2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W) | 979-8-3315-1205-7/25/$31.00 ©2025 IEEE | DOI: 10.1109/DSN-W65791.2025.00077

Securing AI Code Generation Through Automated Pattern-Based Patching

Francesco Altiero, Domenico Cotroneo, Roberta De Luca, Pietro Liguori
University of Naples Federico II, Naples, Italy
franc.altiero@studenti.unina.it, {cotroneo, roberta.deluca2, pietro.liguori}@unina.it

***Abstract*—In an era where AI code generators are reshaping software development, the risk of introducing vulnerabilities is a growing concern. With the increasing reliance on machine learning-driven tools, it is essential to support developers with effective automated vulnerability patching while prioritizing the reliability and security of AI. However, current methods face significant challenges, including a high rate of false positives and difficulties in generating high-quality patches.**

This paper presents *PatchitPy*, a new patching solution that leverages a pattern-matching approach to detect and patch Python vulnerabilities. We generated 609 Python code using three popular AI code generators (i.e., GitHub Copilot, Claude-3.7-Sonnet, and DeepSeek-V3) and evaluated our solution's performance on these samples. The results demonstrate the effectiveness of *PatchitPy*, which outperforms state-of-the-art solutions in both detection and patching. *PatchitPy* achieved an F_1 score of 93% and an Accuracy of 89% for vulnerability detection, and produced high-quality patches with a 80% repair rate for identified vulnerabilities. Furthermore, the patches preserve code quality with minimal impact on complexity, ensuring long-term code maintainability.

***Index Terms*—Vulnerability Patching, Static Analysis, Vulnerability Detection, AI-generated Code**

I. Introduction

Ensuring the reliability and security of software has become a complex challenge, especially with the rise of AI-powered code generators in software development. A survey from GitHub Octoverse found that 92% of developers are using AI coding tools, like GitHub Copilot [1], to boost their productivity [2]. However, AI-powered coding assistants can produce insecure code [3], [4]. Recent work highlighted that Large Language Models (LLMs) often rely on a large amount of source code obtained from public repositories [5], which may include deprecated or vulnerable functions that can *poison* the models [6]–[9], causing the generation of vulnerable code. This scenario emphasizes the critical need for automated solutions to support developers in identifying and patching vulnerabilities, ensuring the security and robustness of the generated code [10], [11].

Automated vulnerability patching is a challenge for developers. Unpatched vulnerabilities expose software systems to potential attacks, providing entry points for cybercriminals [12]. The number of identified software vulnerabilities is constantly rising, reaching over 34.660 Common Vulnerabilities and Exposures (CVEs) in November 2024, compared to 29.066 in 2023, 25.084 in 2022, and 20.153 in 2021 [13]. Another critical factor is the cost of cybercrime, which is estimated to reach $10.5 trillion for companies and users by 2025 [14]. This escalating threat underscores the urgent need for effective automated detection and patching solutions.

Manual patching, although effective, is often impractical due to its time-consuming nature and the high level of expertise required to identify and address security flaws [15], [16]. Therefore, automated solutions are crucial to support developers, minimizing manual effort and improving software reliability [17]. Unfortunately, current state-of-the-art tools face notable limitations. For instance, traditional Automatic Program Repair (APR) tools are specific for patching bugs, limiting their generalization for security vulnerabilities [18], [19]. Additionally, their patch quality is a frequent issue, often causing the introduction of new faults and the reduction of program maintainability [20], [21]. Literature has also examined LLMs for vulnerability patching, showing some limitations, like the generation of oversimplified patches [19], [22], that can be improved using prompt-tuning techniques to refine the model with the vulnerability description [23], [24]. However, a generic software developer may not have the security knowledge to describe the vulnerabilities. Another critical challenge is the issue of False Positives (FPs) and False Negatives (FNs) [25]–[32] as they can create unnecessary overhead or expose code to security risks.

This paper presents *PatchitPy*, a new patching tool to support developers in detecting and patching software vulnerabilities for Python, the most widely used programming language in 2024 on GitHub [33]. Multiple security weaknesses are related to this language, like improper input handling and insecure use of libraries, underscoring the need for reliable patching solutions tailored for Python code [34]. This solution builds upon a previous work that introduced a tool designed only for vulnerability detection [35]. In this paper, we present a pattern-matching approach to automate the patching process, transforming vulnerable code into safe alternatives. Then, we implemented these functionalities as an extension for Visual Studio Code (VS Code), the most used integrated development environment (IDE) implemented by Microsoft [36]–[40]. Developers using built-in code generators (e.g., GitHub Copilot) can assess the generated code with *PatchitPy*, an effective solution for enhancing code security during development.

We evaluated *PatchitPy* to detect and patch vulnerabilities in the code generated by 3 different AI models (i.e., GitHub

2325-6664/25/$31.00 ©2025 IEEE
DOI 10.1109/DSN-W65791.2025.00077

Copilot [1], Claude-3.7-Sonnet [41], and DeepSeek-V3 [42]). Our key results are as follows:

1) **Vulnerability detection with 93% of F_1 Score and 89% of Accuracy.** *PatchitPy* correctly identified code vulnerable to 51 distinct CWEs[1] for Copilot-generated code, 41 CWEs for Claude-generated code, and 47 CWEs for DeepSeek-generated code. Moreover, it achieved the highest F_1 Score and Accuracy, surpassing state-of-the-art approaches (i.e., CodeQL [44], Bandit [45], Semgrep [46], ChatGPT-4o [47], Claude-3.7 [41], and Gemini-2.0-Flash [48]).
2) **High-quality patches with minimal impact on code complexity.** *PatchitPy* achieved a 68% repair rate for the vulnerabilities detected in Copilot-generated code, 89% in Claude-generated code, and 84% in DeepSeek-generated code, outperforming the baseline. Additionally, the applied patches demonstrate the effectiveness of our pattern-matching approach, introducing only minimal changes that reduce the impact on cyclomatic complexity and preserve the original code structure.

In the following, Section II presents the details of our solution; Section III describes the results; Section IV discusses related work; Section V concludes the paper.

II. Workflow

PatchitPy is a patching tool designed to assist developers in promptly assessing software vulnerabilities in Python code. It extends a previous work [35], exclusively focused on vulnerability detection via rules based on regular expressions, without relying on Abstract Syntax Tree (AST) modeling. To define the detection rules, the authors analyzed 240 vulnerable Python code samples collected from two datasets (i.e., *SecurityEval* [49] and *Copilot CWE Scenarios Dataset* [50]), mapping them to OWASP Top 10:2021 [51] categories using CWE labels [43]. This approach was particularly impactful on AI-generated code, which often consists of portions of code, overcoming the detection performance of static analysis tools, such as Bandit [45], CodeQL [44], and Semgrep [46].

Building on this foundation, *PatchitPy* introduces a two-phase workflow for comprehensive vulnerability management, encompassing both detection and remediation, as shown in Fig.1. The first phase analyzes Python code to identify security flaws using static pattern matching. Once vulnerabilities are reported, the second phase activates an automated remediation pipeline that interprets the intent of the code, and generates safe patches by replacing insecure functions with recommended alternatives. This end-to-end approach enhances the previous detection-only methodology, offering developers a practical and automated solution for identifying and mitigating security risks in Python.

A. Safe Patterns Definition

To enrich the rules implemented in previous work with the patching logic, we developed safe alternatives for vulnerable samples collected in previous work. Fig. 2 shows a detailed flowchart outlining the pattern substitution logic.

[1]The Common Weakness Enumeration (CWE) is a list of common software and hardware weaknesses maintained by the MITRE Corporation [43].

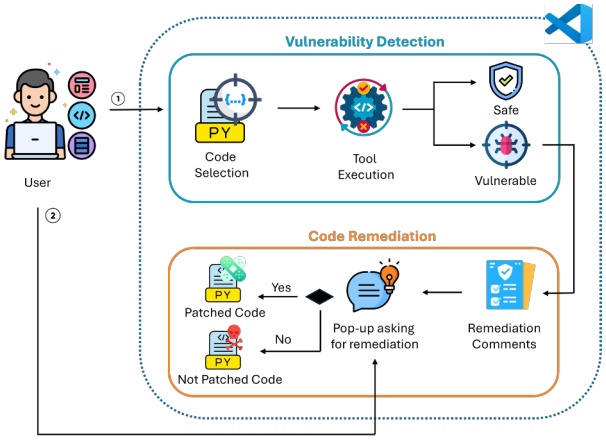

Fig. 1. Overview of *PatchitPy* for vulnerability detection and patching.

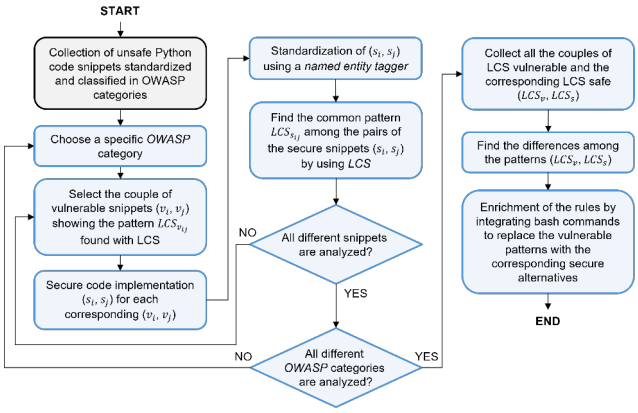

Fig. 2. Workflow for vulnerable pattern substitution with safe alternatives.

For each OWASP category, we selected all pairs of vulnerable samples (v_i,v_j) from the collection defined in previous work. Then, we manually created the safe implementations (s_i,s_j) for each pair of unsafe Python samples. This task was done by 3 members of our research group (i.e., 2 Ph.D. students and a post-doctoral researcher), all with a strong background and expertise in cybersecurity and Python programming, reflecting diverse professional experience and academic qualifications. To ensure the correctness of this process, the 3 members closely collaborated to write safe versions for vulnerable samples, utilizing OWASP's Cheat Sheet Series [52]–[56] and MITRE [43], [57] reports. Finally, they verified the security of these safe samples using a static analysis tool (i.e., CodeQL [44]).

To create rules for patching, we extracted meaningful common patterns among vulnerable collections and safe samples. First, we *standardized* all vulnerable and safe snippets using a *named entity tagger*, which returns a dictionary of standardizable tokens for the input and output parameters

TABLE I
EXAMPLES OF STANDARDIZED CODE SAMPLES. IN **BOLD** ARE SHOWN THE COMMON PATTERNS $LCS_{v_{12}}$ FOUND BETWEEN THE VULNERABLE SNIPPETS (v_1,v_2), AND THE $LCS_{s_{12}}$ RELATED TO THE SAFE IMPLEMENTATIONS (s_1,s_2). IN **BLUE** ARE SHOWN THE DIFFERENCES EXTRACTED BETWEEN ($LCS_{v_{12}}$,$LCS_{s_{12}}$).

	Vulnerable Standardized Code (v_i)	**Secure Standardized Code** (s_i)
(v_1,s_1)	**from flask import Flask, request\n app = Flask(__name__)\n @app.route("/comments") def comments():\n var0 = request.args.get(var1, var2)\n return f" {var0} "\n if __name__ == "__main__":\n app.run(debug=True)**	**from flask import Flask, request, escape\n app = Flask(__name__)\n @app.route("/comments") def comments():\n var0 = request.args.get(var1, var2)\n return f" {escape(var0)} "\n if __name__ == "__main__":\n app.run(debug=False, use_debugger=False, use_reloader=False)**
(v_2,s_2)	**from flask import Flask, request, make_response\n app1 = Flask(__name__)\n @app1.route("/showName") def name():\n var0 = request.args.get(var1)\n return** make_response(**f"Hello {var0}")\n if __name__ == "__main__":\n app1.run(debug=True)**	**from flask import Flask, request, make_response, escape\n app1 = Flask(__name__)\n @app1.route("/showName") def name():\n var0 = request.args.get(var1)\n return** make_response(**f"Hello {escape(var0)}")\n if __name__ == "__main__":\n app1.run(debug=False, use_debugger=False, use_reloader=False)**

of functions, extracted through regular expressions. These regular expressions prevent the standardization of parameters of a function, preserving the details about its behavior (e.g., configurative parameters are recognized by the "=" symbol or keywords like `True`, `False`, etc.). The other tokens selected for the standardization are transformed in "*var#*", where # denotes a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize.

TABLE I - "Vulnerable Standardized Code (v_i)" column exhibits instances of standardized snippets vulnerable to CWE-079 (*Cross-Site Scripting*) due to the direct inclusion of user-controlled input (`var0`) in the HTML response without proper sanitization or escaping, enabling injection attacks [55]. Additionally, the snippets are vulnerable to CWE-209 (*Information Exposure Through an Error Message*), as the application runs in debug mode (i.e., `debug=True`), potentially leaking sensitive information in error responses that could aid an attacker in identifying exploitable weaknesses [58].

These CWEs belong to 2 different OWASP categories, i.e., *Injection* for CWE-079 and *Insecure Design* for CWE-209. This aspect highlights that some couples (v_i,v_j) can belong to multiple OWASP categories, as a single piece of code may introduce multiple vulnerabilities. Furthermore, the vulnerable code in row #2 exemplifies how a *configuration parameter* can impact security. The `debug` mode is explicitly enabled with `debug=True`, which is not replaced with `var#` during standardization since it represents a configuration setting for the Flask framework, recognized with the "=" symbol.

TABLE I - "Secure Standardized Code (s_i)" column shows safe alternatives for the corresponding vulnerable samples. To mitigate the risk associated with CWE-079, samples s_1 and s_2 escape user-provided input using the `escape()` function from `Flask`, ensuring that any HTML special characters in `var0` are properly encoded before being rendered in the response. This prevents attackers from injecting and executing arbitrary scripts in the client's browser [55]. Additionally, to mitigate CWE-209, samples s_1 and s_2 disable Flask's debug mode by setting `debug=False`, `use_debugger=False`, and `use_reloader=False`. This prevents sensitive debugging information, such as stack traces and internal server details, from being exposed in error messages, thus reducing the risk of leaking critical system details to attackers [58].

After the standardization, we used the LCS method to extract meaningful common implementation patterns (i.e., $LCS_{v_{ij}}$ and $LCS_{s_{ij}}$) from each pair of vulnerable (v_1,v_2) and safe samples (s_i,s_j). Subsequently, we compared every couple of vulnerable and safe patterns ($LCS_{v_{ij}}$, $LCS_{s_{ij}}$) to extract the additional parts of code in $LCS_{s_{ij}}$ that are missing in $LCS_{v_{ij}}$. To achieve this, we use the `SequenceMatcher` class from the Python `difflib` module. For instance, the two columns of TABLE I display in **bold** the common implementation patterns $LCS_{v_{12}}$ for the couple (v_1,v_2) and $LCS_{s_{12}}$ for (s_1,s_2), marking in **blue** the additional checks to mitigate the risk of Cross-Site Scripting attack and the Information Exposure.

Finally, we improved the regular expressions implemented in previous work by adding the remediation logic to map the safe alternatives to the vulnerable patterns. When the rules detect a vulnerable pattern, the tool replaces it with the appropriate patches, eventually adding the necessary modules to import (e.g., when the patch contains APIs from a module missing in the vulnerable code). The tool executes 85 detection rules, ensuring the detection of diverse vulnerable implementations. Similarly, each triggered rule is associated with a specific patch to address the corresponding vulnerable part of the detected code.

B. Implementation

We implemented the tool as an extension for Visual Studio Code (VS Code), which is a lightweight, open-source, and popular integrated development environment (IDE) implemented by Microsoft. Recent Stack Overflow surveys show that, since 2018, VS Code has become the most used IDE by software developers [36]–[40]. To support code development, VS Code offers additional features with the extension marketplace [59], allowing users to customize the IDE according to their needs. VS Code provides a well-documented `Extension API` [60], [61] to support the extensions' development. In particular, VS Code leverages the `Electron` [62] framework to enable extension development employing `Node.js`. The availability of various guides and tutorials makes the API easy to use [63].

PatchitPy is a VS Code extension that introduce a feature for evaluating specific code segments, assisting users who want to assess code blocks produced by a code generator (e.g., GitHub Copilot [1]). Thanks to its lightweight pattern-matching approach, it is particularly effective on AI-generated code, which is often incomplete or lacks broader context. Naturally, users can also launch the extension on the entire program. To implement this feature, we configured the extension manifest file (i.e., `package.json`) to add a command to the right-click menu after code selection. Then, we configured the `activate()` function in the `extension.js` file to execute the bash script that analyzes the selected code. When the tool intercepts a vulnerable pattern, the extension shows pop-ups informing users about the detection, displaying fix suggestions, and asking to patch the code. If users click `Yes`, the extension triggers the bash script to modify the vulnerable source code with the appropriate safe patches. The code patching feature leverages VS Code's `TextEdit` API, using the `replace()` method of the `editBuilder` object to modify code. Furthermore, VS Code's `Position` API accurately places new module imports used in the patch at the beginning of the file.

The source code to configure the extension[2], the files needed to reproduce our experiments and the Appendix are available at the following URL: https://github.com/dessertlab/PatchitPy

III. Case Study

We evaluated the performance of *PatchitPy* on code generated by 3 publicly available AI models: GitHub Copilot [1], Claude-3.7-Sonnet [41], and DeepSeek-V3 [42]. These models, accessible via APIs, generate code suggestions based on Natural Language (NL) prompts. Our evaluation covered detection performance, measuring Precision, Recall, F_1 Score, and Accuracy, along with an in-depth analysis of patching performance, including an assessment of patch correctness through a rigorous manual review process, and measuring code quality and cyclomatic complexity, as detailed in § III-C.

A. Code Generation using NL Prompts

We selected 203 NL prompts (121 from *SecurityEval* [49] and 82 from *LLMSecEval* [64]) to generate Python code using GitHub Copilot, Claude-3.7-Sonnet, and DeepSeek-V3. Each model was provided with all 203 prompts, obtaining 609 samples to assess *PatchitPy* for vulnerability detection and patching. *SecurityEval* and *LLMSecEval* provide NL prompts for evaluating LLMs in code generation across various security scenarios, including external input handling, file operations, and HTTP requests. The generated code may exhibit vulnerabilities linked to specific CWEs mapped in the NL prompts. In particular, *SecurityEval* covers 69 CWEs, while *LLMSecEval* includes 18 from the Top 25 CWEs of 2021 [65]. The number of tokens for these 203 prompts has an average value of 21 (median value is 15), with a minimum of 3 and a maximum of 63. In particular, 75% of prompts show less than 35 tokens. Prompt length variations stem from the need for additional details to clarify the request.

[2]We plan to make the extension available in the VS Code Marketplace.

B. Manual Evaluation

After submitting the NL prompts, we executed *PatchitPy* to detect and patch vulnerable implementation patterns in the outcomes. The assessment of the detection results needs manual inspection, involving the evaluation of each code produced in terms of True Positives (TPs), False Positives (FPs), True Negatives (TNs), and False Negatives (FNs). A TP case occurs when both the tool and manual analysis detect a vulnerability, and a TN case happens when both confirm the absence of vulnerabilities. An FP case arises when the tool detects a vulnerability unconfirmed by manual inspection, while an FN case is when the tool fails to detect a vulnerability confirmed by manual analysis.

As manual classification can be susceptible to errors, it was conducted by 3 human evaluators (i.e., 2 Ph.D. students and a post-doctoral researcher), all with a strong background and expertise in cyber security, Python programming, and AI code generators. To minimize the potential for human error, the 3 evaluators independently examined each generated code to check whether it was vulnerable (score of 1) or not (score of 0). Then, the evaluators compared their results and performed an in-depth analysis of the few discrepancy cases, representing about 3%. These discrepancies, attributed to human misclassification, were resolved with a complete alignment, resulting in a 100% consensus in the final evaluation. Furthermore, the evaluators assessed the patches using the same approach. They independently evaluated the generated patches by consulting OWASP's Cheat Sheet Series [52]–[56] and MITRE reports [43]. After comparing their analysis, they executed a static analysis tool (i.e., CodeQL) to ensure the absence of vulnerabilities, obtaining a 100% consensus in the final result. The diversity and expertise of evaluators, coupled with the iterative analysis approach, ensured the reliability of our human evaluation process.

Across the generated samples, GitHub Copilot produced 169 vulnerable instances out of 203 (i.e., 84%), while Claude and DeepSeek produced 126 (i.e., 62%) and 166 (i.e., 82%), respectively. Collectively, the 3 models generated vulnerable code in 76% of the 609 total samples. Furthermore, we analyzed the distribution of generated vulnerable code across different CWEs associated with the NL prompts in *SecurityEval* and *LLMSecEval*. In total, the prompts triggered the generation of code vulnerable to 63 distinct CWEs, with the highest frequency observed for CWE-502 (*Deserialization of Untrusted Data*), CWE-522 (*Insufficiently Protected Credentials*), CWE-434 (*Unrestricted Upload of File with Dangerous Type*), CWE-089 (*SQL Injection*), and CWE-200 (*Exposure of Sensitive Information to an Unauthorized Actor*).

C. Detection and Patching Results

We evaluated *PatchitPy* for vulnerability detection and patching, comparing it with a representative set of static

analysis tools and LLM-based approaches. To ensure a rigorous selection process, we followed criteria similar to those proposed by Li *et al.* [66]. We prioritized widely used, well-documented, free, and open-source tools that explicitly support Python for security scanning. Based on these criteria, we selected CodeQL [44], Semgrep [46], and Bandit [45]. We employed CodeQL version `v2.16.4` with the two `Security` test suites of queries [67], [68] for Python, Semgrep version `1.116.0` with the ready-to-use rules for Python available in the official Registry [69], and Bandit version `1.7.7`. To integrate LLM-based solutions, we followed the methodology of Nong *et al.* [70], employing ChatGPT-4o [47], Claude-3.7-Sonnet [41], and Gemini-2.0 [71]. ChatGPT-4o is an optimized version of GPT-4, developed by OpenAI for faster performance and improved handling of complex prompts. Claude-3.7-Sonnet, developed by Anthropic AI, sets new standards in various tasks [72]. Additionally, Google's Gemini offers advanced capabilities in multimodal tasks, focusing on language understanding for more versatile applications. To perform the analysis with LLMs, we used a Zero-Shot Role-Oriented (ZS-RO) prompt [26], i.e., we assigned to the model the role of a security expert (i.e., RO), successively asking to perform vulnerability detection (and subsequently remediation) for each generated code (NL prompt: *"Act as a security expert. I give you this Python code: [PYTHON_CODE]. Is this code vulnerable? Answer in only Yes or No. If it is vulnerable, patch the code."* [27], [73], [74]).

To evaluate detection, we applied key metrics employed in the field (i.e., Precision, Recall, F_1 Score, and Accuracy) by using the TPs, TNs, FPs, and FNs identified during manual evaluation (see § III-B). TABLE II summarizes the detection results across the total of 609 generated samples, showing that *PatchitPy* outperforms the baseline with a Precision of 97%, Accuracy of 89%, and F_1 Score of 93% (which balances Precision and Recall). Furthermore, *PatchitPy* correctly identified code vulnerable to 51 distinct CWEs for Copilot-generated code, 41 CWEs for Claude-generated code, and 47 CWEs for DeepSeek-generated code.

To evaluate patching, we analyzed the quality of the fix to ensure that changes did not introduce new vulnerabilities or code smells. Then, we analyzed cyclomatic complexity to assess how the patches influenced code complexity, as less complex code is more maintainable and easier to understand. After generating patches with *PatchitPy* and the baseline tools, a team of experts rigorously assessed their correctness to ensure the resulting code was safe (see § III-B). TABLE III presents the evaluation results, excluding CodeQL, Bandit and Semgrep. CodeQL does not offer patching features, while Bandit and Semgrep only provided patching suggestions via comments, without modifying the code. Specifically, Semgrep and Bandit suggested fixes for 19% and 17% of the detected vulnerabilities across the 609 generated code samples, respectively. Unlike these tools, *PatchitPy* applies patches directly, demonstrating its ability to actively improve code security, despite not being an AI-based static analyzer. Notably, *PatchitPy* outperformed all models by correctly applying patches relative

TABLE II
DETECTION RESULTS: PERFORMANCE METRICS FOR *PatchitPy* AND BASELINE. BEST RESULTS FOR EACH METRIC ARE SHOWN IN **BOLD**.

	Detection Solutions	Copilot	Claude	DeepSeek	All models
Precision	*PatchitPy*	**0.97**	**0.96**	**0.98**	**0.97**
	CodeQL	**0.97**	0.78	0.93	0.89
	Semgrep	0.96	0.81	0.92	0.89
	Bandit	0.92	0.85	0.90	0.89
	ChatGPT-4o	0.89	0.66	0.83	0.79
	Claude-3.7-Sonnet	0.91	0.63	0.90	0.81
	Gemini-2.0-Flash	0.87	0.64	0.84	0.78
Recall	*PatchitPy*	0.84	0.93	0.89	0.88
	CodeQL	0.71	0.66	0.68	0.68
	Semgrep	0.69	0.75	0.72	0.72
	Bandit	0.61	0.71	0.72	0.68
	ChatGPT-4o	0.83	**0.98**	**0.98**	0.93
	Claude-3.7-Sonnet	0.89	0.92	0.93	0.91
	Gemini-2.0-Flash	**0.90**	**0.98**	0.97	**0.95**
F_1 Score	*PatchitPy*	**0.90**	**0.94**	**0.93**	**0.93**
	CodeQL	0.82	0.72	0.79	0.77
	Semgrep	0.80	0.78	0.81	0.80
	Bandit	0.73	0.78	0.80	0.77
	ChatGPT-4o	0.86	0.79	0.90	0.85
	Claude-3.7-Sonnet	**0.90**	0.75	0.91	0.85
	Gemini-2.0-Flash	0.88	0.77	0.90	0.85
Accuracy	*PatchitPy*	**0.85**	**0.93**	**0.89**	**0.89**
	CodeQL	0.74	0.67	0.70	0.70
	Semgrep	0.71	0.73	0.72	0.72
	Bandit	0.63	0.74	0.71	0.69
	ChatGPT-4o	0.77	0.67	0.82	0.76
	Claude-3.7-Sonnet	0.84	0.62	0.85	0.77
	Gemini-2.0-Flash	0.80	0.64	0.82	0.76

TABLE III
PATCHING RESULTS: EVALUATING PATCH CORRECTNESS FOR *PatchitPy* AND BASELINE. BEST RESULTS ARE SHOWN IN **BOLD**.

	Patching Solutions	Copilot	Claude	DeepSeek	All models
Patched [Det.]	*PatchitPy*	**0.68**	**0.89**	**0.84**	**0.80**
	ChatGPT-4o	0.36	0.59	0.59	0.52
	Claude-3.7-Sonnet	0.52	0.78	0.67	0.65
	Gemini-2.0-Flash	0.39	0.49	0.58	0.49
Patched [Tot.]	*PatchitPy*	**0.57**	**0.83**	**0.74**	**0.70**
	ChatGPT-4o	0.30	0.58	0.58	0.48
	Claude-3.7-Sonnet	0.46	0.72	0.62	0.59
	Gemini-2.0-Flash	0.36	0.48	0.57	0.46

to both the detected (i.e., Patched [Det.]) and the total number of vulnerabilities for each set of code generated (i.e., Patched [Tot.]). For Copilot, it patched 68% of detected and 57% of total vulnerabilities. For Claude, the rates were 89% and 83%, and for DeepSeek, 84% and 74%, respectively.

To evaluate quality, we compared the generated patches with a ground truth. *LLMSecEval* provides secure code implementations for each NL prompt [75], while *SecurityEval* lacks them. For this reason, the same group of security experts employed for manual analysis (see § III-B) metic-

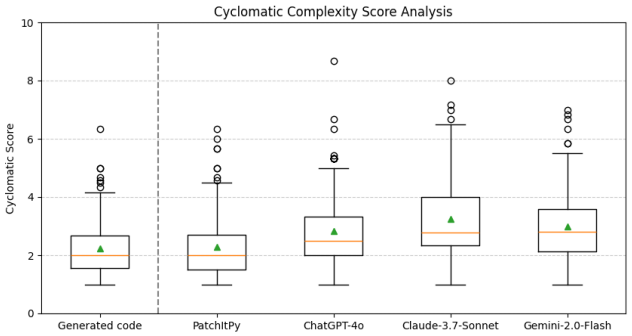

Fig. 3. Cyclomatic complexity distribution across generated code, *PatchitPy*, and AI-based models.

ulously collaborated to write safe samples for each prompt in *SecurityEval*, utilizing security guidelines [43], [52]–[56] and subsequentially verifying the security with CodeQL. We examined the patches using `Pylint` [76], a static code analyzer for Python that checks code quality by identifying errors and code smells and assigning a score based on these evaluations. The non-parametric *Wilcoxon rank sum* test on `Pylint` scores showed that the quality of patches generated by *PatchitPy* is statistically equivalent to the ground truth and to the patches produced by ChatGPT-4o, Claude-3.7, and Gemini-2.0, with all median score values $\sim 9/10$. This result highlights that *PatchitPy* provided patch quality comparable to advanced LLMs.

Finally, we analyzed cyclomatic complexity across the 609 generated code samples to assess the impact of patches applied by different tools [77]. As shown in Fig. 3, *PatchitPy* exhibits a complexity distribution (mean: 2.29, interquartile range - IQR: 1.21) closely aligned with the generated test set (mean: 2.4, IQR: 1.11), maintaining a lower median and IQR compared to the AI-based models. Conversely, ChatGPT-4o (mean: 2.84, IQR: 1.33), Claude-3.7 (mean: 3.26, IQR: 1.67), and Gemini-2.0 (mean: 2.99, IQR: 1.43) display a higher median complexity and a broader IQR, reflecting a greater tendency to modify the code structure. This increase is primarily due to function completions beyond the original signatures, introducing additional logic not present in the generated code. The *Wilcoxon rank sum* test confirms that *PatchitPy* does not introduce statistically significant complexity changes compared to the generated test set. In contrast, the AI-based models exhibit significant complexity increases. These results highlight that *PatchitPy* is a minimally intrusive solution, applying patches without unnecessary modifications and preserving code readability.

IV. Related Work

The impact of software vulnerabilities underscores the need to assist developers in quickly detecting and patching security issues [78]. Recent work proposed VS Code extensions to assess coding rule violations (e.g., *DevReplay* [79]), repair bugs (e.g., *pAPRika* [21], specific for JavaScript), and detect software vulnerabilities (e.g., *CodeQL* [44], *Semgrep* [46], *Bandit* [45], *AIBugHunter* [80], etc). CodeQL and Semgrep are popular, configurable, multi-language static analysis tools widely used for vulnerability analysis [81]–[85]. CodeQL analyzes source code by transforming it into a relational database via its Abstract Syntax Tree (AST) representation and uses a query-based approach for detection; however, its ruleset does not support code patching [67], [68]. Semgrep, which uses pattern matching with regular expressions to detect vulnerabilities, offers the fix feature [86] but demands users to write accurate regex for detection and patching, which may not always be feasible. Moreover, its public rulesets provide fixes via suggestion comments rather than code replacements [69]. Bandit is a Python-specific tool that builds the AST and applies detection plugins [87] to generate a final report with detection and patching suggestions provided only via comments. AIBugHunter [88], an AI-based tool specific for C/C++ language proposed by Fu *et al.* [80], uses Byte-Pair Encoding (BPE) tokenization and Transformer architecture for detecting and repairing vulnerabilities, generating patches through a Neural Machine Translation (NMT) model [89]. Despite their detection capabilities, these tools often overlook or partially implement vulnerability fixing.

Recent work explored the LLMs' performance in software vulnerability detection and patching [22], [90]. Zibaeirad *et al.* [19] introduced VulnLLMEval, a framework to evaluate 10 LLMs on a dataset of 307 real-world vulnerabilities in C code from the Linux kernel, finding that LLMs often failed to address vulnerabilities comprehensively. Fan *et al.* [22] showed that, despite the LLMs' positive impact on automated software repair, the hallucinations and scalability remain open problems. Fu *et al.* [23] and Khan *et al.* [24] used ChatGPT and GitHub Copilot, respectively, for vulnerability detection and remediation in C/C++, highlighting that LLMs often failed to generate correct repairs while improving their performance when using prompt-tuning techniques. However, generic users may lack the security expertise to refine models with vulnerability descriptions.

V. Conclusion

In this work, we introduced *PatchitPy*, an automated patching solution to fix security vulnerabilities in Python code using a lightweight pattern-matching approach. Designed to enhance the reliability of AI-assisted development, *PatchitPy* detects and patches Python vulnerabilities even in incomplete code thanks to its lightweight pattern-matching approach, filling the gap with current solutions. We evaluated the performance of *PatchitPy* on code generated by 3 different publicly accessible AI models (i.e., GitHub Copilot, Claude-3.7-Sonnet, and DeepSeek-V3) obtaining results that overcame baseline solutions in both detection and patching tasks. In particular, across the entire test set of 609 code samples, *PatchitPy* outperformed the baseline in both detection and patching, achieving higher F_1 Score (93%) and Accuracy (89%) values for detection, and obtaining a repair rate of 80% in patching the identified vulnerabilities. The patches show high quality with minimal complexity impact, ensuring code maintainability. This result

poses our solution as an efficient tool for developers to check code security, especially when using AI code generators. Future work aims to support other programming languages and extend the integration to other popular IDEs.

ACKNOWLEDGMENTS

This work has been partially supported by the MUR PRIN 2022 program, project *FLEGREA*, CUP E53D23007950001, 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, the *SERENA-IIoT* project, which has been funded by EU - NGEU, Mission 4 Component 1, CUP J53D23007090006, under the PRIN 2022 (MUR - Ministero dell'Università e della Ricerca) program (project code 2022CN4EBH). We are grateful to our former student Stefano Guarino for his help in the early stages of this work.

REFERENCES

[1] GitHub, "GitHub Copilot," https://github.com/features/copilot, 2023.

[2] ——, "Octoverse: The state of open source and rise of AI in 2023," https://github.blog/news-insights/research/the-state-of-open-source-and-ai/.

[3] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 754–768.

[4] M. L. Siddiq and J. C. 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*, 2022, pp. 29–33.

[5] GitHub/Blog, "GitHub Copilot now has a better AI model and new capabilities," https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/, 2023.

[6] S. Oh, K. Lee, S. Park, D. Kim, and H. Kim, "Poisoned chatgpt finds work for idle hands: Exploring developers' coding practices with insecure suggestions from poisoned ai models," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1141–1159.

[7] J. Li, Z. Li, H. Zhang, G. Li, Z. Jin, X. Hu, and X. Xia, "Poison attack and poison detection on deep source code processing models," *ACM Transactions on Software Engineering and Methodology*, 2023.

[8] A. Hussain, M. R. I. Rabin, and M. A. Alipour, "Trojanedcm: A repository for poisoned neural models of source code," *arXiv preprint arXiv:2311.14850*, 2023.

[9] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers," *arXiv preprint arXiv:2403.15600*, 2024.

[10] X. Wang, S. Wang, P. Feng, K. Sun, S. Jajodia, S. Benchaaboun, and F. Geck, "Patchrnn: A deep learning-based system for security patch identification," in *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE, 2021, pp. 595–600.

[11] R. Lin, Y. Fu, W. Yi, J. Yang, J. Cao, Z. Dong, F. Xie, and H. Li, "Vulnerabilities and security patches detection in oss: A survey," *ACM Computing Surveys*, vol. 57, no. 1, pp. 1–37, 2024.

[12] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 771–781.

[13] CVE Details Documentation, "Vulnerabilities by type & year," https://www.cvedetails.com/, 2024.

[14] Forbes, "10.5 Trillion Reasons Why We Need A United Response To Cyber Risk," https://www.forbes.com/councils/forbestechcouncil/2023/02/22/105-trillion-reasons-why-we-need-a-united-response-to-cyber-risk/.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[16] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," *arXiv preprint arXiv:2303.18184*, 2023.

[17] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, Dec. 2023. [Online]. Available: https://doi.org/10.1145/3631974

[18] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *2021 IEEE 32nd international symposium on software reliability engineering (ISSRE)*. IEEE, 2021, pp. 196–207.

[19] A. Zibaeirad and M. Vieira, "Vulnllmeval: A framework for evaluating large language models in software vulnerability detection and patching," *arXiv preprint arXiv:2409.10756*, 2024.

[20] F. Y. Assiri and J. M. Bieman, "An assessment of the quality of automated program operator repair," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 273–282.

[21] D. Campos, A. Restivo, H. S. Ferreira, and A. Ramos, "Automatic program repair as semantic suggestions: An empirical study," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 217–228.

[22] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.

[23] M. Fu, C. K. Tantithamthavorn, V. Nguyen, and T. Le, "Chatgpt for vulnerability detection, classification, and repair: How far are we?" in *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2023, pp. 632–636.

[24] A. Khan, G. Liu, and X. Gao, "Code vulnerability repair with large language model using context-aware prompt tuning," *arXiv preprint arXiv:2409.18395*, 2024.

[25] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[26] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. 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.

[27] 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)*. IEEE, 2023, pp. 112–119.

[28] M. Al-Hawawreh, A. Aljuhani, and Y. Jararweh, "Chatgpt for cybersecurity: practical applications, challenges, and future directions," *Cluster Computing*, pp. 1–16, 2023.

[29] A. Cheshkov, P. Zadorozhny, and R. Levichev, "Evaluation of chatgpt model for vulnerability detection," *arXiv preprint arXiv:2304.07232*, 2023.

[30] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. 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] Y. Li, S. Wang, and T. 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] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[33] GitHub, "Octoverse: AI leads Python to top language as the number of global developers surges," https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages.

[34] F. C. Bogaerts, N. Ivaki, and J. Fonseca, "A taxonomy for python vulnerabilities," *IEEE Open Journal of the Computer Society*, no. 01, pp. 1–12, 2024.

[35] D. Cotroneo, R. De Luca, and P. Liguori, "Devaic: A tool for security assessment of ai-generated code," *Information and Software Technology*, vol. 177, p. 107572, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584924001770

[36] Stack Overflow, "Stack Overflow Developer Survey 2018," https://survey.stackoverflow.co/2018?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#technology-_-most-popular-development-environments, 2018.

[37] ——, "Stack Overflow Developer Survey 2019," https://survey.stackoverflow.co/2019?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#technology-_-most-popular-development-environments, 2019.

[38] ——, "Stack Overflow Developer Survey 2021," https://survey.stackoverflow.co/2021/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#section-most-popular-technologies-integrated-development, 2021.

[39] ——, "Stack Overflow Developer Survey 2022," https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#section-most-popular-technologies-integrated-development, 2022.

[40] ——, "Stack Overflow Developer Survey 2023," https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022#section-most-popular-technologies-integrated-development, 2023.

[41] Anthropic, "Claude-3.7-Sonnet," https://www.anthropic.com/claude/sonnet, 2025.

[42] DeepSeek, "DeepSeek-V3," https://deepseekv3.org/, 2025.

[43] MITRE, "Common Weakness Enumeration (CWE)," https://cwe.mitre.org/, 2024.

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

[45] PyCQA, "Bandit," https://github.com/PyCQA/bandit/tree/main.

[46] returntocorp, "Semgrep," https://github.com/returntocorp/semgrep.

[47] OpenAI, "OpenAI ChatGPT," https://openai.com/blog/chatgpt/, 2024.

[48] Google, "Gemini AI," https://gemini.google.com/?hl=en.

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

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

[51] MITRE, "CWE VIEW: Weaknesses in OWASP Top Ten (2021)," https://cwe.mitre.org/data/definitions/1344.html.

[52] OWASP, "OWASP Cheat Sheet Series - Introduction," https://cheatsheetseries.owasp.org/index.html, 2024.

[53] ——, "Input Validation Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html, 2024.

[54] ——, "Injection Prevention Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Injection_Prevention_Cheat_Sheet.html, 2024.

[55] ——, "Cross Site Scripting Prevention Cheat Sheet," https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html, 2024.

[56] ——, "Symfony - OWASP Cheat Sheet Series," https://cheatsheetseries.owasp.org/cheatsheets/Symfony_Cheat_Sheet.html.

[57] CVE Details Documentation, "Vulnerabilities By Types/Categories," https://www.cvedetails.com/vulnerabilities-by-types.php.

[58] MITRE, "Generation of Error Message Containing Sensitive Information," https://cwe.mitre.org/data/definitions/209.html, 2024.

[59] Microsoft, "Extension Marketplace - Visual Studio Code," https://code.visualstudio.com/docs/editor/extension-marketplace.

[60] ——, "VS Code API - Visual Studio Code Extension API," https://code.visualstudio.com/api/references/vscode-api.

[61] ——, "Extension API - Visual Studio Code Extension API," https://code.visualstudio.com/api.

[62] Electron, "Debugging in VS Code - Electron," https://www.electronjs.org/docs/latest/tutorial/debugging-vscode.

[63] Microsoft, "Your First Extension - Visual Studio Code Extension API," https://code.visualstudio.com/api/get-started/your-first-extension.

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

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

[66] 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*, 2023, pp. 921–933.

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

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

[69] Semgrep, "Semgrep Registry," https://semgrep.dev/r.

[70] Y. Nong, H. Yang, L. Cheng, H. Hu, and H. Cai, "Appatch: Automated adaptive prompting large language models for real-world software vulnerability patching."

[71] G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut, J. Schalkwyk, A. M. Dai, A. Hauth, K. Millican *et al.*, "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

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

[73] A. Khare, S. Dutta, Z. Li, A. Solko-Breslin, R. Alur, and M. Naik, "Understanding the effectiveness of large language models in detecting security vulnerabilities," *arXiv preprint arXiv:2311.16169*, 2023.

[74] X. Zhou, S. Cao, X. Sun, and D. Lo, "Large language model for vulnerability detection and repair: Literature review and the road ahead," *ACM Transactions on Software Engineering and Methodology*, 2024.

[75] SoftSec Institute, "LLMSecEval Dataset - Secure Code Samples," https://github.com/tuhh-softsec/LLMSecEval/tree/main/Dataset/Secure%20Code%20Samples, 2023.

[76] Logilab and Pylint contributors, "Pylint," https://pylint.readthedocs.io/en/stable/.

[77] Radon 4.1.0 documentation, "Introduction to Code Metrics," https://radon.readthedocs.io/en/latest/intro.html.

[78] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garcerán, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2019.

[79] Y. Ueda, T. Ishio, and K. Matsumoto, "Devreplay: Linter that generates regular expressions for repeating code changes," *Science of Computer Programming*, vol. 223, p. 102857, 2022.

[80] M. Fu, C. Tantithamthavorn, T. Le, Y. Kume, V. Nguyen, D. Phung, and J. Grundy, "Aibughunter: A practical tool for predicting, classifying and repairing software vulnerabilities," *Empirical Software Engineering*, vol. 29, no. 1, p. 4, 2024.

[81] M. F. Gobbi and J. 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.

[82] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the security of github copilot's generated code-a targeted replication study," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 435–444.

[83] A. Kavian, M. M. Pourhashem Kallehbasti, S. Kazemi, E. Firouzi, and M. Ghafari, "Llm security guard for code," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 600–603.

[84] G. Bennett, T. Hall, E. Winter, and S. Counsell, "Semgrep*: Improving the limited performance of static application security testing (sast) tools," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, 2024, pp. 614–623.

[85] S. Gholami and Z. Amri, "Automated secure code review for web-applications," 2021.

[86] Semgrep, "Autofix," https://semgrep.dev/docs/writing-rules/autofix.

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

[88] AIBugHunter, "AIBugHunter," https://aibughunter.github.io/.

[89] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 935–947.

[90] J. Yang, H. Jin, R. Tang, X. Han, Q. Feng, H. Jiang, S. Zhong, B. Yin, and X. Hu, "Harnessing the power of llms in practice: A survey on chatgpt and beyond," *ACM Transactions on Knowledge Discovery from Data*, vol. 18, no. 6, pp. 1–32, 2024.