An Empirical Study of Vulnerabilities in Python Packages and Their Detection

Haowei Quan
Monash University
Melbourne, Australia
haowei.quan@monash.edu

Junjie Wang
College of Intelligence and Computing, Tianjin University
Tianjin, China
junjie.wang@tju.edu.cn

Xinzhe Li
College of Intelligence and Computing, Tianjin University
Tianjin, China
3022001754@tju.edu.cn

Terry Yue Zhuo
Monash University
Melbourne, Australia
terry.zhuo@monash.edu

Xiao Chen
University of Newcastle
Newcastle, Australia
xiao.chen@newcastle.edu.au

Xiaoning Du
Monash University
Melbourne, Australia
Xiaoning.Du@monash.edu

Abstract

In today's software development landscape, Python stands out for its simplicity, versatility, and extensive ecosystem. Python packages, as units of organization, reusability, and distribution, have become a pressing concern, highlighted by the considerable number of vulnerability reports. Existing benchmarks either do not target Python package-vulnerabilities or faces label accuracy issues stem from non-security-related changes within patching commits.

This paper addresses these gaps by introducing *PyVul*, the first comprehensive benchmark suite of Python-package vulnerabilities. *PyVul* includes 1,157 publicly reported, developer-verified vulnerabilities, annotated at both the commit level and function level. To enhance labeling quality, we propose *LLM-VDC*, a generic vulnerability benchmark cleansing method that leverages the code semantic understanding capability of LLMs. *LLM-VDC* improves *PyVul*'s function-level label accuracy by 2.0 fold and establishes *PyVul* the most precise automatically collected vulnerability benchmark. Based on *PyVul*, we conduct the first empirical study to unveil the characteristics of Python-package vulnerabilities and the limitations of state-of-the-art detection tools. Our empirical analysis reveals that current rule-based vulnerability detectors suffer from mismatches between their assumptions and real-world security scenarios, and limited support for high-order vulnerabilities, cross-language interactions, and Python's unique language features. On the other hand, ML-based detectors suffer from their inability to reach the necessary context. *PyVul* provides a solid foundation for advancing vulnerability research and tool development in this domain.

1 Introduction

Over recent years, Python has become the leading programming language due to its user-friendly syntax, versatility, and rich ecosystem [1]. With nearly 600,000 packages hosted on PyPI [2], Python's growing application across domains like web development and machine learning (ML) raises critical concerns about the security of its package ecosystem [3, 4]. For instance, web development often faces vulnerabilities such as cross-site request forgery (CSRF) and resource exhaustion, while ML packages are prone to issues like improper input validation. GitHub Advisory [5] reported 507 vulnerabilities in Python packages in 2023, highlighting its growing security importance, comparable to npm (394) and Maven (937). These vulnerabilities are particularly critical because Python packages play a central role in the modern software supply chain. Vulnerabilities in widely used packages can be transitively inherited and propagate to a large number of downstream applications without direct awareness from end developers, significantly amplifying their security impact.

Despite this, no benchmark comprehensively captures real-world Python package vulnerabilities with high accuracy. A better understanding of real-world vulnerabilities is essential for improving their detection. By systematically characterizing which types of vulnerabilities are most prevalent in Python packages, which vulnerabilities have the most severe downstream impact, and how these vulnerabilities arise and are exploited in practice, we can derive actionable insights to guide which features and analysis capabilities should be prioritized in vulnerability detection tools.

Current vulnerability benchmarks, composed of vulnerabilities at either the commit or function level, either do not derive from or are difficult to associate with Python packages. For instance, CVE-Fixes [6] and CrossVul [7] are collected based on code repositories from security platforms such as National Vulnerability Database (NVD) [8] and do not effectively map to Python packages. In addition, datasets such as VUDENC [9] and SVEN [10] focus on Python code changes, consequently overlooking cross-language vulnerabilities. This motivates us to collect the first benchmark of real-world vulnerabilities in Python packages.

Additionally, concerns regarding the quality of existing benchmarks have been raised [11, 12]. In vulnerability-fixing commits, modified functions are often labeled as vulnerable, even when changes address non-vulnerability-related objectives like refactoring. This leads to inaccuracies in vulnerability assessments. Recent studies have attempted to address this challenge. Wang et al. [13] combined LLMs and static vulnerability detectors to determine vulnerable samples; however, as static vulnerability detectors are used for validating samples, the resulting dataset can no longer serve as a benchmark for them. As revealed in our empirical assessment, current state-of-the-art static vulnerability detectors for Python suffer from low accuracy and excessive number of warnings, underscoring both their need for an effective benchmark and the inadequacy of relying on these detectors as validation mechanisms. Ding et al. [14] proposed a heuristic-based labeling approach that significantly improves label accuracy but incurs substantial data loss and restricts the dataset to single-function vulnerabilities. To overcome these issues, we make the first attempt to refine and cleanse the dataset using LLMs, with manual verification to ensure accuracy.

In this study, we present *PyVul*, **the first large-scale, high-quality vulnerability benchmark suite for Python packages**, and we evaluate **how effectively current vulnerability detectors identify these vulnerabilities**. *PyVul* consists of 1,157 verified vulnerabilities across 151 Common Weakness Enumeration (CWE) categories, identified in Python packages and refined through our LLM-based cleansing method, LLM-VDC. To cater to the needs of vulnerability detectors operating at different granularities, we prepare our benchmark at both the commit level and function level. To enhance label accuracy of our benchmark, we developed and applied an LLM-assisted data cleansing method, *LLM-VDC*. After cleansing, our benchmark, PyVul, achieves an accuracy rate of 100% at the commit level with 1,157 repository snapshots, and an accuracy rate of 94.0% at the function level with 2,082 vulnerable functions, as validated through random sampling. This makes *PyVul* 82.5% to 92.8% more accurate than previous automatically collected function-level datasets [6, 7] and comparable to the human-annotated small dataset, SVEN [10]. LLM-VDC demonstrates superior universality and outperforms the state-of-the-art labeling method introduced by Ding et al. [14] in a multi-lingual vulnerability dataset, with a 33.1% greater improvement in function-level label accuracy. We evaluate the effectiveness of our LLM-assisted data cleansing method and the quality of *PyVul* in RQ1.

Furthermore, leveraging *PyVul*, we assess state-of-the-art rule-based and ML-based static vulnerability detectors (RQ2 and RQ3). We leave the evaluation of dynamic vulnerability detectors to future work considering the common reproducibility issues in the open-source repositories [15]. The results of our evaluation show a significant gap in the ability of current Python static vulnerability detection tools to effectively report real-world security issues. In addition to this assessment, we empirically review six most frequently reported CWEs in Python packages, aiming to provide insights into the limitations of current static tools and fuel future tools to detect zero-day vulnerabilities. Our empirical study reveals substantial discrepancies between the assumptions of current rule-based detectors and real-world security scenarios, compounded by a lack of support for most prevalent types of vulnerabilities such as those high-order vulnerabilities embedded in web applications, and a lack of support for Python's language features such as dynamic typing. On the other hand, current ML-based detectors suffer from their unrealistic training data and function-level settings. Taking functions as input may result in models observing great variance in vulnerable samples or missing important context.

To summarize, our work makes the following contributions:

- The first Python package vulnerability benchmark, *PyVul*, containing 1,157 commit-level and 2,082 function-level vulnerabilities. It demonstrates an accuracy that is 82.5% to 92.8% higher than that of existing automatically collected function-level vulnerability datasets, and it also excels in benchmark size.
- An LLM-assisted approach, *LLM-VDC*, for cleansing function-level vulnerability datasets, which demonstrates a 2.0 fold improvement in function-level label accuracy and enhances commit-level label accuracy to 100%.
- A thorough evaluation of how well existing rule-based and ML-based detectors can identify vulnerabilities in *PyVul*, accompanied by an in-depth diagnosis of their major performance shortcomings for both approaches.
- Our benchmark, code, experimental scripts, and supplementary material are made openly accessible.
- Our benchmark, code, experimental scripts, and supplementary material are made openly accessible at https://github.com/wqmqdjd/PyVul.

2 Benchmark Construction

In this section, we elaborate on the three steps used to establish a large and high-quality benchmark for Python package vulnerabilities, *PyVul*. These steps include data collection, benchmark curation, and data cleansing.

2.1 Data Collection

We collect Python package vulnerabilities from three vulnerability reporting platforms that detail the ecosystems from which the vulnerabilities originate: GitHub Advisories [5], Snyk [16], and Huntr [17]. These platforms are widely used by developers and serve as data sources for other empirical research on vulnerabilities [18–21]. Our data collection covers vulnerability reports published up to Jan 16, 2024. In total, we gathered 2,379 vulnerability reports from GitHub Advisories, 930 from Snyk, and 321 from Huntr, totaling 3,630 reports. We conduct an initial screening, retaining those that meet the following criteria: 1) they include fix commits that address the corresponding vulnerabilities; 2) the repositories are accessible at the time of data collection, and the commits have not been rolled back or deleted; and 3) they are not duplicates of any other reports, as identified by fix commits. Ultimately, our initial collection resulted in 1,767 unique vulnerability reports.

2.2 Benchmark Curation

A comprehensive and accurate benchmark for vulnerabilities is essential for evaluating vulnerability detectors [21]. Currently, there are two main categories of static vulnerability detection methods applicable to Python packages: rule-based and ML-based methods. These methods differ significantly in terms of the context granularity they use for identification. Rule-based static analysis methods, such as CodeQL [22], PySA [23], and Bandit [24], typically operate at the project level. In contrast, ML-based static analysis methods [12, 14, 25] generally work at the function level. To cater the needs of both types of analysis, we have constructed a benchmark that accommodates vulnerabilities at both the commit and function levels.

Commit-level Benchmark. To construct the commit-level benchmark, we checkout the 1,767 collected commits as patched, non-vulnerability samples and their direct parent version on the main branch as vulnerability samples.

Function-level Benchmark. To construct the function-level vulnerability dataset, we employed a common methodology utilized in previous studies [12, 14, 25]. For each commit, we consider the functions involved as vulnerability samples in their prefix version and as non-vulnerability samples in their post-fix version. From a total of 1,767 commits, we collected 8,374 vulnerability samples and 8,374 non-vulnerability samples, resulting in a comprehensive dataset of 16,748 samples.

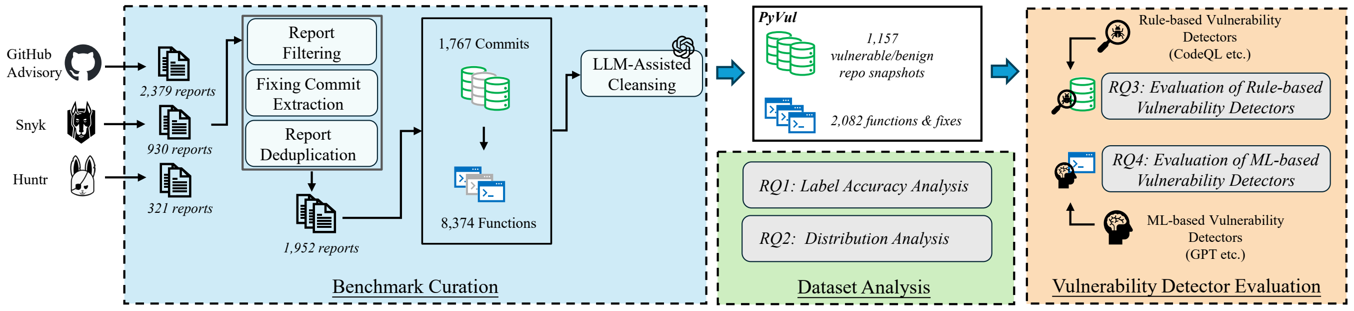

GitHub Advisory
Snyk
Huntr
2,379 reports
930 reports
321 reports
Report Filtering
Fixing Commit Extraction
Report Deduplication
1,952 reports
1,767 Commits
8,374 Functions
LLM-Assisted Cleansing
Benchmark Curation
PyVul
1,157 vulnerable/benign repo snapshots
2,082 functions & fixes
RQ1: Label Accuracy Analysis
RQ2: Distribution Analysis
Dataset Analysis
Rule-based Vulnerability Detectors (CodeQL etc.)
RQ3: Evaluation of Rule-based Vulnerability Detectors
RQ4: Evaluation of ML-based Vulnerability Detectors
ML-based Vulnerability Detectors (GPT etc.)
Vulnerability Detector Evaluation

Figure 1: The overview of our study

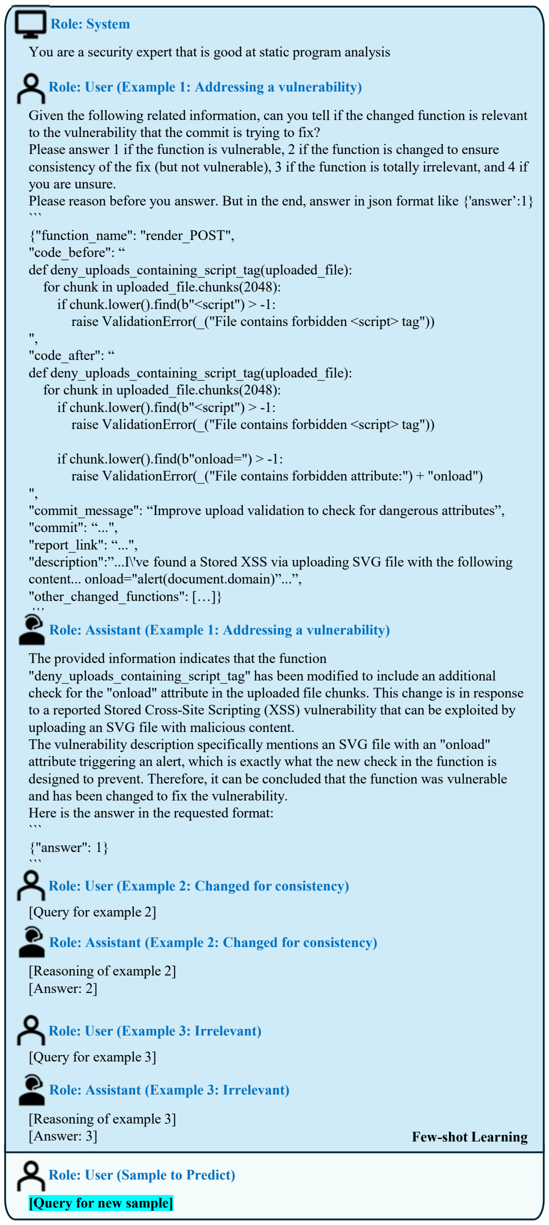
2.3 Data Cleansing

To evaluate the quality of the curated benchmark and compare it with baseline benchmarks, we manually validated a statistically significant number of randomly sampled vulnerable commits and vulnerable functions from *PyVul*, *CVEFixes* [6], and *CrossVul* [7] in Section 4.1. The sample sizes were determined following [26], and the results are summarized in Table 1. The accuracy of commit-level labels significantly surpasses that of function-level labels across all three benchmarks, achieving rates of 99.7%, 99.5%, and 99.4%, respectively. On the other hand, the function-level label accuracy stands at only 40.5%, 48.3%, and 51.2% for the three benchmarks. The low quality arises mainly because numerous changes in these commits do not pertain directly to vulnerabilities; instead, they involve code refactoring, consistent code style maintenance, or improvements in readability, which are also highlighted by other researchers [14]. For developers with security background, pinpointing the actual vulnerability-fixing changes within a commit is not notably difficult. However, manually annotating all samples within the benchmark is labor-intensive and not scalable. As an alternative, we propose an approach, *LLM-VDC*, that leverages the code semantic understanding capabilities of LLMs to help filter out function-level changes that are unrelated to vulnerability fixes. This approach additionally improves the commit-level label accuracy, as we only retain the commits with at least one relevant function change.

We utilize the in-context learning capabilities of LLMs due to the insufficient fine-tuning data available for our annotation task. We implement established practices of prompt engineering when presenting the task to LLMs. These practices include system role definition [27], few-shot learning [28], and chain-of-thought (CoT) prompting [29], all of which have proven effective in recent studies. The system role defines how LLMs should function during interactions, influencing the tone, focus, and limitations of their responses. In this case, we designate the system role as a security expert. Few-shot learning enables LLMs to grasp and perform specific tasks through illustrative examples, while the CoT prompting technique enhances these examples by outlining the reasoning process behind the answers. In our design, we specifically ask LLMs to state the reasons before yielding the final answer.

We now elaborate on the task formulation, emphasizing the clarity and inclusiveness of classification categories, the adequacy of context information, and the adaptive truncation of context to accommodate the token limitations of LLMs.

- **Adequacy of Context.** The task involves determining whether the changes made to a function are intended to address security



Role: System

You are a security expert that is good at static program analysis

Role: User (Example 1: Addressing a vulnerability)

Given the following related information, can you tell if the changed function is relevant to the vulnerability that the commit is trying to fix?
Please answer 1 if the function is vulnerable, 2 if the function is changed to ensure consistency of the fix (but not vulnerable), 3 if the function is totally irrelevant, and 4 if you are unsure.
Please reason before you answer. But in the end, answer in json format like {'answer':1}

```
{"function_name": "render_POST",
"code_before": "
def deny_uploads_containing_script_tag(uploaded_file):
    for chunk in uploaded_file.chunks(2048):
        if chunk.lower().find(b"<script") > -1:
            raise ValidationError(_("File contains forbidden <script> tag"))
",
"code_after": "
def deny_uploads_containing_script_tag(uploaded_file):
    for chunk in uploaded_file.chunks(2048):
        if chunk.lower().find(b"<script") > -1:
            raise ValidationError(_("File contains forbidden <script> tag"))

        if chunk.lower().find(b"onload=") > -1:
            raise ValidationError(_("File contains forbidden attribute:") + "onload")
",
"commit_message": "Improve upload validation to check for dangerous attributes",
"commit": "...",
"report_link": "...",
"description":"...I\'ve found a Stored XSS via uploading SVG file with the following content... onload="alert(document.domain)"...",
"other_changed_functions": […]}
```

Role: Assistant (Example 1: Addressing a vulnerability)

The provided information indicates that the function "deny_uploads_containing_script_tag" has been modified to include an additional check for the "onload" attribute in the uploaded file chunks. This change is in response to a reported Stored Cross-Site Scripting (XSS) vulnerability that can be exploited by uploading an SVG file with malicious content.
The vulnerability description specifically mentions an SVG file with an "onload" attribute triggering an alert, which is exactly what the new check in the function is designed to prevent. Therefore, it can be concluded that the function was vulnerable and has been changed to fix the vulnerability.
Here is the answer in the requested format:

```
{"answer": 1}
```

Role: User (Example 2: Changed for consistency)

[Query for example 2]

Role: Assistant (Example 2: Changed for consistency)

[Reasoning of example 2]
[Answer: 2]

Role: User (Example 3: Irrelevant)

[Query for example 3]

Role: Assistant (Example 3: Irrelevant)

[Reasoning of example 3]
[Answer: 3]

Few-shot Learning

Role: User (Sample to Predict)

[Query for new sample]

Figure 2: The prompt used for annotating the relevance of function-level changes to vulnerability fixes.

issues identified in the associated commit. To assist in this assessment, we provide two main pieces of information: 1) Details about the focal function, including its name and complete function definition both before and after the changes, and 2) Contextual information regarding the vulnerability being addressed, which includes the commit message, a link to the advisory report, a description of the vulnerability from the report, and a list of all other functions that were modified in this commit.
- **Classification Categories.** To help LLMs better understand our task, we explicitly ask the LLMs to classify each change into one of four categories: 1) the function is patched against a vulnerability; 2) the function is not vulnerable but has been changed for consistency; 3) the function is irrelevant to the vulnerability; or 4) no decision can be made. **We provide LLMs one example for each of the first three categories through few-shot learning, accompanied by detailed reasoning steps** elucidating why the example aligns with the respective category. This method not only aids LLMs in accurately understanding the category definitions but also in adopting the intended reasoning processes. It is important to note that LLMs have the option to indicate when they cannot reach a clear conclusion. This feature helps prevent the model from generating hallucinations by avoiding incorrect answers when it is uncertain.
- **Adapt to Token Limitations.** Due to the limited context length of LLMs, we implement strategies to sacrifice part of the context information when the limit is exceeded. Specifically, we employ the following measures: 1) Commit messages or descriptions from advisory reports will be truncated with a note stating "collapsed due to token limitation" if they exceed a certain threshold (e.g., 2,000 characters, as used in our experiment), and 2) We adopt a stepwise reduction method to supply information about other changed functions in the commit. Depending on whether the context limitation of the prompt has been exceeded, we will attempt the following methods from ❶ to ❸ and only choose the latter option if all former options are infeasible: ❶ We supply all other changed function information in the commit; ❷ We supply all other changed functions in the same file; ❸ We do not supply other changed functions.

The prompt used to annotate the relevance of each modified function with the vulnerability-fixing commit is shown in Figure 2. In this study, we adopt GPT-4 [30], one of the top-performing LLMs available at the time of writing [31]. Our cleansing method results in a collection of 1,157 commits and 2,082 function pairs.

3 Study Design

In this study, we aim to investigate the characteristics of vulnerabilities in Python packages and assess the performance of current static vulnerability detection methods on these vulnerabilities. We begin by evaluating the label accuracy of *PyVul* and use *PyVul* as a foundation for our analysis in the following sections. The evaluation focuses on addressing the following research questions (RQs):
- RQ1: How accurate are the vulnerability labels in *PyVul*?
- RQ2: How effective are current rule-based approaches for detecting vulnerabilities in *PyVul*?
- RQ3: How effective are current ML-based approaches for detecting vulnerabilities in *PyVul*?

Next, we will introduce the subjects used for comparison and evaluation when addressing these RQs.

3.1 Existing Python Vulnerability Benchmarks

To the best of our knowledge, there is currently no dataset specifically focused on vulnerabilities in Python packages. However, several datasets have been curated that concentrate on vulnerabilities within Python code. In this study, we utilize the label accuracy of these existing datasets to benchmark the label accuracy of *PyVul*. We select the subject datasets based on two criteria: 1) the dataset must contain vulnerable Python code; and 2) the vulnerabilities in the dataset should either be manually verified or linked to corresponding advisory reports or Common Vulnerabilities and Exposures (CVE) entries, ensuring the dataset's high quality.
- *SVEN* [10]. SVEN is a manually annotated vulnerability dataset that contains 808 pairs of vulnerable and non-vulnerable functions across various programming languages. Among these, 380 pairs refer specifically to Python functions. The original SVEN dataset does not include commit-level vulnerabilities. As such, we supplement the SVEN dataset with relevant fix commits for each function pairs, resulting in a total of 143 commit-level vulnerability and non-vulnerability samples.
- *CVEFixes [6] and CrossVul [7].* Both CVEFixes and CrossVul are multi-language datasets derived from the CVE database, which include the fixing commits for various vulnerabilities. CVEFixes is annotated at the function level, whereas CrossVul provides annotations at the file level. Previous research [12] has successfully extracted vulnerable and benign functions from CrossVul. We employ the approach to obtain the function-level data. As a result, we have 1,360 pairs of Python functions related to 508 commits in CVEFixes, and 777 pairs of Python functions related to 319 commits in CrossVul.

3.2 Vulnerability Dataset Cleansing Methods

The development of effective cleaning approaches and highly accurate techniques for annotating vulnerable functions has been limited. To our knowledge, the most advanced automated labeling method is presented by Ding et al. [14], who designed heuristic rules to retain functions only when their likelihood of being vulnerable is high. These rules prioritize precision over recall. Specifically, a function's prior version is labeled as vulnerable only if: 1) it is the sole function modified in a vulnerability-fixing commit; 2) its name appears in the linked CVE description; 3) its file name appears in the CVE description, and it is the only function changed in that file. Using this approach, PrimeVul achieves a high accuracy rate of around 90%. However, its strict rules may lead to a substantial loss of vulnerable samples, as many vulnerabilities are not limited to a single function. Additionally, CVE descriptions can be incomplete or may not always specify particular functions or file names. In contrast, our data cleaning method, *LLM-VDC*, is more adaptable and can accurately identify genuine vulnerable functions across a wide range of commits with LLM assistance. In the following, we will use the name *PrimeVul* to also refer to this rule-based method for cleansing vulnerability datasets.

3.3 Static Vulnerability Detectors

Below, we introduce the three rule-based approaches and three machine learning-based approaches used to analyze their effectiveness in detecting vulnerabilities in *PyVul*.

3.3.1 Rule-based Approaches. Our selection criteria are as follows: the methods must support vulnerability detection for Python, be executable, and be widely recognized in the field of vulnerability detection [32–34], reflecting the highest standard in this task.

- *CodeQL* [22]. CodeQL is a comprehensive static analysis engine developed by GitHub that uses queries to identify vulnerable patterns in code. It converts the source code of a program into a queryable database that maintains the program's semantics, such as data and control flows. CodeQL supports multiple programming languages, including Python. For Python, there are 101 built-in queries [35], including an extended set focused on security. Each query is annotated with the CWEs. In total, 123 CWEs are covered by these queries.
- *PySA* [23]. PySA is a static taint analysis tool developed as part of Facebook's Pyre-check project. PySA identifies and flags these vulnerabilities by analyzing the code to trace data flows from untrusted input sources to potentially vulnerable sinks. The tool features 38 clearly defined taint analysis rules that describe the characteristics of the sources and sinks , mapping to 67 different CWEs.
- *Bandit* [24]. Bandit is a popular open-source static analysis tool (with 7.6k stars on GitHub) designed specifically to identify security issues in Python code. Bandit focuses on detecting vulnerabilities primarily through pattern matching against Abstract Syntax Trees (ASTs) and does not take into account control flows or data flows within the code. It encompasses 39 rules, mapping to 17 distinct CWEs.

3.3.2 ML-based Approaches. The state-of-the-art ML-based approaches for Python vulnerability detection primarily include GNN-based and LLM-based methods. GNN is a widely used model architecture for vulnerability detection. To the best of our knowledge, the state-of-the-art GNN method trained to detect Python vulnerabilities is VUDENC [9]. However, due to unresolved bugs in its published implementation [36] and our unsuccessful attempts to contact the authors, we were unable to replicate VUDENC and therefore had to exclude it from our study.

LLMs pretrained on code have significant potential for language agnostic vulnerability detection, either through direct prompting [14] or fine-tuning [12, 14]. Research indicates that models from the GPT-2 family can achieve performance comparable to GNN-based methods when fine-tuned on small datasets, such as CVEFixes [6], and demonstrate superior performance on larger datasets [12]. However, due to a lack of specifically curated Python datasets, most existing studies have focused on other programming languages, particularly C/C++. In this paper, using *PyVul*, we conduct the first study of LLM-based vulnerability detection in Python packages. In RQ3, we will evaluate the performance of LLMs employing two distinct methods: 1) direct prompting, which assesses the LLM's inherent knowledge of vulnerabilities and its ability to identify vulnerable code patterns, and 2) fine-tuning, which examines the models' capacity to learn from vulnerability samples and adapt to the task of vulnerability detection. For our experiments, we selected three representative LLMs, including one open-source model and two proprietary models from OpenAI. The selection was guided by the goal of covering commonly used LLM categories in practice, rather than performing an exhaustive comparison across all available LLMs.

- *CodeQwen1.5-Chat* [37]. CodeQwen1.5-Chat is one of the latest code LLMs from the open-source community, featuring 7B parameters. This model has been pretrained on approximately 3 trillion tokens of code-related data. We have selected CodeQwen1.5-Chat, the instruction-tuned version of CodeQwen1.5, to evaluate its performance in both direct prompting and fine-tuning scenarios. Despite having only 7B parameters, CodeQwen1.5-Chat has demonstrated state-of-the-art performance on the HumanEval benchmark [31], outperforming GPT-3.5 Turbo and showing performance comparable to GPT-4.
- *GPT-3.5 Turbo* and *GPT-4* [30]. The GPT family of models demonstrates exceptional performance among LLMs and shows superior capabilities compared to open-source models in vulnerability detection [14]. As of this study, GPT-3.5 Turbo is the highest-performing proprietary model available for fine-tuning, while GPT-4 ranks as the top-performing proprietary model on the HumanEval leaderboard.

3.3.3 Experimental Setup. When evaluating the rule-based approaches, we used their default settings. Due to time constraints, we set a timeout of 60 minutes for each run of the detectors. All evaluations of the rule-based approaches were conducted on a computer equipped with a 14-core Intel Xeon W-2175 CPU and 32 GB of RAM, running Ubuntu 20.04.6.

We follow existing research [14] to setup the fine-tuning framework for CodeQwen1.5-Chat, with Axolotl [38]. We load the model's weights from Hugging Face Models [39] and fine-tune it with a learning rate of 2×10^{-5} for four epochs using LoRA [40], which balances between the training efficiency and task performance [41, 42]. To directly prompt CodeQwen1.5-Chat, we utilize the default parameters, and all experiments are conducted on an NVIDIA A100 GPU with 80 GB of memory. For the proprietary models, we interact with them through the OpenAI APIs. We fine-tune GPT-3.5 Turbo for just one epoch following [14]. Since fine-tuning for GPT-4 was not available during this study, our evaluation does not include this model. During the model inference, we use the models' default parameters while setting the temperature parameter to 0 for deterministic results.

4 Experiment Results

4.1 RQ1: Benchmark Quality

4.1.1 Effectiveness of LLM-VDC. To assess the effectiveness of our LLM-assisted method for cleansing vulnerability datasets, we evaluate the label accuracies of *PyVul* before any data cleansing, after applying *PrimeVul*, and after applying *LLM-VDC*, respectively.

Validating the raw data, which includes 1,767 commits and 8,374 functions, is not feasible to do manually. Therefore, we evaluate the data by randomly sampling commits and vulnerable functions from each version of the dataset and manually checking the accuracy of their labels. The sample sizes n_{sample} are computed by the formula [26] for statistical significance, $n_{sample} = \frac{z^2 \times p(1-p)/e^2}{1+z^2 \times p(1-p)/e^2 N}$, where N is the population size, z is the z-score, p is the standard deviation, and e is the margin of error. We adopt a z-score of 1.96

Table 1: Label accuracy of existing benchmarks and our newly curated benchmark, and the impact of data cleansing method on label accuracy. 95% confidence intervals are computed using the Wilson score method [43].

Benchmark	Data Cleansing	#Commits	#Sampled Commits	Commit-lvl Acc (%)	Commit-lvl 95% CI (%)	#Vuln Funcs	#Sampled Funcs	Function-lvl Acc (%)	Function-lvl 95% CI (%)
	N/A	1,767	316	99.7	[98.2, 99.9]	8,374	368	40.5	[35.6, 45.6]
PyVul	PrimeVul	745	254	100.0	[98.5, 100.0]	1,012	279	70.6	[65.0, 75.7]
	LLM-VDC	**1,157**	289	**100.0**	[98.7, 100.0]	**2,082**	325	**94.2**	[91.1, 96.2]
SVEN[10]	N/A	143	105	100.0	[96.5, 100.0]	380	192	96.3	[92.7, 98.2]
CVEFixes[6]	N/A	508	219	99.5	[97.5, 99.9]	1,360	300	48.3	[42.7, 54.0]
CrossVul[7]	N/A	319	175	99.4	[96.8, 99.9]	777	258	51.2	[45.1, 57.2]
PyVul (Python)	**LLM-VDC**	**788**	259	**100.0**	[98.5, 100.0]	**1,480**	306	**93.1**	[89.7, 95.5]

corresponding to 95% confidence level, a standard deviation of 0.5, which is the maximum standard deviation as the exact distribution unknown, and a margin of error of 5%. Columns four and seven in Table 1 lists the sample sizes. The first and the third authors, both with expertise in general software vulnerabilities and over four years of experience in Python development, independently assess whether each commit or function is genuinely related to the reported vulnerability. Their evaluation is based on a thorough review of the fix commit, the code before and after the commit, the related vulnerability report, the CVE descriptions, and any available discussions among developers. We quantify inter-rater agreement on the initial independent assessments using Cohen's Kappa [44], which is 0.718 and within the range of fair to good. In cases where the authors disagree, they collaboratively review the sample and discuss it until they reach a consensus. For our assessment criteria, a commit is considered erroneously labeled if: 1) it is irrelevant to the associated vulnerability, or 2) it does not fully resolve the vulnerability. A function is deemed mistakenly labeled as vulnerable if: 1) it has been modified for consistency rather than directly addressing the vulnerability, or 2) it is entirely irrelevant to the vulnerability. We calculate label accuracy as the percentage of correctly labeled samples out of all evaluated samples. The results of this analysis are presented in Table 1. To provide accurate coverage for binomial proportions in boundary cases such as near-perfect detection rates, we additionally report 95% confidence intervals (CIs) using the Wilson score method [43].

When no data cleansing method is applied, the collected dataset shows a label accuracy of 99.7% at the commit level. Both *PrimeVul* and *LLM-VDC* successfully increase the commit-level label accuracy to 100.0%. Note that these accuracies are calculated from samples; we therefore include Wilson intervals to make the uncertainty explicit. This improvement is also reflected in the Wilson 95% confidence intervals, which become tighter and shift upward with increased lower bounds, indicating improved label accuracy under sampling uncertainty. At the function level, when no data cleansing is applied, the label accuracy of the collected raw data is 40.5%. After applying the *PrimeVul* method, the label accuracy increases to 70.8%. However, even the upper bound of the 95% CI is significantly lower than the accuracy of around 90% reported in their own datasets [14]. This indicates that *PrimeVul* shows limited effectiveness in our *PyVul*. The primary reason for its underperformance is presumably the inclusion of diverse programming languages, particularly the obfuscated JavaScript code. *PrimeVul* relies on searching for function names in the CVE descriptions, but changes in obfuscated JavaScript code often involve short, non-descriptive function names, such as a single letter "p". These functions are likely to be inaccurately associated with vulnerability fixes. This highlights *PrimeVul*'s limitations in generalization ability. In contrast, the *LLM-VDC* method significantly enhances the label accuracy of *PyVul* to 94.2%.

Additionally, *LLM-VDC* retains a significantly larger number of samples compared to *PrimeVul*. Initially, the benchmark shows 1,767 commits and 8,374 vulnerable functions before any data cleansing. After applying *PrimeVul*, these numbers decrease to 745 commits and 1,012 functions. In contrast, the *LLM-VDC* method retains 1,157 commits and 2,082 functions. Remarkably, at the function level, *LLM-VDC* retains twice as many samples as *PrimeVul*, highlighting its superior effectiveness in data preservation.

LLM-VDC significantly enhances the function-level label accuracy of *PyVul* to 94.2%. Compared to the baseline cleansing method, *LLM-VDC* not only achieves a greater improvement in label accuracy but also retains twice as many samples.

4.1.2 *Label Accuracy of PyVul.*

Since both *PyVul* and the baseline benchmarks include code written in Python and other programming languages, we extracted and evaluated only the segments pertinent to Python code to ensure a fair comparison. We refer to this subset of our *PyVul* benchmark as *PyVul (Python)*. In this section, we randomly sample commits and vulnerable functions from each benchmark and follow the same procedure for manually validating their label accuracy as described in Section 4.1.1.

We present the results in Table 1. As indicated, all four benchmarks demonstrate strong performance in commit-level labeling accuracy, achieving 99.4% or higher. At the function level, we observe that the label accuracy of the automatically collected vulnerability datasets, CVEFixes and CrossVul, is 48.3% and 51.2%, respectively. In contrast, the manually curated vulnerability function dataset, SVEN, boasts a label accuracy of as high as 96.3%. However, this dataset is limited in size due to the high cost of manual annotation, containing only 380 vulnerable functions. On the other hand, the vulnerability function dataset we collected automatically has a label accuracy of 93.1%. This accuracy is between 82.5% and 92.8% higher than that of the other two automated baseline benchmarks and is comparable to the manually curated dataset SVEN. This advantage is also supported by Wilson intervals: PyVul (Python) attains 93.1% with a narrow 95% CI [89.7%, 95.5%], while the automatically collected baselines remain around 50% with substantially lower intervals. With the advanced data cleansing method, opportunities arise to establish a high-quality vulnerability benchmark on a large scale.

All the Python vulnerability benchmarks examined demonstrate high accuracy at the commit level. However, the accuracy at the function level varies significantly. *PyVul (Python)* achieves a label accuracy of 93.1%, which is between 82.5% and 92.8% higher than the accuracies of the automatically collected baseline datasets. Additionally, this accuracy is comparable to that of the manually curated dataset SVEN.

4.2 RQ2: Evaluation of Rule-based Detectors

With a more accurately annotated vulnerability benchmark, we can perform a more precise evaluation of existing vulnerability detection methods. Since rule-based and ML-based vulnerability detection methods typically operate at different levels (commit or function), we evaluate them separately. For rule-based vulnerability detection methods, we use the commit-level vulnerability dataset for evaluation.

Due to the limitations of rule-based static vulnerability detection methods, such as CodeQL, PySA, and Bandit, which can only detect vulnerabilities for a specific programming language at a time, we use samples in *PyVul* that involve only the Python language, accounting for 68.1% (788/1,157) of the Commit vulnerability dataset. In this *PyVul (Python)* subset, we selected six vulnerability types with the highest occurrence frequency, totaling 244 commit-level vulnerability samples, to assess the efficiency of rule-based detectors, as shown in the second column of Table 2.

We apply the detectors to scan the vulnerability samples from CWEs that they target and report the number of complete runs in the third column. Among the three detectors, PySA exhibits a notable number of failed runs. Out of 244 scans, only 54 of them finish within a one-hour time window without any interrupting runtime errors. On average, the completion rate of the three detectors is 62.2%.

The detection results of rule-based static analysis methods are difficult to verify automatically. Even when the detection results identify the target vulnerability, the reported locations of the vulnerable code can vary from the patch, especially for taint-style vulnerabilities. As a result, we manually interpret and verify each of the detection results that match the CWE of the target vulnerabilities. Our manual verification protocol follows the same procedure used in RQ1. Specifically, the first and third authors independently review each vulnerability and the corresponding alerts, and determine whether any alert truly reports the target vulnerability. The initial agreement between the two authors is 0.601 in terms of Cohen's Kappa, indicating fair-to-good agreement. When disagreements occur, the two authors jointly revisit the case and discuss it until reaching consensus. For each vulnerability, the authors examine the vulnerability description, the fix commit message, and any available developer discussions to identify the vulnerable code location or the taint path responsible for the vulnerability. They then inspect the reported alerts and check whether any alert covers the identified vulnerable location, or overlaps with the relevant taint path. The results of this manual verification are presented in the fifth column of Table 2. Overall, the best-performing detector in our evaluation, CodeQL, successfully detects 10.8% (23/212) of these reported real-world vulnerabilities. Bandit detects 5.3% (10/189) of these vulnerabilities, and PySA fails to detect any of them. For the detection of specific CWE categories of vulnerability, CodeQL again demonstrates the best performance and detects 30.0% (15/50) of the CWE-22 vulnerabilities.

Table 2: Performance of rule-based detectors.

Detector	CWE	#Commit	#Complete	#Warnings	#Verified Positives
CodeQL	CWE-79	73	73	1,633	5
	CWE-22	50	50	1,414	15
	CWE-400	37	37	1,302	2
	CWE-362	32	-	-	-
	CWE-89	29	29	1,341	1
	CWE-352	23	23	388	0
	Total	244	212	6,078	23
PySA	CWE-79	73	24	25	0
	CWE-22	50	30	143	0
	CWE-400	37	-	-	-
	CWE-362	32	-	-	-
	CWE-89	29	0	0	0
	CWE-352	23	-	-	-
	Total	244	54	168	0
Bandit	CWE-79	73	73	73,933	8
	CWE-22	50	50	127,273	0
	CWE-400	37	37	90,889	0
	CWE-362	32	-	-	-
	CWE-89	29	29	30,928	2
	CWE-352	23	-	-	-
	Total	244	189	323,023	10

CWE-79: Cross-site Scripting; CWE-22: Path Traversal; CWE-400: Uncontrolled Resource Consumption; CWE-362: Race Condition; CWE-89: SQL Injection; CWE-352: Cross-Site Request Forgery.

We further list the total number of warnings given by the detectors after scanning the target vulnerabilities in the fourth column of Table 2. In total, PySA generates 168 warnings for 54 vulnerability samples, CodeQL generates 6,078 warnings for 212 samples, and Bandit generates 323,023 warnings for 189 samples. Notably, on average, Bandit outputs 1,709 warnings for each sample. The high volume of warnings produced by Bandit underscores a potentially significant, or even impractical, manual auditing effort.

Overall, none of the evaluated detectors are capable of effectively identifying vulnerabilities in *PyVul (Python)*. In addition, low completion rate and excessive warning numbers further undermine the applicability of the detectors in real-world scenarios.

The best-performing rule-based vulnerability detection approach detects merely 10.8% of the real-world vulnerabilities in *PyVul (Python)*. Additionally, two issues spotted in the evaluated detectors, low completion rate and excessive volume of warnings, potentially undermine their applicability in the real world.

4.2.1 Limitations of Rule-based Detectors. To understand the underlying reasons for the inefficiency of these static analysis tools, we randomly select and manually audit 30 cases where detection failed and summarize the causes of detection failures for each of the six CWE vulnerability types. The following lists the discussion of two CWEs, with the remaining cases provided in the supplementary material.

CWE-79: Cross-site Scripting. Cross-site Scripting (XSS) vulnerabilities are a type of injection vulnerability where user-supplied data gets rendered in web pages without adequate safety checks, resulting in potential malicious code execution. The examined XSS vulnerabilities can be categorized into three types: 1) Reflected XSS **(12/30)**. 2) Stored XSS **(16/30)**. 3) Improper URL parameter validation leading to potential XSS in downstream applications **(2/30)**. Reflected XSS and stored XSS, the predominant types of XSS identified in Python packages, ultimately share the same taint sources, i.e., user input from remote flows, and taint sinks, i.e., server responses. Involved in web applications, these vulnerabilities exhibit complex data flows and require sophisticated taint analysis, which is evidenced by the variety of the fixing locations. For reflected XSS, we observe two commonly adopted fixes in real-world reports: 1) sanitizing the fields of web pages (7/12), 2) sanitizing the data parsed from user requests (3/12). As evidenced by the fixes, accurate detection of reflected XSS requires cross-language taint analysis that takes both server-side code and client-side code into consideration. Neither CodeQL or PySA supports cross-language taint analysis.

Stored XSS extends further. We observe fixes have been applied in various locations, including: 1) proper sanitization in object-relational mappings (ORM) (4/17), 2) proper sanitization in response crafting (5/17), 3) proper sanitization in client-side code (2/17), 4) setting Content Security Policy (CSP) in the server's configuration (4/17). This requires a more comprehensive scope of analysis. In addition, stored XSS is considered a high-order taint-style vulnerability [45] where there exist two phases of taint flows [46]. In the first phase, taint flows from remote flow sources to data storage sinks. In the second phase, the exact taint is loaded from data storage and flows to server response generation sinks. As such, for detecting stored XSS, the detectors are required to identify the stored location of the taint and bridge the two phases. Neither of the two detectors makes any effort to model these complex taint flows for stored XSS.

The third type of XSS, improper validation of URL parameters leading to potential XSS in downstream applications, commonly occurs in non-standalone packages—those not self-contained and primarily serve as utility providers for other applications. Effective taint analysis relies on the precise definition of taint sources and sinks, which can be predefined by the detectors or supplied by users. Both CodeQL and PySA have a comprehensive range of predefined sources and sinks. However, for non-standalone packages where their downstream applications need to be taken into consideration, these predefined sources and sinks can hardly be effective. For example, in the context of a web framework such as Django [47], a function parameter may be used by downstream applications to pass user-supplied data, thus qualifying as a taint source. Vulnerabilities associated with these package-specific sources and sinks cannot be automatically identified by the subject detectors.

Apart from taint analysis rules targeting specific types of XSS, Bandit, and CodeQL have rules checking HTML escaping project-wide to address general XSS, such as checking if the Jinja2 environment is set to auto-escape. Such rules mitigate certain XSS vulnerabilities but face several problems: 1) Jinja2, despite its popularity, is not universally adopted across all Python web applications. Even for those that employ Jinja2 templates, it is not guaranteed that all client-side web pages are generated using Jinja2. 2) Dynamic content on web pages that uses JavaScript can also introduce XSS vulnerabilities; 3) Setting Jinja2 environment to auto-escape may be against the project's business logic. For example, the project *nbdime* involved with CVE-2021-41134 [48] is a tool for diffing and merging of Jupyter notebooks and offers web-based extensions. In such a case where user-uploaded code is displayed on web pages, project-wide automatic escaping may not be a viable solution.

CWE-352: Cross-Site Request Forgery (CSRF). CSRF is an attack that allows the attacker to exploit a user's authentication credentials on a logged-in website and send malicious requests to that site. Two causes of CSRF are identified in the examined reports: 1) Using GET requests to modify database **(16/23)**. GET requests are supposed to be used only for viewing data, and using GET requests to change anything in the database is not protected by any CSRF protection policies; 2) CSRF protection is not applied to certain pages **(6/23)** (e.g., not setting CSRF tokens for certain forms).

Among the three evaluated detectors, only CodeQL includes a rule targeting CSRF vulnerabilities. However, CodeQL over-simplifies the vulnerability without addressing vulnerable code patterns in the real world. CodeQL operates under the assumption that modern web frameworks include built-in CSRF protections and that vulnerabilities only arise when these protections are explicitly disabled. CodeQL's rule checks if a web framework is used and whether CSRF protections have been disabled in the global settings. Yet, we have not observed such cases in real-world vulnerability reports. Most CSRF vulnerabilities exist with certain CSRF protection turned on. As such, CodeQL's rule is completely ineffective against these real-world vulnerabilities.

Our evaluation has yielded two key insights regarding the current state of rule-based detectors:

- In terms of the detector rules, we observe a substantial discrepancy between the assumptions of the evaluated detectors and the real-world security landscape. To quantify the prevalence of this issue, we measure the proportion of in-scope rule-missing failures, where the detector claims to target the corresponding CWE but no rules cover the real-world pattern. As these measurements are binomial proportions, we additionally report 95% confidence intervals using the Wilson score interval, which provides more accurate coverage without relying on large-sample normal approximations. The proportions are 52.5% (95% CI: [44.3%, 60.6%]), 29.2% (95% CI: [20.8%, 39.4%]), and 40.7% (95% CI: [32.2%, 49.7%]) for CodeQL, PySA, and Bandit, respectively. Notably, even at the lower bound of the 95% confidence interval, at least 20.8% of failures can be attributed to missing detector rules.
- In terms of the detector architecture, current rule-based vulnerability detectors for Python lack: 1) support for high-order vulnerabilities, web application-related vulnerabilities can be high-order, such as stored XSS; 2) capability of modeling cross-language vulnerabilities; 3) accurate data flow modeling that addresses Python's language features.

Our empirical evaluation reveals two primary limitations in the current state of rule-based vulnerability detectors for Python: 1. substantial discrepancy between the assumptions of the detectors and real-world security scenarios, 2. lack of support for high-order and cross-language vulnerabilities, and Python's language features.

Table 3: Performance of ML-based approaches in detecting vulnerabilities in *PyVul*.

Model	Train	Test	Dataset	Invalid	Acc.	Prec.	Recall	F1
CodeQwen1.5-Chat	-	300	Paired	19	53.4%	51.9%	59.1%	55.3%
			Non-paired	23	58.5%	57.4%	66.9%	61.8%
GPT-3.5 Turbo	-	300	Paired	0	49.5%	49.5%	60.4%	54.4%
			Non-paired	3	50.8%	50.8%	61.1%	55.5%
GPT-4	-	300	Paired	0	58.0%	52.1%	32.7%	40.2%
			Non-paired	0	51.3%	65.8%	33.3%	44.3%
CodeQwen1.5-Chat finetuned	300	300	Paired	0	51.0%	50.7%	72.0%	59.5%
			Non-paired	0	62.7%	60.1%	75.3%	66.9%
GPT-3.5 Turbo finetuned	300	300	Paired	0	50.0%	-	0%	0%
			Non-paired	0	67.7%	63.9%	81.3%	71.6%
GPT-3.5 Turbo finetuned	1500	300	Paired	0	50.0%	50.0%	99.3%	66.5%
			Non-paired	-	-	-	-	-

4.3 RQ3: Evaluation of ML-based Detectors

In this RQ, we examine how well the real-world Python package vulnerabilities in *PyVul* can be identified by ML-based detectors. Existing approaches [12, 14] commonly detect vulnerabilities at the function level. To address this requirement, we utilize function-level samples from *PyVul* for our evaluation.

4.3.1 Performance of Vulnerability Prediction. To train and evaluate the ML models, we require both vulnerable and non-vulnerable samples. There are several strategies for curating non-vulnerable samples. One approach is to collect the patched versions of vulnerable functions, while another is to gather unrelated functions. Using the patched versions creates a more challenging scenario, as vulnerable and benign samples tend to be very similar, differing by only a few lines of code. This requires ML-based approaches to have a deeper understanding of the intrinsic characteristics of the vulnerabilities. In our experimental setup, we use both strategies, referring to them as paired and non-paired datasets. To create the non-paired dataset, we compile a pool of benign samples from two sources: 1) newly added functions in *PyVul*'s commit, which do not have pre-fix versions, and 2) patched versions of functions that are labeled by *LLM-VDC* as irrelevant to the vulnerabilities. This results in a total of 1,462 functions. For each vulnerable sample, we randomly select a benign sample written in the same programming language from this pool.

The three LLMs introduced in Section 3.3.2 are evaluated under two different setups: direct prompting and fine-tuning. For the evaluation across all settings, we use a total of 300 samples, including 150 vulnerable samples and 150 benign samples, following [49]. In the direct prompting setup, we adopt a zero-shot approach and follow [14] for the chain-of-thought prompt. To the best of our knowledge, Ding et al. [14] represents one of the most recent studies evaluating the capability of LLMs to detect vulnerabilities using direct prompting. Furthermore, we fine-tune CodeQwen1.5-Chat and GPT-3.5 Turbo using a different set of 300 samples, following the settings described in Section 3.3.3. To examine the effect of data volume on fine-tuning, we also conduct an additional experiment where we fine-tune GPT-3.5 Turbo with a larger dataset of 1,500 samples. For all setups, we evaluate the models' performance using various metrics and present the results in Table 3. As LLMs occasionally generate answers that are not in the required format, we report the number of such invalid answers in the fifth column.

As shown in Table 3, LLMs without additional adjustments achieve accuracies ranging from 49.5% to 58.5% and F1 scores ranging from 40.2% to 61.8%, which are only marginally better than random guesses, indicating that they do not inherently support vulnerability detection tasks. Fine-tuning on non-paired data greatly improves the performance of LLMs in terms of vulnerability detection, making it a potentially promising direction. Specifically, GPT-3.5 Turbo and CodeQwen1.5-Chat fine-tuned on 300 non-paired data yields a 29.0% and 8.3% increase in F1 scores respectively, achieving F1 scores of 71.6% and 66.9%.

However, fine-tuning on paired data reveals a severe problem. LLMs fine-tuned on paired data achieve even worse performance than in a zero-shot setting. Specifically, CodeQwen1.5-Chat, despite an increase of 4.2% in F1 score, shows a decrease of 2.4% in accuracy. GPT-3.5 Turbo completely fails to derive any meaningful learning from the paired data and consistently predicts every test case as vulnerable. Moreover, despite being trained on a larger dataset of 1500 samples, GPT-3.5 Turbo does not demonstrate any learning improvements, maintaining an accuracy of 50.0%. This indicates that LLMs are not able to differentiate the vulnerable functions and their patched version. In real world, vulnerable code and benign code are largely similar and often differ in a small number of lines. The inability of LLMs to differentiate between these subtle variations suggests that current LLM-based approaches may struggle to provide practical utility in real-world applications.

The LLM-based vulnerability detection approaches, though achieve relatively promising performance on non-paired data with a best F1 score of 75%, fail to differentiate vulnerable sample with their largely similar patched version, indicating their limited capability in real-world scenarios.

4.3.2 Performance Discrepancies Across CWEs. We further investigate the performance of ML-based approaches fine-tuned and tested on different types of CWE vulnerabilities. Due to limited samples per category, we select the five most prevalent CWEs from our function-level benchmark and use all vulnerable samples from each. We adopt a non-paired setting because LLMs struggle to learn effectively from paired data, which leads to uninformative metrics. For this evaluation, we choose GPT-3.5 Turbo, as it shows the highest F1 score when fine-tuned on non-paired data in Section 4.3.1. We follow the same method as Section 4.3.1 to prepare the non-paired datasets. The datasets curated for each selected CWE category are then divided using an 80/20 split for fine-tuning and testing.

Table 4: Performance of GPT-3.5 Turbo when fine-tuned and tested on different CWE types of vulnerabilities.

CWE	Train	Test	Invalid	Acc.	Prec.	Recall	F1
CWE-79 (Cross-site Scripting)	232	58	0	58.1%	56.4%	71.0%	62.9%
CWE-22 (Path Traversal)	160	42	0	72.5%	90.9%	50.0%	64.5%
CWE-20 (Improper Input Validation)	158	40	0	75.0%	69.2%	90.0%	78.3%
CWE-476 (NULL Pointer Dereference)	84	22	0	68.2%	62.5%	90.9%	74.1%
CWE-125 (Out-of-bound Read)	84	22	0	77.3%	75.0%	81.8%	78.3%

Table 4 shows that GPT-3.5 Turbo yields varying results across CWEs. The performance on CWE-79 and CWE-22 is notably lower

```
+ function htmlEntities(str) {
+     return String(str).replace(/&/g, '&amp;').
     replace(/</g, '&lt;').replace(/>/g, '&gt;').
     replace(/"/g, '&quot;');
+ }
  ...
  {
      data: "ticket",
      render: function (data, type, row, meta) {
          if (type === 'display') {
              data = '<div class="tickettitle"><a
     href="' + get_url(row) + '" >' +
                   row.id + '. ' +
-                  row.title + '</a></div>';
+                  htmlEntities(row.title) + '</a></
     div>';
          }
          return data
      }
  }
```

Figure 3: Example of XSS vulnerability CVE-2021-3945 [51]. The fix in commit 2c7065e [52].

```
def form_valid(self, form):
    request = self.request
    if text_is_spam(form.cleaned_data['body'],
      request):
        return render(request, template_name='
      helpdesk/public_spam.html')
    else:
        ticket = form.save(user=self.request.user if
      self.request.user.is_authenticated else None)
        try:
            return HttpResponseRedirect('%s?ticket=%s
      &email=%s&key=%s' % (
                reverse('helpdesk:public_view'),
                ticket.ticket_for_url,
                urlquote(ticket.submitter_email),
                ticket.secret_key)
                                    )
        except ValueError:
            return HttpResponseRedirect(reverse('
      helpdesk:home'))
```

Figure 4: Relevant data storing function of XSS vulnerability CVE-2021-3945.

compared to other categories: CWE-79, despite being trained on the largest sample size of 232, attained the lowest F1 score of 62.9%. In contrast, CWE-125, which was trained on a smaller sample set of 84, achieved a higher F1 score of 78.3%.

To understand this gap, we manually examined 50 vulnerable functions from each category and identified two contributing factors: high variance among vulnerable functions and limited contextual visibility.

1) **The great variance of the vulnerable functions.** CWE-79 (Cross-Site Scripting) is a taint-style vulnerability [50], where taint flows span multiple functions and sanitization may occur at various points. We observed sanitization applied in diverse locations—from database interactions and embeded client-side code such as HTML/JavaScript, to server configurations. In this case, in the function-level setting where the changed functions prior to the fixing commit are marked as vulnerable, a diverse range of vulnerable function samples may be observed. Such great variance can confuse the model and hinder its ability to learn effective patterns for identifying vulnerabilities.

Figure 3 shows an example of XSS vulnerability. This vulnerability arises when a user creates a ticket with a malicious title that injects html code. As the ticket titles are never sanitized, when the tickets are rendered on the admin's page, the injected HTML code will take effect. The fix adopted by the developers sanitizes the ticket titles in the `render` function where the tickets are rendered. Alternatively, the vulnerability could also be effectively mitigated by sanitizing the data before it is stored in the database, specifically within the `form_valid` function, as shown in Figure 4. In this instance, if developers opt to sanitize the data in `render`, then `render` is identified as the vulnerable function; if sanitization occurs in `form_valid`, then `form_valid` is marked as vulnerable. In contrast, the patches of CWE-125 (Out-of-bound Read) typically exhibit shorter taint flows. When the vulnerabilities are processed as vulnerable functions, the models are able to observe more stable vulnerable code patterns compared to CWE-79.

2) **Inability to see important context.** In the patches of CWE-79 vulnerabilities, sanitizations are commonly implemented as new functions and then applied to the input data. However, in the function-level setting, the model may not be able to access the content of such sanitization functions, resulting in additional difficulty in differentiating between the vulnerable code and fixed code pair.

For instance, in the same example of Figure 3, the developers create a sanitization function `htmlEntities`, which sanitizes potential HTML injection, and invoke it in the `render` function. In the function-level settings, including the ones that take a step further and group functions implicated in cross-function vulnerabilities, such as [53], as the function `htmlEntities` is newly created and does not have a pre-existing vulnerable counterpart, it will not be included in the dataset that is used to train or test models. Consequently, this omission impedes the model's ability to discern between the pre-fix and post-fix versions of the `render` function.

Therefore, we argue that the typical function-level setting is problematic to real-world scenarios, failing to capture the characteristics of real-world vulnerabilities. This echoes the observation reported by Risse et al. [54]. Future work is encouraged to explore innovative training methods to incorporate relevant contexts of vulnerabilities and enable models to effectively learn vulnerable code patterns despite the variance of fixing patches.

The performance of fine-tuned LLMs for vulnerability detection varies substantially across CWEs. The commonly adopted function-level setting fails on complex real-world vulnerabilities for two reasons: 1) the great variance of the vulnerable functions, and 2) the potential absence of important context.

5 Related Work

Vulnerability empirical study. Many empirical studies [55–57] have been conducted to study vulnerabilities in other software systems or software ecosystems. Regarding the Python ecosystem, Alfadel et al. [58] make the first move to study the propagation and life span of Python security vulnerabilities. Besides, there has been research to study the bugs in machine learning (ML) libraries in Python [59–62]. Despite these efforts, the characteristics of security vulnerabilities in the whole Python package ecosystem have not been well studied, and how well current vulnerability detection tools perform on real-world vulnerabilities in Python packages remains unknown.

Vulnerability datasets. Different datasets [7, 9, 12, 14, 25, 63] have been presented to facilitate vulnerability detection. Apart from the datasets we have compared *PyVul* with, notably, the VUDENC [9] dataset contains more than 20,000 vulnerable Python functions. However, since VUDENC is collected by keyword matching in commit messages, it suffers from low label accuracies [10] and is thus excluded from our comparison. Cheng et al. [12] presented DiverseVul with their empirical study, a new C/C++ vulnerable source code dataset that is 60% larger than the previous largest dataset for C/C++. In contrast, our dataset is collected with a focus on vulnerabilities in Python packages and cleansed with the assistance of LLMs to achieve high label accuracy.

Vulnerability dataset cleansing. Apart from PrimeVul [14], ReposVul [13] also targets the inaccurate labels in vulnerability datasets by combining LLMs and static vulnerability detectors. ReposVul determines a file as related to a vulnerability fix if both LLMs indicate its relevance and static vulnerability detectors identify vulnerabilities in its before-fixing version. Comparing to ReposVul, our cleansing method, LLM-VDC, cleanses the datasets at a finer granularity with a higher accuracy. In addition, since ReposVul inherently depends on the outputs of static vulnerability detectors, its resulting dataset is unsuitable as a benchmark for evaluating static detectors.

Vulnerability detection. Rule-based static vulnerability detection has been a commonly adopted approach, characterized by scalability and comprehensiveness. Previously, the method has been extensively explored in the context of statically typed languages [45, 64–67]. Recently, its application in dynamic languages such as JavaScript has also become a popular research area [19, 68–70]. However, rule-based static vulnerability detection in Python is yet to be explored by the research community, which may be attributed to Python's complex features [71]. Even the most fundamental elements of static analysis, such as call graphs, have only recently been explored [72]. The state-of-the-art static vulnerability detectors in Python are predominantly developed by industry, with notable tools such as CodeQL [22], PySA [23], and Bandit [24] leading the efforts.

For ML-based vulnerability detection, previous papers have used LSTM [9, 73], CNNs and RNNs [74], Bidirectional RNNS [75], and Graph Neural Networks [25, 76, 77] to detect vulnerable source code. Among them, only VUDENC [9] was trained and evaluated on Python code. VUDENC applies a word2vec model to identify semantically similar code tokens and to provide a vector representation, and then uses an LSTM network to classify vulnerable code token sequences.

6 Discussion & Threats to Validity

Discussion. Our LLM-assisted cleansing method labeled 72 out of 8,374 vulnerable functions as "4) no decision can be made". To ensure the integrity and validity of the evaluation on this automated cleansing method, especially when measuring the precision and recall, we exclude the commits associated with these 72 functions from our dataset. Future work could explore altering the composition of contextual information provided to LLMs or incorporating additional context to help LLMs resolve such cases.

In our empirical evaluation of vulnerability detectors, we evaluated current rule-based and ML-based detectors and investigated their limitations independently. A direct comparison between these two methodologies was not conducted due to inherent differences in their operational granularities. Rule-based detectors scan whole projects and locate vulnerabilities precisely with detailed information such as the causes of the vulnerabilities and taint flow paths, while current ML-based detectors typically analyze individual functions and solely classify them as vulnerable or not.

Threats to validity. Our dataset is curated from multiple vulnerability-reporting platforms, which differ in reporting criteria, disclosure practices, update frequencies, and historical coverage. As a result, platform bias and temporal bias may exist and potentially affect the representativeness of certain vulnerability types or time periods. However, our aim is to obtain an ecosystem-level understanding of Python-package vulnerabilities; therefore, maximizing coverage and curating a comprehensive set of known vulnerabilities are of greater importance to our study.

In addition, fix commits are used to identify and de-duplicate vulnerability reports collected from different platforms. Two special situations exist: 1. a single fix addresses multiple unrelated vulnerabilities; 2. a vulnerability is fixed by multiple distinct commits. The former leads to under-counting vulnerabilities and the latter results in incomplete patches. However, based on our manual review of a statistically significant number of commits, such cases are rare, and their overall impact on our conclusions is limited. This could potentially be mitigated by incorporating LLMs into the de-duplication process, which we leave for future work.

For RQ1 and RQ2, the dataset labels and the rule-based detectors' results are validated manually. The reliability of these decisions can be influenced by factors such as the evaluators' expertise in relevant areas and their personal interpretations of the vulnerabilities. To mitigate potential biases, we involve two authors, both with solid backgrounds in Python programming and software security, to independently assess the correctness of the labels and then resolve disputes.

In RQ3, the ML-based detectors are trained and evaluated using our dataset cleansed by *LLM-VDC*, which achieves high but not perfect accuracy. Such label noises may affect the measured performance of ML-based detectors. Nevertheless, our manual validation shows that the labeling accuracy exceeds 93%, suggesting that such noise is relatively limited in scale. We therefore expect its impact to primarily affect absolute performance values rather than invalidate the overall conclusions of RQ3.

7 Conclusion

In this paper, we presented *PyVul*, the first large-scale, high-quality benchmark suite of Python-package vulnerabilities, consisting of 1,157 vulnerable repository snapshots, and 2,082 vulnerable functions along with their respective patched versions. We introduce LLM-VDC, an LLM-assisted data cleansing method, to cleanse *PyVul* and achieve a 62.1% to 74.1% improvement in function-level label accuracy compared to previous automatically collected vulnerability datasets, underscoring the effectiveness of our cleansing method in vulnerability labels. Utilizing *PyVul*, we further evaluate current rule-based and ML-based static vulnerability detectors in Python. Our experimental results reveal that none of the current approaches is satisfactory for detecting these real-world vulnerabilities in Python packages. Additionally, our empirical study delves into the limitations of these detectors, offering critical insights to fuel future development of static vulnerability detectors.

References

[1] (2024) The top programming languages. Accessed: 2024-02-13. [Online]. Available: https://spectrum.ieee.org/top-programming-languages/

[2] (2024) Pypi · the python package index. [Online]. Available: https://pypi.org/

[3] N. Ivaki *et al.*, "A taxonomy for python vulnerabilities," *IEEE Open Journal of the Computer Society*, 2024.

[4] H.-C. Tran, A.-D. Tran, and K.-H. Le, "Detectvul: A statement-level code vulnerability detection for python," *Future Generation Computer Systems*, vol. 163, p. 107504, 2025.

[5] "Github advisory," https://github.com/advisories/, 2024, accessed: 2024-02-13.

[6] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: automated collection of vulnerabilities and their fixes from open-source software," in *Proc. PROMISE '21*, 2021, pp. 30–39.

[7] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, "Crossvul: a cross-language vulnerability dataset with commit data," in *Proc. ESEC/FSE '21*, 2021, pp. 1565–1569.

[8] (2024) National vulnerability database. Accessed: 2024-02-13. [Online]. Available: https://nvd.nist.gov/

[9] L. Wartschinski, Y. Noller, T. Vogel, T. Kehrer, and L. Grunske, "Vudenc: Vulnerability detection with deep learning on a natural codebase for python," *Information and Software Technology*, vol. 144, p. 106809, 2022.

[10] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proc. CCS '23*, 2023, pp. 1865–1879.

[11] R. Croft, M. A. Babar, and M. M. Kholoosi, "Data quality for software vulnerability datasets," in *Proc. ICSE '23*. IEEE, 2023, pp. 121–133.

[12] 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 *Proc. RAID '23*, 2023, pp. 654–668.

[13] X. Wang, R. Hu, C. Gao, X.-C. Wen, Y. Chen, and Q. Liao, "Reposvul: A repository-level high-quality vulnerability dataset," in *Proc. ICSE Companion '24*, 2024, pp. 472–483.

[14] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, and Y. Chen, "Vulnerability detection with code language models: How far are we?" *arXiv preprint arXiv:2403.18624*, 2024.

[15] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for Python build reproducibility," in *Proc. ISSTA '21*, 2021, pp. 439–451.

[16] "Snyk vulnerability database," https://security.snyk.io/, 2024, accessed: 2024-02-13.

[17] (2024) Huntr: The world's first bug bounty platform for ai/ml. Accessed: 2024-02-13. [Online]. Available: https://huntr.com/

[18] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. USENIX Security '19*, 2019, pp. 995–1010.

[19] S. Li, M. Kang, J. Hou, and Y. Cao, "Mining node. js vulnerabilities via object dependence graph and query," in *Proc. USENIX Security '22*, 2022, pp. 143–160.

[20] N. Vasilakis, C.-A. Staicu, G. Ntousakis, K. Kallas, B. Karel, A. DeHon, and M. Pradel, "Preventing dynamic library compromise on node. js via rwx-based privilege reduction," in *Proc. CCS '21*, 2021, pp. 1821–1838.

[21] M. H. M. Bhuiyan, A. S. Parthasarathy, N. Vasilakis, M. Pradel, and C.-A. Staicu, "Secbench. js: An executable security benchmark suite for server-side javascript," in *Proc. ICSE '23*. IEEE, 2023, pp. 1059–1070.

[22] GitHub, "Codeql," 2024. [Online]. Available: https://codeql.github.com

[23] Meta, "Pysa," 2024. [Online]. Available: https://github.com/facebook/pyre-check

[24] PyCQA, "Bandit," 2024. [Online]. Available: https://github.com/PyCQA/bandit

[25] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3280–3296, 2021.

[26] W. G. Cochran, *Sampling techniques*. Wiley, 1977.

[27] M. Zheng, J. Pei, and D. Jurgens, "Is" a helpful assistant" the best role for large language models? a systematic evaluation of social roles in system prompts," *arXiv preprint arXiv:2311.10054*, vol. 8, 2023.

[28] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proc. NeurIPS '20*, 2020.

[29] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[30] OpenAI, "Chatgpt," 2024. [Online]. Available: https://chatgpt.com/

[31] "EvalPlus evaluates AI Coders with rigorous tests." https://evalplus.github.io/leaderboard.html, 2024, accessed: 2024-02-13.

[32] G. Raffa, J. Blasco, D. O'Keeffe, and S. K. Dash, "{CloudFlow}: Identifying security-sensitive data flows in serverless applications," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 1073–1090.

[33] J. Gong, N. Duan, Z. Tao, Z. Gong, Y. Yuan, and M. Huang, "How well do large language models serve as end-to-end secure code agents for python?" in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, 2025, pp. 1004–1013.

[34] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot-generated code in github projects: An empirical study," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–34, 2025.

[35] "Codeql python queries," https://docs.GitHub.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/python-built-in-queries, 2024, accessed: 2024-02-13.

[36] (2024) Vudenc repository. Accessed: 2024-02-13. [Online]. Available: https://github.com/LauraWartschinski/VulnerabilityDetection

[37] QwenLM, "Codeqwen1.5," 2024. [Online]. Available: https://github.com/QwenLM/CodeQwen1.5

[38] "Go ahead and axolotl questions," https://github.com/axolotl-ai-cloud/axolotl, 2024, accessed: 2024-02-13.

[39] "Hugging Face – The AI community building the future." https://huggingface.co/, 2024, accessed: 2024-02-13.

[40] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[41] T. Y. Zhuo, A. Zebaze, N. Suppattarachai, L. von Werra, H. de Vries, Q. Liu, and N. Muennighoff, "Astraios: Parameter-efficient instruction tuning code large language models," *arXiv preprint arXiv:2401.00788*, 2024.

[42] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *arXiv preprint arXiv:2308.10462*, 2023.

[43] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

[44] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[45] H. Zhang, W. Chen, Y. Hao, G. Li, Y. Zhai, X. Zou, and Z. Qian, "Statically discovering high-order taint style vulnerabilities in os kernels," in *Proc. CCS '21*, 2021, pp. 811–824.

[46] H. Su, F. Li, L. Xu, W. Hu, Y. Sun, Q. Sun, H. Chao, and W. Huo, "Splendor: Static detection of stored xss in modern web applications," in *Proc. ISSTA '23*, 2023, pp. 1043–1054.

[47] Django, "Django: The web framework for perfectionists with deadlines," 2024. [Online]. Available: https://www.djangoproject.com/

[48] "Cve-2021-41134," 2021, accessed: 2024-02-13. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-41134

[49] M. D. Purba, A. Ghosh, B. J. Radford, and B. Chu, "Software vulnerability detection using large language models," in *Proc. ISSREW '23*. IEEE, 2023, pp. 112–119.

[50] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. S&P '15*. IEEE, 2015, pp. 797–812.

[51] "CVE-2021-3945," https://nvd.nist.gov/vuln/detail/CVE-2021-3945, 2021, accessed: 2024-02-13.

[52] "Commit 2c7065e, django-helpdesk," https://github.com/django-helpdesk/django-helpdesk/commit/2c7065e0c4296e0c692fb4a7ee19c7357583af30, 2021, accessed: 2024-02-13.

[53] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, "Toward improved deep learning-based vulnerability detection," in *Proc. ICSE '24*, 2024, pp. 1–12.

[54] N. Risse and M. Böhme, "Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection," *arXiv preprint arXiv:2408.12986*, 2024.

[55] M. Jimenez, M. Papadakis, and Y. Le Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *Proc. APSEC '16*. IEEE, 2016, pp. 105–112.

[56] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, pp. 1665–1705, 2014.

[57] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velá squez, "An empirical study on android-related vulnerabilities," in *Proc. MSR '17*. IEEE, 2017, pp. 2–13.

[58] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in python packages," *Empirical Software Engineering*, vol. 28, no. 3, p. 59, 2023.

[59] N. S. Harzevili, J. Shin, J. Wang, S. Wang, and N. Nagappan, "Characterizing and understanding software security vulnerabilities in machine learning libraries," in *Proc. MSR '23*. IEEE, 2023, pp. 27–38.

[60] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proc. ISSTA '18*, 2018, pp. 129–140.

[61] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," in *Proc. ISSRE '12*. IEEE, 2012, pp. 271–280.

[62] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "The symptoms, causes, and repairs of bugs inside a deep learning library," *Journal of Systems and Software*, vol. 177, p. 110935, 2021.

[63] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A c/c++ code vulnerability dataset with code changes and cve summaries," in *Proc. MSR '20*, 2020, pp. 508–512.

[64] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM sigplan notices*, vol. 49, no. 6, pp. 259–269, 2014.

[65] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE S&P '14*, 2014, pp. 590–604.

[66] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. ICSE '15*, vol. 1. IEEE, 2015, pp. 280–291.

[67] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, pp. 1–32, 2018.

[68] V. Kashyap, K. Dewey, E. A. Kuefner, J. Wagner, K. Gibbons, J. Sarracino, B. Wiedermann, and B. Hardekopf, "Jsai: A static analysis platform for javascript," in *Proc. ISSTA '14*, 2014, pp. 121–132.

[69] S. Khodayari and G. Pellegrino, "{JAW}: Studying client-side {CSRF} with hybrid property graphs and declarative traversals," in *Proc. USENIX Security '21*, 2021, pp. 2525–2542.

[70] M. Kang, Y. Xu, S. Li, R. Gjomemo, J. Hou, V. Venkatakrishnan, and Y. Cao, "Scaling javascript abstract interpretation to detect and exploit node. js taint-style vulnerability," in *Proc. S&P '23*. IEEE, 2023, pp. 1059–1076.

[71] Y. Yang, M. Fazzini, and M. Hirzel, "Complex Python features in the wild?" in *Proc. MSR '22*, 2022.

[72] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in Python," in *Proc. ICSE '21*. IEEE, 2021, pp. 1646–1657.

[73] 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.

[74] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. ICMLA '18*. IEEE, 2018, pp. 757–762.

[75] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[76] Y. Mirsky, G. Macon, M. Brown, C. Yagemann, M. Pruett, E. Downing, S. Mertoguno, and W. Lee, "{VulChecker}: Graph-based vulnerability localization in source code," in *Proc. USENIX Security '23*, 2023, pp. 6557–6574.

[77] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.