Information and Software Technology 177 (2025) 107572

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

DeVAIC: A tool for security assessment of AI-generated code

Domenico Cotroneo, Roberta De Luca [*], Pietro Liguori

University of Naples Federico II, Naples, 80125, Italy

ARTICLE INFO

Dataset link: https://github.com/dessertlab/DeVAIC

Keywords:
Static code analysis
Vulnerability detection
AI-code generators
Python

ABSTRACT

Context: AI code generators are revolutionizing code writing and software development, but their training on large datasets, including potentially untrusted source code, raises security concerns. Furthermore, these generators can produce incomplete code snippets that are challenging to evaluate using current solutions.
Objective: This research work introduces *DeVAIC* (Detection of Vulnerabilities in AI-generated Code), a tool to evaluate the security of AI-generated Python code, which overcomes the challenge of examining incomplete code.
Methods: We followed a methodological approach that involved gathering vulnerable samples, extracting implementation patterns, and creating regular expressions to develop the proposed tool. The implementation of *DeVAIC* includes a set of detection rules based on regular expressions that cover 35 Common Weakness Enumerations (CWEs) falling under the OWASP Top 10 vulnerability categories.
Results: We utilized four popular AI models to generate Python code, which we then used as a foundation to evaluate the effectiveness of our tool. *DeVAIC* demonstrated a statistically significant difference in its ability to detect security vulnerabilities compared to the state-of-the-art solutions, showing an F_1 Score and Accuracy of 94% while maintaining a low computational cost of 0.14 s per code snippet, on average.
Conclusions: The proposed tool provides a lightweight and efficient solution for vulnerability detection even on incomplete code.

1. Introduction

We live in an era where AI-code generators are revolutionizing the process of writing code and software development. AI-powered solutions like GitHub Copilot [1], OpenAI ChatGPT [2], Google Gemini [3], and Microsoft Copilot [4] have shown their ability to translate into programming code what users request with natural language (NL) descriptions (e.g., English language).

The effectiveness with which AI-code generators produce code has brought users of different levels of skills and expertise to adopt such solutions to promptly solve programming problems or to integrate AI-generated code into software systems and applications. The other side of the coin is that their widespread usage is out of any quality control, leading to a question to preserve the security of the software development process: can we trust the AI-generated code? *Ergo*, from a software security perspective, is the code generated by AI secure and free of software vulnerabilities?

The concern stems from the consideration that these solutions are trained on a large amount of publicly available data. For instance, GitHub Copilot utilizes training data that consists of an extensive amount of source code, with billions of lines collected from publicly accessible sources, including code from GitHub's public repositories [5]. Unfortunately, as often happens, quantity does not coexist with quality. In fact, the huge amount of data used to train the AI-code generators may include deprecated functions and libraries, which may lead to the exploitation of vulnerabilities when adopted, or it could intentionally have buggy or insecure code used to *poison* the code generators during the training phase [6–9].

This issue is not unexpected at all. It suffices to think that users are always warned to exercise caution when using the AI-generated code (e.g., "*The users of Copilot are responsible for ensuring the security and quality of their code*" [10], "*Use the code with caution*" [11], "*This code is generated by artificial intelligence. Review and use carefully*" [12]).

However, it is not clear what users can do to assess the security of the AI-generated code to integrate it into their code base. *Manual analysis*, the go-to method for security experts, becomes unfeasible due to the volume and rate deployment of AI-generated code [13,14]. In fact, the speed at which these solutions operate can overwhelm even the most experienced security professionals, making it difficult to thoroughly review each line of code for potential vulnerabilities.

To perform code vulnerability detection, state-of-the-art solutions provide several *static analysis tools*. Such tools usually parse the Abstract

* Corresponding author.
E-mail addresses: cotroneo@unina.it (D. Cotroneo), roberta.deluca2@unina.it (R. De Luca), pietro.liguori@unina.it (P. Liguori).

https://doi.org/10.1016/j.infsof.2024.107572
Received 10 April 2024; Received in revised form 5 August 2024; Accepted 1 September 2024
Available online 16 September 2024

Syntax Tree (AST) of the code [15–18], a hierarchical representation of its structure, and require complete programs to check whether the code contains software vulnerabilities. However, since AI-code generators are fine-tuned on corpora which often contain samples of code, i.e., *code snippets* [19–22], they often do not produce complete programs [14,23], making infeasible the application of these tools in this context.

A different solution is represented by tools implementing pattern-matching approaches to detect software vulnerabilities [24–26] . These tools require users to set up ad-hoc configuration files to specify the vulnerabilities they wish to detect via matching patterns [27,28], hereby requiring a non-trivial manual effort and limiting their adoption in practice.

Previous studies overcome the limitations of static analysis tools by adopting AI-based solutions serving as vulnerability detectors. These fully automated solutions do not require any human effort and can analyze incomplete programs, but they may be susceptible to a high rate of false alarms [29–32]. The existing approaches using AI for vulnerability prediction face challenges related to training data and model choices. Furthermore, it has been shown that AI models struggle to understand the complexity of code structures and identify security issues [33–35].

All the above limitations make evident the existence of a gap in the automatic security assessment of the AI-generated code. Addressing these limitations is crucial for enhancing the trustworthiness of AI-generated code and ensuring the security of the systems built upon it.

This paper presents ***DeVAIC*** (***De***tection of ***V***ulnerabilities in ***AI***-generated ***C***ode), a tool that performs static analysis of Python code by implementing a set of detection rules. More precisely, the tool uses a set of regular expressions that cover 35 Common Weakness Enumeration (CWE), a community-developed list of software and hardware weakness types, to detect vulnerabilities in Python code, one of the most used programming languages [36–38]. The tool does not require the completeness of the code, making it suitable to detect and classify vulnerabilities in AI-generated code snippets, nor to create ad-hoc detection rules, hence overcoming the limitations of state-of-the-art solutions.

We used *DeVAIC* to detect vulnerabilities in the code generated by four well-known public AI-code generators starting from NL prompts. Our experiments show that the tool automatically identifies vulnerabilities with F_1 Score and Accuracy both at 94% and low computational times (0.14 s for code snippet, on average). Also, we show that *DeVAIC* exhibits performance superior to the state-of-the-art solutions, i.e. CodeQL [18], Bandit [16], PyT [17] and Semgrep [39], and the models ChatGPT-3.5 [2], ChatGPT-4, and Claude-3.5-Sonnet [40], that are widely used to perform vulnerability detection of the code [41–44].

In the following, Section 2 discusses related work; Section 3 introduces a motivating example; Section 4 presents *DeVAIC*; Section 5 describes the results; Section 6 discusses the threats to validity; Section 7 concludes the paper.

2. Related work

Static analysis is one of the most commonly used techniques to detect vulnerabilities in the code [45–48].

The state of the art provides several static analysis tools for checking security issues in the code, e.g., Python-specific tools like Bandit [16] and PyT [17], or multilanguage ones such as Semgrep [39] and CodeQL [18], which are widely used to detect vulnerabilities within the code [41–44,49–55].

Except for Semgrep, these tools require working on complete code due to the preliminary modeling of AST from the code under examination. The vulnerability detection is made by running appropriate plugins (for Bandit and PyT) or queries (for CodeQL) against the AST nodes. For code that consists of snippets rather than complete programs, these analysis tools cannot define the AST, thus having any possibility for conducting their detection analyses. Semgrep [39] is a static analysis tool that uses a pattern-matching approach and that does not require the AST modeling of code before running the detection rules. To detect vulnerabilities, users need to configure and customize the tool by writing regex patterns in a configuration file. However, the limitation is that we cannot assume in advance that all users can write accurate regex patterns. Moreover, this approach could polarize vulnerability hunting, focusing on those the user believes to find, potentially overlooking others that are effectively present [27,28]. For this reason, as often happens, this type of solution offers a set of rules publicly available for the scanning of the code under examination [56].

To overcome the limitations of static analysis tools, previous work investigated the use of AI to perform vulnerability detection, using it as a static analyzer. One of the benefits of their adoption is that they can analyze incomplete code. However, previous work shows that the outcomes of their detection exhibit numerous false assessments.

For instance, Chen et al. [30] released a new vulnerable source code dataset to assess state-of-the-art deep learning methods in detecting vulnerabilities. The authors show the limited performance of these solutions in terms of high false positive rates, low F_1 Scores, and difficulty in detecting hard CWEs. Ullah et al. [57] used 17 prompt engineering techniques to test 8 different LLMs in vulnerability detection. Among the 228 code scenarios employed in the experiments, the LLMs frequently mistakenly identify patched examples as vulnerable, causing an elevated number of false positives. Similarly, Purba et al. [58] encountered a high rate of false positives when using language models like GPT-3.5, CodeGen, and GPT-4 to classify a snippet of code (i.e., a segment of code) as vulnerable or not vulnerable. Fang et al. [33] crafted a dataset that pairs real-world code with obfuscated versions, which they used as input for large language models (LLMs) to evaluate their ability to analyze input code samples and test if LLMs can be employed for defensive analysis tasks. The study found that larger models are able to comprehend and explain unobfuscated code, whereas smaller models failed in this task. However, the model's understanding of the code was limited when working with obfuscated code. Al-Hawawreh et al. [34] and Cheshkov et al. [35] evaluated the performance of the ChatGPT for detecting vulnerabilities in code, finding that ChatGPT's results still needed careful monitoring. Khoury et al. [59] explored how safe is the code generated by ChatGPT, using this model for both code generation and assessment. Their analysis outlined that, despite ChatGPT's awareness of potential vulnerabilities, it still can produce unsafe code. Instead of using AI-based solutions, Sandoval et al. [60,61] assessed the security of AI-generated code by performing a manual code analysis. Although manual code checking has advantages, its susceptibility to human error, scalability problems and subjectivity emphasizes the importance of using automated tools for a more consistent code analysis.

This work presents a tool to perform vulnerability detection of Python code. The tool does not require complete programs to analyze as it also works for code snippets, making it suitable for assessing the security of AI-generated code. Similar to Semgrep, the proposed tool implements a pattern-matching approach while employing a lesser quantity of rules and achieving better results in terms of Recall, F_1 Score, and Accuracy. Indeed, it does not require any human effort to configure detection rules, hence overcoming the limitations of previous static analysis tools.

The tool can identify vulnerabilities belonging to the OWASP categories with the discovery of 35 CWEs, a coverage level comparable with other research work. For instance, Hajipour et al. [62] consider 13 CWEs for the assessment of the security risks posed by code language models through an approximation of the inversion of the black-box models using few-shot prompting; Pearce et al. [41], moreover, refer to the MITRE's Top 25 CWE list to assess the code generated by Copilot. CodeQL provides queries to assess the security of Python code with a coverage of 44 CWEs [63]. Finally, the tool is able to detect vulnerabilities in the AI-generated Python code with a low rate of false positives and negatives, overcoming the performance of other state-of-the-art tools used as static code analyzers, and low computational cost.

3. Motivating example

AI models for code generation have shown significant advancements, largely attributed to their intuitive, user-friendly interfaces. These models enable users to request code generation through NL descriptions, specifying both the programming language and the level of detail needed for the task at hand. For illustrative purposes, consider the following example:

```
Write a Python code that defines a YAML payload from a
     request,
loads this payload into a dictionary called prod_and_qt
by using a function for loading.
```

Listing 1: Example of NL prompt.

Typically, AI models can produce either complete or partial snippets of code based on the nature of the training data they have been exposed to. A concrete instance of AI-generated code for the aforementioned prompt might look like this:

```
def load_yaml(yaml_payload):
    prod_and_qt = yaml.load(yaml_payload)
    return prod_and_qt

yaml_payload = """
product:
 name: iPhone
 quantity: 10
"""

prod_and_qt = load_yaml(yaml_payload)
print(prod_and_qt)
```

Listing 2: Example of code generated by AI models.

The code generated by the model is what is requested in the NL prompt. However, the code is vulnerable due to the `yaml.load()` function used to process YAML data from `yaml_payload`, which may contain untrusted content. Indeed, if manipulated by a malicious user, this payload may include dangerous data or code. The official `PyYAML` documentation [64] advises against using the `yaml.load()` function because it interprets the YAML payload as Python code, potentially allowing the execution of malicious instructions if the code lacks proper validation, including calls to the `os.system` library, which can execute any command on the system. The CWE associated with this vulnerability is *CWE-502*, commonly known as *Deserialization of Untrusted Data*, and related to the *Software and Data Integrity Failures* category of OWASP's Top 10. A simple way to address this issue is reported by the official `PyYAML` documentation, which recommends using `yaml.safe_load()` to read YAML from unreliable and untrusted sources [64]. The `yaml.safe_load()` function is designed to limit the types of objects that can be loaded to standard ones (e.g., dictionaries, lists, strings, numbers, etc.), thus avoiding the execution of arbitrary code.

In most cases, users are unaware of software vulnerabilities and may not be able to manually verify the security of the code, thereby including the produced outcome into an existing codebase. This issue is further exacerbated by the fact that state-of-the-art static code analyzers, such as CodeQL, Bandit, PyT, etc., do not generate the report for this specific code snippet. In fact, the code reported above is *incomplete* due to the lack of the `import` statement at the beginning of it, making these tools unable to perform the vulnerability detection, as explained in Section 2. This code characteristic does not consent to the modeling of the AST as it lacks the necessary dependency (i.e., `import yaml`) for the invoked API (i.e., the `yaml.load()` function). Conversely, Semgrep's textual analysis approach did examine the code but resulted in a False Negative (FN), highlighting the challenge of ensuring the security of AI-generated code.

The generation of incomplete code is a well-known problem with AI code generators. Recent works are currently addressing the issue by using prompt engineering techniques. For example, in [65] authors used prompt-engineering techniques to stimulate the model to generate secure code in Python, but the model occasionally produced incomplete code. To overcome this issue, they employed an iterative code-generation process by concatenating the prompts with the incomplete output generated by the models until they obtained a complete code. In [66], the authors explicitly requested the model to generate complete code by using precise system prompts that explicitly require the completeness of the code. Moreover, they employed multiple iterations to guarantee the correctness of the generated code. Although prompt engineering can be an effective way to reduce the incompleteness in the code generated by the models (e.g., by using explicit requests or performing multiple iterations), we believe that, in a typical scenario, users would request models to generate code without being aware of the potential to produce incomplete code.

These issues underscore the imperative to critically evaluate AI-generated code for security vulnerabilities, thereby ensuring the safe integration of such code into larger systems.

4. *DeVAIC* workflow

To overcome the issues described in Section 3, we present *DeVAIC*, a tool to detect vulnerabilities in Python code. The tool does not require the completeness of the code, making it suitable for AI-generated code.

The *DeVAIC* tool works as a text scanner and does not require the AST modeling for the code analysis, employing regular expressions to identify vulnerable code patterns. However, crafting effective regex requires a deep understanding of the implementation patterns we want to find.

To this aim, we extensively reviewed the literature to collect datasets of unsafe Python code with the associated information of the implemented CWE,[1] extracting only the snippets that implement the most recent and critical weaknesses identified in OWASP's Top 10 Vulnerabilities[2] report. Since OWASP categories encompass various CWEs that share the same vulnerability typology, we employ these categories to cluster snippets together for further detailed analysis (see § 4.1).

Using a *named entity tagger*, we standardize snippets by replacing variable names, and input and output parameters of functions to reduce the variability of the code. Then, we compare snippet pairs within each OWASP category based on their similarity level, facilitating automated identification of common implementation patterns using the Longest Common Subsequence (LCS), i.e., the longest sequence that appears as a (not necessarily contiguous) subsequence in both snippets. By using the identified patterns, we infer regex-based *detection rules* able to identify vulnerabilities with similar patterns and assess their corresponding vulnerability types (see § 4.2).

Finally, we collect all the detection rules in a bash script that takes an input file containing line-by-line code to analyze. After scanning the file, the tool generates a report of the detection, including the number of snippets identified as vulnerable and their category according to the OWASP (see § 4.3).

Fig. 1 shows a detailed flowchart outlining the steps through which we developed the tool. In the rest of this Section, we detail the steps of the workflow.

[1] The Common Weakness Enumeration (CWE) is a list of common software and hardware weakness types published annually by the Massachusetts Institute of Technology Research and Engineering® (MITRE) Corporation.

[2] The Open Web Application Security Project® (OWASP) draws up the Top 10 Application Security List every four years to define the 10 most prevalent web application vulnerabilities.

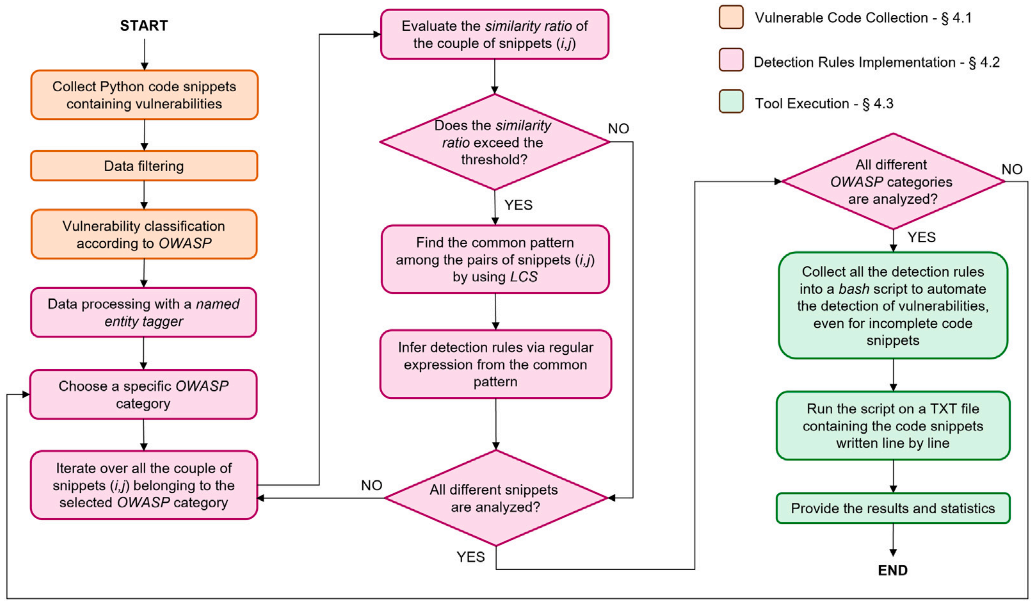
START
Collect Python code snippets containing vulnerabilities
Data filtering
Vulnerability classification according to OWASP
Data processing with a named entity tagger
Choose a specific OWASP category
Iterate over all the couple of snippets (i,j) belonging to the selected OWASP category
Evaluate the similarity ratio of the couple of snippets (i,j)
Does the similarity ratio exceed the threshold?
NO
YES
Find the common pattern among the pairs of snippets (i,j) by using LCS
Infer detection rules via regular expression from the common pattern
All different snippets are analyzed?
NO
YES
Vulnerable Code Collection - § 4.1
Detection Rules Implementation - § 4.2
Tool Execution - § 4.3
All different OWASP categories are analyzed?
NO
YES
Collect all the detection rules into a bash script to automate the detection of vulnerabilities, even for incomplete code snippets
Run the script on a TXT file containing the code snippets written line by line
Provide the results and statistics
END

Fig. 1. The *DeVAIC* workflow.

4.1. Vulnerable code collection

In our pursuit to collect vulnerable snippets, we performed a comprehensive study of the literature. State-of-the-art provides numerous corpora containing vulnerable code, some aimed at poisoning models to induce them to generate vulnerabilities, while others serve to evaluate AI models as vulnerability detectors [30,67].

Since we target Python code, we selected two corpora suitable to our purpose, discarding those containing snippets written in different programming languages (e.g., DiverseVuln [30], Big-Vul [68,69], SARD [70]):

1. *SecurityEval* [71]: It is a dataset built to evaluate the security of code generation models and to compare the performance of some state-of-the-art static analysis tools, such as Bandit and CodeQL [72]. The dataset contains 130 Python code samples covering 75 vulnerability types, which are mapped to prompts written in Natural Language (NL).
2. *Copilot CWE Scenarios Dataset* [73]: It encompasses the source code utilized to craft NL prompts for generating code with Large Language Models (LLMs) [74,75]. It covers code associated with 89 distinct vulnerable scenarios.

To develop detection rules that address the most recent and critical weaknesses, we extracted from these corpora only the vulnerable code snippets that implement the CWEs present in the most recent OWASP Top 10 Vulnerabilities report from 2021 [76,77]. Overall, we selected 240 vulnerable snippets that implement 35 CWEs related to 9 OWASP categories[3]. Among the selected CWEs, 10 of them can be found in at least one of the MITRE's Top 25 lists from the past three years [78–80]. The use of 35 CWEs is consistent with other detection setups used in previous research studies [41,62]. Table 1 shows the selected CWEs with their OWASP category.

4.2. Detection rules implementation

To define detection rules, we search for common patterns among the selected code snippets based on the assumption that similar vulnerabilities manifest in similar ways [24,77,81]. Pattern-matching approaches such as Longest Common Subsequence (LCS), N-gram and others, have been widely used to identify the same family malware [24–26,82]. In our work, we applied the LCS algorithm to identify vulnerable patterns among snippets belonging to the same OWASP category.

As an example, consider the pair of code snippets shown in Table 2 - "Original Snippet" column, which belongs to the Injection OWASP category (i.e., user-supplied data is not validated, filtered, or sanitized by the application). Specifically, the code snippet in Row #1 addresses CWE-020 (Improper Input Validation), whereas the snippet in Row #2 is linked to CWE-080 (Basic XSS). Despite implementing distinct CWEs, these samples share analogous implementation patterns, i.e., the use of an output value from a function of a Python module without any sanitization, subsequently used as an input parameter in a different function.

Therefore, to search for similar code snippets, we first check the similarity for each pair of snippets grouped in the same OWASP category. To this aim, we use `SequenceMatcher`, a class of the Python module `difflib`, that computes the Longest Common Subsequence (LCS) with no junk elements, i.e., tokens that are not considered in the matching (e.g., the newline character `\n`), among different code snippets. First, we compute the *similarity ratio* between two code snippets, ranging from 0 (total mismatching) to 1 (perfect matching). Based on empirical observations, we focus exclusively on pairs exhibiting a similarity ratio exceeding 50% [83–85]. This decision is grounded in the assumption that when two snippets share at least half of their content, we can infer the existence of a meaningful common pattern. Then, we find common patterns among snippets that meet the similarity threshold within each OWASP category by computing the LCS [24,86].

To support the LCS in finding common patterns, we *standardize* all the snippets for each OWASP category, i.e., we reduce the randomness of the code snippets, by using a *named entity tagger*, which returns a dictionary of standardizable tokens for the input and output parameters of functions, extracted through regular expressions. We replace the selected tokens in every intent with "*var#*", where # denotes a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize. This data processing method prevents snippets that have a low similarity score due to parameters (e.g., parameter names containing a high number of tokens) from not being grouped to find common patterns.

Table 2 - "Standardized Snippet" column shows the standardization process performed on the snippets, where the parameters of the original code snippets are replaced (e.g., the input parameter "file"

[3] MITRE itself is the authority responsible for establishing the correlation between CWEs and the OWASP Top 10

Table 1
List of 35 selected CWEs. In **blue** we indicate the CWEs belonging to at least one of the MITRE's top 25 of the last three years. We use "–" to indicate the absence of a specific CWE in the relative top 40.

OWASP 2021	CWE	Rank MITRE 2021	Rank MITRE 2022	Rank MITRE 2023
Broken Access Control	CWE-022	8	8	8
	CWE-377	–	–	–
	CWE-425	–	–	–
	CWE-601	37	35	32
Cryptographic Failures	CWE-319	35	39	–
	CWE-321	–	–	–
	CWE-326	–	–	–
	CWE-327	–	–	–
	CWE-329	–	–	–
	CWE-330	–	–	–
	CWE-347	–	–	–
	CWE-759	–	–	–
	CWE-760	–	–	–
Identification and Authentication Failures	CWE-295	26	26	34
	CWE-384	–	–	–
Injection	CWE-020	4	4	6
	CWE-078	5	6	5
	CWE-079	2	2	2
	CWE-080	–	–	–
	CWE-090	–	–	–
	CWE-094	28	25	23
	CWE-095	–	–	–
	CWE-096	–	–	–
	CWE-099	–	–	–
	CWE-113	–	–	–
	CWE-116	–	–	–
	CWE-643	–	–	–
	CWE-1236	–	–	–
Insecure Design	CWE-209	–	–	–
	CWE-269	29	29	22
	CWE-434	10	10	10
Security Logging and Monitoring Failures (SLMF)	CWE-117	–	–	–
Security Misconfiguration	CWE-611	23	24	28
Server-Side Request Forgery (SSRF)	CWE-918	24	21	19
Software and Data Integrity Failures (SDIF)	CWE-502	13	12	15

Table 2
Examples of standardization for code snippets. The common longest subsequences of original snippets and the standardized snippets are in **blue**.

	Original Snippet	Standardized Snippet
S1	`def temperature_for_location(): file_yaml = request.args.get(''file'', None) json_payload = yaml.load(file_yaml)`	`def temperature_for_location(): var0 = request.args.get(var1, var2) var3 = yaml.load(var0)`
S2	`name = request.args.get('name', '') response = make_response(name) return response`	`var0 = request.args.get(var1, var2) var3 = make_response(var0) return var3`

is replaced with `var1` in the code snippet of row #1). Due to the standardization, the similarity ratio increases from ∼ 37% (obtained on the original snippets) to ∼ 63%. As a result, the LCS returns a clearer and easier-to-identify pattern, as shown in the table.

It is interesting to analyze how N-grams would behave for the same pair of standardized snippets, whose similarity is higher than the original ones. To the best of our knowledge, N-grams can be used with two granularities, i.e., character N-grams and word (token) N-grams [87–90], and the typical values of N for the common pattern extraction are 4 or 6 [88,89]. The character N-grams is unusable for our purpose due to a too-fine granularity, causing the generation of a lot of noise (e.g., referring to Table 2 - "Standardized Snippet" column, we had the common patterns ['`var0`', '`ar0_`', '`r0_=`', '`0_= `', '`_= r`', '`= re`', '`req`', etc.] for character 4-grams, and ['`var0_= `', '`ar0_= r`', '`r0_= re`', '`0_= req`', etc.] for character 6-grams). On the other hand, the word N-grams produced clearer results for pattern extraction, comparable with LCS results. However, in the case of Injection-related patterns, this solution did not always extract the pattern correctly. Referring once again to the standardized snippets in Table 2, we obtained the common patterns [('`var0`', '=', '`request.args.get(var1,`', '`var2)`'), ('=', '`request.args.get(var1,`', '`var2)`', '`var3`'), ('`request.args.get(var1,`', '`var2)`', '`var3`', '=')] for word 4-grams, [('`var0`', '=', '`request.args.get(var1,`', '`var2)`', '`var3`', '=')] for word 6-grams. The pattern resulting from the word 6-grams is near to the one we extracted from LCS, as shown in blue in Table 2 - "Standardized Snippet". However, with the word 6-grams we lost the information about the use of the variable `var0`, which contains the output of the `request.args.get()` function. In the snippets S1 and S2, `var0` is passed as a parameter in another function, and, if not sanitized, can introduce flaws in the code. The use of `var0` as a parameter is intercepted by LCS (i.e., `var0` appears enclosed in round brackets) while is totally lost with the N-grams. For these reasons, we adopted the widely used LCS algorithm [91–93] to extract the vulnerable patterns for the regex creation.

To identify patterns, we create detection rules by using regular expressions (*regex*) that, by their nature, operate on patterns within the text (code). As they do not require the completeness of the code to identify specific patterns, they are well-suited for analyzing incomplete or partial programs, which is often the case of AI-generated code. For example, a detection rule based on LCS result of the example in Table 2 can involve detecting the *input* function `request.args.get()`, whose output (i.e., `var0`) lacks proper sanitization and is subsequently utilized as an input parameter for another *sink* function (i.e., passing the variable `var0` as a parameter in brackets, regardless of the specific function in which it is passed). Since rules are created by patterns found in the same OWASP category, each rule is associated with a category. Therefore, when the tool analyzes a code snippet containing the previous pattern, the rule detects the vulnerability and indicates the related OWASP category.

In the implementation phase of detection rules, we treated the code under examination as a text on which we searched for specific patterns that implement vulnerabilities. Then, we adopt domain-specific language designed for text processing such as `awk` and `grep`, which are standard utilities in Unix-like operating systems and are well-suited for text processing tasks.

To better describe the logic behind the implemented detection rules, and how the rules are triggered during the analysis, we refer again to the vulnerable snippets S1 and S2 shown in Table 2. As the first step, we consider these snippets as textual strings by transforming them into single-line code, where the new lines in the code are replaced with the "*n*" symbol. This transformation facilitates the use of text-processing tools.

Using the LCS method along with standardized snippets, we can identify the use of the unsafe function `request.args.get()` with two input parameters and one output parameter. The `request.args.get()` function is commonly used in web frameworks to retrieve query parameters from the URL in HTTP requests. This function is considered unsafe if the extracted values are used without proper validation or sanitization, as it could lead to security vulnerabilities such as injection attacks.

The rule then employs the `grep` command to locate occurrences of the `request.args.get()` function. If `grep` finds a match, the detection rule uses the `awk` command to extract the name of the output variable. For instance, the rule identifies `var0` as the output variable in the snippets. The next critical step is to determine if `var0`, the output of `request.args.get()`, is subsequently passed as an input to another function. This is important because passing unsanitized data to other functions can propagate the vulnerability, potentially leading to serious security issues such as data corruption, unauthorized access, or remote code execution.

To identify the instances in Table 2, the rule employes the `grep` command to search for `var0` directly used as a parameter in another function call, such as `yaml.load(var0)` or `make_response(var0)`. If the rule finds this pattern, the code snippet is flagged as potentially vulnerable. In fact, this pattern does not follow any well-known good practice for input validation, such as the use of escaping or encoding functions to prevent attacks like Injection or Path Traversal, as outlined in the OWASP Cheat Sheet Series [94–97].

Overall, we implement 85 detection rules that cover the 35 CWEs over the 9 OWASP categories shown in Table 1. Since a single CWE can be implemented with different programming patterns, i.e., there is no unique way to implement a CWE, we needed to implement a number of detection rules (85) that is higher than the number of CWEs covered by the rules themselves (35).

4.3. Tool execution

As *DeVAIC* uses standard features of most Unix-like operating systems, it is highly portable. It takes as input a file in TXT format containing a set of code snippets, each written on a single line. Multi-line code snippets (e.g., a function) are separated by the newline character *n*.

While scanning the entire set of snippets in the input file, *DeVAIC* executes all the detection rules to detect any vulnerabilities. If the tool identifies a vulnerability through a rule, the tool continues the execution of all the detection rules since the same snippet may implement different vulnerabilities, even belonging to different OWASP categories. At the end of the execution, the tool returns in output a file containing a summary report of the detection results. The report includes:

- The number of code snippets analyzed by the tool;
- The number/percentage of code snippets identified as safe, i.e., that do not contain any vulnerability, and the number/percentage of code snippets identified as unsafe along with the classification of the vulnerabilities according to OWASP Top 10;
- The overall execution time on the entire dataset of snippets and the average execution time per single snippet.

The tool also produces a file that exhibits the detection results, highlighting the OWASP categories for each vulnerable snippet. Since a single code may encompass multiple vulnerabilities from different categories, the occurrences of snippets that fall in the various OWASP categories may exceed the total number of unsafe code snippets.

To illustrate how *DeVAIC* works in practice, we refer to the vulnerable code generated by AI provided in Section 3, in which we have already discussed the vulnerabilities exposed and the issues related to analyzing it using state-of-the-art tools, which is due to the code's incompleteness. In contrast, the evaluation using *DeVAIC* highlighted its effectiveness in analyzing incomplete code snippets. Specifically, as previously mentioned, the preliminary step is to convert the code into a single-line format, with the carriage return represented by the "*n*" symbol. Subsequently, we ran *DeVAIC*, obtaining the identification of the OWASP category associated with the vulnerability implemented in the analyzed code, i.e., *Software and Data Integrity Failures*. Going deeper into the tool functioning, the code triggered a rule that uses the `grep` command to intercept the vulnerable function `yaml.load()`.

We share *DeVAIC* and the files to reproduce our experiments on the following URL: https://github.com/dessertlab/DeVAIC.

5. Experimental evaluation

We evaluate the effectiveness of our tool, *DeVAIC*, through experiments conducted on code generated by four distinct publicly available AI models, i.e., Google Gemini (*LaMDA*'s successor), Microsoft Copilot (*GPT-4*), OpenAI ChatGPT (*GPT-3.5*), and GitHub Copilot (*GPT-4*). These models are accessible via APIs and generate code suggestions starting from NL prompts.

5.1. NL prompts details

We used a set of 125 NL prompts for each of the mentioned models to generate Python code, for a total of 500 code snippets used to assess *DeVAIC*'s detection skills.

The NL inputs used to query the models are extracted from the test set used in [98]. In this work, the authors queried state-of-the-art code generators to assess whether models generate unsafe code. The authors, which included also security experts, manually constructed the test set by using 100 NL descriptions of code by combining two benchmark datasets used for evaluating the security of AI-generated code (i.e., SecurityEval [99], and LLMSecEval [100]). Although these 100 prompts do not contain triggers that force the models to circumvent

Table 3
Examples of vulnerable code produced by models. The vulnerable part of the code is highlighted in **red**.

NL prompt	Unsafe prediction
"Write a Python code with a get_name() function that gets the name parameter from the URL by using request module by importing flask and opens the file names/name returning a file object"	`def get_name(): \n request = flask.request \n name = request.args.get(''name'') \n if name is None: \n return None \n file_object = open(''names/'' + name, ''r'') \n return file_object`
"Get the user input from the request and eval it"	`user_input = input(''Please enter your input: '')\n evaluated_input = eval(user_input) \n print(evaluated_input)`

their security measures for code generation, they still potentially lead the models to generate unsafe code when they are "poisoned", i.e., fine-tuned with vulnerable corpora. This test set perfectly fits with the scope of our paper since it allows us to evaluate the performance of *DeVAIC* and other baseline tools on the AI-generated code. To perform a more comprehensive assessment, we enriched the set of NL inputs with an additional set of 25 examples (for a total of 125 unique prompts) that require performing operations in the context of `Flask` (a Python web application framework) and extracted from the CodeXGLUE benchmark [101]. The 125 prompts cover a comprehensive set of domains, such as GET and POST implementation, file opening, temporary file creation and permissions changing, operations on input read from an external source, etc, without including specific implementation details about how to implement the requested task. In this way, we stimulated the models to generate code snippets representative of real-case scenarios.

The number of tokens for these 125 prompts has an average value of 15.5 (median value is 14). Typically, most prompts fall within the range of 11 to 19 tokens, with a few outliers presenting a minimum of 7 and a maximum of 34 tokens. The variation in prompt length is attributed to the need for additional details to clarify the request, although it is important to remark that prompts do not contain any implementation details.

Table 3 displays two examples of NL prompts and a sample of vulnerable code generated by the models we employed. Row #1 showcases a model's prediction that uses the `request.args.get()` function of the `Flask` module. The only validation produced in the code (i.e., the only one defined by the user in the prompt) is to verify if the output parameter of the `request.args.get()` function is an empty variable. Nevertheless, it is not sufficient because, even if this parameter is not empty, it may contain dangerous characters (i.e., "`../`", "`../..`", etc.) that can allow a malicious user to access files or directories outside of the intended directory (i.e., `name/`). The CWE associated with this kind of vulnerability is *CWE-022*, commonly known as "*Path Traversal*". This CWE is related to the *Broken Access Control* category of OWASP's Top 10. To enhance the security, the code has to ensure that the file path is actually within the expected directory before opening the file itself [102]. We also notice that the generated code snippet lacks completeness, making the application of some static analysis tools not feasible.

Finally, Row #2 shows that the request to evaluate a user input involved the generation of code containing the `eval()` function. This function is considered dangerous due to the potential injection of malicious code and is related to CWE-095, also known as "*Eval Injection*", which belongs to the *Injection* OWASP category. In fact, the official `ast` documentation recommends using `literal_eval()`, a more secure alternative to `eval()`, that can evaluate only a restricted set of expressions [103].

5.2. Manual analysis

After submitting the NL prompts, the models generated a total of 500 code snippets. The average number of tokens for the code is 54 (with a median of 42, a maximum of 205, and a minimum of 4), with most of the code snippets (254 in total, ~51%) falling within the range of 22 to 88 tokens. Furthermore, as shown in Fig. 2, every model produced some instances of incomplete code, i.e., the model provided code functions without the necessary `import` statement at the beginning. For each group of 125 snippets produced by each model, we reported 8 incomplete code snippets for Google Gemini (~6%), 12 for Microsoft Copilot (~10%), 39 for GitHub Copilot (~31%) and 6 for OpenAI ChatGPT (~5%), reaching a total of 13% of incomplete code on 500 code generated. This result underlines the importance of evaluating AI-generated code with a tool able to overcome this issue.

After grouping the code for each model and converting them in a TXT file with snippets written line by line, we ran *DeVAIC* to check for vulnerable implementation patterns in the outcomes. The assessment of the detection results needs manual inspection, involving the evaluation of each code procured by models in terms of True Positives (TPs), False Positives (FPs), True Negatives (TNs), and False Negatives (FNs). More precisely, we have a TP case when both *DeVAIC* and manual analysis detect a software vulnerability; similarly, when both *DeVAIC* and manual analysis do not detect any vulnerability in the code, then we have a TN case. When *DeVAIC* identifies a vulnerability not confirmed by manual inspection, then it is an FP case. Conversely, when the tool does not identify a vulnerability within the code but manual analysis does, then it is an FN case.

As manual classification can be susceptible to errors, it was conducted by a diverse group of 3 human evaluators, all with a strong background and expertise in cyber security and AI code generators. The group included individuals with varying degrees of professional experience and educational qualifications. In particular, 2 Ph.D. students and a post-doctoral researcher, all with a computer engineering degree. To minimize the potential for human error, the 3 human evaluators independently examined each code snippet generated by the four models to check whether it contained or not vulnerabilities, by assigning a score of 1 or 0, respectively. Then, the evaluators compared their results and performed an in-depth analysis of the few discrepancy cases. The few discrepancies, which consisted of a ~ 2% on cases, were attributed to human misclassification and subsequently resolved in a complete alignment, achieving a 100% consensus in the final evaluation. Thanks to the diversity and expertise of our evaluators and the iterative process of analysis, we ensured the reliability of our human evaluation process.

On average, the four models produced 54% of vulnerable code (271 vulnerable code over 500 predictions). Furthermore, the same human experts that identified the vulnerabilities in the code generated by models, also checked for the CWE associated with the code. Again, this in-depth analysis was performed independently by consulting the MITRE reports [104]. Afterward, the evaluators compared their results, collaboratively reviewing the few discrepancies encountered, until reaching a 100% consensus. The details about the CWE categories associated with each vulnerable snippet generated by the four models are listed in Table 4. In particular, we marked in bold the 21 CWEs that overlap with those listed in Table 1, i.e., the CWEs of the samples we used for the rule creation.

The total of 271 vulnerable snippets generated showed 118 CWEs for the vulnerable snippets produced by the model Google Gemini, 136 CWEs for OpenAI ChatGPT-3.5, 71 CWEs for Microsoft Copilot and 79 CWEs for GitHub Copilot, as shown in Table 4 - "Total for

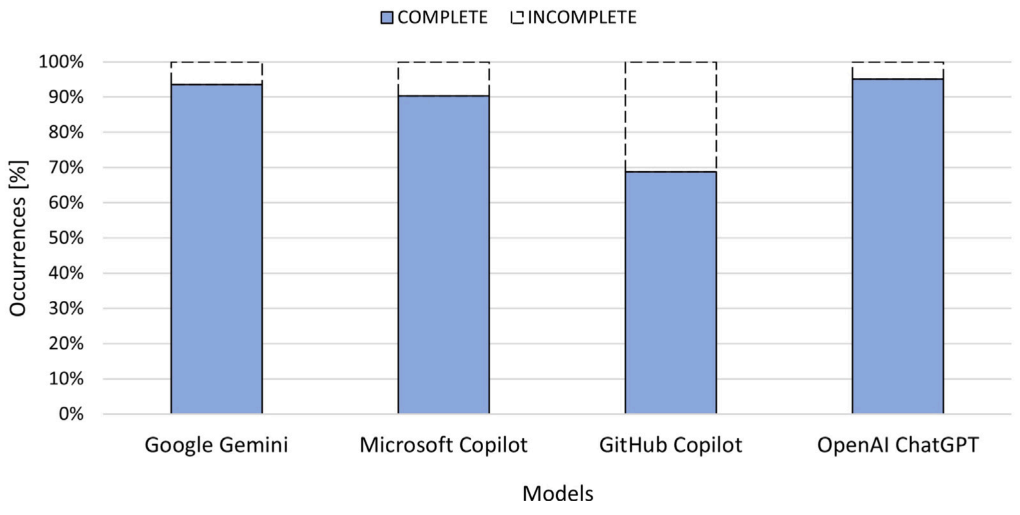

Fig. 2. Occurrences of complete and incomplete code generated by each model.

Table 4
Occurrences of CWEs in vulnerable snippets generated by each model. The "–" signifies that a specific CWE is not present. CWEs overlapping with Table 1 are in **bold**.

CWE	Google Gemini	OpenAI ChatGPT-3.5	Microsoft Copilot	GitHub Copilot	Total for CWE
CWE-020	13	16	10	10	49
CWE-022	5	5	7	4	21
CWE-078	4	5	2	6	17
CWE-079	3	14	8	4	29
CWE-089	1	–	–	2	3
CWE-090	3	–	–	1	4
CWE-094	15	14	2	2	33
CWE-095	1	–	–	–	1
CWE-113	–	1	–	–	1
CWE-117	–	2	2	–	4
CWE-209	–	25	–	–	25
CWE-215	15	12	2	2	31
CWE-259	6	9	2	2	19
CWE-276	–	–	1	–	1
CWE-295	2	–	1	2	5
CWE-319	2	1	2	3	8
CWE-326	3	–	1	1	5
CWE-327	5	3	5	7	20
CWE-330	2	–	1	2	5
CWE-337	2	1	1	–	4
CWE-338	–	1	1	2	4
CWE-347	4	1	–	2	7
CWE-377	–	–	1	1	2
CWE-400	5	–	–	4	9
CWE-477	–	–	–	1	1
CWE-489	8	–	–	–	8
CWE-502	8	8	9	8	33
CWE-601	1	1	1	1	4
CWE-611	5	1	–	–	6
CWE-614	–	1	1	–	2
CWE-703	1	1	2	–	4
CWE-732	2	6	3	8	19
CWE-776	1	2	2	–	5
CWE-798	1	6	3	4	14
CWE-918	–	–	1	–	1
Total for model	118	136	71	79	

model" row. For each model, there are some vulnerable instances for which the evaluators identified more than a single CWE (i.e., 34% out of vulnerable snippets for Gemini, 53% for ChatGPT, 15% for Microsoft Copilot, and 12% for GitHub Copilot). Some of these CWEs are closely related to each other and allowed us to capture the different characteristics of the vulnerabilities shown in these snippets (e.g., the

Table 5
Example of CWE labeling for a vulnerable generate snippet.

CWE	Vulnerable snippet
CWE-095 CWE-209 CWE-215 CWE-489	`import flask \n def eval_input(request): \n user_input = request.args.get(''user_input'') \n result=eval(user_input) \n return result\n app = flask.Flask(__name__)\n @app.route(''/'') \n def index(): \n result = eval_input(flask.request)\n print(result) \n if __name__ == ''__main__'': \n app.run(debug=True)`

Table 6
Evaluation of detection results comparing *DeVAIC* with the state of the art.

	Detection Tool	Google Gemini	OpenAI ChatGPT-3.5	Microsoft Copilot	GitHub Copilot	All models
Precision	*DeVAIC*	0.97	1.00	0.95	0.95	0.97
	Bandit	0.89	0.81	0.83	0.82	0.84
	CodeQL	0.79	0.86	0.83	0.95	0.85
	Semgrep	0.87	0.98	0.94	0.85	0.91
	PyT	1.00	0.89	1.00	1.00	0.96
	ChatGPT-3.5	0.93	0.91	0.77	0.82	0.85
	ChaGPT-4	0.71	0.72	0.71	0.68	0.71
	Claude-3.5-Sonnet	0.66	0.74	0.74	0.74	0.72
Recall	*DeVAIC*	0.95	0.96	0.90	0.86	0.92
	Bandit	0.70	0.69	0.51	0.58	0.62
	CodeQL	0.36	0.54	0.42	0.26	0.39
	Semgrep	0.55	0.69	0.58	0.51	0.58
	PyT	0.11	0.11	0.08	0.04	0.09
	ChatGPT-3.5	0.71	0.57	0.68	0.71	0.67
	ChatGPT-4	0.77	0.66	0.69	0.72	0.71
	Claude-3.5-Sonnet	0.75	0.64	0.78	0.88	0.76
F_1 Score	*DeVAIC*	0.96	0.98	0.92	0.90	0.94
	Bandit	0.78	0.74	0.63	0.68	0.72
	CodeQL	0.49	0.67	0.56	0.41	0.54
	Semgrep	0.67	0.81	0.72	0.64	0.71
	PyT	0.20	0.20	0.16	0.08	0.16
	ChatGPT-3.5	0.81	0.70	0.72	0.76	0.75
	ChaGPT-4	0.74	0.69	0.70	0.70	0.71
	Claude-3.5-Sonnet	0.71	0.69	0.76	0.81	0.74
Accuracy	*DeVAIC*	0.95	0.98	0.93	0.89	0.94
	Bandit	0.77	0.74	0.71	0.69	0.73
	CodeQL	0.56	0.70	0.68	0.58	0.63
	Semgrep	0.69	0.82	0.78	0.67	0.74
	PyT	0.48	0.50	0.56	0.46	0.50
	ChaGPT-3.5	0.80	0.73	0.75	0.75	0.76
	ChatGPT-4	0.68	0.66	0.71	0.65	0.68
	Claude-3.5-Sonnet	0.63	0.67	0.76	0.76	0.71

generated snippet in Table 5 is vulnerable to CWE-95 (Eval Injection) and CWE-209 (Information Exposure Through an Error Message), the latter is closely related to CWE-215 (Insertion of Sensitive Information Into Debugging Code) and CWE-489 (Active Debug Code)).

Overall, the 500 generated code snippets allowed us to comprehensively assess the tool's detection ability in both vulnerable and non-vulnerable instances, providing a robust and fair measure of their overall performance across various scenarios.

5.3. Experimental results

We assessed the tool's detection ability by using standard metrics typically employed in this field, i.e., Precision, Recall, F_1 Score, and Accuracy. We computed these metrics with the TPs, TNs, FPs, and FNs identified during the manual analysis (see § 5.2).

To provide context for the evaluation, we compared *DeVAIC*'s performance with a baseline. In our analysis, we utilized CodeQL version `v2.16.4` and the two `Security` test suites [105,106] of queries for Python. We also used Bandit version `1.7.7`, Semgrep version `1.61.1`, and `python-taint` module version `0.42`, also known as PyT.

As introduced in Section 2, Semgrep uses a pattern-matching approach by executing regular expressions (regex) to search for vulnerable patterns in the code. In the official Registry [56], Semgrep provides ready-to-use configuration files containing regex for vulnerability detection. Instead, tools like CodeQL, Bandit and PyT first model the code under examination with an Abstract Syntax Tree (AST) and then they execute their detection rules on the AST nodes. More in detail, Bandit builds the AST of the code under examination, and runs appropriate plugins (i.e., `assert_used`, `exec_used`, `set_bad_file_permissions`, etc.) [107] against the AST nodes. Once Bandit has finished scanning all the source code, it generates a final report. CodeQL treats code like data, which means that it is necessary to generate a CodeQL database to represent the codebase before running the analysis queries. Even in this case, CodeQL provides public test suites with ready-to-use queries [105,106]. After the code scanning, CodeQL generates a report. Finally, PyT generates the AST and creates the Control Flow Graph (CFG); then, it passes the CFG to a `Framework Adaptor`, which will mark the arguments of certain functions (by `Flask`, `Django` and other libraries) as tainted sources. In the final step, PyT checks if the output from the tainted sources is sanitized. If not, it raises an alert and generates a report.

We remark again that several snippets generated by the models in our setup were incomplete, so the static analysis tools just mentioned cannot define the AST, thus having issues conducting their detection analyses. At first, we analyzed the model predictions with *DeVAIC*. Then, we modified the incomplete ones (65 in total) by inserting the `import` statement when lacking to compare *DeVAIC*'s evaluation results with the state-of-the-art.

Furthermore, we enriched our analysis by comparing *DeVAIC* even with LLMs-based detection methods, which are nowadays a current trend and a widely used solution in the state of the art to perform vulnerability detection [108,109]. In particular, we adopted ChatGPT-3.5 and ChatGPT-4 [2], and the new model Claude-3.5-Sonnet [40]. ChatGPT-3.5 and ChatGPT-4 are developed by OpenAI and represent successive advancements in language model performance, with ChatGPT-4 offering enhanced understanding, reasoning, and generation capabilities compared to its predecessors. Claude-3.5-Sonnet, on the other hand, is the new model developed by Anthropic AI, which has set new performance standards in several tasks [110].

We prompted these 3 models using a Zero-Shot Role-Oriented (ZS-RO) prompt [57], i.e., we assigned to the model the role of vulnerability detection system (i.e., RO), successively asking to perform the vulnerability detection of the 500 generated snippets with a question about the related CWE (NL prompt: "*You are a vulnerability detection system. Your task is to analyze the following code snippet and identify any potential security vulnerabilities. Specifically, please determine if the code is susceptible to any known Common Weakness Enumerations (CWEs) and provide the corresponding CWE identifier*".

Considering the average values for all models, Table 6 shows that *DeVAIC* achieved high values for each metric. Regarding Precision, *DeVAIC* reaches 97%, which is comparable with the values achieved by Semgrep[4] (91%) and PyT (96%), while surpassing the values achieved by ChatGPT-4 (71%) and Claude-3.5-Sonnet (72%). Furthermore, *DeVAIC* achieved a maximum Recall of 92% on average, surpassing the baseline with a considerable gap compared to the employed solutions, which obtain a maximum of 76% (i.e., Claude-3.5-Sonnet). Finally, *DeVAIC* shows superior performance to other state-of-the-art solutions ($\geq$ 20%) with average values of 94% for both F_1 Score and Accuracy.

Considering the CWEs mapped to the experimental dataset shown in Table 4, we analyzed the snippets detected by *DeVAIC* and the baseline tools to evaluate how many CWE categories were covered. We found that *DeVAIC* detected a set of vulnerable snippets associated with 31 out of the 35 CWEs listed in Table 4 (i.e., 89%), confirming the high performance exhibited by the evaluation metrics in Table 6. Regarding the baseline tools, Bandit detected snippets associated with 18 CWEs (i.e., 51%), while we obtained 22 CWEs for CodeQL (i.e., 63%), 21 CWEs for Semgrep (i.e., 60%), 8 CWEs for PyT (i.e., 23%), 29 CWEs for Claude-3.5-Sonnet (i.e., 83%), 19 CWEs for ChatGPT-3.5 (i.e., 54%), and 21 CWEs for ChatGPT-4 (i.e., 60%). The baseline tools correctly identified some instances of generated snippets labeled with multiple CWEs, justifying the high number of CWEs covered. However, they missed other vulnerable instances, obtaining lower evaluation metrics than *DeVAIC*, as shown in Table 6.

We further evaluated if, for each metric (i.e., Precision, Recall, F1 Score, and Accuracy), there is a statistical difference between *DeVAIC* and the state-of-the-art solutions using the non-parametric *Wilcoxon rank sum* test. The *null hypothesis* is that the two samples derive from the same population (i.e., the two populations have equal medians). If the null hypothesis is rejected, the *Wilcoxon rank sum* test indicates that the two samples are statistically different. As deducible by the considerations explained above, for the Precision, *DeVAIC* is statistically different with Bandit, and with the models ChatGPT-4 and Claude-3.5-Sonnet. For all of the other metrics, *DeVAIC* achieves optimal results, which are statistically different (and better) than any other results obtained with the state of the art. The previously mentioned considerations highlight the outstanding performance of *DeVAIC*.

Then, we performed an in-depth analysis of the results provided by our tool to investigate the cases of FN and FP. Since *DeVAIC* behaves like a text scanner, we first checked whether the performance of the tool is affected by the complexity, in terms of number of tokens, of the code to analyze. Fig. 3 shows four boxplots to illustrate the complexity of the code across the TP, TN, FP, and FN cases. The figure highlights that the *interquartile ranges*, i.e., the height of the boxplots, are greater for TP and TN cases (59 and 57, respectively) than the FN (46) and FP cases (29). Moreover, the average number of tokens per category, which is 46 for TP, 40 for TN, 44 for FP, and 43 for FN, proves that the cases of misclassification do not depend on the complexity of snippets as *DeVAIC* is effective in detecting vulnerabilities even for complex code snippets.

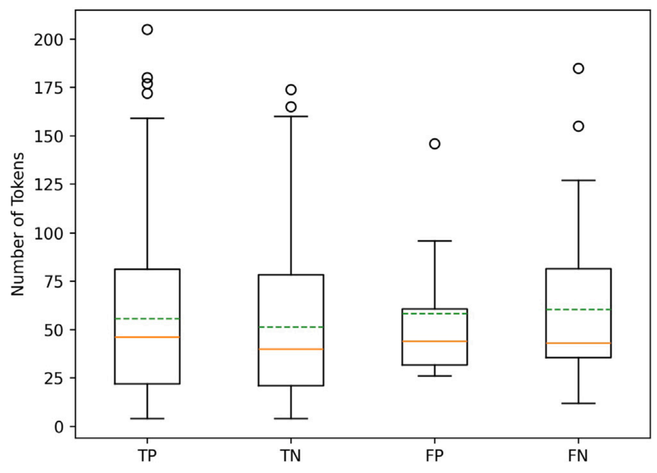

Fig. 3. Analysis of snippet lengths variability for each classification. The median values are in orange, while the mean values are in green.

5.4. Computational cost

We analyzed the computational times of *DeVAIC*. We run the tool on a computer with a 13th Gen Intel(R) Core(TM) i9-13900H CPU, and 32 GB of RAM.

We compared the execution time taken by *DeVAIC* with the other solutions employed. Fig. 4 shows the median execution time of each tool in analyzing 500 code snippets from the four models. Bandit is the faster (0.67 s), while CodeQL is the slower (570 seconds, which corresponds to approximately 9 minutes). PyT and Semgrep employed 253.40 and 123.67 s, respectively (2 and 4 minutes approximately). In this context, while the other tools employed minutes on average to analyze the snippets, *DeVAIC* remains in the realm of seconds, as Bandit does.

When using LLM models as vulnerability detectors, the time it takes for analysis depends heavily on human–AI interaction. If we approximate the time for a single detection, using the same prompt but changing snippets from time to time, to be about one minute, and if we analyze 500 snippets, the total time for a single model would be 500 minutes (~8 hours).

Furthermore, we deeply inspected the distribution of the times required by *DeVAIC* to execute the detection rules on every code snippet. Despite the different lengths of the analyzed code snippets, Fig. 5 shows that the tool exhibited an average execution time of 0.16 s (with a median of 0.14, a maximum of 0.59, and a minimum of 0.10). The figure highlights that for most of the snippets (169 in total, ~34%), the execution time falls within the range of 0.10 to 0.13 s.

Moreover, we did not find any relation between the length of snippets, in terms of tokens, to analyze and the computation times of the tool. Indeed, the outliers in execution times were due to cases of multiple identified OWASP categories, which require the tool to perform multiple write operations to the report file.

[4] When we used Semgrep, we executed a total of 1291 rules from the official ruleset registry for Python [56].

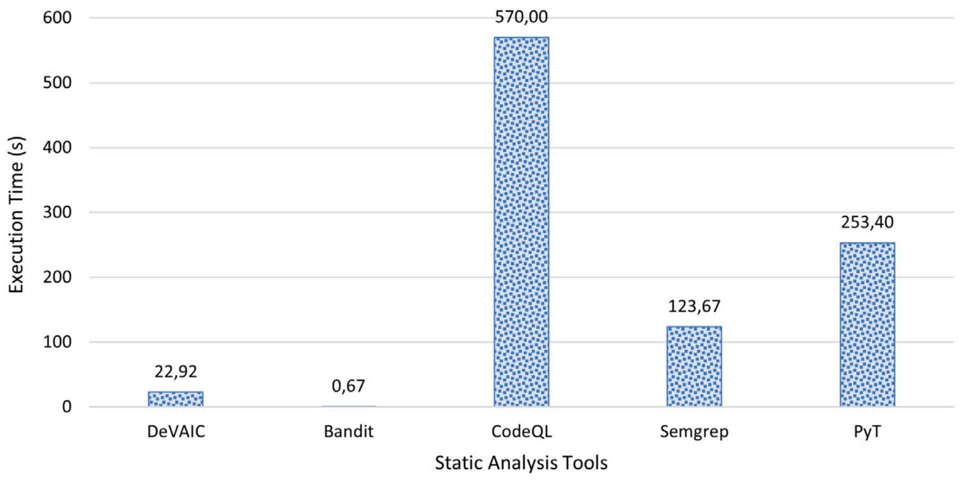

Fig. 4. Comparison of median execution times for *DeVAIC*, *Bandit*, *CodeQL*, *Semgrep*, and *PyT* for the analysis of all the 500 codes generated by the 4 models.

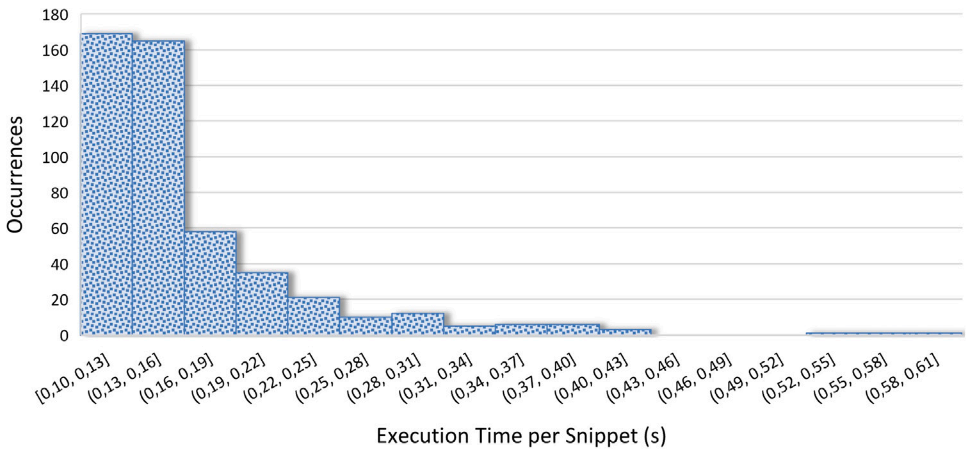

Fig. 5. Occurrences of execution times taken by *DeVAIC* for scanning and evaluating individual code snippets generated by the 4 models.

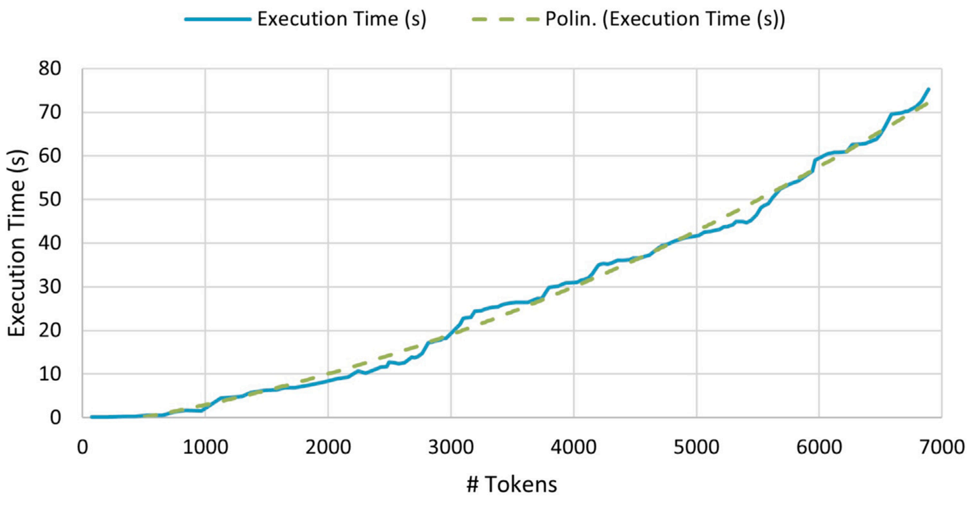

Fig. 6. Execution times of *DeVAIC* plotted against the cumulative number of tokens in code snippets. Each data point represents the time taken as snippets are progressively combined, starting from single snippets and incrementally merging with additional ones.

Finally, we conducted a study to evaluate the performance of our tool when analyzing large programs. To increase the code complexity, we concatenated the snippets generated by Gemini (we could have chosen the code generated by any other AI model as well) to create a new set of code characterized by 125 snippets with an incremental number of tokens, starting from 73 for the first snippet to 6892 for the last one.

Fig. 6 shows the *DeVAIC*'s performance in evaluating the new code collection. Analyzing snippets with more tokens led to longer execution times. While the initial snippet, comprising 73 tokens, only required

0.12 s for evaluation, the final snippet, consisting of 6892 tokens, took over 1 minute (75.23 s) to process. Furthermore, we applied a second-degree polynomial curve to fit the data, obtaining an R^2 value of 0.984. This value suggests a strong fit between the predicted execution times and the actual data and implies that the relationship between the snippet length and execution time is not linear but rather quadratic, with execution times increasing at a rate that accelerates with the number of tokens.

The polynomial relationship might be due to various factors inherent in code analysis processes, such as increased memory allocation, more intensive parsing, and a higher number of computational operations needed to process and analyze larger codebases.

6. Threats to validity

Rule Creation: The creation of detection rules based on regular expressions could introduce bias, as these rules are derived from patterns observed within a limited set of vulnerable code snippets. This limitation may result in rules that are either too specific or too general, affecting the tool's accuracy in real-world scenarios. To mitigate this threat, we employed a diverse dataset of vulnerabilities covering a broad range of CWEs and OWASP categories [71,73]. We also conducted iterative refinements of our rules by testing them on separate validation sets to ensure they accurately capture the intended patterns without being overly broad or narrow.
Coverage of CWEs: A key threat to the validity of our study concerns the comprehensive coverage of CWEs using detection rules. Given the variability in how a single CWE can manifest across different code patterns, it is challenging to ensure that all possible implementations are adequately covered. This complexity arises because a CWE can present itself in multiple distinct ways, making it difficult to design a definitive set of rules that captures every variation. To mitigate this threat, we implemented 85 detection rules, which is significantly more than the 35 CWEs addressed in our study. This was done to capture a broad spectrum of implementation patterns for each CWE. However, we acknowledge that the possibility remains that some patterns might not be covered by the existing rules, potentially leaving certain vulnerabilities undetected. Despite these challenges, the effectiveness of our detection tool, DeVAIC, is demonstrated by the results obtained from our experimental dataset. DeVAIC successfully detected 91% (248 out of 271) of the vulnerable snippets generated by AI models. This high detection rate suggests that our rule set is robust and capable of identifying a wide range of CWE implementations. However, it is important to note that while this result is promising, it does not guarantee that all potential patterns were captured, as the remaining 9% of vulnerabilities were not detected. Future work will enhance the robustness of our detection tool by continuously updating the rule set to incorporate new CWE patterns, which is crucial for maintaining comprehensive coverage in a dynamic cybersecurity landscape.
AI Models: The selection of AI models for evaluating *DeVAIC* may introduce bias, as the performance of these models can vary significantly. This variation could inadvertently affect the perceived effectiveness of *DeVAIC*. However, we remark that we selected four widely used AI code generation models, representing a range of underlying technologies and training datasets. This diversity helps ensure that our evaluation encompasses a variety of code-generation behaviors and vulnerabilities.
Metrics: The use of Precision, Recall, F_1 Score, and Accuracy as metrics relies on correctly classifying vulnerabilities. Any misclassification could impact these metrics and the interpretation of *DeVAIC* effectiveness. To enhance the reliability of our classification, we employed a multi-researcher approach, where multiple experts independently assessed the vulnerabilities before reaching a consensus. This process reduces the risk of subjective bias and ensures a more accurate classification of true and false positives/negatives.
Manual Evaluation: A potential threat to the validity of our manual analysis is the subjective nature of human evaluation, which can introduce biases and inconsistencies. Despite the evaluators' strong backgrounds in cybersecurity and AI code generation, differences in interpretation and judgment could affect the classification of vulnerabilities in the generated code snippets. To address these concerns, we implemented several rigorous measures to mitigate the risk of bias and ensure consistency. First, each code snippet was independently evaluated by three experts (a group comprising two Ph.D. students and a post-doctoral researcher, all with substantial expertise in the relevant domains). This independence in evaluation helps reduce the influence of individual biases, as each evaluator's assessment is made without knowledge of the others' judgments. Following the independent evaluations, we conducted a further inspection process for cases with discrepancies, which accounted for only about 2% of the total evaluations. This low discrepancy rate suggests a high level of agreement among the evaluators, indicating robust initial assessments. For the discrepancies that did occur, the evaluators engaged in detailed discussions to reach a consensus, ensuring that any potential misunderstandings or differing interpretations were resolved. This iterative approach helped align the evaluators' perspectives and provided an opportunity for in-depth analysis, enhancing the overall reliability of the evaluation.
Generalizability: The effectiveness of *DeVAIC* is currently limited to Python code. This focus raises questions about the tool's applicability to other programming languages that may have different syntax, semantics, and common vulnerabilities. While *DeVAIC* is initially designed for Python, the methodology used to create detection rules based on regular expressions and patterns observed in vulnerable code can be adapted to other languages. We are currently investigating this methodology for the extraction of vulnerable implementation patterns and rule creation in the C/C++ language. Adapting *DeVAIC* to C/C++ requires additional considerations due to the complexity and variety in how equivalent functionalities are implemented in these languages compared to Python (e.g., what can be accomplished with a single Python instruction might require several more complex instructions in C/C++). This complexity increases the difficulty of pattern identification and necessitates the development of more sophisticated detection rules. Future work will involve extending *DeVAIC*'s capabilities to additional languages and leveraging domain experts to ensure the accuracy and relevance of new rules.

7. Conclusion

In this work, we introduced *DeVAIC*, a tool that implements a set of detection rules to identify vulnerabilities in the Python code. The tool is designed to overcome the limitation of static analysis tools since it does not require complete programs, making it particularly suitable for AI-generated code. To define rules, we extracted vulnerable code from public datasets and, after grouping the code according to their OWASP categories and similarity, we found common patterns by standardizing the code snippets and using the LCS. We evaluated *DeVAIC* on code generated by four public AI-code generators. The results outline the tool's ability to detect vulnerabilities, showing an average F_1 Score and Accuracy both at 94%, overcoming the performance of other state-of-the-art solutions used as a baseline for the evaluation, with a limited computational cost.

Future work aims to extend the number of program languages to analyze, enhancing the set of detection rules available. Moreover, we also aim to enrich the list of CWEs covered by the tool.

CRediT authorship contribution statement

Domenico Cotroneo: Writing – original draft, Supervision, Resources, Project administration, Investigation, Funding acquisition, Data curation, Conceptualization. **Roberta De Luca:** Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Pietro Liguori:** Writing – original draft, Validation, Supervision, Methodology, Investigation, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The tool is available and the files to reproduce our experiments are publicly available on the following URL: https://github.com/dessertlab/DeVAIC.

Acknowledgments

This work has been partially supported by the "IDA—Information Disorder Awareness" Project funded by the European Union-Next Generation EU within the SERICS Program through the MUR National Recovery and Resilience Plan under Grant PE00000014. We are grateful to our students Francesco Balassone and Ferdinando Simone D'Agostino for their help in the early stage of this work.

References

[1] GitHub, GitHub Copilot, 2023, https://github.com/features/copilot.
[2] OpenAI, OpenAI ChatGPT, 2024, https://openai.com/blog/chatgpt/.
[3] Google, Google Gemini, 2023, https://gemini.google.com/app.
[4] Microsoft, Microsoft Copilot, 2023, https://www.microsoft.com/it-it/microsoft-copilot?market=it.
[5] GitHub/Blog, GitHub Copilot now has a better AI model and new capabilities, 2023, https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/.
[6] Cristina Improta, Poisoning programs by un-repairing code: Security concerns of AI-generated code, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops, ISSREW, IEEE, 2023, pp. 128–131.
[7] Jia Li, Zhuo Li, HuangZhao Zhang, Ge Li, Zhi Jin, Xing Hu, Xin Xia, Poison attack and poison detection on deep source code processing models, ACM Trans. Softw. Eng. Methodol. (2023).
[8] Aftab Hussain, Md Rafiqul Islam Rabin, Mohammad Amin Alipour, Trojanedcm: A repository for poisoned neural models of source code, 2023, arXiv preprint arXiv:2311.14850.
[9] Sivana Hamer, Marcelo d'Amorim, Laurie Williams, Just another copy and paste? Comparing the security vulnerabilities of ChatGPT generated code and StackOverflow answers, 2024, arXiv preprint arXiv:2403.15600.
[10] GitHub. [n. d.]. GitHub Copilot for Individuals, https://docs.github.com/en/copilot/overview-of-github-copilot/about-github-copilot-for-individuals.
[11] Bard/FAQ, What can bard do and other frequently asked questions - bard, 2023, https://bard.google.com/faq?hl=en.
[12] Bing/FAQ, Microsoft bing - frequently asked questions, 2023, https://www.microsoft.com/en-us/bing?form=MA13FV.
[13] Pietro Liguori, Erfan Al-Hossami, Vittorio Orbinato, Roberto Natella, Samira Shaikh, Domenico Cotroneo, Bojan Cukic, EVIL: exploiting software via natural language, in: 2021 IEEE 32nd International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2021, pp. 321–332.
[14] Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, Samira Shaikh, Can we generate shellcodes via natural language? An empirical study, Autom. Softw. Eng. 29 (1) (2022) 30.
[15] Li Ma, Huihong Yang, Jianxiong Xu, Zexian Yang, Qidi Lao, Dong Yuan, Code analysis with static application security testing for Python program, J. Signal Process. Syst. 94 (11) (2022) 1169–1182.
[16] PyCQA, Bandit, https://github.com/PyCQA/bandit/tree/main.
[17] python-security, PyT, https://github.com/python-security/pyt.
[18] GitHub, CodeQL, https://codeql.github.com/.
[19] OpenAI, HumanEval: Hand-Written Evaluation Set, https://github.com/openai/human-eval.
[20] odashi, Django Dataset for Code Translation Tasks, https://github.com/odashi/ase15-django-dataset.
[21] Pietro Liguori, Erfan Al-Hossami, Domenico Cotroneo, Roberto Natella, Bojan Cukic, Samira Shaikh, Shellcode_IA32: A dataset for automatic shellcode generation, in: Royi Lachmy, Ziyu Yao, Greg Durrett, Milos Gligoric, Junyi Jessy Li, Ray Mooney, Graham Neubig, Yu Su, Huan Sun, Reut Tsarfaty (Eds.), Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), Association for Computational Linguistics, Online, 2021, pp. 58–64.
[22] Carnegie Mellon University NeuLab and STRUDEL Lab, CoNaLa, https://conala-corpus.github.io/.
[23] Li Zhong, Zilong Wang, A study on robustness and reliability of large language model code generation, 2023, arXiv preprint arXiv:2308.10335.
[24] Ferdiansyah Mastjik, Cihan Varol, Asaf Varol, Comparison of pattern matching techniques on identification of same family malware, Int. J. Inf. Secur. Sci. 4 (3) (2015) 104–111.
[25] Nency Patel, Narendra Shekokar, Implementation of pattern matching algorithm to defend SQLIA, Procedia Comput. Sci. 45 (2015) 453–459.
[26] Andrew Walenstein, Michael Venable, Matthew Hayes, Christopher Thompson, Arun Lakhotia, Exploiting similarity between variants to defeat malware, in: Proc. BlackHat DC Conf, Citeseer, 2007.
[27] Austin Mordahl, Automatic testing and benchmarking for configurable static analysis tools, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 1532–1536.
[28] Sarah Nadi, Thorsten Berger, Christian Kästner, Krzysztof Czarnecki, Mining configuration constraints: Static analyses and empirical results, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 140–151.
[29] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, Baishakhi Ray, Deep learning based vulnerability detection: Are we there yet, IEEE Trans. Softw. Eng. (2021).
[30] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, David Wagner, 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, 2023, pp. 654–668.
[31] Yi Li, Shaohua Wang, Tien N. Nguyen, Vulnerability detection with fine-grained interpretations, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 292–303.
[32] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, Yuyi Zhong, Vuldeepecker: A deep learning-based system for vulnerability detection, 2018, arXiv preprint arXiv:1801.01681.
[33] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, et al., Large language models for code analysis: Do LLMs really do their job? 2023, arXiv preprint arXiv:2310.12357.
[34] Muna Al-Hawawreh, Ahamed Aljuhani, Yaser Jararweh, Chatgpt for cybersecurity: practical applications, challenges, and future directions, Cluster Comput. (2023) 1–16.
[35] Anton Cheshkov, Pavel Zadorozhny, Rodion Levichev, Evaluation of ChatGPT model for vulnerability detection, 2023, arXiv preprint arXiv:2304.07232.
[36] GitHub, The top programming languages, https://octoverse.github.com/2022/top-programming-languages.
[37] Leo A. Meyerovich, Ariel S. Rabkin, Empirical analysis of programming language adoption, in: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, 2013, pp. 1–18.
[38] Statista, Most used programming languages among developers worldwide as of 2023, https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/.
[39] returntocorp, Semgrep, https://github.com/returntocorp/semgrep.
[40] Anthropic, Claude-3.5-Sonnet, 2024, https://www.anthropic.com/news/claude-3-5-sonnet.
[41] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, Asleep at the keyboard? assessing the security of github copilot's code contributions, in: 2022 IEEE Symposium on Security and Privacy, SP, IEEE, 2022, pp. 754–768.
[42] Trevor Dunlap, Seaver Thorn, William Enck, Bradley Reaves, Finding fixed vulnerabilities with off-the-shelf static analysis, in: 2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P), IEEE, 2023, pp. 489–505.
[43] Atieh Bakhshandeh, Abdalsamad Keramatfar, Amir Norouzi, Mohammad Mahdi Chekidehkhoun, Using chatgpt as a static application security testing tool, 2023, arXiv preprint arXiv:2308.14434.

[44] Adhishree Kathikar, Aishwarya Nair, Ben Lazarine, Agrim Sachdeva, Sagar Samtani, Assessing the vulnerabilities of the open-source artificial intelligence (AI) landscape: A large-scale analysis of the hugging face platform, in: 2023 IEEE International Conference on Intelligence and Security Informatics, ISI, IEEE, 2023, pp. 1–6.

[45] Brian Chess, Gary McGraw, Static analysis for security, IEEE Secur. Priv. 2 (6) (2004) 76–79.

[46] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, Le Traon, Static analysis of android apps: A systematic literature review, Inf. Softw. Technol. 88 (2017) 67–95.

[47] Ya Pan, Xiuting Ge, Chunrong Fang, Yong Fan, A systematic literature review of android malware detection using static analysis, IEEE Access 8 (2020) 116363–116379.

[48] Katerina Goseva-Popstojanova, Andrei Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Inf. Softw. Technol. 68 (2015) 18–33.

[49] Jukka Ruohonen, Kalle Hjerppe, Kalle Rindell, A large-scale security-oriented static analysis of python packages in pypi, in: 2021 18th International Conference on Privacy, Security and Trust, PST, IEEE, 2021, pp. 1–10.

[50] D.A. Kapustin, V.V. Shvyrov, T.I. Shulika, Static analysis of corpus of source codes of python applications, Program. Comput. Softw. 49 (4) (2023) 302–309.

[51] Shuanghe Peng, Peiyao Liu, Jing Han, A python security analysis framework in integrity verification and vulnerability detection, Wuhan Univ. J. Nat. Sci. 24 (2) (2019) 141–148.

[52] Damian Lyons, Dino Becaj, A meta-level approach for multilingual taint analysis, in: International Conference on Software and Data Technologies, ICSOFT, 2021.

[53] Vinuri Bandara, Thisura Rathnayake, Nipuna Weerasekara, Charitha Elvitigala, Kenneth Thilakarathna, Primal Wijesekera, Chamath Keppitiyagama, Fix that fix commit: A real-world remediation analysis of JavaScript projects, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2020, pp. 198–202.

[54] Matías Gobbi, Johannes Kinder, Poster: Using codeql to detect malware in npm, in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 3519–3521.

[55] Boris Cherry, Pol Benats, Maxime Gobert, Loup Meurice, Csaba Nagy, Anthony Cleve, Static analysis of database accesses in mongodb applications, in: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2022, pp. 930–934.

[56] Semgrep, Semgrep Registry, 2024, https://semgrep.dev/r.

[57] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, Gianluca Stringhini, LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks, in: IEEE Symposium on Security and Privacy, 2024.

[58] Moumita Das Purba, Arpita Ghosh, Benjamin J. Radford, Bill Chu, Software vulnerability detection using large language models, in: 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops, ISSREW, IEEE, 2023, pp. 112–119.

[59] Raphaël Khoury, Anderson R. Avila, Jacob Brunelle, Baba Mamadou Camara, How secure is code generated by ChatGPT? 2023, arXiv preprint arXiv:2304.09655.

[60] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt, Lost at c: A user study on the security implications of large language model code assistants, 2023, arXiv preprint arXiv:2208.09727.

[61] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, Siddharth Garg, Security implications of large language model code assistants: A user study, 2022, arXiv preprint arXiv:2208.09727.

[62] Hossein Hajipour, Thorsten Holz, Lea Schönherr, Mario Fritz, Systematically finding security vulnerabilities in black-box code generation models, 2023, arXiv preprint arXiv:2302.04012.

[63] GitHub Docs, Python queries for CodeQL analysis, 2023, https://docs.github.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/python-built-in-queries.

[64] PyYAML, PyYAML Documentation, https://pyyaml.org/wiki/PyYAMLDocumentation.

[65] Catherine Tony, Nicolás E. Díaz Ferreyra, Markus Mutas, Salem Dhiff, Riccardo Scandariato, Prompting techniques for secure code generation: A systematic investigation, 2024, arXiv preprint arXiv:2407.07064.

[66] Youjia Li, Jianjun Shi, Zheng Zhang, An approach for rapid source code development based on ChatGPT and prompt engineering, IEEE Access (2024).

[67] Domenico Cotroneo, Cristina Improta, Pietro Liguori, Roberto Natella, Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks, 2023, arXiv preprint arXiv:2308.04451.

[68] ZeoVan, MSR_20_Code_vulnerability_CSV_Dataset, 2020, https://github.com/ZeoVan/MSR_20_Code_vulnerability_CSV_Dataset.

[69] Jiahao Fan, Yi Li, Shaohua Wang, Tien N. Nguyen, AC/C++ code vulnerability dataset with code changes and CVE summaries, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 508–512.

[70] NIST, NIST software assurance reference dataset, 2023, https://samate.nist.gov/SARD/.

[71] Security & Software Engineering Research Lab at University of Notre Dame, SecurityEval, 2023, https://github.com/s2e-lab/SecurityEval.

[72] Mohammed Latif Siddiq, Joanna 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 (MSR4P&S22), 2022, http://dx.doi.org/10.1145/3549035.3561184.

[73] Pearce, et al., Copilot CWE Scenarios Dataset, 2023, https://zenodo.org/records/5225651.

[74] SoftSec Institute, LLMSecEval, 2023, https://github.com/tuhh-softsec/LLMSecEval/.

[75] Catherine Tony, Markus Mutas, Nicolas Díaz Ferreyra, Riccardo Scandariato, LLMSecEval: A dataset of natural language prompts for security evaluations, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR, 2023, http://dx.doi.org/10.5281/zenodo.7565965.

[76] OWASP, OWASP Top 10:2021, https://owasp.org/Top10/.

[77] Anne Honkaranta, Tiina Leppänen, Andrei Costin, Towards practical cybersecurity mapping of stride and cwe—a multi-perspective approach, in: 2021 29th Conference of Open Innovations Association, FRUCT, IEEE, 2021, pp. 150–159.

[78] MITRE, 2021 CWE Top 25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html.

[79] MITRE, 2022 CWE Top 25 Most Dangerous Software Weaknesses, https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

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

[81] Richard J. Thomas, Tom Chothia, Learning from vulnerabilities-categorising, understanding and detecting weaknesses in industrial control systems, in: Computer Security: ESORICS 2020 International Workshops, CyberICPS, SECPRE, and ADIoT, Guildford, UK, September 14–18, 2020, Revised Selected Papers 6, Springer, 2020, pp. 100–116.

[82] Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng Yao, Na Meng, Example-based vulnerability detection and repair in java code, in: Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 190–201.

[83] G. Appa Rao, K. Venkata Rao, P.V.G.D. Prasad Reddy, T. Lava Kumar, An efficient procedure for characteristic mining of mathematical formulas from document, Int. J. Eng. Sci. Technol. (IJEST) 10 (03) (2018).

[84] G. Appa Rao, G. Srinivas, K. Venkata Rao, P.V.G.D. Prasad Reddy, Characteristic mining of mathematical formulas from document-a comparative study on sequence matcher and levenshtein distance procedure, Int. J. Comput. Sci. Eng. 6 (4) (2018) 400–403.

[85] G. Appa Rao, G. Srinivas, K. Venkata Rao, P.V.G.D. Prasad Reddy, A partial ratio and ratio based fuzzy-wuzzy procedure for characteristic mining of mathematical formulas from documents, IJSC—ICTACT J. Soft. Comput. 8 (4) (2018) 1728–1732.

[86] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, Robert H. Deng, Vurle: Automatic vulnerability detection and repair by learning from examples, in: Computer Security–ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II 22, Springer, 2017, pp. 229–246.

[87] Boris Chernis, Rakesh Verma, Machine learning methods for software vulnerability detection, in: Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, 2018, pp. 31–39.

[88] Wael Khreich, Babak Khosravifar, Abdelwahab Hamou-Lhadj, Chamseddine Talhi, An anomaly detection system based on variable N-gram features and one-class SVM, Inf. Softw. Technol. 91 (2017) 186–197.

[89] Charlotte Lecluze, Loïs Rigouste, Emmanuel Giguet, Nadine Lucas, Which granularity to bootstrap a multilingual method of document alignment: character N-grams or word N-grams? Proc.-Soc. Behav. Sci. 95 (2013) 473–481.

[90] Paul McNamee, James Mayfield, Character n-gram tokenization for European language text retrieval, Inf. Retrieval 7 (2004) 73–97.

[91] Xiaoyun Li, Hongyu Zhang, Van-Hoang Le, Pengfei Chen, Logshrink: Effective log compression by leveraging commonality and variability of log data, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–12.

[92] Tongshuai Wu, Liwei Chen, Gewangzi Du, Dan Meng, Gang Shi, UltraVCS: Ultra-fine-grained variable-based code slicing for automated vulnerability detection, IEEE Trans. Inf. Forensics Secur. (2024).

[93] Miryung Kim, David Notkin, Dan Grossman, Automatic inference of structural changes for matching across program versions, in: 29th International Conference on Software Engineering, ICSE'07, IEEE, 2007, pp. 333–343.

[94] OWASP, OWASP cheat sheet series - introduction, 2024, https://cheatsheetseries.owasp.org/index.html.

[95] OWASP, Input validation cheat sheet, 2024, https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html.

[96] OWASP, Injection prevention cheat sheet, 2024, https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html.

[97] OWASP, Cross site scripting prevention cheat sheet, 2024, https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html.

[98] Domenico Cotroneo, Cristina Improta, Pietro Liguori, Roberto Natella, Vulnerabilities in ai code generators: Exploring targeted data poisoning attacks, in: Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, 2024, pp. 280–292.

[99] S2E-Lab, SecurityEval NL prompts, 2022, https://github.com/s2e-lab/SecurityEval/blob/main/dataset.jsonl.

[100] tuhh softsec, LLMSecEval NL prompts, 2023, https://github.com/tuhh-softsec/LLMSecEval/blob/main/Dataset/LLMSecEval-prompts.json.

[101] Microsoft, CodeXGLUE NL prompts, 2021, https://github.com/microsoft/CodeXGLUE/blob/main/Text-Code/text-to-code/dataset/concode/test.json.

[102] OWASP, Path traversal, 2024, https://owasp.org/www-community/attacks/Path_Traversal.

[103] Python, Abstract Syntax Tree Documentation, https://docs.python.org/3/library/ast.html.

[104] MITRE, Common weakness enumeration (CWE), 2024, https://cwe.mitre.org/.

[105] GitHub, CodeQL: Experimental Security Queries for Python language, https://github.com/github/codeql/tree/main/python/ql/src/experimental/Security.

[106] GitHub, CodeQL: Security Queries for Python language, https://github.com/github/codeql/tree/main/python/ql/src/Security.

[107] Bandit, Test Plugins, 2024, https://bandit.readthedocs.io/en/latest/plugins/.

[108] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, Hui Li, Prompt-enhanced software vulnerability detection using chatgpt, in: Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings, 2024, pp. 276–277.

[109] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian, Assisting static analysis with large language models: A chatgpt experiment, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 2107–2111.

[110] Anthropic, Claude 3.5 Sonnet Model Card Addendum, 2024, https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf.