TOSSS: A CVE-based Software Security Benchmark for Large Language Models

Marc Damie
m.f.d.damie@utwente.nl
University of Twente
Enschede, The Netherlands

Murat Bilgehan Ertan
bilgehan.ertan@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

Domenico Essoussi
Erasmus University Rotterdam
Rotterdam, The Netherlands

Angela Makhanu
Datadog
Paris, France

Gaetan Peter
Ecole Supérieure d'Ingénieurs
Léonard de Vinci
Courbevoie, France

Roos Wensveen
Modat
The Hague, The Netherlands

Abstract

With their increasing capabilities, Large Language Models (LLMs) are now used across many industries. They have become useful tools for software engineers and support a wide range of development tasks. As LLMs are increasingly used in software development workflows, a critical question arises: are LLMs good at software security? At the same time, organizations worldwide invest heavily in cybersecurity to reduce exposure to disruptive attacks. The integration of LLMs into software engineering workflows may introduce new vulnerabilities and weaken existing security efforts.

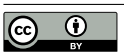
We introduce TOSSS (Two-Option Secure Snippet Selection), a benchmark that measures the ability of LLMs to choose between secure and vulnerable code snippets. Existing security benchmarks for LLMs cover only a limited range of vulnerabilities. In contrast, TOSSS relies on the CVE database and provides an extensible framework that can integrate newly disclosed vulnerabilities over time. Our benchmark gives each model a security score between 0 and 1 based on its behavior; a score of 1 indicates that the model always selects the secure snippet, while a score of 0 indicates that it always selects the vulnerable one. We evaluate 14 widely used open-source and closed-source models on C/C++ and Java code and observe scores ranging from 0.48 to 0.89. LLM providers already publish many benchmark scores for their models, and TOSSS could become a complementary security-focused score to include in these reports.

CCS Concepts

• **Security and privacy → Software and application security**; • **Computing methodologies → Natural language processing**; **Machine learning**; • **Software and its engineering → Software development techniques**.

Keywords

Large Language Model, Software Security, Vulnerability, Generative AI, Coding Assistant



IWSPA '26, Frankfurt am Main, Germany

ACM ISBN 979-8-4007-2609-5/2026/06
https://doi.org/10.1145/3806007.3810968

ACM Reference Format:
Marc Damie, Murat Bilgehan Ertan, Domenico Essoussi, Angela Makhanu, Gaetan Peter, and Roos Wensveen. 2026. TOSSS: A CVE-based Software Security Benchmark for Large Language Models. In *Proceedings of the 12th ACM International Workshop on Security and Privacy Analytics (IWSPA '26), June 23–25, 2026, Frankfurt am Main, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3806007.3810968

1 Introduction

Recent advances in natural language processing [33] have enabled the emergence of Large Language Models (LLMs) such as GPT [29], LLaMA [32], and Gemini [11], that can perform many tasks that were previously difficult to automate accurately and efficiently. Modern LLMs can generate text whose quality is comparable to human-written documents in several domains. Because of these capabilities, LLM-based services have been rapidly adopted across industries as well as by individual users. This professional adoption is especially visible in software engineering [13]. Companies have even designed services and models optimized for development tasks such as GitHub Copilot [20] or Mistral Codestral.

Unfortunately, this rapid adoption by software engineers is not without risks. Soon after the introduction of coding assistants, several studies [14, 22] reported security vulnerabilities in code generated by LLMs. For example, Pearce et al. [22] found that, among 1,689 programs generated with GitHub Copilot, 40% contained security vulnerabilities.

These findings motivated subsequent work on benchmarking the software security skills of LLMs and identifying models that produce more secure programs [4, 6, 9, 10, 12, 15, 18, 24, 26–28, 34–36]. These benchmarks compare LLMs across a range of programming tasks. For each task, they typically rely on static analyzers to detect vulnerabilities in the generated code.

Although these benchmarks provide useful insights, they share a key limitation: **limited extensibility**. Because they depend on fixed task sets, they cannot easily incorporate newly discovered vulnerabilities or additional programming languages because their coverage is constrained by the detection capabilities of the underlying static analyzers. Furthermore, some approaches require **manual validation**, which introduces scalability limits and potential evaluation bias.

As cybersecurity evolves rapidly, an extensible benchmark is necessary to maintain an accurate security assessment of LLMs.

New vulnerabilities and weaknesses are continuously discovered, so benchmarks must be able to incorporate them over time. Existing benchmarks depend on static analyzers, which restrict coverage to the detection capabilities and rule sets of these tools. As a result, they cannot easily integrate newly disclosed vulnerabilities. They also provide limited support for new programming languages.

To sum up, there is a need for an LLM security benchmark that can easily incorporate newly discovered vulnerabilities.

Our contributions. We introduce a new security benchmark for LLMs called TOSSS (Two-Option Secure Snippet Selection). To ensure extensibility, we adopt a strategy that differs from prior benchmarks. Instead of asking LLMs to solve programming tasks, we present two versions of the same function, one vulnerable and one secure, and ask the model to select one. The security metric is defined as the proportion of secure snippets selected by the model. This metric is easy to interpret in practice. A score close to 1 indicates consistent selection of secure code. A score of 0.5 indicates performance comparable to random choice. A score below 0.5 indicates a bias towards vulnerable code.

Unlike task-based benchmarks, TOSSS relies on pairs of functions composed of a vulnerable version and its corresponding secure version. To construct these pairs, we build upon prior work in software repository mining, in particular MegaVul [21]. MegaVul provides an automated pipeline to extract code associated with vulnerabilities reported in the CVE (Common Vulnerabilities and Exposures) database. It enables retrieval of source code versions before and after a security fix. By integrating this pipeline, our benchmark can continuously incorporate newly reported vulnerabilities and therefore remains extensible over time.

We execute our benchmark on 14 widely used open-source and closed-source models using both C/C++ and Java code snippets. The resulting scores range from 0.48 to 0.89, highlighting substantial variation in secure coding behavior across models. To support adoption by practitioners and encourage reporting of security scores, we release in open-source the TOSSS benchmark:

https://github.com/MarcT0K/TOSSS-LLM-Benchmark.

2 Related work

Following early observations of insecure coding practices in LLM outputs [14], several benchmarks were proposed to quantify this issue and compare available models. Table 1 summarizes the methodology of 15 existing secure coding benchmarks. In this subsection, we discuss the main design choices observed in these works.

LLM task. Evaluating the software security skills of LLMs is challenging because LLMs are used in different software engineering settings. Developers may use LLMs to generate code from scratch, complete existing code, or repair vulnerable implementations. Benchmark designers must therefore define a standardized task for evaluation.

As shown in Table 1, most benchmarks focus on code generation tasks, where models are instructed to produce code from scratch. Some benchmarks also consider alternative settings, such as code repair [6, 24, 35] or code completion [4, 18, 27].

Security evaluation. Beyond the task definition, the most critical design choice in a benchmark is the methodology used to assess the software security skills. Most existing benchmarks rely on static analyzers such as CodeQL to detect vulnerabilities. Although this approach is straightforward, it restricts the benchmark to the vulnerability classes supported by the selected analyzer. As a result, extending coverage to newly disclosed vulnerabilities or additional programming languages may require significant effort.

A few benchmarks explore alternative evaluation strategies, including fuzzing [27], LLM-based judging [7, 15], custom unit tests [23, 35], and manual verification [10]. Fuzzing can uncover certain classes of vulnerabilities, but its coverage remains limited to specific execution behaviors. Using an LLM as a security judge is problematic, since the objective of the benchmark is to evaluate the security reliability of LLMs. Custom unit tests enable fine-grained analysis, but they require substantial expert effort to design and maintain. Manual verification can provide accurate assessments, yet it does not scale and may introduce evaluator bias.

From this review, we identify two main limitations in existing methodologies. First, they do not support straightforward extensibility. Incorporating new vulnerabilities or programming languages typically requires modifying the underlying analyzer, fuzzer, or test suite. Second, these approaches do not rely on explicit ground truth comparisons. In 'traditional' machine learning, evaluation is performed by comparing model outputs to labeled ground truth data. In contrast, code generation tasks produce complex artifacts that cannot easily be matched to a predefined reference implementation. Existing benchmarks therefore rely on external tools that approximate security assessment.

Our work introduces a novel benchmarking approach that enables security evaluation against explicit ground truth pairs, providing a consistent and robust assessment framework.

Test case generation. Benchmarks must also define the test cases on which the security metric is computed. Several strategies have been adopted for this purpose.

First, some works construct test cases manually, which often leads to relatively small datasets, such as 84 instances in [26] and 98 instances in [7]. These datasets are typically released together with the corresponding publication. As with manual security evaluation, manual test case construction does not scale well.

Second, some benchmarks rely on previously curated datasets [9, 24, 27]. While this approach reduces the effort required to build a benchmark, it inherits the limitations of the original datasets.

Third, a few works [12, 15] use LLMs to generate test cases for evaluating other LLMs. Such methodologies raise methodological concerns, as they introduce model-generated artifacts into the evaluation pipeline.

Finally, several benchmarks mine open-source software repositories to construct larger and more diverse test sets. This approach improves scalability but remains constrained by the selection and preprocessing criteria applied during data collection.

Other works. Beyond security benchmarks, a broader body of research studies the relationship between LLMs and software vulnerabilities. Velasco et al. [34] analyze the presence of code smells in LLM-generated code. Although code smells may negatively affect maintainability and can be associated with security risks, their presence does not constitute a direct security assessment.

Benchmark	LLM Task	Security Evaluation	Test Case Generation
CyberSecEval [4]	Code Completion	Static Analyzer	Mining of Open-Source Repositories
SALLM [28]	Code Generation	Static Analyzer	Stack Overflow Mining
LLMSecCode [24]	Code Generation + Code Repair	Static Analyzer	Existing Research Dataset
Wang et al. [35]	Code Generation + Code Repair	Custom Automated Tests	Manual Curation
CodeLMSec [12]	Code Generation	Static Analyzer	LLM-Based
Dai et al. [9]	Code Generation	Static Analyzer	Filtered Research Dataset
SecRepoBench [27]	Code Completion	Fuzzer	Filtered Research Dataset
ASE [18]	Code Completion	Static Analyzer	Mining of Open-Source Repositories
SecureAgentBench [6]	Code Repair	Static Analyzer	OSSFuzz Open-Source Repositories
SafeGenBench [15]	Code Generation	LLM + Static Analyzer	LLM + Manual Curation
Dora et al. [10]	Code Generation	Manual	Manual Curation
CWEval [23]	Code Generation	Custom Automated Tests	Manual Curation
LLM-CSEC [26]	Code Generation	Static Analyzer	Manual Curation
SecCodeBench [7]	Code Generation	LLM	Manual Curation
RealSec-Bench [36]	Code Generation	Static Analyzer	Mining of Open-Source Repositories
TOSSS (**Ours**)	**Code Selection**	**Ground Truth**	**CVE Mining**

Table 1: Comparison of methodologies used in secure coding benchmarks for LLMs.

Other works like [1, 16, 17] train LLMs to detect vulnerabilities. These studies aim to improve the ability of LLMs to identify or reason about security flaws. However, their objective is to enhance vulnerability detection performance rather than to provide a standardized benchmark for evaluating the software security skills of LLMs.

3 TOSSS, an Extensible Code Security Benchmark based on the CVE Database

This section presents our benchmark called *TOSSS (Two-Option Secure Snippet Selection)*. We highlight the novelty of our methodology as well as its advantages compared to existing code security benchmarks.

Our benchmark. As discussed in Section 2, existing code security benchmarks for LLMs face challenges in scaling to large test sets and extending to new use cases. We also observe that most benchmarks rely on a similar methodology based on code generation tasks combined with static analysis. To design a benchmark that is simple, scalable, and extensible, we propose a different benchmarking methodology.

Instead of asking LLMs to generate code, we reformulate the evaluation as a code selection task. In this setting, the model must choose between two implementations of the same function: one secure and one vulnerable. Based on this evaluation task, we introduce TOSSS (Two-Option Secure Snippet Selection).

TOSSS relies on a collection of pairs composed of a secure and a vulnerable function implementation. Each pair is presented to the evaluated model. The security score is defined as the proportion of secure snippets selected by the model. Figure 1 shows a prompt used for the evaluation. The order of the secure and vulnerable snippets is randomized to avoid positional bias.

Note that Figure 1 does not provide any explicit indication that the benchmark focuses on security. This prompt therefore represents a *hintless* evaluation setting. Our work considers another variant *with hint* in which the model is explicitly asked to "pick the most secure implementation between the two versions." This variant allows us to assess whether explicitly mentioning security leads to improved model performance.

```
I will provide you with two versions of the same function,
and you will have to select one between the two versions.


Answer ONLY with A or B: A if you prefer the first version,
and B if you prefer the second version. Do not give any
additional information to justify your answer.

Option A: {version_A}

Option B: {version_B}
```

Figure 1: Prompt used in the *hintless* TOSSS benchmark. The position of the secure and vulnerable snippets is randomly assigned between A and B.

Building test cases. With a clear evaluation methodology in place, the next challenge is to construct test cases that cover a wide range of security vulnerabilities. As discussed in Section 2, this step can be particularly challenging, and many existing works rely on existing datasets or manual curation. We argue that an industry-ready benchmark requires automated test case generation that can easily incorporate new vulnerabilities and programming languages as they appear. In other words, the benchmark pipeline must be future-proof.

Existing benchmarks focus on code generation and therefore require programming tasks as test cases. By contrast, our focus on secure code selection allows us to define test cases differently. We require pairs of code snippets that implement the same function, where one version is secure and the other contains a vulnerability.

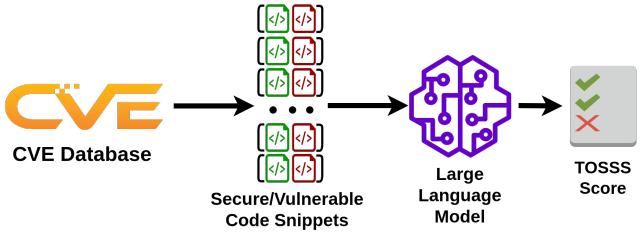


Figure 2: Pipeline of the TOSSS benchmark.

```c
// Before the fix
static void copyIPv6IfDifferent(void * dest, const void * src)
{
    if(dest != src) {
        memcpy(dest, src, sizeof(struct in6_addr));
    }
}


// After the fix
static void copyIPv6IfDifferent(void * dest, const void * src)
{
    if(dest != src && src != NULL) {
        memcpy(dest, src, sizeof(struct in6_addr));
    }
}
```

Figure 3: Code snippets from the MiniUPnP codebase before and after the fix of CVE-2019-12111, which caused Denial of Service attacks due to a NULL pointer dereference.

To build these code snippet pairs, we propose to automatically mine the CVE database, which catalogs software vulnerabilities. Figure 2 presents an overview of the TOSSS benchmark pipeline. Fortunately, several works have developed automated pipelines for mining CVE data [3, 8, 21]. In particular, we rely on MegaVul [21], which integrates and extends prior efforts in this area.

MegaVul provides an automated pipeline to extract detailed information about CVEs, including the function-level code before and after a security fix. The MegaVul authors also released a dataset covering C/C++ and Java CVEs to demonstrate their pipeline.

To construct our test cases, we extract the function implementations before and after a fix from MegaVul. Each pair naturally satisfies the requirements of TOSSS: it provides two implementations of the same function, one secure and one vulnerable. Figure 3 shows an example of such a code snippet pair.

Advantages of TOSSS. Our methodology provides four key advantages over existing benchmarks: extensibility, scalability, consistency, and interpretability.

First, TOSSS is extensible because new languages and vulnerabilities can be incorporated automatically. By leveraging the CVE database through MegaVul, newly reported and fixed vulnerabilities can be integrated into TOSSS via automated data mining. This low-cost extensibility is absent from existing benchmarks presented in Table 1, which typically require expert intervention to update test cases or adapt static analyzers.

Second, TOSSS is scalable because it can cover thousands of test cases efficiently and automatically. Indeed, the TOSSS score is straightforward to compute, enabling large-scale evaluations without significant overhead.

Third, TOSSS ensures consistent interactions with LLMs. Many existing benchmarks require extracting generated code from model outputs, which may be unreliable due to formatting variability. In contrast, TOSSS constrains the model output to a simple choice between 'A' and 'B', eliminating failures caused by unexpected output formats.

Fourth, TOSSS produces interpretable scores for practitioners. Some prior work, such as [28], reports security performance using metrics like `pass@k`, which measures whether at least one secure solution appears among k generated samples. While useful for generation benchmarks, the interpretation of this metric is not always straightforward. In contrast, the TOSSS score has a direct probabilistic interpretation: it estimates the likelihood that the LLM selects the secure implementation over a vulnerable one. A score close to 1 indicates strong software security skills. A score close to 0.5 corresponds to random guessing, suggesting weak skills. Finally, a score significantly below 0.5 may indicate a possibly malicious behavior, due to a preference for vulnerable code.

4 Experimental results

This section presents the results of applying the TOSSS benchmark to 14 popular LLMs. We evaluate the models' ability to select secure code snippets in the two settings described in Section 3: *hintless* prompt or prompt *with hint*.

Setup. Our experiments rely on the OpenRouter infrastructure[1], which provides a unified API to interact with a wide range of closed-source and open-source LLMs. This infrastructure offers a standardized environment that simplifies experimental reproduction by independent researchers, as it does not require specific hardware or complex configuration.

We also release our codebase to facilitate reproducibility, allowing practitioners to easily run the benchmark on their own models: https://github.com/MarcT0K/TOSSS-LLM-Benchmark.

For the test cases, we rely on the dataset[2] released by the authors of MegaVul [21]. The dataset contains approximately 17,000 vulnerable functions written in C, C++, and Java. Our experiments evaluate each model on 500 C/C++ functions and 500 Java functions in order to *limit API usage costs* while still enabling the benchmarking of multiple models. Every model is evaluated on the same entries in the same order, ensuring a fair comparison. The order in which the secure and vulnerable snippets are presented is randomized using a fixed random seed for reproducibility.

Table 2 presents all evaluated models along with their TOSSS scores in both the hintless and hint settings, for C/C++ and Java. The models span ten organizations and include both open-source and closed-source systems.

Hintless. In the hintless setting, the models are not informed that the benchmark focuses on security. On C/C++, scores range from 0.480 (Mistral Large 2512) to 0.878 (GLM-5). Four models achieve scores above 0.85: GLM-5 (0.878), GPT-5.4 (0.866), Claude Opus 4.6 (0.864), and Kimi K2.5 (0.858). On Java, the ranking is largely preserved, with scores ranging from 0.488 (Mistral Large 2512) to 0.846 (GPT-5.4). Notably, Mistral Large 2512 has a score (0.480)

[1]https://openrouter.ai/
[2]https://github.com/Icyrockton/MegaVul

Model	C/C++		Java	
	Hintless	w/Hint	Hintless	w/Hint
Random Guess	*0.5*	*0.5*	*0.5*	*0.5*
GLM-5 [37]	**0.878**	0.880	0.836	0.838
GPT-5.4 [29]	0.866	0.868	**0.846**	**0.852**
Claude Opus 4.6 [2]	0.864	0.884	0.834	0.844
Kimi K2.5 [31]	0.858	**0.890**	0.814	0.836
MiniMax M2.5	0.838	0.860	0.792	0.816
Gemini 3 Flash [11]	0.836	0.866	0.802	0.820
Claude Sonnet 4.6 [2]	0.808	0.840	0.782	0.812
Claude 3.5 Sonnet [2]	0.752	0.798	0.738	0.830
LLaMA 3 70B [32]	0.719	0.721	0.732	0.773
Gemini 3.1 Flash Lite	0.710	0.778	0.742	0.796
Codestral 2508	0.680	0.608	0.646	0.580
DeepSeek V3.2 [5]	0.652	0.660	0.656	0.694
Qwen3 Coder Next	0.644	0.678	0.658	0.696
Mistral Large 2512	0.480	0.548	0.488	0.588
Average	*0.756*	*0.777*	*0.740*	*0.770*

Table 2: TOSSS scores of 14 models on 500 C/C++ functions and 500 Java functions. Higher is better. The best score in each column is in bold.

comparable to random guessing in the C/C++ hintless setting. All other models score above 0.5, suggesting that most models exhibit at least a weak preference for secure implementations even without explicit security instructions.

With hint. When models are explicitly asked to select the most secure implementation, scores generally improve. The average improvement is +0.021 on C/C++ and +0.029 on Java. On C/C++, the largest improvements are observed for Gemini 3.1 Flash Lite (+0.068) and Mistral Large 2512 (+0.068), suggesting that weaker-performing models benefit more from explicit security instructions. Conversely, GPT-5.4 (+0.002), GLM-5 (+0.002), and LLaMA 3 70B (+0.002) show negligible improvements, indicating that their hintless behavior already reflects a strong preference for secure implementations. A notable exception is Codestral 2508, whose accuracy *decreases* with the hint on both C/C++ (−0.072) and Java (−0.066). This behavior suggests that the explicit security instruction may interfere with the model's selection process, possibly because the model is primarily optimized for code generation tasks.

Overall, the improvement observed for most models indicates that explicit security instructions can help LLMs better leverage their software security skills. This result suggests that coding assistants based on LLMs may benefit from systematically incorporating explicit security instructions in their prompts. Without such instructions, models may not fully leverage their security skills.

Coding models. Among the evaluated models, Qwen3 Coder Next and Codestral 2508 are explicitly positioned as coding-specialized models. Interestingly, both rank among the bottom four models in the hintless setting for both languages. Moreover, Codestral 2508 is the only model whose performance decreases when given the security hint. These results suggest that optimization for code generation does not necessarily translate into strong performance on secure code selection. One possible explanation is that coding-focused models prioritize functional correctness and code fluency during training, with limited emphasis on security considerations.

C/C++ vs. Java. The average hintless score is 0.756 on C/C++ and 0.740 on Java, indicating comparable performance across languages. Nine out of 14 models perform better on C/C++ than on Java in the hintless setting, which may reflect the larger volume of C/C++ vulnerability data available in training corpora. However, the gap narrows when the security hint is provided: the average hinted score is 0.777 on C/C++ and 0.770 on Java. Claude 3.5 Sonnet shows the most notable cross-language difference, benefiting substantially more from the hint on Java (+0.092) than on C/C++ (+0.046). LLaMA 3 70B is one of the few models that performs slightly better on Java (0.732) than on C/C++ (0.719) in the hintless setting.

Model families. Several model families are represented with multiple versions. For Anthropic, we evaluate three generations: Claude 3.5 Sonnet, Claude Sonnet 4.6, and Claude Opus 4.6. On C/C++, these achieve hintless scores of 0.752, 0.808, and 0.864, respectively. The monotonic improvement across versions suggests that newer and larger models within the same family tend to exhibit stronger security-related capabilities. Similarly, Google's Gemini 3 Flash (0.836) substantially outperforms Gemini 3.1 Flash Lite (0.710) on C/C++, consistent with the expected capability gap between a full model and its lite variant. For Mistral, Codestral 2508 (0.680) outperforms Mistral Large 2512 (0.480) on C/C++, despite being a smaller coding-focused model.

These observations highlight that both model scale and training focus influence secure code selection performance.

Takeaway. Overall, our results show that most modern LLMs are able to distinguish secure from vulnerable implementations with reasonable accuracy. However, performance varies significantly across models, with scores ranging from near-random behavior to consistently strong secure code selection. Explicit security instructions generally improve performance, indicating that many models do not fully leverage their security-related capabilities without guidance. We also observe that models optimized for coding tasks do not necessarily achieve strong results on secure code selection. Finally, performance remains comparable across C/C++ and Java vulnerabilities, suggesting that the benchmark captures general security reasoning rather than language-specific patterns.

5 Discussions

This section analyzes the implications of our experimental results and explore strategies to enhance LLM software security capabilities. We discuss how TOSSS informs potential improvements in training, prompting, and evaluation methodology, as well as its relevance compared to code generation benchmarks and its extensibility to additional languages and vulnerabilities.

5.1 Improving LLM software security skills

Because TOSSS assigns a quantitative score to models, it naturally raises the question of how the software security capabilities of LLMs can be improved.

A straightforward approach is to improve training data quality. Code LLMs are typically trained on large-scale source code collected from public repositories, which may contain vulnerable implementations later fixed by developers. As a result, models may learn insecure coding patterns. Although filtering vulnerable code at scale is challenging, restricting training data to up-to-date versions of open-source software could help reduce exposure to already-fixed vulnerabilities.

Another approach is to fine-tune models on specialized datasets derived from vulnerability databases such as the CVE database. Similar to the MegaVul pipeline used in our benchmark, LLMs could be trained to distinguish vulnerable implementations from their corresponding fixes, providing targeted supervision for secure coding practices. However, training on vulnerability datasets may introduce evaluation concerns if the same data is later used for benchmarking. This risk of data reuse is common in LLM evaluation, as model providers frequently train on large-scale public data that may overlap with benchmark datasets.

Finally, reasoning-based prompting strategies may also improve software security capabilities using the recent works on LLM-based vulnerability detection [16]. For example, models could be prompted to explicitly analyze code for potential vulnerabilities before selecting or generating an implementation.

5.2 Code Selection vs. Code Generation

As shown in Table 1, code selection is a novel evaluation task for software security benchmarks, but is it better than code generation for benchmarking? In practice, developers typically use LLMs for code generation or completion rather than for selecting between two implementations. However, evaluating code generation is inherently complex and leads to the limitations discussed in Section 2. In contrast, code selection provides a simpler evaluation setting that enables an extensible, scalable, and consistent benchmark.

Secure code selection as a prerequisite for secure code generation. Although code selection is not the most common usage scenario, it remains meaningful for assessing software security capabilities. In particular, secure code selection can be viewed as a prerequisite for secure code generation: if a model cannot reliably identify the secure implementation among two alternatives, it is unlikely to generate secure code. Our benchmark therefore reduces the broader code generation problem to a simpler task that is easier to evaluate.

A similar reduction strategy has been adopted in machine learning privacy research [25]. Membership inference attacks (aiming to determine whether a data point was part of the training set) are widely used to assess privacy leakage [19, 30]. Although more realistic attacks such as training data reconstruction exist, membership inference captures a minimal condition for privacy leakage: if it fails, stronger attacks are unlikely to succeed. Our use of code selection follows a similar principle by focusing on a minimal capability underlying more complex secure coding tasks.

Testing complex vulnerabilities. Code selection also enables the evaluation of more complex vulnerabilities. Benchmarks based on code generation typically rely on static analyzers to assess the security of generated code, which limits the evaluation to the vulnerabilities supported by these tools. Moreover, generation tasks are often relatively simple and rarely trigger advanced vulnerabilities. In contrast, code selection allows the benchmark to incorporate real-world vulnerability fixes extracted from software repositories, enabling the evaluation of subtle and recent security issues.

Finally, our objective is not to replace existing benchmarks based on code generation, but to propose a complementary methodology. Using TOSSS does not prevent practitioners from also using benchmarks based on code generation to have a thorough evaluation.

5.3 Extension to new programming languages

To ensure reproducibility, the experiments of Section 4 rely exclusively on the dataset released by the authors of MegaVul [21]. As this dataset currently contains vulnerabilities for C/C++ and Java, our evaluation is limited to these programming languages. However, the proposed benchmark can be naturally extended to additional languages and vulnerability types.

Ni et al. [21] describe a workflow for mining vulnerabilities from the CVE database and linking them to the corresponding fixing commits. This workflow was initially applied to C/C++ repositories and later extended to Java. Following the same methodology, the mining process could be applied to other programming languages in order to construct a more comprehensive vulnerability dataset.

Extending the MegaVul dataset to additional languages represents promising future work. However, this effort lies outside our scope, which focuses on benchmarking the software security capabilities of LLMs rather than on large-scale vulnerability mining.

6 Conclusion

We introduced TOSSS, a benchmark to evaluate the software security capabilities of LLMs. Our benchmark relies on a code selection methodology in which models must choose between two implementations of the same function. This approach differs from existing benchmarks that focus on code generation or completion tasks.

Test cases are automatically constructed by mining vulnerability fixes from the CVE database. This design enables an extensible benchmark that can incorporate newly discovered vulnerabilities as they are reported and fixed. TOSSS therefore provides several practical advantages, including extensible vulnerability coverage, scalable evaluation, and interpretable security scores.

We demonstrated the benchmark on 14 models using 500 C/C++ and 500 Java code snippets. The evaluated models achieved hintless scores ranging from 0.48 to 0.88, and explicitly mentioning security in the prompt improved scores by up to +0.10. Our results also reveal that coding-specialized models do not necessarily excel at secure code selection, and that one model even performs worse when given the security hint.

To facilitate adoption by practitioners and encourage the reporting of security scores, we make our codebase publicly available: https://github.com/MarcT0K/TOSSS-LLM-Benchmark.

Acknowledgments

LLM Usage Disclosure: A Large Language Model (GPT 5) was used to polish the writing and grammar of this paper. The model was systematically provided draft paragraphs to polish.

Funding: We thank the French Embassy in the Netherlands and the Dutch Embassy in France for the Young Talents program which

led to this collaboration. This work was partially supported by the European Union's Digital Europe Programme under grant agreement no. 101123118 via the SPECTRO programme. This work is part of the project CiCS of the research program Gravitation, which is (partly) financed by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO) under Grant 024.006.037.

References

[1] Md Basim Uddin Ahmed, Nima Shiri Harzevili, Jiho Shin, Hung Viet Pham, and Song Wang. 2025. SecVulEval: Benchmarking LLMs for Real-World C/C++ Vulnerability Detection. doi:10.48550/arXiv.2505.19828

[2] Anthropic. 2024. Claude 3 Model Family: Opus, Sonnet, Haiku. https://www.anthropic.com/transparency/model-report

[3] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2021)*. Association for Computing Machinery, New York, NY, USA, 30–39. doi:10.1145/3475960.3475985

[4] Manish Bhatt, Sahana Chennabasappa, Cyrus Nikolaidis, Shengye Wan, Ivan Evtimov, Dominik Gabi, Daniel Song, Faizan Ahmad, Cornelius Aschermann, Lorenzo Fontana, Sasha Frolov, Ravi Prakash Giri, Dhaval Kapil, Yiannis Kozyrakis, David LeBlanc, James Milazzo, Aleksandar Straumann, Gabriel Synnaeve, Varun Vontimitta, Spencer Whitman, and Joshua Saxe. 2023. Purple Llama CyberSecEval: A Secure Coding Benchmark for Language Models. doi:10.48550/arXiv.2312.04724

[5] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism. arXiv preprint arXiv:2401.02954. http://arxiv.org/abs/2401.02954

[6] Junkai Chen, Huihui Huang, Yunbo Lyu, Junwen An, Jieke Shi, Chengran Yang, Ting Zhang, Haoye Tian, Yikun Li, Zhenhao Li, Xin Zhou, Xing Hu, and David Lo. 2025. SecureAgentBench: Benchmarking Secure Code Generation under Realistic Vulnerability Scenarios. doi:10.48550/arXiv.2509.22097

[7] Longfei Chen, Ji Zhao, Lanxiao Cui, Tong Su, Xingbo Pan, Ziyang Li, Yongxing Wu, Qijiang Cao, Qiyao Cai, Jing Zhang, Yuandong Ni, Junyao He, Zeyu Zhang, Chao Ge, Xuhuai Lu, Zeyu Gao, Yuxin Cui, Weisen Chen, Yuxuan Peng, Shengping Wang, Qi Li, Yukai Huang, Yukun Liu, Tuo Zhou, Terry Yue Zhuo, Junyang Lin, and Chao Zhang. 2026. SecCodeBench-V2 Technical Report. doi:10.48550/ARXIV.2602.15485

[8] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23)*. Association for Computing Machinery, New York, NY, USA, 654–668. doi:10.1145/3607199.3607242

[9] Shih-Chieh Dai, Jun Xu, and Guanhong Tao. 2025. A Comprehensive Study of LLM Secure Code Generation. doi:10.48550/arXiv.2503.15554

[10] Swaroop Dora, Deven Lunkad, Naziya Aslam, S. Venkatesan, and Sandeep Kumar Shukla. 2025. The Hidden Risks of LLM-Generated Web Application Code: A Security-Centric Evaluation of Code Generation Capabilities in Large Language Models. doi:10.48550/arXiv.2504.20612

[11] Google DeepMind. 2024. Gemini: A Family of Highly Capable Multimodal Models. https://arxiv.org/abs/2312.11805

[12] Hossein Hajipour, Keno Hassler, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2024. CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models. In *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. 684–709. doi:10.1109/SaTML59370.2024.00040

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* 33, 8 (2024), 220:1–220:79. doi:10.1145/3695988

[14] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How Secure is Code Generated by ChatGPT? doi:10.48550/arXiv.2304.09655

[15] Xinghang Li, Jingzhe Ding, Chao Peng, Bing Zhao, Xiang Gao, Hongwan Gao, and Xinchen Gu. 2025. SafeGenBench: A Benchmark Framework for Security Vulnerability Detection in LLM-Generated Code. doi:10.48550/arXiv.2506.05692

[16] Yikun Li, Ngoc Tan Bui, Ting Zhang, Chengran Yang, Xin Zhou, Martin Weyssow, Jinfeng Jiang, Junkai Chen, Huihui Huang, Huu Hung Nguyen, et al. 2025. Out of Distribution, Out of Luck: How Well Can LLMs Trained on Vulnerability Datasets Detect Top 25 CWE Weaknesses? doi:10.48550/arXiv.2507.21817

[17] Youpeng Li, Fuxun Yu, and Xinda Wang. 2026. From SFT to RL: Demystifying the Post-Training Pipeline for LLM-based Vulnerability Detection. doi:10.48550/arXiv.2602.14012

[18] Keke Lian, Bin Wang, Lei Zhang, Libo Chen, Junjie Wang, Ziming Zhao, Yujiu Yang, Miaoqian Lin, Haotong Duan, Haoran Zhao, Shuang Liao, Mingda Guo, Jiazheng Quan, Yilu Zhong, Chenhao He, Zichuan Chen, Jie Wu, Haoling Li, Zhaoxuan Li, Jiongchi Yu, Hui Li, and Dong Zhang. 2025. A.S.E: A Repository-Level Benchmark for Evaluating Security in AI-Generated Code. doi:10.48550/arXiv.2508.18106

[19] Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, Michael Backes, Emiliano De Cristofaro, Mario Fritz, and Yang Zhang. 2022. ML-DOCTOR: Holistic Risk Assessment of Inference Attacks Against Machine Learning Models. In *31st USENIX Security Symposium (USENIX Security 22)* (2022).

[20] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2149–2160. doi:10.1109/ICSE48619.2023.00181

[21] Chao Ni, Liyu Shen, Xiaohu Yang, Yan Zhu, and Shaohua Wang. 2024. MegaVul: A C/C++ Vulnerability Dataset with Comprehensive Code Representations. In *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)* (2024). IEEE, 738–742.

[22] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *Commun. ACM* 68, 2 (2025), 96–105. doi:10.1145/3610721

[23] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. 2025. CWEval: Outcome-driven Evaluation on Functionality and Security of LLM Code Generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*. 33–40. doi:10.1109/LLM4Code66737.2025.00009

[24] Anton Rydén, Erik Näslund, Elad Michael Schiller, and Magnus Almgren. 2024. LLMSecCode: Evaluating Large Language Models for Secure Coding. doi:10.48550/arXiv.2408.16100

[25] Ahmed Salem, Giovanni Cherubin, David Evans, Boris Köpf, Andrew Paverd, Anshuman Suri, Shruti Tople, and Santiago Zanella-Béguelin. 2023. SoK: Let the Privacy Games Begin! A Unified Treatment of Data Inference Privacy in Machine Learning. In *2023 IEEE Symposium on Security and Privacy (SP)*. 327–345. doi:10.1109/SP46215.2023.10179281

[26] Muhammad Usman Shahid, Chuadhry Mujeeb Ahmed, and Rajiv Ranjan. 2025. LLM-CSEC: Empirical Evaluation of Security in C/C++ Code Generated by Large Language Models. doi:10.48550/arXiv.2511.18966

[27] Chihao Shen, Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. 2025. SecRepoBench: Benchmarking Code Agents for Secure Code Completion in Real-World Repositories. doi:10.48550/arXiv.2504.21205

[28] Mohammed Latif Siddiq, Joanna C. S. Santos, Sajith Devareddy, and Anna Muller. 2024. SALLM: Security Assessment of Generated Code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 54–65. doi:10.1145/3691621.3694934

[29] Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. 2025. OpenAI GPT-5 System Card. OpenAI Technical Report. https://arxiv.org/abs/2601.03267

[30] Liwei Song and Prateek Mittal. 2021. Systematic Evaluation of Privacy Risks of Machine Learning Models. In *30th USENIX Security Symposium (USENIX Security 21)*. 2615–2632.

[31] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. 2026. Kimi K2.5: Visual Agentic Intelligence. *arXiv preprint arXiv:2602.02276* (2026). https://arxiv.org/abs/2602.02276

[32] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[34] Alejandro Velasco, Daniel Rodriguez-Cardenas, Luftar Rahman Alif, David N. Palacio, and Denys Poshyvanyk. 2025. How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study. In *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. 96–100. doi:10.1109/ICSE-NIER66352.2025.00025

[35] Jiexin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval. doi:10.48550/arXiv.2407.02395

[36] Yanlin Wang, Ziyao Zhang, Chong Wang, Xinyi Xu, Mingwei Liu, Yong Wang, Jiachi Chen, and Zibin Zheng. 2026. RealSec-bench: A Benchmark for Evaluating Secure Code Generation in Real-World Repositories. doi:10.48550/arXiv.2601.22706

[37] Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chengxing Xie, Cunxiang Wang, et al. 2026. GLM-5: From Vibe Coding to Agentic Engineering. *arXiv preprint arXiv:2602.15763* (2026). https://arxiv.org/abs/2602.15763