2024 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-5487-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/SPW63631.2024.00014

Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers

Sivana Hamer, Marcelo d'Amorim, Laurie Williams
Department of Computer Science, North Carolina State University, Raleigh, North Carolina
Email: sahamer@ncsu.edu, mdamori@ncsu.edu, lawilli3@ncsu.edu

***Abstract*—Sonatype's 2023 report found that 97% of developers and security leads integrate generative Artificial Intelligence (AI), particularly Large Language Models (LLMs), into their development process. Concerns about the security implications of this trend have been raised. Developers are now weighing the benefits and risks of LLMs against other relied-upon information sources, such as StackOverflow (SO), requiring empirical data to inform their choice. In this work, our goal is to raise software developers' awareness of the security implications when selecting code snippets by empirically comparing the vulnerabilities of ChatGPT and StackOverflow. To achieve this, we used an existing Java dataset from SO with security-related questions and answers. Then, we asked ChatGPT the same SO questions, gathering the generated code for comparison. After curating the dataset, we analyzed the number and types of Common Weakness Enumeration (CWE) vulnerabilities of 108 snippets from each platform using CodeQL. ChatGPT-generated code contained 248 vulnerabilities compared to the 302 vulnerabilities found in SO snippets, producing 20% fewer vulnerabilities with a statistically significant difference. Additionally, ChatGPT generated 19 types of CWE, fewer than the 22 found in SO. Our findings suggest developers are under-educated on insecure code propagation from both platforms, as we found 274 unique vulnerabilities and 25 types of CWE. Any code copied and pasted, created by AI or humans, cannot be trusted blindly, requiring good software engineering practices to reduce risk. Future work can help minimize insecure code propagation from any platform.**

Keywords: Software Engineering Security, Empirical Study, Large Language Models, Software Supply Chain, Code Generation

1. Introduction

Artificial Intelligence (AI) as assistant tools have become commonplace within software development. According to Sonatype's 2023 State of the Supply Chain report [1], 97% of developers and security leads integrate generative AI into their software engineering processes. In particular, the report found that Large Language Models (LLMs) are commonly utilized by developers, with ChatGPT as the most used tool. Moreover, GitHub's State of Open Source Software report for 2023 [2] revealed that almost a third of open-source projects with stars have a maintainer using GitHub Copilot.

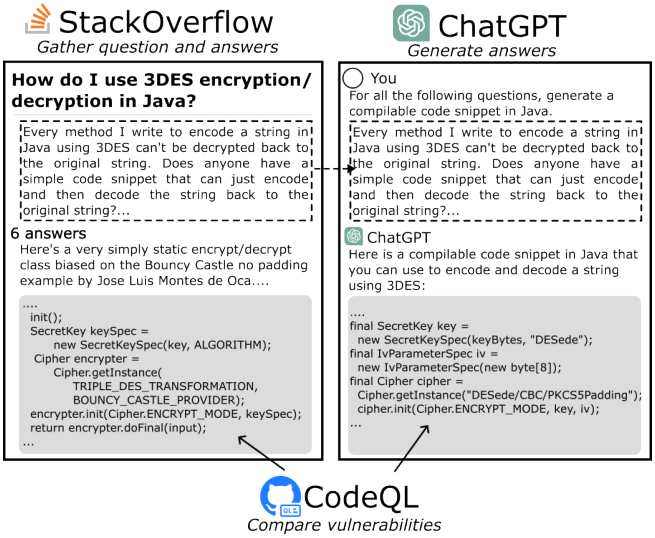

Figure 1. Example of how we compared ChatGPT and StackOverflow

As LLMs are now widely adopted within software engineering, practitioners have raised concerns regarding the security of the tool [3], [4]. Code generated by LLMs can contain vulnerabilities [5], [6] and security-related code smells [7]. Research has found vulnerable code generated by LLMs in GitHub [8]. Consequently, AI-generated code may affect the software supply chain. Sonatype's 2023 report on Generative AI in software development [9] highlighted that security is a primary concern among respondents. Alarmingly, despite security concerns, 74% of respondents felt pressured into using generative AI.

Developers are weighing the benefits and risks of using LLMs by contrasting them with other information sources, notably online forums like StackOverflow (SO), commonly used during development [10], [11]. Consequently, a developer may want to know how the security of LLMs-generated code compares with other web-based information sources. The choice of development source influences the security and functionality of the resulting code [12]. As current web-based information sources contain insecure code [13], [14], [15], [16], developers must know which information source is less insecure. Hence, choosing a more secure source can propagate fewer security issues to developers and guide policy decisions about using LLMs. Empirical

DOI 10.1109/SPW63631.2024.00014

data is needed to provide insights about the differences in code vulnerabilities between LLMs and other information sources for informed decision-making.

The goal of this work is *to raise software developers' awareness of the security implications when selecting code snippets by empirically comparing the vulnerabilities of ChatGPT and StackOverflow*. To achieve this goal, the following research questions were answered:

RQ1: What vulnerabilities differences are there between ChatGPT and SO code snippets?

RQ2: What types of vulnerabilities in terms of Common Weakness Enumeration (CWE) types are present for ChatGPT-generated code versus SO-answered?

To answer these questions, we conducted an experimental study comparing two widely adopted web-based information sources used in software development: ChatGPT [17] and SO [18]. Fig. 1 provides an example of how we compared the ChatGPT and SO. In this study, we used 108 Java security-related code snippets from SO taken from Chen et al. [14]. We queried ChatGPT, reusing the SO question as our prompt, to gather AI-generated code snippets for comparison. We then detected the vulnerabilities in the code with CodeQL [19], a static analysis tool developed by GitHub that provides CWE.

We found that ChatGPT-generated code contains fewer vulnerabilities and types of vulnerabilities compared to SO. Still, both platforms can be sources of insecure code propagation as we found 274 unique vulnerabilities and 25 types of CWE. Additionally, the vulnerabilities found in the ChatGPT-generated code and SO overlapped only in 25% of the vulnerabilities. The contributions of our work are:

- A comparison of the vulnerabilities found in 108 of ChatGPT-generated and SO-answered Java security-related code snippets.
- An enhanced dataset of 108 ChatGPT snippets associated with 87 questions and 90 answers from security-related SO questions [1].
- A list of 25 different vulnerabilities found in ChatGPT or SO snippets with their respective CWEs.

The remainder of this work is divided as follows. Section 2 details the methodology. Section 3 describes our results and Section 4 discusses our findings. Section 5 delineates the limitations of the study. Section 6 presents the related work. Finally, Section 7 concludes the work.

2. Methodology

We conducted an experimental study to compare both information sources in five steps. First, we selected the platforms under study, ChatGPT and SO (Step 1). We then selected security-related questions and answers from SO (Step 2) and collected the code snippets from the answers (Step 3). We prompted ChatGPT with the SO questions to generate code (Step 4) and gathered the generated snippets (Step 3). We then compare the gathered vulnerabilities of SO and ChatGPT using CodeQL (Step 5). In the following subsections, we detail each step.

1. Our dataset: https://zenodo.org/records/10806611

2.1. Step 1: Platform selection

We choose the two platforms to compare the code in LLMs and web-based information sources. For our LLM we chose ChatGPT, developed by OpenAI, for two main reasons. First, in the 2023 report on Generative AI in software development, Sonatype found that ChatGPT was the most used tool by 86% of the respondents [9]. Second, LLMs can be interacted with as a conversational agent in a similar way that a developer may ask a question and receive an answer in SO. The tools are thus directly comparable. For our traditional web-based information source we selected SO, an online question-and-answer forum for programmers, for the following reasons. SO remains widely utilized and frequented by its users. Based on data from February 2024, the site hosts 22 million users with 2,700 questions per day [20]. 92.5% of users visit the site at least weekly or a few times a month [21]. Additionally, prior work about insecure code propagation has investigated the SO site [12], [13], [14], [15], [16]. By choosing SO, we directly build upon and extend prior knowledge on the security of online information sources.

2.2. Step 2: Question and answer selection

We collect a dataset of questions with answers to sample SO answers and generate code with LLMs. We intentionally sampled the questions and answers to serve the purpose of our study through a purposeful sample [22]. We used a previously curated Java code snippet dataset by Chen et al. [14]. The dataset provided 1,429 SO answers with secure or insecure code snippets for security-related questions from 2008-2017. The dataset was chosen as it was previously manually curated and is publicly available. Additionally, a dataset with security-related questions was selected as not all questions in the wild have software vulnerabilities. Our sampling approach can thus serve as an upper bound for vulnerabilities found in SO posts. At the same time, Java is consistently a top programming language for developers in open source software [2]. After gathering the answers identifiers from the datasets, when analyzing the data we noticed that some snippets were partially stored. For example, imports were missing from the snippet in SO. At the same time, the dataset did not store the text of the associated questions and answers. As we needed the complete snippets, we mined SO to gather the answers through the StackExchange API [23]. Specifically, we collected the title and body of the questions with the answers. In total, 1,216 questions and 1,377 answers were mined. We could not gather 52 answers indicated in the dataset with their respective questions as they were no longer available in SO.

2.3. Step 3: Code snippet filtration

We curate the code snippets to analyze the answers' vulnerabilities. Hence, we collected the associated code

snippets for each of the 1,377 answers. To achieve this in an automated manner, we utilized the `Beautiful Soup 4` Python library [24] to parse the HTML of the answers to find the code blocks. We further checked the found code blocks to verify they were Java snippets using the `javalang` Python library [25]. We gathered in total 3,739 code snippets.

We further curated our data to verify that the snippets could compile further analysis (Step 5). To find code snippets that were not single lines of code, as one-liners are not compilable in Java, we gathered snippets stored in code blocks. Based on our analysis, the criteria were verified if the code block's parent HTML tag was of type `<pre>`. After applying the criteria, 2,432 snippets remained. As code snippets may have multiple Java classes in the same snippet or none at all, we then filtered that snippets had only one class name. We verified the criteria using the `javalang` Python library [25]. After the step, 526 snippets remained.

Finally, we compiled each snippet to satisfy the vulnerability detection tool requirements. We did not modify any code snippets to avoid introducing bias into the code. Hence, we verified and tried to fix compilation errors due to missing package libraries for all close snippets with a missing package error. In our initial compilation of the snippets, we gathered all errors due to missing packages through the regular expression `error: package [a-zA-z.]+ does not exist`. We manually searched and downloaded all missing packages through Maven the `.jar` files, searching all packages. Then, we recompiled all the snippets. We were unsuccessful for 62 packages missing in 86 code snippets, which may have also contained other compilation errors. In total, 216 snippets remained, corresponding to 189 questions and 204 answers.

2.4. Step 4: ChatGPT answers generation

We generate answers from ChatGPT to compare LLMs generated code. For each of the 216 remaining snippets, we asked the ChatGPT model the following prompt: *For all the following questions, generate a compilable code snippet in Java. [BODY].*

We selected a prompt that leveraged the SO post as we did not want to introduce additional bias to the model. Still, based on our tests, we were required to detail which programming language the question was for, which in our case was Java. Additionally, to verify that the code given was a complete program and not a code snippet, we specified that the code must be compilable due to the limitations of our vulnerability detection tool for Java. Finally, we prompted the model with the complete body of the SO question, though more expensive, was more comparable to how users asked questions in SO.

We thus prompted ChatGPT (`gpt-3.5-turbo-0613`) through the OpenAI REST API in Python [26] and stored each reply. Combining the answers with the questions generated a total of 237 pairs of SO snippets. We then followed the process outlined in Step 3 with some minor differences.

TABLE 1. SUMMARY OF VULNERABILITIES IN CHATGPT AND STACKOVERFLOW (SO). THE PLATFORM WITH FEWER VULNERABILITIES IS HIGHLIGHTED.

	GPT	SO	Overlap
Questions with vulnerabilities	77	75	81%
Answers with vulnerabilities	79	77	79%
Code snippets with vulnerabilities	87	83	79%
Vulnerabilities in snippets*	248	302	-
Unique vulnerabilities in snippets	158	183	25%

Chi-squared test significance: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$

First, as the API returned markdown, we processed the result with the regular expression `` ```[^`]*``` `` finding code blocks based on whether the text was enclosed by a code snippet block (`` ``` ``). Second, as we only selected code block snippets, we did not need to filter if they were code blocks. Additionally, we downloaded missing libraries for the ChatGPT snippets with the same procedure. We were unsuccessful for 29 imports for 11 code snippets that could have other compilation errors. After filtering 108 snippets pairs related to SO 87 questions and 90 answers remained.

2.5. Step 5: Vulnerability detection

We utilized a static analysis tool to determine the security vulnerabilities of each code snippet from ChatGPT and SO. These tools are one way to detect vulnerabilities and have been used in industry [27]. Specifically, we utilized CodeQL [19], a semantic code analysis engine developed by GitHub. CodeQL has been used to evaluate the code generated by LLMs in prior research [5], [8]. At the same time, CodeQL is one of the better-performing static analysis tools for Java [28]. CodeQL detects vulnerabilities by querying the code on a database with vulnerability variants. We utilized version `v2.16.1` and `java-security-and-quality.qls` test suite that contains queries for security and quality of Java code. We analyzed 234 queries for each snippet.

To determine the significance of the differences during our analysis, we utilized standard statistical analysis tests in R [29]. We mapped the CWE we found with MITRE's 2023 Top-25 Most Dangerous Software Weaknesses list [30].

3. Results

3.1. Vulnerabilities

To understand the differences between platforms, we start by analyzing the vulnerabilities detected by CodeQL. Table 1 shows a summary of the results. We found vulnerabilities in 77 questions for ChatGPT. On the other hand, the number of questions with vulnerabilities for SO was 75. Hence, SO had fewer questions with vulnerabilities, yet the differences were insignificant when we performed a Chi-squared test (p = 0.87). Additionally, the overlap in questions with vulnerabilities was 81%. Similarly, the number of answers with vulnerabilities was 79 for ChatGPT

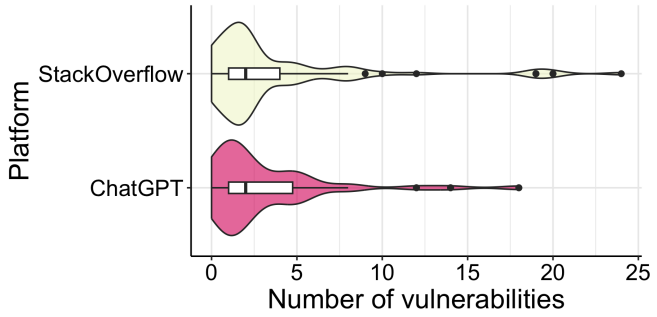

Figure 2. Number of vulnerabilities in the code snippets in each platform.

and 77 for SO, with an overlap of 79%. SO had fewer answers with vulnerabilities, yet the differences were not statistically significant (p = 0.87). Likewise, ChatGPT generated 87 snippets with vulnerabilities, higher than the 83 found in SO with an overlap of 79%. The differences were not statistically significant (p = 0.76).

We found 248 vulnerabilities across all ChatGPT code snippets. Meanwhile, we found 302 vulnerabilities in the SO code snippets. Hence, the number of vulnerabilities is 20% fewer in ChatGPT snippets than SO. We performed a Chi-squared test to determine if the differences in the number of vulnerabilities of the code snippets by platform were statistically significant. We found a statistically significant difference (p = 0.02).

Fig. 2 shows a violin chart with a box plot of the number of vulnerabilities of the snippets for each platform. The average and median number of vulnerabilities per snippet for ChatGPT were 3.04 and 2, respectively. The average number of vulnerabilities for SO code snippets was 3.62, while 2 was the median. The maximum vulnerabilities found in a snippet was 24, created in SO. To compare the number of vulnerabilities in each code snippet produced in SO versus ChatGPT, we utilized a paired t-test. We did not find a statistically significant difference in the number of vulnerabilities in each code snippet (p = 0.25). Therefore, the differences were not statistically significant despite ChatGPT producing on average code snippets with fewer vulnerabilities than SO.

We gathered the unique vulnerabilities in each snippet to compare the overlap of vulnerabilities, as vulnerabilities could be present in different lines. The number of unique vulnerabilities in the code snippets is 158 for ChatGPT and 183 for SO. We found no statistically significant difference with the unique snippet vulnerabilities for the platforms using a Chi-squared test (p = 0.18). The vulnerabilities generated by ChatGPT and present in SO code snippets are only the same in 25% of snippets. Noticeably, there is a difference of at least 54% between the overlap of unique vulnerabilities in snippets and the other overlaps.

3.2. Vulnerabilities CWEs

We gathered the vulnerabilities associated with CWEs to understand how the security issue types varied between platforms. The CWEs we found are shown in Table 2. We found 25 different types of CWEs in both platforms.

TABLE 2. THE TYPES OF CWE FROM THE INFORMATION SOURCES. THE PLATFORM WITH FEWER VULNERABILITIES IS HIGHLIGHTED. THE DELTA (Δ) REPRESENTS THE DIFFERENCE BETWEEN PLATFORMS.

CWE-ID	# GPT	# SO	Δ	Top 25
CWE-078	2	0	2	5
CWE-088	2	0	2	-
CWE-248	4	10	6	-
CWE-295	13	12	1	-
CWE-297	4	4	0	-
CWE-326	4	7	3	-
CWE-327	95	122	27	-
CWE-328	95	122	27	-
CWE-329	7	8	1	-
CWE-330	0	3	3	-
CWE-335***	0	16	16	-
CWE-338	0	3	3	-
CWE-391	14	18	4	-
CWE-404	37	29	8	-
CWE-476	1	6	5	12
CWE-477	19	15	4	-
CWE-561	13	13	0	-
CWE-570*	0	4	4	-
CWE-571*	0	4	4	-
CWE-581	0	2	2	-
CWE-772	37	29	8	-
CWE-780	3	6	3	-
CWE-798*	36	20	16	18
CWE-835**	8	0	8	-
CWE-1204	7	8	1	-

Chi-squared test significance: '***' $p < 0.001$, '**' $p < 0.01$, '*' $p < 0.05$

In ChatGPT we found 19 CWEs. The most frequent type of CWEs are **CWE-327: Use of a Broken or Risky Cryptographic Algorithm** and **CWE-328: Use of Weak Hash**, both tied in first place with 95 snippets. CWE-327 captures when cryptographic algorithms used are insecure. Hence, the desired security cannot be guaranteed. Meanwhile, CWE-328 is similar as it covers when the hash algorithm does not meet security expectations. The CWE is found by the CodeQL rules *"Use of a broken or risky cryptographic algorithm"* and *"Use of a potentially broken or risky cryptographic algorithm"*. Tied in second place is **CWE-404: Improper Resource Shutdown or Release** and **CWE-772: Missing Release of Resource after Effective Lifetime**, found in 37 snippets. Both CWEs cover when resources are incorrectly released. CWE-404 captures a release before it becomes available, whereas CWE-772 is after the resource is no longer needed. The CWE is found by the CodeQL rules *"Improper Resource Shutdown or Release"* and *"Missing Release of Resource after Effective Lifetime'*. Lastly, the third-most present is **CWE-798: Use of Hard-coded Credentials**. The vulnerability describes when credentials like passwords or cryptographic keys are hard-coded in the code and can be leveraged by attackers to bypass authentication. Additionally, CWE-798 is in 18th place in MITRE's Top 25. The rule *"Hard-coded credential in API call"* captures the CWE.

In SO we found 22 types of CWEs. Hence, ChatGPT generated fewer types of CWEs with a difference of 14.63%. Tied for first place, **CWE-327: Use of a Broken or Risky**

Cryptographic Algorithm and **CWE-328: Use of Weak Hash** were found in 122 snippets. In the second place, with 29 snippets, was **CWE-404: Improper Resource Shutdown or Release** and **CWE-772: Missing Release of Resource after Effective Lifetime**. In third place was **CWE-798: Use of Hard-coded Credentials** with 20 snippets. Interestingly, the order of the top most common CWEs is the same for both ChatGPT and SO. However, the number of vulnerabilities for the most frequent CWEs found in each platform differed with a delta ranging from 8 to 27. For the top CWEs, ChatGPT created fewer vulnerabilities for CWE-327 and CWE-328. Meanwhile, SO produced fewer vulnerabilities for CWE-404, CWE-772, and CWE-798.

ChatGPT snippets were better than SO for 15 types of CWEs. Meanwhile, SO was better for 8 types of CWEs. Finally, 2 types of CWEs had the same number of snippets. The CWEs with the largest differences were **CWE-327: Use of a Broken or Risky Cryptographic Algorithm**, **CWE-328: Use of Weak Hash**, **CWE-335: Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)**, and **CWE-798: Use of Hard-coded Credentials**. CWE-335 occurs when a pseudo-random number generator is used, but the seed is incorrectly managed. The vulnerability is captured by the rule *"Random used only once"*. The difference between CWE-327 and CWE-328 CWEs was 27, while for CWE-335 and CWE-798 it was 16. The remainder of CWEs had differences ranging from 0 to 8 occurrences in the code snippets.

We also compared the statistical significance of each type of CWE between the platforms using a Chi-squared test. We found five types of CWE with statistical significance. Ordered by the CWEs with the most statistically significant results, **CWE-335: Incorrect Usage of Seeds in Pseudo-Random Number Generator (PRNG)** had the highest significance ($p < 0.001$). ChatGPT generated the vulnerability 0 times compared to the 16 occurrences in snippets in SO. **CWE-835: Loop with Unreachable Exit Condition ('Infinite Loop')** had the second highest statistical significance ($p = 0.005$). The vulnerability occurs when there is an infinite loop in an iterator. The rule *"Constant loop condition"* can cover the vulnerability. The CWE occurred in ChatGPT code 8 times, while 0 times for SO. **CWE-798: Use of Hard-coded Credentials** was the third highest statically significant difference ($p = 0.03$). We found 36 occurrences in ChatGPT code, compared to the 20 occurrences in SO code. Lastly, **CWE-570: Expression is Always False** and **CWE-571: Expression is Always True** have statistically significant results with the same significance ($p = 0.046$). The CWEs capture when the evaluation of an expression always returns true or false, respectively. The rule *"Useless comparison test"* captures both CWEs. ChatGPT generated 0 CWE instances compared to SO that produced 4 for each.

We found three different CWEs within MITRE'S Top 25 CWEs for 2023. Ranked by position, **CWE-078: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')** is the highest ranked CWE we found in 5th place. The CWE covers when an input is used in operative systems commands that an attacker could influence externally. The vulnerability was captured by the rule *"Executing a command with a relative path"*. Only 2 snippets generated by ChatGPT contained the CWE. Following is **CWE-476: NULL Pointer Dereference** located in the 12th place. The vulnerability captures when a pointer deference occurs by expecting a valid pointer, yet the pointer is null. The CWE was present in 1 ChatGPT snippet and 6 SO snippets. Finally, **CWE-798: Use of Hard-coded Credentials** is the last CWE we found on the list, positioned in 18th place. We found the CWE in 36 times in ChatGPT snippets, while 20 times in SO snippets.

4. Discussion

4.1. Developer recommendations

As with any disruption to software development, there are potential benefits and risks with using LLMs. Still, developers are under-educated on insecure code propagation from both platforms, as both contain vulnerable code that can propagate to developers. We found 274 different vulnerabilities and 25 different types of CWE in the platforms. Adding another element to the software supply chain carries an inherent risk. Information gathered by developers from an outside source should be handled with caution as it can become an attack vector for malicious actors. Hence, our first recommendation is *do not blindly trust code, AI-generated or human-created, from outside sources*. LLMs have provided an excellent opportunity to become more conscientious of the complete software supply chain.

Consequently, developers may wonder if they should even use LLMs and other web-based information sources due to security risks. Our second recommendation is despite risks *to use the platforms, but apply good software security practices*. Static analysis tools and software testing can help detect copied and pasted code vulnerabilities. Practices can be adopted starting with those with the highest impact [31]. At the same time, developers need to pay more attention to CWE-335, CWE-570, and CWE-571 for LLM-generated code, while CWE-798 and CWE-835 for SO answers.

4.2. Future work

There are several avenues for future work. First, reducing insecure code can propagate fewer security risks to users. In line with prior work [32], [33], *approaches to stop insecure code propagation* need further research. Second, we found that ChatGPT generated less vulnerable code compared to SO using CodeQL. A question remains: *why are there even any differences* as LLMs are trained using the internet, including sites such as SO? We hypothesize that the compression and aggregation of information of LLMs may reduce vulnerabilities. Still, work must further study why such differences occur, validating or refuting our hypothesis. Companies can leverage this finding by incorporating LLMs within their development tools or information

sources, as there may be a possible added security benefit of the technology. Finally, in our work, we focused on code generation. Still, LLMs are incorporated in various software tasks, including code summarization and translation [34]. As such, future work should evaluate the *security implications of LLMs for other software tasks*.

5. Limitations

The main limitation of our work is the representativeness and generalizability of our analysis with the diversity of questions and answers in software development. Our findings are limited by our sampling strategy, the number of code snippets analyzed, and the Java programming language of the snippets. Future work can expand upon the generalizability of our findings by analyzing different development information sources, LLMs, and programming languages with more code snippets. Another limitation is based on how we leveraged ChatGPT as our prompt may have biased our results, despite our mitigation efforts to construct and refine the prompt iteratively. Prompt engineering approaches can be investigated to potentially reduce vulnerabilities in the generated code. At the same time, our findings are constrained by the time frame and version of ChatGPT analyzed, given the evolving nature of LLMs and their usage. Additionally, how we measured and detected vulnerabilities in the code snippets scope our findings. Software vulnerabilities are a commonly used measure [35], while static analysis tools to detect vulnerabilities are a common practice within software development [27]. CodeQL, the tool we use to detect vulnerabilities, has been used in prior research to evaluate code generated by LLMs [5], [8]. Static analyzers still generate false positives, though CodeQL is one of the less sensitive tools in Java [28]. Lastly, the non-deterministic nature of LLMs limits the reproducibility of our work. To combat this threat, we make our dataset public.

6. Related work

Software supply chain attacks leverage software components to compromise downstream users [36]. For example, relying on third-party packages from ecosystems like npm and PyPi could compromise consumer packages [37], [38]. As the software supply chain continues evolving, a recent concern for practitioners is the usage of LLMs in development [3], [4]. Research has revealed that code generated by GitHub Copilot can introduce vulnerabilities in public repositories [8]. LLMs have thus become an additional consideration in the security of the software supply chain.

Regarding software security, works have leveraged LLMs for vulnerability repair [39], [40] and detection [41]. Works have also shown uses for LLMs in software security testing [42], [43] and security code reviews [44]. Complementary to prior works, data poisoning [45] and prompt injection attacks [46] exploits have been found for LLMs in software development contexts. Research has evaluated several quality attributes of code generated by LLMs, including the correctness and usability [6], [10], [11], [47], [48], [49], [50]. At the same time, studies have evaluated the security of the generated code. Insecure code propagation has previously been studied within software security for platforms as SO [51]. Research has identified security risks such as vulnerabilities [5], [6] and code smells [7] in code generated by LLMs. Datasets [52] and frameworks [53] have also been created to evaluate the security of LLMs for code generation.

Prior work comparing the security of LLM-generated code is the most related to our study. Research has contrasted differences in the code when developers use LLMs [54], [55]. At the same, work has evaluated if LLMs have introduced the same vulnerabilities as humans [56]. In line with Asare et al. [56] and Sandoval et al. [55], the security impact of LLMs is low. At the same time, Asare et al. [56] also found that the vulnerabilities generated by LLMs were different than human-produced. Contrary to the findings of the other works and our study, Perry et al. [54] found that participants were more likely to generate less secure code. Still, Perry et al. [54] found participants who trusted AI less provided code with fewer security vulnerabilities. Hence, LLMs may be a positive addition to the security of software projects if leveraged intentionally. The studies analyzed C, C++, Python, and JavaScript code, while we studied Java code. Hence, we hypothesize that the findings are applicable across different programming languages.

Our study complements prior work by contrasting code vulnerabilities of LLMs with a web-based information source, SO. By comparing LLMs with an established platform like SO, we enhance our understanding of the security risks associated with using generative AI. Hence, we help developers by increasing the awareness of the security risks when selecting the information source for code snippets.

7. Conclusions

With the widespread adoption of LLMs in software engineering, developers have raised concerns about the security risk implications and the potential impact on the software supply chain. Developers are weighing the benefits and risks of using LLMs compared to other web-based information sources, such as online forums like SO. Notably, SO also contains code with security issues. Hence, developers require empirical data comparing the security of both platforms to inform their choices.

We compared ChatGPT and SO vulnerabilities for 108 Java code snippets using CodeQL. Based on our findings, software developers are under-educated on insecure code propagation from any information online source, be it AI-generated or human-created code. ChatGPT as a platform generated more vulnerabilities and types of CWE than SO. Still, the code in ChatGPT and SO had 274 different vulnerabilities and overlapped only in 25% of snippets. *Any code that can be copied and pasted from an outside source, AI-generated or human-created, cannot be blindly trusted, requiring applying good software security practices.* The security concerns surrounding generative AI are an opportunity to increase conscientiousness about software security.

Acknowledgments

This work was supported and funded by the National Science Foundation Grant No. 2207008, CNS-2026928, North Carolina State University Provost Doctoral Fellowship, and Goodnight Doctoral Fellowship. Any opinions expressed in this material are those of the authors and do not necessarily reflect the views of any of the funding organizations. We thank the Realsearch and WSPR research groups from North Carolina State University for their support and feedback. Additionally, we are grateful for the reviewer's time and feedback.

References

[1] Sonatype, "Annual State of the Software Supply Chain," 2023.

[2] GitHub, "The state of open source software." https://octoverse.github.com/, 2023.

[3] W. Enck, Y. Acar, M. Cukier, A. Kapravelos, C. Kästner, and L. Williams, "S3C2 Summit 2023-06: Government Secure Supply Chain Summit," *arXiv preprint arXiv:2308.06850*, 2023.

[4] T. Dunlap, Y. Acar, M. Cucker, W. Enck, A. Kapravelos, C. Kastner, and L. Williams, "S3C2 Summit 2023-02: Industry Secure Supply Chain Summit," *arXiv preprint arXiv:2307.16557*, 2023.

[5] 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," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768, IEEE, 2022.

[6] Z. Liu, Y. Tang, X. Luo, Y. Zhou, and L. F. Zhang, "No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT," *arXiv preprint arXiv:2308.04838*, 2023.

[7] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An Empirical Study of Code Smells in Transformer-based Code Generation Techniques," in *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 71–82, IEEE, 2022.

[8] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, and J. Yu, "Security Weaknesses of Copilot Generated Code in GitHub," *arXiv preprint arXiv:2310.02059*, 2023.

[9] Sonatype, "The Risks and Rewards of Generative AI in Software Development," 2023.

[10] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Who answers it better? an in-depth analysis of ChatGPT and Stack Overflow answers to software engineering questions," *arXiv preprint arXiv:2308.02312*, 2023.

[11] B. Xu, T.-D. Nguyen, T. Le-Cong, T. Hoang, J. Liu, K. Kim, C. Gong, C. Niu, C. Wang, B. Le, *et al.*, "Are we ready to embrace generative ai for software q&a?," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1713–1717, IEEE, 2023.

[12] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You Get Where You're Looking for: The Impact of Information Sources on Code Security," in *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 289–305, IEEE, 2016.

[13] F. Fischer, K. Bottinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 121–136, IEEE, 2017.

[14] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How Reliable is the Crowdsourced Knowledge of Security Implementation?," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 536–547, IEEE, 2019.

[15] A. Rahman, E. Farhana, and N. Imtiaz, "Snakes in paradise?: Insecure python-related coding practices in stack overflow," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pp. 200–204, IEEE, 2019.

[16] H. Zhang, S. Wang, H. Li, T.-H. Chen, and A. E. Hassan, "A Study of C/C++ Code Weaknesses on Stack Overflow," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2359–2375, 2022.

[17] OpenAI, "ChatGPT." https://chat.openai.com/, 2024.

[18] StackOverflow, "StackOverflow." https://stackoverflow.com/, 2024.

[19] GitHub, "CodeQL." https://codeql.github.com/, 2024.

[20] "Stack Exchange." https://stackexchange.com/sites, 2023.

[21] StackOverflow, "2023 Developer Survey." https://survey.stackoverflow.co/2023, 2023.

[22] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, p. 94, 2022.

[23] StackExchange, "StackExchange API." https://api.stackexchange.com/, 2024.

[24] B. Soup, "Beautiful Soup Documentation." https://beautiful-soup-4.readthedocs.io/, 2024.

[25] javalang, "javalang." https://github.com/c2nes/javalang, 2024.

[26] OpenAI, "OpenAI Python API library." https://github.com/openai/openai-python, 2024.

[27] S. Elder, N. Zahan, R. Shu, M. Metro, V. Kozarev, T. Menzies, and L. Williams, "Do I really need all this work to find vulnerabilities?: An empirical case study comparing vulnerability detection techniques on a Java application," *Empirical Software Engineering*, vol. 27, no. 6, p. 154, 2022.

[28] K. Li, S. Chen, L. Fan, R. Feng, H. Liu, C. Liu, Y. Liu, and Y. Chen, "Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 921–933, 2023.

[29] J. Kloke and J. W. McKean, *Nonparametric statistical methods using R*. CRC Press, 2014.

[30] MITRE, "2023 CWE Top 25 Most Dangerous Software Weaknesses." https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.

[31] N. Zahan, S. Shohan, D. Harris, and L. Williams, "Do software security practices yield fewer vulnerabilities?," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pp. 292–303, IEEE, 2023.

[32] H. Hong, S. Woo, and H. Lee, "Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions," in *Annual Computer Security Applications Conference*, pp. 194–206, ACM, 2021.

[33] F. Fischer and J. Grossklags, "Nudging software developers toward secure code," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 76–79, 2022.

[34] Z. Zheng, K. Ning, J. Chen, Y. Wang, W. Chen, L. Guo, and W. Wang, "Towards an understanding of large language models in software engineering tasks," *arXiv preprint arXiv:2308.11396*, 2023.

[35] P. Morrison, D. Moye, R. Pandita, and L. Williams, "Mapping the field of software life cycle security metrics," *Information and Software Technology*, vol. 102, pp. 146–159, 2018.

[36] P. Ladisa, H. Plate, M. Martinez, and O. Barais, "SoK: Taxonomy of Attacks on Open-Source Software Supply Chains," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 1509–1526, IEEE, 2023.

[37] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's knife collection: A review of open source software supply chain attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pp. 23–43, Springer, 2020.

[38] 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*, pp. 331–340, 2022.

[39] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2339–2356, IEEE, 2023.

[40] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1162–1174, IEEE, 2023.

[41] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 112–119, IEEE, 2023.

[42] A. Happe and J. Cito, "Getting pwn'd by ai: Penetration testing with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 2082–2086, 2023.

[43] Y. Zhang, W. Song, Z. Ji, Danfeng, Yao, and N. Meng, "How well does LLM generate security tests?," *arXiv preprint arXiv:2310.00710*, Oct. 2023.

[44] J. Yu, P. Liang, Y. Fu, A. Tahir, M. Shahin, C. Wang, and Y. Cai, "Security Code Review by LLMs: A Deep Dive into Responses," *arXiv preprint arXiv:2401.16310*, 2024.

[45] D. Cotroneo, C. Improta, P. Liguori, and R. Natella, "Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks," *arXiv preprint arXiv:2308.04451*, 2023.

[46] F. Wu, X. Liu, and C. Xiao, "DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions," *arXiv preprint arXiv:2312.04730*, 2023.

[47] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, pp. 1–5, ACM, 2022.

[48] N. Al Madi, "How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5, ACM, 2022.

[49] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[50] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, "GitHub Copilot AI pair programmer: Asset or Liability?," *Journal of Systems and Software*, vol. 203, p. 111734, 2023.

[51] W. Bai, O. Akgul, and M. L. Mazurek, "A Qualitative Investigation of Insecure Code Propagation from Online Forums," in *2019 IEEE Cybersecurity Development (SecDev)*, pp. 34–48, IEEE, 2019.

[52] M. L. Siddiq and J. C. S. Santos, "SecurityEval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques," in *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*, pp. 29–33, ACM, 2022.

[53] M. L. Siddiq and J. C. S. Santos, "Generate and pray: Using sallms to evaluate the security of llm generated code," *arXiv preprint arXiv:2311.00889*, 2023.

[54] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2785–2799, 2023.

[55] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, (Anaheim, CA), pp. 2205–2222, USENIX Association, Aug. 2023.

[56] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?," *Empirical Software Engineering*, vol. 28, no. 6, p. 129, 2023.