LLMsec: Evaluating the Security of Code Generated by Large Language Models

Maksim Makaranka, Krzysztof Stawarski
Faculty of Electronics and Information Technology,
Warsaw University of Technology, Warsaw, Poland

***Abstract*—We present LLMsec, a framework for evaluating the security of package dependencies generated by large language models in software engineering tasks. Our method uses Trivy scanner to automatically scan code outputs from models such as GPT-4.1 and DeepSeek V3, assessing vulnerability severity with CVSS metrics. Experiments across diverse tasks reveal significant variation in the security of dependencies proposed by different models. The findings support the need for automated supply-chain risk analysis in LLM-based development workflows.**

***Index Terms*—LLM, Large Language Models, Software Supply-Chain Security, Vulnerability Detection, Trivy, CVSS, LangChain, LangGraph**

I. Introduction

Large Language Models (LLMs) such as GPT-4.1 and DeepSeek V3 have recently achieved remarkable capabilities in automated code generation for a variety of software tasks. However, ensuring the security of code created by LLMs remains a major challenge, particularly with respect to third-party dependencies that may introduce critical vulnerabilities. Supply-chain attacks exploiting insecure packages have become a growing threat in modern software development. In this work, we systematically evaluate the security of package dependencies generated by leading LLMs, using Trivy—a state-of-the-art vulnerability scanner that classifies risks according to the CVSS (Common Vulnerability Scoring System) framework and public vulnerability databases. Our primary goal is to benchmark models based on the quantity and severity of detected vulnerabilities in the dependencies suggested by their outputs. The results highlight differences in security posture among models and provide insights for practitioners regarding the safe use of LLM-based code generation.

II. Background and Related Work

Recent advances in Large Language Models (LLMs) have enabled automatic source code generation that rivals or exceeds human-level performance in a wide range of programming tasks [1], [2]. For example, Codex achieves up to 70% functional correctness on benchmark datasets, a significant jump from previous models [1]. However, security and reliability remain pressing concerns, with studies revealing that LLMs frequently recommend packages with known vulnerabilities or synthesize code containing exploitable weaknesses [3], [4]. One systematic evaluation found around 40% of Copilot-generated code to be vulnerable across 89 cybersecurity scenarios [3]. Recent work demonstrates that adversaries can inject backdoors into instruction-tuned code LLMs, achieving attack success rates as high as 86% even with minimal poisoning of training data [4]. Traditional static and dynamic analysis methods are limited in scalability and integration with AI-based assistants, prompting a shift toward machine learning and automated approaches for vulnerability detection and mitigation [5]. Tools like Trivy [6] and Snyk [7] offer comprehensive post-hoc vulnerability scanning of dependencies, but their coupling with LLM pipelines remains ad hoc and lacks systematic benchmarks. While many studies have focused on code correctness or bug prediction [2], there is comparatively little research addressing supply-chain risks stemming from the dependencies that LLMs propose. This work addresses this gap by providing a unified framework for benchmarking and comparing the vulnerability exposure of LLM-suggested packages in realistic development scenarios.

III. Problem Statement and Research Questions

The rapid adoption of code-generating large language models raises concerns about the security of their recommended software dependencies. Even when the generated source code appears correct, included third-party packages may introduce hidden vulnerabilities that can compromise software integrity. To understand these risks, we focus on the following core problem: How secure are the package dependencies suggested by leading LLMs for common programming tasks, as measured by the presence and severity of known vulnerabilities?

Based on this, our study is guided by the following research questions:

- **RQ1:** What is the overall number and severity of vulnerabilities present in the dependencies generated by different LLMs?
- **RQ2:** Are there meaningful differences between state-of-the-art language models in terms of the risk profile of their recommended dependencies?
- **RQ3:** Which types of vulnerabilities and package patterns are most frequently introduced during LLM-driven code generation?

Addressing these questions aims to provide practical insights into the security implications of LLM-assisted programming and inform safe usage guidelines.

IV. Methodology Overview

This study aims to evaluate and compare the security posture of various leading Large Language Models (LLMs) when tasked with generating code for common software development projects. The core methodology involves prompting these LLMs with a diverse suite of 20 well-defined technical project descriptions. For each prompt, the LLMs are expected to generate the necessary code, including crucial dependency files (e.g., package.json, requirements.txt, pom.xml, Gemfile). These dependency files, representing the libraries and versions selected by each LLM for each task, are then systematically collected for further analysis.

Subsequently, each set of generated dependencies is subjected to a security vulnerability scan using Trivy, an open-source, comprehensive vulnerability scanner. The Trivy output, which details identified vulnerabilities, their severity (based on CVSS scores), and fix status, is captured in JSON format for each LLM and each task.

Finally, this vulnerability data is aggregated and analyzed. We generate comparative statistics and visualizations to illustrate the number and severity of vulnerabilities introduced by the library choices of each LLM across the entire benchmark suite, as well as the fix status of these vulnerabilities. This allows for a quantitative comparison of the LLMs' tendencies to select secure versus vulnerable software components.

A. Evaluated Language Models

The language models evaluated in this study include several prominent and widely-used LLMs recognized for their code generation capabilities. The models compared are:

- OpenAI's GPT-4.1
- DeepSeek-V3
- xAI's Grok-3

These models were chosen to represent a spectrum of current AI code generation technologies, potentially differing in their training data, architectural nuances, and underlying safety mechanisms regarding library selection.

B. Security Metric

The primary security metric employed in this research is the identification and classification of known vulnerabilities within the software dependencies selected by each LLM. This is achieved using Trivy, a comprehensive open-source vulnerability scanner.

For each project task completed by an LLM, Trivy scans the generated dependency files (e.g., package.json for Node.js, requirements.txt for Python, pom.xml for Java/Maven, Gemfile for Ruby). The scanner cross-references the declared libraries and their versions against extensive vulnerability databases.

The key outputs from Trivy, which form our security metrics, are:

- Vulnerability Severity: Categorized into standard CVSS based classes such as CRITICAL, HIGH, MEDIUM, LOW, and UNKNOWN. This allows for a risk-based assessment of the LLM's library choices.
- Vulnerability Fix Status: Information such as fixed (a patched version is available), affected (vulnerable, no immediate fix from vendor), will_not_fix, etc. This provides insight into the actionability of mitigating the identified risks.

By aggregating these metrics across all tasks and LLMs, we can quantify and compare their propensity to introduce vulnerable dependencies.

C. Benchmark Task Suite

The benchmark suite consists of 20 distinct technical project tasks. These tasks were carefully designed to cover a broad range of common software development scenarios and complexities, thereby compelling the LLMs to make non-trivial decisions about library and framework selection.

The tasks were chosen to ensure diversity in programming languages as well as in application types. Prompts covered Python, Node.js, Java, Go, and Ruby, among others, which were used across tasks such as: backend development (with RESTful and GrapQL APIs), database interactions (both SQL and NoSQL) or communicating through APIs and message brokers.

The rationale for this diverse and complex suite was to simulate real-world development challenges. This forces LLMs to go beyond trivial code snippets and select full-fledged libraries for functionalities like database abstraction, web serving, request handling, security, and data manipulation, which are the primary subjects of our vulnerability scanning. The number of tasks (20) was chosen to provide a sufficiently large sample size for meaningful statistical analysis while remaining manageable for execution and data collection.

V. System Implementation

A. System Overview

The LLMsec framework is implemented as a modular, multi-agent system built on top of the LangGraph library, which extends LangChain's capabilities to enable graph-based orchestration of complex workflows involving Large Language Models (LLMs). The system exposes a single HTTP endpoint via a FastAPI server and automates the entire flow: from validating the user's request, through code or dependency generation with various LLMs, to security analysis of the produced artifacts. All code and documentation are openly available in the LLMsec GitHub repository [8].

B. Agent-Based Pipeline and Workflow

At the core of the LLMsec system is a dynamically constructed workflow graph, managed in `graph.py` using LangGraph's *StateGraph* abstraction. Each node in this graph represents an autonomous agent that is responsible for a distinct stage in the overall evaluation process:

- **Validation Agent:** Initially, the system verifies that the user-submitted task description is a valid, actionable software engineering problem. This is achieved using a language model in a low-cost configuration and is implemented in `validation_agent.py`.

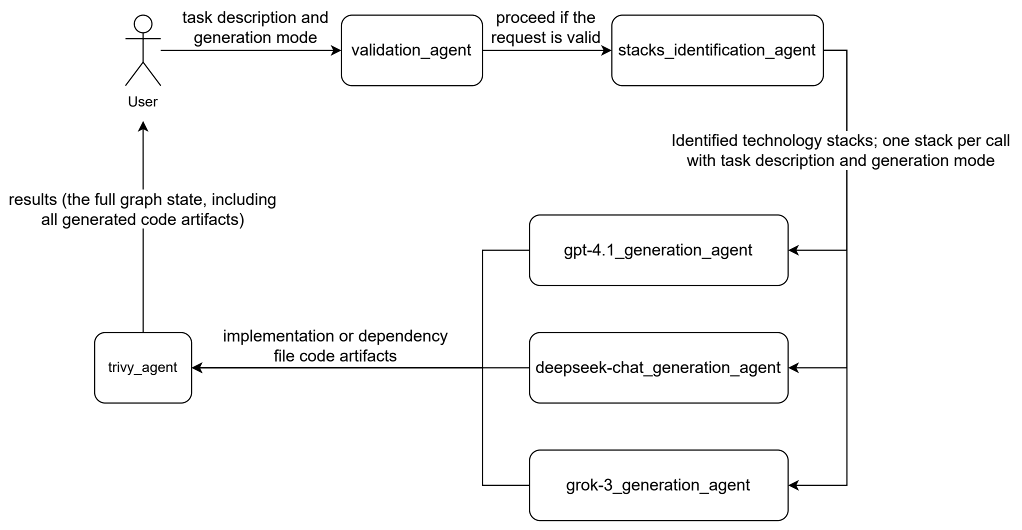
task description and generation mode
User
validation_agent
proceed if the request is valid
stacks_identification_agent
Identified technology stacks; one stack per call with task description and generation mode
results (the full graph state, including all generated code artifacts)
gpt-4.1_generation_agent
implementation or dependency file code artifacts
trivy_agent
deepseek-chat_generation_agent
grok-3_generation_agent

Fig. 1. Agent-based graph workflow in LLMsec system. Each node represents an autonomous agent responsible for a specific analysis step. Edges indicate transitions and data flow between agents.

- **Stacks Identification Agent:** Once the instruction passes validation, another agent identifies appropriate technology stacks for the task (e.g., combinations of language, framework, and database), as implemented in `stacks_identification_agent.py`.
- **Generation Agents:** For each combination of LLM and technology stack, a dedicated agent (see `generation_agent.py`) generates implementation code or dependency files according to the specified generation mode ("code" or "dependencies").
- **Trivy Agent:** This agent aggregates all generated files and invokes Trivy, an industry-grade vulnerability scanner, to analyze the produced dependencies or code artifacts for known vulnerabilities, based on public security databases and the CVSS scoring system. The relevant logic is in `trivy_agent.py`.

The graph-based architecture allows for scalability (easy integration of new agents or LLMs) and clear separation of concerns between different stages of the processing pipeline (see Figure 1).

C. LLM Abstraction and Extensibility

Support for multiple LLMs (e.g., OpenAI GPT-4.1, DeepSeek V3, Grok-3) is facilitated by abstraction mechanisms implemented in `llm_provider.py`. This design allows the system to easily incorporate new models for benchmarking. Each LLM's configuration, API credentials, and model selection logic are centralized in this module for maintainability and extensibility.

D. Prompt Engineering and Chain Design

Prompt templates for all major processing steps are defined using LangChain's compositional primitives. Dedicated prompt chains in files like `code_generation_chain.py` and `dependencies_generation_chain.py` ensure that every LLM invocation is deterministic (via temperature and JSON output constraints) and domain-specific for the type of artifact being generated. This approach supports systematic comparison across models and tasks.

E. API Endpoint

The LLMsec system provides a single, streamlined API endpoint accessible via a POST request to `/ask`. The endpoint accepts two parameters:

- `task_description` – a natural language description of the desired software engineering task,
- `generation_mode` – a string flag indicating the analysis focus: either `"code"` (for the code blocks generation) or `"dependencies"` (to only generate the packages or dependency files).

Upon receiving a request, the FastAPI server (see `server.py`) orchestrates the execution of the graph pipeline, aggregates agent outputs, and returns the full final state of the graph for comprehensive process transparency.

F. Summary

Through its compositional, agent-based pipeline leveraging LangGraph, LLMsec provides a reproducible, extensible, and systematic framework for benchmarking the security of LLM-generated code and dependencies across various technology stacks. The modular design of LLMsec allows new language models to be integrated and evaluated with minimal effort, making it well-suited for tracking rapid developments in the field. As a result, LLMsec not only streamlines present-day LLM benchmarking, but also offers a practical foundation for future research and automation in the secure analysis of AI-generated code.

VI. Experimental Setup

A. Hardware and Software Environment

The LLMs evaluated in this study were accessed via their respective third-party APIs. Consequently, no specialized local hardware (e.g., GPUs or high-performance CPUs) was required for model inference. The evaluation scripts, orchestrating the experiments and data collection, were executed on a Linux host system.

Vulnerability scanning was performed using Trivy version 0.63.0. The vulnerability database utilized by Trivy was version 2, with its last update recorded on 2025-06-02 at 12:20:25 UTC. The LLM interaction and orchestration layer, implemented in `server.py` (as detailed below), leveraged the LangChain and LangGraph libraries for managing communication flows.

B. Evaluation Procedure

The end-to-end evaluation process involved two main stages: data generation and vulnerability scanning, followed by data analysis and visualization.

First, the `generate_data.py` script initiated the process. This script dispatched a predefined set of prompts (described previously) to a custom API endpoint managed by `server.py`. The `server.py` component then relayed these prompts to each of the three selected LLMs. The code outputs generated by each LLM were then automatically analyzed for known vulnerabilities using the configured Trivy scanner. The raw scan results, including details of any identified vulnerabilities and their corresponding CVSS metrics, were systematically stored in JSON files for subsequent analysis.

In the second stage, the `analyze_data.py` script processed these JSON files. This script aggregated the vulnerability data, specifically counting the number of vulnerabilities detected in the dependencies proposed by each LLM and categorizing them by severity based on CVSS scores and status. This processed and aggregated data was then used to generate the histograms and other quantitative comparisons presented in this article, allowing for an assessment of the relative security risks associated with dependencies suggested by different models.

VII. Results

A. Overall Vulnerability Statistics

The comprehensive analysis of code outputs generated by the three LLMs—DeepSeek-V3, GPT-4.1, and Grok-3—revealed a total of 141 vulnerabilities across all suggested dependencies. Since different models could generate code using the same vulnerable libraries, these vulnerabilities are not unique to the model and do overlap to some degree.

As detailed in Table I, there was considerable variation in the number of vulnerabilities introduced by each model. Grok-3 generated dependencies with the highest number of vulnerabilities (80), followed by GPT-4.1 (33), and DeepSeek-V3 (28). Notably, for all vulnerabilities detected during our experiments, corresponding fixes or patched versions of the affected packages were available, indicating a clear path to remediation for developers.

TABLE I
Vulnerability Analysis Results by LLM Model.

Model	**Total Vulns.**	**Critical**	**High**	**Medium**	**Low**
DeepSeek-V3	28	0	16	10	2
GPT-4.1	33	3	12	12	6
Grok-3	80	3	36	36	5
Overall Summary	**141**	**6**	**64**	**58**	**13**

B. Severity Distribution Analysis

To understand the potential impact of the identified vulnerabilities, we analyzed their distribution according to CVSS severity scores. Figure 2 illustrates the aggregated severity breakdown across all dependencies suggested by the LLMs. The most prevalent category was 'High' severity, accounting for 64 vulnerabilities. This was followed by 'Medium' severity with 58 instances, 'Low' severity with 13 instances, and 'Critical' severity with 6 instances. The presence of 'Critical' and a significant number of 'High' severity vulnerabilities underscores the potential security risks that can be inadvertently introduced into software projects through LLM-generated dependency recommendations if not carefully vetted.

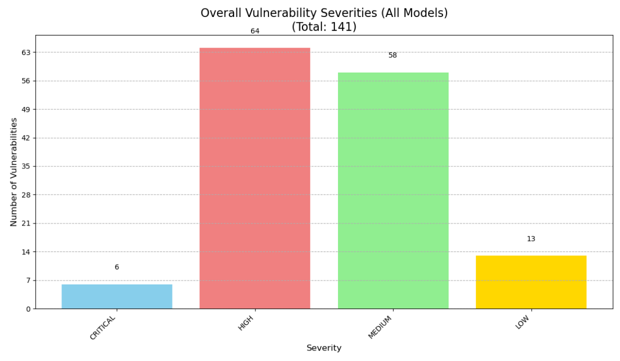

Fig. 2. Vulnerabilities detected in code produced by all LLMs, grouped by severity.

C. Model-Specific Error Patterns

Further examination of the vulnerability data reveals distinct patterns in the security profiles of dependencies suggested by each LLM.

Figure 3 shows that while DeepSeek-V3 introduced the fewest total vulnerabilities (28), the majority of these were classified as 'High' (16) or 'Medium' (10) severity, with no 'Critical' vulnerabilities detected.

In contrast, GPT-4.1, as depicted in Figure 4, suggested dependencies containing 33 vulnerabilities, including 3 'Critical', 12 'High', and 12 'Medium' severity issues.

Grok-3 (Figure 5) not only proposed dependencies with the highest overall count of vulnerabilities (80) but also contributed significantly to the 'High' (36) and 'Medium'

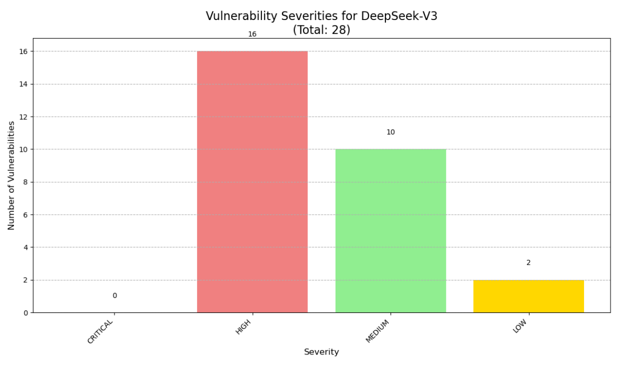

Fig. 3. Vulnerabilities detected in code produced by DeepSeek-V3, grouped by severity.

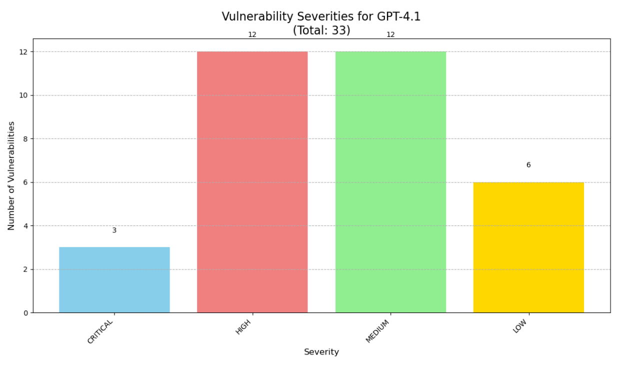

Fig. 4. Vulnerabilities detected in code produced by GPT-4.1, grouped by severity.

(36) categories, along with 3 'Critical' vulnerabilities. These model-specific distributions highlight that different LLMs exhibit varying propensities for recommending dependencies with specific types and severities of vulnerabilities, suggesting that the choice of LLM can influence the baseline security posture of the generated code.

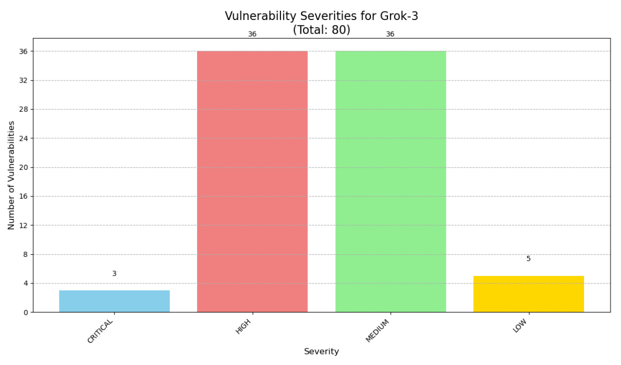

Fig. 5. Vulnerabilities detected in code produced by Grok-3, grouped by severity.

VIII. Discussion

A. Answering the Research Questions

The findings from our experimental evaluation of LLM-generated package dependencies offer valuable insights into the security landscape of AI-assisted software development. Our study was guided by three primary research questions, which we now address based on the presented results.

Regarding RQ1, which inquired about the overall number and severity of vulnerabilities, our findings indicate a non-trivial presence of such issues. Across the three evaluated LLMs, a total of 141 vulnerabilities were identified in the suggested package dependencies, as detailed in Table I. The severity distribution, illustrated in Figure 2, revealed that 'High' severity vulnerabilities were the most common (64 instances), followed by 'Medium' (58), 'Low' (13), and 'Critical' (6). This directly quantifies the extent and provides a general risk profile of vulnerabilities introduced. While it is a crucial practical detail that all 141 detected vulnerabilities had available fixes, their initial introduction by LLMs remains a pertinent concern.

Addressing RQ2, which asked if meaningful differences exist between state-of-the-art language models in terms of the risk profile of their recommended dependencies, our experiments affirm that such differences are indeed present. Grok-3 introduced the highest number of total vulnerabilities (80), significantly more than GPT-4.1 (33) and DeepSeek-V3 (28). Furthermore, the severity profiles varied; for instance, DeepSeek-V3 did not suggest dependencies with 'Critical' vulnerabilities, whereas both GPT-4.1 and Grok-3 did, as shown in Figures 3, 4, and 5. GPT-4.1, for example, had a notable proportion of its vulnerabilities in the 'Critical' and 'Low' categories, contrasting with DeepSeek-V3, which predominantly featured 'High' and 'Medium' severity issues. These variations confirm that the choice of LLM can influence the security risk associated with generated dependencies.

Finally, for RQ3, concerning the types of vulnerabilities and package patterns most frequently introduced, our current results primarily highlight that 'High' and 'Medium' severity vulnerabilities were the most common overall (Figure 2). A more granular analysis of specific CWEs or commonly implicated vulnerable packages, while not explicitly detailed in the preceding results, represents a vital area for deeper qualitative investigation, and the data collected forms a solid basis for such future work. The current findings suggest that LLMs may not consistently prioritize the latest, most secure package versions, potentially drawing from training data that includes outdated or insecure examples.

B. Threats to Validity

Several factors could potentially influence the validity of our findings. Internal validity might be affected by the design of prompts used to elicit code, as specific phrasings could inadvertently guide LLMs towards more or less secure suggestions despite our aim for diverse tasks. The inherent non-determinism of LLM APIs could also introduce slight variations upon re-running experiments, though we mitigated this by querying each model for every prompt and adjusting the model's temperature to minimize the nondeterministic aspects of token generation. Furthermore, the accuracy of Trivy, while robust, is not absolute; false positives or negatives are possible, although its recent vulnerability database update (UpdatedAt: 2025-06-02) is a strength.

With respect to external validity, our focus on three prominent LLMs means the findings may not universally generalize to all available models, particularly those with differing architectures or training data. The "diverse tasks" covered a specific set, and vulnerability patterns might differ for other programming languages, package ecosystems, or more specialized software engineering tasks. The continuous evolution of LLMs also means that the observed security performance might change with newer model versions, impacting temporal validity.

Construct validity concerns arise from using vulnerability counts and CVSS severity as proxies for actual security risk, as true exploitability is context-dependent. Additionally, our method relies on Trivy's capability to identify dependencies, and complex or unconventional declarations might be overlooked.

Regarding conclusion validity, while observed differences between models are based on direct counts and severity distributions, formal statistical tests for significance were not detailed in the results but would strengthen the assertions about "meaningful differences."

C. Practical Implications

The results of this study carry several practical implications for different stakeholders in the software development lifecycle.

For practitioners and developers, a key takeaway is the necessity for vigilance; they should not blindly trust or directly implement package dependencies suggested by LLMs without thorough vetting. Integrating automated vulnerability scanners like Trivy into the development workflow, especially when incorporating LLM-generated code, is crucial, and our LLMsec framework exemplifies such an approach. Given that all detected vulnerabilities had fixes, practitioners should also prioritize applying these patches and maintaining up-to-date dependencies. Furthermore, an awareness that different LLMs may carry different risk profiles regarding suggested dependencies is important.

Model vendors should consider these findings as motivation for enhancing the security-awareness of their models. This could involve incorporating security best practices and up-to-date vulnerability information into training and fine-tuning processes. Implementing safety layers or mechanisms that warn users about potentially insecure or outdated package suggestions could also prove beneficial. Greater transparency regarding the recency of training data concerning software packages would help users assess potential risks.

Looking towards future research and tooling, efforts should be directed towards expanding the scope of such studies to include a broader range of LLMs, programming languages, and more complex software engineering tasks. A deeper root cause analysis is needed to understand why LLMs suggest vulnerable dependencies, whether due to prevalence in training data or knowledge cut-off dates. The development of standardized benchmarks for evaluating the security of LLM-generated code and dependencies would be valuable. Moreover, exploring mitigation techniques, such as prompting LLMs for more secure code or automatically suggesting safer alternatives, is a promising avenue. Our findings strongly support the continued development and adoption of tools like LLMsec for automated supply-chain risk analysis in LLM-augmented development environments.

References

[1] M. Chen, J. Tworek, H. Jun *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] J. Austin, A. Odena, M. Nye *et al.*, "Program synthesis with large language models," in *Advances in Neural Information Processing Systems*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[3] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of github copilot's code contributions," *IEEE Symposium on Security and Privacy*, 2021. [Online]. Available: https://arxiv.org/abs/2108.09293

[4] M. I. Hossen, J. Zhang, Y. Cao, and X. Hei, "Assessing cybersecurity vulnerabilities in code large language models," *Proceedings of the 45th IEEE Symposium on Security and Privacy*, 2024, preprint available at https://arxiv.org/abs/2404.18567v1.

[5] T. Marjanov, I. Pashchenko, and F. Massacci, "Machine learning for source code vulnerability detection: What works and what isn't there yet," *IEEE Security & Privacy*, vol. 20, no. 5, pp. 23–31, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9859261

[6] "Trivy - simple and comprehensive vulnerability scanner," https://github.com/aquasecurity/trivy.

[7] "Snyk - find and automatically fix vulnerabilities in your code," https://snyk.io/.

[8] M. Makaranka and K. Stawarski, "LLMsec: Code generation security benchmark," 2025, available at https://github.com/maksanm/LLMsec.