

[7] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.

[8] T. Dunlap, E. Lin, W. Enck, and B. Reaves. Vfcfinder: Seamlessly pairing security advisories and patches, 2023.

[9] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[10] Github advisory database.

[11] Y. Guo, C. Patsakis, Q. Hu, Q. Tang, and F. Casino. Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In *European symposium on research in computer security*, pages 271–289. Springer, 2024.

[12] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[13] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[14] N. Imtiaz, A. Khanom, and L. Williams. Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages. *IEEE Transactions on Software Engineering*, 49(4):1540–1560, 2023.

[15] M. Kazerounian, J. S. Foster, and B. Min. Simtyper: sound type inference for ruby using type equality prediction. 5(OOPSLA), 2021.

[16] R. G. Kula, A. Ouni, D. M. German, and K. Inoue. On the impact of micro-packages: An empirical study of the npm javascript ecosystem, 2017.

[17] X. Li, W. Li, Y. Zhang, and L. Zhang. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. ISSTA 2019, page 169–180, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Log4shell: Rce 0-day exploit found in log4j 2, a popular java logging package.

[19] N. S. Mathews, Y. Brus, Y. Aafer, M. Nagappan, and S. McIntosh. Llbezpeky: Leveraging large language models for vulnerability detection. *arXiv preprint arXiv:2401.01269*, 2024.

[20] T. G. Nguyen, T. Le-Cong, H. J. Kang, R. Widyasari, C. Yang, Z. Zhao, B. Xu, J. Zhou, X. Xia, A. E. Hassan, X.-B. D. Le, and D. Lo. Multi-granularity detector for vulnerability fixes. *IEEE Trans. Softw. Eng.*, 49(8):4035–4057, Aug. 2023.

[21] National vulnerability database.

[22] M. Omar and S. Shiaeles. Vuldetect: A novel technique for detecting software vulnerabilities using language models. In *2023 IEEE International*

Conference on Cyber Security and Resilience (CSR), pages 105–110, 2023.

[23] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, New York, NY, USA, 2015. Association for Computing Machinery.

[24] D. Pinckney, F. Cassano, A. Guha, and J. Bell. A large scale analysis of semantic versioning in npm. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 485–497, 2023.

[25] D. Pinckney, F. Cassano, A. Guha, and J. Bell. npm-follower: A complete dataset tracking the npm ecosystem. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 2132–2136, New York, NY, USA, 2023. Association for Computing Machinery.

[26] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo. Out of sight, out of mind? how vulnerable dependencies affect open-source projects. *Empirical Softw. Engg.*, 26(4), July 2021.

[27] A. Sejfia and M. Schäfer. Practical automated detection of malicious npm packages. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1681–1692, 2022.

[28] C.-A. Staicu and M. Pradel. Freezing the web: a study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, page 361–376, USA, 2018. USENIX Association.

[29] C.-A. Staicu, M. Pradel, and B. Livshits. Understanding and automatically preventing injection attacks on node. js. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[30] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, E. T. Barr, and W. Le. A comprehensive study of the capabilities of large language models for vulnerability detection. *arXiv preprint arXiv:2403.17218*, 2024.

[31] E. Tolhurst. How npm is jeopardizing open source projects. 2018.

[32] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. H. Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint*, 2023.

[33] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 351–361, New York, NY, USA, 2016. Association for Computing Machinery.

[34] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams. Malwarebench: Malware samples are not enough. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, pages 728–732, 2024.

[35] N. Zahan, P. Burckhardt, M. Lysenko, F. Aboukhadijeh, and L. Williams. Shifting the lens: Detecting malware in npm ecosystem with large language models. *arXiv preprint arXiv:2403.12196*, 2024.

[36] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, page 331–340, New York, NY, USA, 2022. Association for Computing Machinery.

[37] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 705–716, 2021.

[38] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, Santa Clara, CA, Aug. 2019. USENIX Association.