

An Empirical Study of Vulnerabilities in Python Packages and Their Detection

Haowei Quan

Monash University

Melbourne, Australia

haowei.quan@monash.edu

Terry Yue Zhuo

Monash University

Melbourne, Australia

terry.zhuo@monash.edu

Junjie Wang

College of Intelligence and Computing, Tianjin University

Tianjin, China

junjie.wang@tju.edu.cn

Xiao Chen

University of Newcastle

Newcastle, Australia

xiao.chen@newcastle.edu.au

Xinzhe Li

College of Intelligence and Computing, Tianjin University

Tianjin, China

3022001754@tju.edu.cn

Xiaoning Du

Monash University

Melbourne, Australia

Xiaoning.Du@monash.edu

Abstract

In the rapidly evolving software development landscape, Python stands out for its simplicity, versatility, and extensive ecosystem. Python packages, as the unit for code organization, reusability, and distribution, have become an increasingly pressing concern, highlighted by the considerable number of vulnerability reports. As a scripting language, Python often has to cooperate with other programming languages for usability, including better efficiency and interoperability with other libraries. This also adds complexity to the vulnerabilities inherent to Python packages, and the effectiveness of current vulnerability detection tools in spotting these vulnerabilities is underexplored within the research community.

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, each linked to its affected packages. To accommodate diverse detection techniques, the benchmark provides annotations at both commit and function levels. An LLM-assisted data cleansing method is incorporated to improve label accuracy, achieving 100% commit-level and 94% function-level accuracy, establishing *PyVul* as the most precise large-scale Python vulnerability benchmark. We further carry out a distribution analysis of *PyVul*, which demonstrates that vulnerabilities in Python packages involve multiple programming languages and exhibit a wide variety of types. Moreover, our analysis reveals that multi-lingual Python packages are potentially more susceptible to vulnerabilities. Evaluation of state-of-the-art detectors using this benchmark reveals a significant discrepancy between the capabilities of existing tools and the demands of effectively identifying real-world security issues in Python packages. Additionally, we conduct an empirical review of the top-ranked Common Weakness Enumerations (CWE) observed in Python packages, to diagnose the fine-grained limitations of current detection tools and highlight the necessity for future advancements in the field.

1 Introduction

Over recent years, Python has become the leading programming language due to its user-friendly syntax, versatility, and rich ecosystem [15]. With nearly 600,000 packages hosted on PyPI [13], Python's growing application across domains like web development and machine learning (ML) raises critical concerns about the security of its

package ecosystem [35, 63]. 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 [8] reported 507 vulnerabilities in Python packages in 2023, highlighting its growing security importance, comparable to npm (394) and Maven (937).

Despite this, no benchmark comprehensively captures real-world Python package vulnerabilities with high accuracy. Real-world vulnerabilities in Python packages may involve other programming languages. Due to Python's nature as a scripting language, Python packages frequently embed C/C++ code for performance-critical tasks, such as the implementations of NumPy [51] and PyTorch [47]. Moreover, as a popular candidate for developing web applications, client-side code such as JavaScript and HTML can inevitably be involved. This highlights the necessity of contextualizing vulnerabilities within Python packages rather than focusing solely on the Python code itself. Python packages provide a comprehensive view of execution paths that can reveal vulnerabilities and help reduce biases in vulnerability identification.

Current vulnerability benchmarks for Python, 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, CVEFixes [20] and CrossVul [50] are collected based on projects from security platforms such as National Vulnerability Database (NVD) [12] and do not effectively map to Python packages. In addition, datasets such as VUDENC [66] and SVEN [33] 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.

Concerns regarding the quality of existing benchmarks have been raised [24, 26]. 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. [65] 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. [28] 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**. We evaluate how effectively current vulnerability detectors can 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 the function level. We initially collected 3,630 reports from three security advisories, GitHub Advisories [8], Snyk [14], and Huntr [11]. Subsequent filtering based on the existence of associated fixing commits and eliminating duplicates resulted in 1,767 reports. Following established methodologies from previous studies [20, 50], we constructed a commit-level benchmark consisting of 1,767 vulnerable repository snapshots, and a function-level benchmark comprising 8,374 vulnerable functions, each derived from the respective fixing commits. To enhance the 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 [20, 50] and comparable to the human-annotated small dataset, SVEN [33], which contains only 380 vulnerable functions. LLM-VDC demonstrates superior universality and outperforms the state-of-the-art labeling method introduced by Ding et al. [28] in a multi-lingual vulnerability dataset, with a 33.1% greater improvement in function-level label accuracy. After evaluating the effectiveness of our LLM-assisted data cleansing method and the quality of *PyVul* (RQ1), we analyze the distribution of real-world Python package vulnerabilities regarding programming languages, function compositions, and CWE categories (RQ2).

Furthermore, leveraging *PyVul*, we assess state-of-the-art rule-based and ML-based static vulnerability detectors (RQ3 and RQ4). We leave the evaluation of dynamic vulnerability detectors to future work considering the common reproducibility issues in the open-source repositories [49]. 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 significant 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%.
- The first look into the distribution of Python package vulnerabilities regarding programming languages, function compositions, and CWEs.
- 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, and experimental scripts are made openly accessible at <https://github.com/billquan/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 [8], Snyk [14], and Huntr [11]. These platforms are widely used by developers and serve as data sources for other empirical research on vulnerabilities [21, 42, 64, 80]. 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. 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. 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 [31], PySA [46], and Bandit [54], typically operate

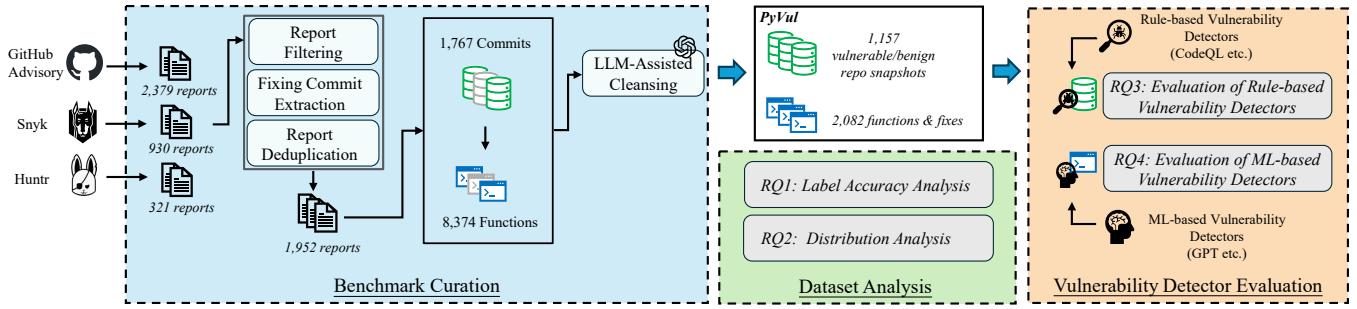


Figure 1: The overview of our study

at the project level. In contrast, ML-based static analysis methods [23, 24, 28] 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 [23, 24, 28]. For each commit, we consider the functions involved as vulnerability samples in their pre-fix 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.

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* [20], and *CrossVul* [50] in Section 4.1. The sample sizes were determined following [25], 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 40.4%, 48.3%, and 51.0% 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 code readability, which are also highlighted by other researchers [28]. 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 [76], few-shot learning [22], and chain-of-thought (CoT) prompting [68], 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 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 with 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.

 **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]

 **Role: User (Sample to Predict)**
[Query for new sample]

Few-shot Learning

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

- **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 ① to ③ and only choose the latter option if all former options are infeasible: ① We supply all other changed function information in the commit; ② We supply all other changed functions in the same file; ③ 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 [52], one of the top-performing LLMs available at the time of writing. 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: What is the vulnerabilities distribution in Python packages?
- RQ3: How effective are current rule-based approaches for detecting vulnerabilities in *PyVul*?
- RQ4: 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. This includes existing benchmarks for Python vulnerabilities, automated filtering methods for vulnerability datasets, rule-based and ML-based vulnerability detectors that are relevant to the Python community.

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 programs. 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** [33]. 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. SVEN was con-

- structed by carefully examining existing data sources, including VUDENC [66] and BigVul [30], to ensure high quality. However, the intensive manual effort involved in creating this dataset limits its scalability. It is important to note that the original SVEN dataset does not include commit-level vulnerabilities, which are essential for rule-based vulnerability detectors that operate on repository snapshots. To address this gap, we enhanced the SVEN dataset by incorporating the relevant fix commits for the identified vulnerable and non-vulnerable function pairs. This supplementation resulted in a total of 143 commit-level vulnerability and non-vulnerability samples.
- *CVEFixes* [20] and *CrossVul* [50]. 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 [24] has successfully extracted vulnerable and benign functions from the CrossVul dataset. We employ a similar approach to obtain function-level data for CrossVul. 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. [28]. They proposed a set of heuristic rules to retain functions only if the likelihood of them being the source of a vulnerability is high. These heuristic rules are heavily focused on the precision of identifying vulnerable functions, rather than on recall. Specifically, a prior version of a function is labeled as vulnerable when it meets all three of the following criteria: 1) the function is the only one modified in a vulnerability-fixing commit; 2) the function's name is mentioned in the linked CVE report's vulnerability description; and 3) the function's file name is noted in the associated CVE report's vulnerability description, and it is the only function modified in that file. Using this approach, PrimeVul achieves a high accuracy rate of around 90%. However, its strict rules may lead to a significant 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 the help of LLMs. 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, reflecting the highest standard in this task.

- *CodeQL* [31]. 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. Additionally, CodeQL comes with a suite of query-based rules designed to detect various types of vulnerabilities. CodeQL supports multiple programming languages, including Python. For Python, there are 101 built-in queries [6], including an extended set focused on security. Each query is annotated with the CWEs that it aims to detect, and in total, 123 CWEs are covered by these queries.
- *PySA* [46]. PySA is a static code analysis tool developed as part of Facebook's Pyre-check project. It is specifically designed to address a variety of taint-style vulnerabilities in Python applications, such as SQL injections. 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, which help in identifying covered CWEs. In total, PySA covers 67 different CWEs.
- *Bandit* [54]. Bandit is a popular open-source static analysis tool (with 6k stars on GitHub) designed specifically to identify security issues in Python code. It scans Python programs to uncover common vulnerabilities, such as the use of potentially dangerous APIs and hard-coded credentials. 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, each annotated with the CWE related to the target vulnerability. In total, these rules cover 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 [66]. However, due to unresolved bugs in its published implementation [17] 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 vulnerability detection, either through direct prompting [28] or fine-tuning [24, 28]. 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 [20], and demonstrate superior performance on larger datasets [24]. 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 RQ4, 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 LLMs, including one open-source model and two from OpenAI.

- *CodeQwen1.5-Chat* [55]. 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 [7], outperforming GPT-3.5 Turbo and showing performance comparable to GPT-4.
- *GPT-3.5 Turbo* and *GPT-4* [52]. The GPT family of models demonstrates exceptional performance among LLMs and shows superior capabilities compared to open-source models in vulnerability detection [28]. 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 [28] to setup the fine-tuning framework for CodeQwen1.5-Chat, with Axolotl [9]. We load the model's weights from Hugging Face Models [10] and fine-tune it with a learning rate of 2×10^{-5} for four epochs using LoRA [34], which balances between the training efficiency and task performance [69, 79]. 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 [28]. 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 [25] 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 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. 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.

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%. At the function level, when no data cleansing is applied, the label accuracy of the collected raw data is 40.4%. After applying the *PrimeVul* method, the label accuracy increases to 70.8%. However, this is significantly lower than the accuracy of around 90% reported in their own datasets [28]. 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

Table 1: Label accuracy of existing benchmarks and our newly curated benchmark, and the impact of data cleansing method on label accuracy.

Benchmark	Data Cleansing	#Commits	#Sampled Commits	Commit-lvl Acc(%)	#Vuln Funcs	#Sampled Funcs	Function-lvl Acc(%)
SVEN[33]	N/A	143	105	100	380	192	96.3
CVEFixes[20]	N/A	508	219	99.5	1,360	300	48.3
CrossVul[50]	N/A	319	175	99.4	777	258	51.0
PyVul (Python)	LLM-VDC	788	259	100.0	1,480	306	93.1
PyVul	N/A	1,767	316	99.7	8,374	368	40.4
	PrimeVul	745	254	100.0	1,012	279	70.8
	LLM-VDC	1,157	289	100.0	2,082	325	94.2

Table 2: The language composition of PyVul.

Language Composition	#Commits	#Functions
Python	775 (67.0%)	1,480 (71.1%)
C/C++	335 (29.0%)	463 (22.2%)
JavaScript/TypeScript	23 (2.0%)	115 (5.5%)
Java	4 (0.3%)	12 (0.6%)
Other Language	2 (0.3%)	12 (0.6%)
Multiple Languages	18 (1.6%)	-
Total	1,157	2,082

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 51.0%, 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. 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: Characteristic Analysis of Python Package Vulnerabilities

Currently, there is no systematic analysis of the characteristics of Python package vulnerabilities due to the absence of a vulnerability benchmark for Python packages. The curated *PyVul* benchmark allows for a comprehensive analysis of various aspects of Python package vulnerabilities. This includes qualitative and quantitative analysis of their language composition, the number of functions

involved, and the types of vulnerabilities present, providing insights necessary for understanding Python package vulnerabilities and guiding the development of corresponding detection tools.

4.2.1 Language Composition Analysis. The analysis of language composition in the benchmark offers valuable insights for developing effective detection tools. Since packages from which the vulnerabilities originate provide a more comprehensive context for understanding them, while the fixing commits directly indicate their causes and fixes, our analysis is conducted against both.

We conducted an analysis of programming languages used in all 349 Python packages associated with *PyVul* by querying the language statistics of their repositories via the GitHub API. The results are presented in Figure 3. As shown, these Python packages predominantly involve multiple programming languages. Approximately 75% (262/349) of the packages used at least two programming languages, while around 36% (127/349) utilized at least five different languages. We additionally counted the total number of vulnerabilities at the commit level encompassed by these packages as light blue bars in Figure 3. The data reveals that over 90% of the vulnerabilities are found in packages that use multiple languages. On average, a Python-only package is associated with 1.18 vulnerabilities, while a multi-lingual Python package is linked to 4.02 vulnerabilities. Notably, 14 packages with more than 12 languages contribute 342 vulnerabilities. The main reason is that two of these packages, TensorFlow and PyTorch, account for 311 vulnerabilities. We further employ the interquartile range method [27] to remove the impact of outliers. After adjustment, a Python-only package is associated with an average of 1.18 vulnerabilities, whereas a multilingual Python package averages 1.29 vulnerabilities. This suggests an increased risk of vulnerabilities in multilingual packages.

To better understand the relationship between vulnerabilities and the multi-language characteristics of Python packages, we compare the language composition distribution of the packages in *PyVul* with that of general PyPI packages. The packages in *PyVul* are quite popular, averaging 13,358.7 stars on GitHub. To effectively control the effect of popularity, we randomly select the same number of packages from the top 8,000 most popular PyPI packages [16] for comparison. As illustrated in Figure 3, the packages in *PyVul* show a clear tendency towards the usage of multiple programming languages. This echoes the observation that multi-lingual Python packages can be more susceptible to vulnerabilities, which is also consistent with the observation in previous work [72].

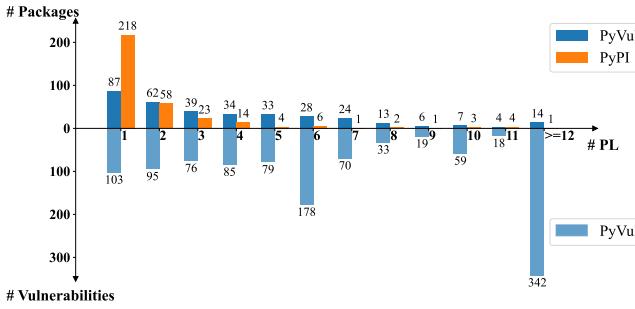


Figure 3: Programming language (PL) distribution in Python packages.

We further analyze the language composition of vulnerabilities at the commit level. We use Guesslang [5] to identify each vulnerable function’s programming language, and aggregate them to derive the language composition at the commit level. We group the commits and vulnerable functions, respectively, according to their programming languages and present the statistics in Table 2. To our surprise, only 1.6% of the vulnerabilities involve more than one programming language. Among the vulnerabilities, 67.0% are exclusively related to Python, while 31.4% are associated with other programming languages, with C/C++ being the most prevalent non-Python language. Two important observations can be drawn: 1) non-Python vulnerabilities are common in Python packages, and 2) most vulnerabilities and their fixes are associated with a single programming language. It is essential to note that this does not imply that they can be effectively detected by tools designed for that specific language. The broader context of these vulnerabilities often involves multiple programming languages. Therefore, effective detection tools must be capable of handling cross-language code contexts, a point which is also supported by our findings in RQ3.

At the package level, multi-language is a common characteristic in Python packages. Our analysis indicates that multi-lingual Python packages are potentially more susceptible to vulnerabilities. At the commit level, the vulnerability samples within Python packages also involve various programming languages. However, only 1.6% of the commits involve vulnerable functions in multiple languages, indicating that these vulnerabilities tend to be fixed within a single language.

4.2.2 Vulnerability Type Distribution. Vulnerabilities come in many different types, each varying in detection difficulty. Beyond simply assessing whether a vulnerability detection method can find vulnerabilities, we are also interested in its performance when detecting different types of vulnerability. Therefore, we additionally annotate the *PyVul* dataset with CWEs from original vulnerability reports. The 1,157 commit-level vulnerabilities in the *PyVul* dataset belong to 151 different CWE vulnerability types. We performed a simple clustering based on the mechanisms, the causes and the consequences of these CWE vulnerability types. For example, CWE-125 (Out-of-bounds Read), CWE-787 (Out-of-bounds Write), CWE-120 (Buffer Copy without Checking Size of Input, ‘Classic Buffer Overflow’),

Table 3: Vulnerability types distribution of *PyVul*.

Type	#Commits	#Functions	Avg. CVSS
Injection	202 (17.5%)	411 (19.7%)	7.4
Improper Access Control	133 (11.5%)	305 (14.6%)	7.3
Out-of-Bound Read/Write	114 (9.9%)	174 (8.4%)	6.0
File Operation Error	80 (6.9%)	165 (7.9%)	6.9
Improper Input Validation	75 (6.5%)	109 (5.2%)	6.7
Calculation Error	66 (5.7%)	79 (3.8%)	4.6
Sensitive Information Exposure	60 (5.1%)	103 (4.9%)	5.8
Request Forgery	53 (4.6%)	109 (5.2%)	7.5
Improper Resource Management	54 (4.7%)	112 (5.4%)	6.5
NULL Pointer Dereference	43 (3.7%)	53 (2.5%)	5.4
Assertion Failures	39 (3.4%)	46 (2.2%)	5.4
Incorrect Synchronization	38 (3.3%)	66 (3.2%)	5.6
Redirect Error	23 (2.0%)	47 (2.3%)	6.1
Use of Uninitialized Resource	24 (2.1%)	29 (1.4%)	5.9
Improper Deserialization	23 (2.0%)	39 (1.9%)	8.9
Incorrect Regular Expression	22 (1.9%)	31 (1.5%)	6.1
Uncontrolled Recursion	16 (1.4%)	24 (1.2%)	5.9
Improper Exception Handling	16 (1.4%)	20 (1.0%)	5.3
Inefficient Algorithmic Complexity	12 (1.0%)	39 (1.9%)	7.0
Incorrect Provision of Specified Functionality	9 (0.8%)	27 (1.3%)	3.5
Incomplete Cleanup	6 (0.5%)	11 (0.5%)	7.2
Side Channel	5 (0.4%)	8 (0.4%)	5.2
Others	43 (3.7%)	70 (3.4%)	6.7
Total	1,157	2,082	6.5

and CWE-122 (Heap-based Buffer Overflow) were merged into one category. We list the details of the clustering in Appendix C.

In Table 3, we list the distribution of vulnerability types in Python packages. Injection vulnerabilities are the most common type, with 195 commits (394 functions), accounting for 17.5% (19.7%) of the total. Injection vulnerabilities consist of 16 CWE vulnerability types, including SQL Injection, Command Injection, Parameter Injection, Cross-site Scripting (XSS) Injection, Static Code Injection, XML External Entity (XXE) Injection, CSV Formula Injection, and others. Access control vulnerabilities are the second most common type, accounting for 11.5% of the total commits (133 commits) and 14.6% of the total functions (305 functions). Access control vulnerabilities consist of 33 CWE vulnerability types, primarily including CWE-284 (Improper Access Control), CWE-287 (Improper Authentication), CWE-305/289/288/290/294 (Authentication Bypass by Primary Weakness/Alternate Name/Using an Alternate Path or Channel/Spoofing/Capture Replay), and CWE-304 (Missing Critical Step in Authentication), among others. Following closely are vulnerability types such as Out-of-Bound Read/Write, File Operation Error, Improper Input Validation, and Calculation Error, which also occur relatively frequently.

From the vulnerability types we can spot a great diversity regarding their origins and attack scenarios. Vulnerabilities such as XSS Injection, Improper Access Control and Request Forgery are predominantly associated with web applications. On the other hand, vulnerabilities such as Out-of-Bound Read/Write, NULL Pointer Dereference and Use of Uninitialized Resource are typically linked to low-level C/C++ code. Additionally, Incorrect Synchronization relates to parallel execution. This diversity in vulnerability type echoes Python’s usage in different fields and may pose extra difficulty to automated static detectors, including both rule-based ones and ML-based ones.

PyVul exhibits a total of 151 distinct CWE vulnerability types. Among the vulnerability types, Injection, Improper Access Control, Out-of-Bound Read/Write are the most common types. The vulnerability types within Python packages are diverse in their origins and attack scenarios, echoing Python's application in various fields and posing additional difficulty to automated static detectors.

4.3 RQ3: 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 4.

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.

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 matches the CWE of the target-reported vulnerabilities. The results of this manual verification are presented in the fifth column of Table 4. 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.

We further list the total number of warnings given by the detectors after scanning the target vulnerabilities in the fourth column of Table 4. 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.

Table 4: 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.

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.

4.3.1 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 three CWEs and we present the rest in Appendix B.

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 [74] where there exist two phases of taint flows [60]. 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 that are 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 the detector 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 [29], 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 [4] 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-22: Path Traversal. Path traversal refers to a situation where an application receives unvalidated user input as parameters for file-related operations, such as reading or viewing files. These parameters contain special characters (e.g., ‘..’ and '/') that can be used to bypass protection mechanisms, gain unauthorized access to protected files or directories, or overwrite sensitive data. Several

types of path traversal vulnerability are spotted: 1) Improper use of other packages (4/30); 2) Unawareness of behavioral differences of used APIs from other packages when executed on different operating systems (3/30); 3) Missing validation in certain taint paths (22/30). The first two causes are not considered by any of these evaluated detectors. For the static detection of the third cause, taint analysis is the typical approach. CodeQL and PySA support static taint analysis. However, four factors hinder their performance in the detection of CWE-22:

- Lack of package-specific taint specifications. Non-standalone packages require package-specific taint specifications.
- Lack of accurate type information. This is an inherited challenge for Python static analyzers. As a dynamic language, variable types in Python are determined at run time. Without type information, static modeling of data flows can be largely incomplete, substantially limiting the effectiveness of taint analysis built upon it. Implementing type inference can mitigate this challenge. However, neither of the subject detectors incorporate any form of type inference.
- Limited handling of Python's complex language features. Python's advanced language features, such as higher-order functions and dynamic features [73], frequently present in the examined packages. Incomplete addressing of these features further contributes to incomplete data flow modeling.
- Complex data flows in web applications. Web applications are frequently spotted in CWE-22 reports. The inherent complexity of web applications arises from their interaction with client-side components and their capability to execute multiple routes concurrently, often resulting in intricate data flows. Neither of the detectors effectively models these intricate data flows.

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 significant insights regarding the current state of rule-based detectors:

- In terms of the detector rules, we observe a significant discrepancy between the assumptions of the evaluated detectors and the

Table 5: 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	-	-	-	-	-

real-world security landscape. The evaluated rule-based detectors tend to oversimplify the vulnerabilities, resulting in either a high volume of false positives or complete missing detection of real-world cases.

- 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. significant 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.

4.4 RQ4: 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 [24, 28] commonly detect vulnerabilities at the function level. To address this requirement, we utilize function-level samples from *PyVul* for our evaluation.

4.4.1 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, which include 150 vulnerable samples and 150 benign samples, following [53]. In the direct prompting setup, we adopt a zero-shot approach and follows [28] for the chain-of-thought prompt. To the best of our knowledge, [28] 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 experimental setups, we evaluate the models' performance using various metrics and present the results in Table 5. 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 5, 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.

4.4.2 Performance Discrepancies Across CWEs. We further investigate the performance of ML-based approaches when they are fine-tuned and tested on different types of CWE vulnerabilities. Due to the limited amount of data available for each CWE category, we select the five most prevalent CWEs from our function-level benchmark and utilize all vulnerable samples from each category.

Table 6: 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%

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 4.4.1. We follow the same method as Section 4.4.1 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.

The data in Table 6 indicates that fine-tuning GPT-3.5 Turbo on different CWE categories results in varying performance levels. Notably, the performance on CWE-79 and CWE-22 is significantly lower compared to other categories. Specifically, 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%.

We further investigate the performance discrepancy between CWE-79 and CWE-125 by manually examining 50 vulnerable functions from each category. In comparing CWE-79 with CWE-125, we identify two factors that may contribute to the poor performance associated with fine-tuning CWE-79: the specific characteristics of the CWE category and the training method used.

1) The great variance of the vulnerable functions. CWE-79 (Cross Site Scripting) is a taint-style vulnerability [71]. The taint flows of CWE-79 are typically complex and involve multiple functions. To fix the vulnerability, the sanitizations can be applied at different places of the taint flows. In real-world instances, we have observed sanitization being applied to diverse locations—from data storage functions interacting with databases to various segments of client-side code like HTML, JavaScript, or template creation, to even within 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 4 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 5. In this instance, if developers opt to sanitize the data in render, then

```
+ 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 + '. ' +
+ data + '</a></div>';
      return data
    }
  }
}
```

Figure 4: Example of XSS vulnerability CVE-2021-3945 [3]. The fix in commit 2c7065e [2].

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 4, 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 [59], 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. [56]. 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 significantly across different CWE categories of vulnerabilities. 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.

```

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('?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 5: Relevant data storing function of XSS vulnerability CVE-2021-3945.

5 Related Work

Vulnerability empirical study. Many empirical studies [37, 45, 61] have been conducted to study vulnerabilities in other software systems or software ecosystems. Tan et al. [61] analyzed around 2k real-world vulnerabilities in Linux kernel, Mozilla, and Apache and yielded several guidelines for developing corresponding detectors. Linares et al. [45] conducted a large-scale empirical study to characterize different types of vulnerability that affect Android apps. Regarding the Python ecosystem, Alfadel et al. [18] 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 [32, 36, 62, 75]. 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 [23, 24, 28, 30, 50, 66] have been presented to facilitate vulnerability detection. Apart from the datasets we have compared *PyVul* with, notably, the ReVeal [23] dataset was labeled using the patches to known security issues at Chromium security issues and Debian security tracker. BigVul [30] collect vulnerability-fixing commits from CVE records in the NVD. Cheng et al. [24] 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++, and the most diverse compared to all previous datasets. 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 [28], ReposVul [65] 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 [19, 41, 67, 70, 74]. Recently, its application in dynamic languages such as JavaScript has also become a popular research area [38–40, 42]. 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 [73]. Even the most fundamental elements of static analysis, such as call graphs, have only recently been explored [58]. The state-of-the-art static vulnerability detectors in Python are predominantly developed by industry, with notable tools such as CodeQL [31], PySA [46], and Bandit [54] leading the efforts.

For ML-based vulnerability detection, previous papers have used LSTM [44, 66], CNNs and RNNs [57], Bidirectional RNNS [43], and Graph Neural Networks [23, 48, 78] to detect vulnerable source code. Among them, only VUDENC [66] 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. For RQ1 and RQ3, 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. We additionally measure their agreement level with Cohan’s Kappa before any consensus has been reached, which is 0.718 for RQ1 and 0.601 for RQ3 and within the range of fair to good.

In all experiments in this study where humans are involved, there exist cases where the participant cannot make the decisions either because of limited descriptions or erroneously attached commits in the vulnerability reports. As the number of such cases is relatively

small, we expect they do not affect the overall conclusions. For example, in RQ3, there is one reviewed path traversal vulnerability report that we cannot decide the type and the cause of it, which we expect does not affect our overall observations regarding this category of vulnerabilities.

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] 2014. CVE-2014-0474. <https://nvd.nist.gov/vuln/detail/CVE-2014-0474>. Accessed: 2024-11-30.
- [2] 2021. Commit 2c7065e, django-helpdesk. <https://github.com/django-helpdesk/django-helpdesk/commit/2c7065e0c4296e0c692fb4a7ee19c7357583af30>. Accessed: 2024-02-13.
- [3] 2021. CVE-2021-3945. <https://nvd.nist.gov/vuln/detail/CVE-2021-3945>. Accessed: 2024-02-13.
- [4] 2021. CVE-2021-41134. <https://nvd.nist.gov/vuln/detail/CVE-2021-41134>. Accessed: 2024-02-13.
- [5] 2021. Detect the programming language of a source code . <https://pypi.org/project/guesslang/>. Accessed: 2024-02-13.
- [6] 2024. CodeQL Python Queries. <https://docs.GitHub.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/python-built-in-queries>. Accessed: 2024-02-13.
- [7] 2024. EvalPlus evaluates AI Coders with rigorous tests. <https://evalplus.github.io/leaderboard.html>. Accessed: 2024-02-13.
- [8] 2024. GitHub Advisory. <https://github.com/advisories/>. Accessed: 2024-02-13.
- [9] 2024. Go ahead and axolotl questions. <https://github.com/axolotl-ai-cloud/axolotl>. Accessed: 2024-02-13.
- [10] 2024. Hugging Face - The AI community building the future. <https://huggingface.co/>. Accessed: 2024-02-13.
- [11] 2024. Huntr: The world's first bug bounty platform for AI/ML. <https://huntr.com/>. Accessed: 2024-02-13.
- [12] 2024. National Vulnerability Database. <https://nvd.nist.gov/>. Accessed: 2024-02-13.
- [13] 2024. PyPI - The Python Package Index. <https://pypi.org/>
- [14] 2024. Snyk Vulnerability Database. <https://security.snyk.io/>. Accessed: 2024-02-13.
- [15] 2024. The Top Programming Languages. <https://spectrum.ieee.org/top-programming-languages/>. Accessed: 2024-02-13.
- [16] 2024. Top PyPI Packages. <https://hugovk.github.io/top-pypi-packages/>. Accessed: 2024-02-13.
- [17] 2024. VUDENC repository. <https://github.com/LauraWartschinski/VulnerabilityDetection>. Accessed: 2024-02-13.
- [18] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.
- [19] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Barret, Jacques Klein, Yves Le Traon, Damien Ochteau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *ACM sigplan notices* 49, 6 (2014), 259–269.
- [20] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.
- [21] Masudul Hasan Masud Bhuiyan, Adithya Srinivas Parthasarathy, Nikos Vasilakis, Michael Pradel, and Cristian-Alexandru Staiu. 2023. SeeBench.js: An executable security benchmark suite for server-side JavaScript. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1059–1070.
- [22] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (Vancouver, BC, Canada) (NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 159, 25 pages.
- [23] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.
- [24] Yizheng Chen, Zhoujie Ding, Lamya Alowain, Xinyun Chen, and David Wagner. 2023. Diverseul: 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*. 654–668.
- [25] William G Cochran. 1977. *Sampling techniques*. Wiley.
- [26] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 121–133.
- [27] Frederik Michel Dekking, Cornelis Kraaijkamp, Hendrik Paul Lopuhää, and Ludolf Erwin Meester. 2006. *A Modern Introduction to Probability and Statistics: Understanding why and how*. Springer Science & Business Media.
- [28] Yangruibo Ding, Yanju Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. 2024. Vulnerability Detection with Code Language Models: How Far Are We? *arXiv preprint arXiv:2403.18624* (2024).
- [29] Django. 2024. *Django: The web framework for perfectionists with deadlines*. <https://www.djangoproject.com/>
- [30] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.
- [31] GitHub. 2024. *CodeQL*. <https://codeql.github.com>
- [32] Nima Shiri Harzevili, Jijo Shin, Junjie Wang, Song Wang, and Nachiappan Nagapana. 2023. Characterizing and understanding software security vulnerabilities in machine learning libraries. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 27–38.
- [33] Jingxuan He and Martin Vechev. 2023. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1865–1879.
- [34] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).
- [35] Naghmeh Ivaki et al. 2024. A Taxonomy for Python Vulnerabilities. *IEEE Open Journal of the Computer Society* (2024).
- [36] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.
- [37] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 105–112.
- [38] Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, Jianwei Hou, VN Venkatakrishnan, and Yinzhi Cao. 2023. Scaling javascript abstract interpretation to detect and exploit node.js taint-style vulnerability. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1059–1076.
- [39] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.
- [40] Soheil Khodayari and Giancarlo Pellegrino. 2021. {JAW}: Studying client-side {CSRF} with hybrid property graphs and declarative traversals. In *30th USENIX Security Symposium (USENIX Security 21)*. 2525–2542.
- [41] Li Li, Alexandre Bartel, Tegawende F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Ochteau, and Patrick McDaniel. 2015. Iccfa: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 280–291.
- [42] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining node.js vulnerabilities via object dependence graph and query. In *31st USENIX Security Symposium (USENIX Security 22)*. 143–160.
- [43] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. Sevavr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 4 (2021), 2244–2258.

- [44] Zhen Li, Deqing Zou, Shouhai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).
- [45] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. 2017. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2–13.
- [46] Meta. 2024. PySA. <https://github.com/facebook/pyre-check>
- [47] Meta. 2024. Tensors and Dynamic neural networks in Python with strong GPU acceleration. <https://pytorch.org/>
- [48] Yisroel Mirsky, George Macon, Michael Brown, Carter Yagemann, Matthew Pruitt, Evan Downing, Sukarno Mertoguno, and Wenke Lee. 2023. {VulChecker}: Graph-based Vulnerability Localization in Source Code. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6557–6574.
- [49] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *The 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 439–451.
- [50] Georgios Nikitopoulos, Konstantina Dritts, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569.
- [51] NumPy. 2024. *The fundamental package for scientific computing with Python*. <https://numpy.org/>
- [52] OpenAI. 2024. ChatGPT. <https://chatgpt.com/>
- [53] Mounmita Das Purba, Arpita Ghosh, Benjamin J Radford, and Bill Chu. 2023. Software vulnerability detection using large language models. In *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 112–119.
- [54] PyCQA. 2024. Bandit. <https://github.com/PyCQA/bandit>
- [55] QwenLM. 2024. CodeQwen1.5. <https://github.com/QwenLM/CodeQwen1.5>
- [56] Niklas Risse and Marcel Böhme. 2024. Top score on the wrong exam: On benchmarking in machine learning for vulnerability detection. *arXiv preprint arXiv:2408.12986* (2024).
- [57] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.
- [58] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical call graph generation in Python. In *ICSE 2021*. IEEE, 1646–1657.
- [59] Adriana Sejfta, Satyaki Das, Saad Shafiq, and Nenad Medvidović. 2024. Toward Improved Deep Learning-based Vulnerability Detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
- [60] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. 2023. Splendor: Static Detection of Stored XSS in Modern Web Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1043–1054.
- [61] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19 (2014), 1665–1705.
- [62] Ferdinand Thung, Shaowei Wang, David Lo, and Lingxiao Jiang. 2012. An empirical study of bugs in machine learning systems. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 271–280.
- [63] Hoai-Chau Tran, Anh-Duy Tran, and Kim-Hung Le. 2025. DetectVul: A statement-level code vulnerability detection for Python. *Future Generation Computer Systems* 163 (2025), 107504.
- [64] Nikos Vasilakis, Cristian-Alexandru Staicu, Grigoris Ntousakis, Konstantinos Kallas, Ben Karel, André DeHon, and Michael Pradel. 2021. Preventing dynamic library compromise on node.js via rwx-based privilege reduction. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1821–1838.
- [65] Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. Reposul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 472–483.
- [66] Laura Wartschinski, Yannic Noller, Thomas Vogel, Timo Kehrer, and Lars Grunske. 2022. Vudenc: Vulnerability detection with deep learning on a natural codebase for python. *Information and Software Technology* 144 (2022), 106809.
- [67] Fengguo Wei, Sankardas Roy, Ximeng Ou, and Robby. 2018. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Transactions on Privacy and Security (TOPS)* 21, 3 (2018), 1–32.
- [68] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.
- [69] Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui. 2023. Exploring parameter-efficient fine-tuning techniques for code generation with large language models. *arXiv preprint arXiv:2308.10462* (2023).
- [70] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604. <https://doi.org/10.1109/SP.2014.44>
- [71] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 797–812.
- [72] Haoran Yang, Yu Nong, Shaowei Wang, and Haipeng Cai. 2024. Multi-language software development: Issues, challenges, and solutions. *IEEE Transactions on Software Engineering* 50, 3 (2024), 512–533.
- [73] Yi Yang, Milanova Fazzini, and Martin Hirzel. 2022. Complex Python Features in the Wild?. In *the 19th International Conference on Mining Software Repositories (MSR) 2022*.
- [74] Hang Zhang, Weiteng Chen, Yu Hao, Guoren Li, Yizhuo Zhai, Xiaochen Zou, and Zhiyun Qian. 2021. Statically discovering high-order taint style vulnerabilities in os kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 811–824.
- [75] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 129–140.
- [76] Mingqian Zheng, Jiaxin Pei, and David Jurgens. 2023. 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* 8 (2023).
- [77] Yunhui Zheng and Xiangyu Zhang. 2012. Static detection of resource contention problems in server-side scripts. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 584–594.
- [78] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems* 32 (2019).
- [79] Terry Yue Zhuo, Armel Zebaze, Nitchakarn Suppattarakachai, Leandro von Werra, Harm de Vries, Qian Liu, and Niklas Muennighoff. 2024. Astraios: Parameter-Efficient Instruction Tuning Code Large Language Models. *arXiv preprint arXiv:2401.00788* (2024).
- [80] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*. 995–1010.

A Vulnerability Span Analysis

Span analysis aims to examine the number of functions related to a vulnerability. It provides crucial insights for detecting and addressing vulnerabilities, as the span reveals the minimum context required for effective analysis. However, a precise measurement of the span has not yet been obtained due to the inaccurate identification of code changes relevant to vulnerabilities in prior benchmarks. Given the high quality of *PyVul*, we evaluate how many functions are involved in the vulnerabilities of Python packages and present the statistics in Figure 6. The number of functions involved in the vulnerabilities of *PyVul* ranges from 1 to 13, with fewer vulnerabilities observed in more expansive categories. On average, each vulnerability in *PyVul* is linked to 1.8 function. In particular, 503 (43.5%) vulnerabilities involve more than one vulnerable function. These cross-function vulnerabilities are associated with an average of 2.6 vulnerable functions. The prevalence of cross-function vulnerabilities emphasizes the importance of fully considering the cross-function characteristics when detecting or addressing vulnerabilities.

B Limitations of Rule-based Detectors.

The following presents more analysis of our empirical review of the top CWEs. **CWE-400: Uncontrolled Resource Consumption.** Uncontrolled Resource Consumption refers to a type of vulnerability where a system fails to properly limit resource usage, leading to exhaustion of system resources such as CPU, memory, disk space,

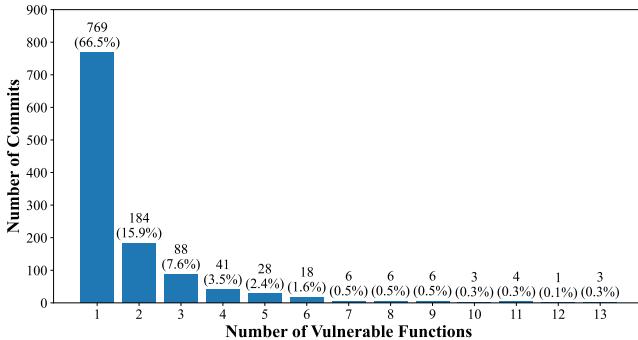


Figure 6: Vulnerable functions count distribution of PyVul.

network bandwidth, or file descriptors. This can result in performance degradation, denial of service (DoS), or even system crashes. The examined Uncontrolled Resource Consumption vulnerabilities can be attributed to four causes: 1. Improper limitations on resource consumption (**23/30**). A typical example of this includes parsing a user-supplied YAML file without setting the maximum number of nodes, which can lead to excessive consumption of space or time. 2. Regular expressions with an inefficient worst-case computational complexity (**4/30**). 3. Algorithm defects (**2/30**). For instance, certain user input can trigger infinite loops in a program. 4. Unclosed resources (**1/30**).

Bandit targets only one specific case of improper limitations on resource consumption, which checks whether the `timeout` parameter has been set in the `request` library's API calls, failing to address this most prevalent type of Uncontrolled Resource Consumption systematically. Improper limitations on resource consumption attribute to the co-existence of two factors: 1) User-consumed resources. There exists data flows from user inputs to resource consumption APIs, such as file storage APIs and XML parsing APIs; 2) Absence of limitations or user supplying limitations, such as the size of user-uploaded data for file storage APIs, or the maximum number of nodes in XML parsing APIs. These limitations are typically implemented either as parameters of the resource consumption APIs or as independent checks before user inputs reach these APIs. As such, effective detection of improper limitations on resource consumption requires an extended taint analysis that not only identifies the taint flows from user inputs to resource consumption APIs, but also backtraces the limitations from these APIs.

On the other hand, CodeQL includes a rule targeting inefficient regular expressions, failing to address most Uncontrolled Resource Consumption vulnerabilities, and PySA does not have any rule targeting Uncontrolled Resource Consumption vulnerabilities.

CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). A race condition can arise when the necessary atomicity of operations is violated in concurrent execution, resulting in unexpected program behavior. Traditional atomicity violations typically involve synchronous operations, such as threads, accessing shared memory without adequate safeguards. In web applications, atomicity violations can also occur when synchronous operations access external resources such as file systems. In the PyPI ecosystem, both traditional (**12/30**) and

web application-related (**18/30**) atomicity violations are commonly observed.

None of the three tools supports detection of race conditions in Python. Detection of traditional atomicity violations involving access to shared memory requires definitions of atomic regions [77]. This detection can potentially be implemented using CodeQL, which provides API modeling based on functionality and a sound data flow analysis engine. Web application-related atomicity violations extend further, requiring an assessment of whether multiple operations access the same external resource, such as a specific data record in a database. Furthermore, as discussed, data flows in web applications are complex to model. As such, detecting atomicity violations in web applications requires sophisticated methods to be developed.

CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'). SQL vulnerabilities occur when developers fail to filter, escape, restrict, or properly handle user input strings in systems that interact with databases. This allows attackers to input carefully crafted strings to illegally access data from the database. The majority (**28/29**) of SQL injection vulnerabilities are caused by improper input validation, except for CVE-2014-0474 [1], which mainly relates to developers' unawareness of MySQL's typecasting behavior.

All three detectors target SQL injection caused by improper input validation. Bandit's rules checks for hard-coded SQL queries and use of potentially dangerous APIs such as Django's RawSQL. However, as Bandit does not exhibit any data flow analysis, these rules exhibit a high false positive rate. CodeQL and PySA adopted taint analysis and are able to more accurately identify SQL injection. However, in the vulnerability reports examined, most of these SQL injections locate in non-standalone packages, and taint analysis in these packages are largely ineffective without package-specific taint specifications.

C Detailed Clustering

Table 7: Clustering of CWE vulnerability types

Cluster Name	CWE Name	Commits	Functions
Injection	CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	89	185
	CWE-89 Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	29	50
	CWE-78 Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	20	29
	CWE-74 Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	12	29
	CWE-94 Improper Control of Generation of Code ('Code Injection')	12	36
	CWE-77 Improper Neutralization of Special Elements used in a Command ('Command Injection')	10	11
	CWE-611 Improper Restriction of XML External Entity Reference	7	17
	CWE-88 Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')	6	11
	CWE-1336 Improper Neutralization of Special Elements Used in a Template Engine	4	11
	CWE-93 Improper Neutralization of CRLF Sequences ('CRLF Injection')	3	11
	CWE-80 Improper Neutralization of Script-Related HTML Tags in a Web Page (Basic XSS)	3	5
	CWE-116 Improper Encoding or Escaping of Output	2	4
	CWE-75 Failure to Sanitize Special Elements into a Different Plane (Special Element Injection)	1	3
	CWE-707 Improper Neutralization	1	1
	CWE-1236 Improper Neutralization of Formula Elements in a CSV File	1	2
	CWE-96 Improper Neutralization of Directives in Statically Saved Code ('Static Code Injection')	1	4
	CWE-91 XML Injection (aka Blind XPath Injection)	1	2
Improper Input Validation	CWE-20 Improper Input Validation	69	99
	CWE-1284 Improper Validation of Specified Quantity in Input	6	10
File Operation Error	CWE-22 Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	51	101
	CWE-59 Improper Link Resolution Before File Access ('Link Following')	10	20
	CWE-377 Insecure Temporary File	5	6
	CWE-434 Unrestricted Upload of File with Dangerous Type	4	19
	CWE-29 Path Traversal: '..\filename'	4	10
	CWE-23 Relative Path Traversal	4	6
	CWE-36 Absolute Path Traversal	1	1
NULL Pointer Dereference	CWE-476 NULL Pointer Dereference	43	53
	CWE-125 Out-of-bounds Read	43	53
Out-of-Bound Read/Write	CWE-787 Out-of-bounds Write	21	48
	CWE-119 Improper Restriction of Operations within the Bounds of a Memory Buffer	17	19
	CWE-120 Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	16	28
	CWE-131 Incorrect Calculation of Buffer Size	11	17
	CWE-122 Heap-based Buffer Overflow	6	9
	CWE-400 Uncontrolled Resource Consumption	41	78
Resource Management Error	CWE-770 Allocation of Resources Without Limits or Throttling	12	31
	CWE-404 Improper Resource Shutdown or Release	1	3

Table 7 continued from previous page

Cluster Name	CWE Name	Commits	Functions
Assertion Failures	CWE-617 Reachable Assertion	39	46
Information Exposure	CWE-200 Exposure of Sensitive Information to an Unauthorized Actor	38	64
	CWE-209 Generation of Error Message Containing Sensitive Information	4	7
	CWE-532 Insertion of Sensitive Information into Log File	4	5
	CWE-212 Improper Removal of Sensitive Information Before Storage or Transfer	4	11
	CWE-312 Cleartext Storage of Sensitive Information	2	6
	CWE-668 Exposure of Resource to Wrong Sphere	2	3
	CWE-614 Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	2	2
	CWE-598 Use of GET Request Method With Sensitive Query Strings	1	1
	CWE-524 Use of Cache Containing Sensitive Information	1	1
	CWE-213 Exposure of Sensitive Information Due to Incompatible Policies	1	2
	CWE-311 Missing Encryption of Sensitive Data	1	1
Incorrect Synchronization	CWE-362 Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	34	62
	CWE-367 Time-of-check Time-of-use (TOCTOU) Race Condition	1	1
	CWE-821 Incorrect Synchronization	1	1
	CWE-662 Improper Synchronization	1	1
	CWE-833 Deadlock	1	1
Open Redirect	CWE-601 URL Redirection to Untrusted Site ('Open Redirect')	23	47
Improper Deserialization	CWE-502 Deserialization of Untrusted Data	23	39
Origin Validation Error	CWE-352 Cross-Site Request Forgery (CSRF)	23	54
	CWE-918 Server-Side Request Forgery (SSRF)	21	36
	CWE-444 Inconsistent Interpretation of HTTP Requests ('HTTP Request/Response Smuggling')	9	23
	CWE-346 Origin Validation Error	1	1
Improper Access Control	CWE-287 Improper Authentication	20	38
	CWE-284 Improper Access Control	14	28
	CWE-863 Incorrect Authorization	10	18
	CWE-347 Improper Verification of Cryptographic Signature	10	30
	CWE-295 Improper Certificate Validation	9	34
	CWE-384 Session Fixation	7	32
	CWE-522 Insufficiently Protected Credentials	6	10
	CWE-285 Improper Authorization	5	12
	CWE-276 Incorrect Default Permissions	5	7
	CWE-269 Improper Privilege Management	4	14
	CWE-345 Insufficient Verification of Data Authenticity	4	4
	CWE-640 Weak Password Recovery Mechanism for Forgotten Password	4	4
	CWE-294 Authentication Bypass by Capture-replay	3	4
	CWE-250 Execution with Unnecessary Privileges	3	4
	CWE-307 Improper Restriction of Excessive Authentication Attempts	3	8
	CWE-521 Weak Password Requirements	3	3
	CWE-290 Authentication Bypass by Spoofing	2	2
	CWE-306 Missing Authentication for Critical Function	2	7
	CWE-862 Missing Authorization	2	5
	CWE-1220 Insufficient Granularity of Access Control	2	6
	CWE-620 Unverified Password Change	2	12
	CWE-305 Authentication Bypass by Primary Weakness	1	4
	CWE-289 Authentication Bypass by Alternate Name	1	2
	CWE-288 Authentication Bypass Using an Alternate Path or Channel	1	4

Table 7 continued from previous page

Cluster Name	CWE Name	Commits	Functions
	CWE-304 Missing Critical Step in Authentication	1	2
	CWE-639 Authorization Bypass Through User-Controlled Key	1	1
	CWE-273 Improper Check for Dropped Privileges	1	1
	CWE-613 Insufficient Session Expiration	1	1
	CWE-749 Exposed Dangerous Method or Function	1	1
	CWE-940 Improper Verification of Source of a Communication Channel	1	2
	CWE-281 Improper Preservation of Permissions	1	1
	CWE-732 Incorrect Permission Assignment for Critical Resource	1	1
	CWE-942 Permissive Cross-domain Policy with Untrusted Domains	1	2
	CWE-322 Key Exchange without Entity Authentication	1	1
Computation Error	CWE-369 Divide By Zero	36	38
	CWE-190 Integer Overflow or Wraparound	19	24
	CWE-681 Incorrect Conversion between Numeric Types	6	9
	CWE-191 Integer Underflow (Wrap or Wraparound)	2	2
	CWE-682 Incorrect Calculation	2	5
	CWE-193 Off-by-one Error	1	1
Regular Expression	CWE-1333 Inefficient Regular Expression Complexity	18	25
	CWE-185 Incorrect Regular Expression	4	6
Uncontrolled Recursion	CWE-674 Uncontrolled Recursion	4	5
	CWE-835 Loop with Unreachable Exit Condition ('Infinite Loop')	10	17
	CWE-776 Improper Restriction of Recursive Entity References in DTDs ('XML Entity Expansion')	1	1
	CWE-834 Excessive Iteration	1	1
Uninitialized	CWE-824 Access of Uninitialized Pointer	10	11
	CWE-665 Improper Initialization	8	8
	CWE-908 Use of Uninitialized Resource	6	10
Improper Exception handling	CWE-754 Improper Check for Unusual or Exceptional Conditions	9	12
	CWE-12 ASP.NET Misconfiguration: Missing Custom Error Page	3	4
	CWE-755 Improper Handling of Exceptional Conditions	2	2
	CWE-460 Improper Cleanup on Thrown Exception	1	1
	CWE-248 Uncaught Exception	1	1
Incomplete Cleanup	CWE-459 Incomplete Cleanup	2	4
	CWE-416 Use After Free	2	4
	CWE-415 Double Free	2	3
Side Channel	CWE-203 Observable Discrepancy	2	4
	CWE-385 Covert Timing Channel	2	3
	CWE-208 Observable Timing Discrepancy	1	1
Format String	CWE-134 Use of Externally-Controlled Format String	2	4
Inefficient Algorithmic Complexity	CWE-330 Use of Insufficiently Random Values	1	3
	CWE-331 Insufficient Entropy	2	5
	CWE-338 Use of Cryptographically Weak Pseudo-Random Number Generator (PRNG)	1	1
	CWE-328 Use of Weak Hash	1	1
	CWE-407 Inefficient Algorithmic Complexity	2	2
	CWE-326 Inadequate Encryption Strength	3	21
	CWE-327 Use of a Broken or Risky Cryptographic Algorithm	2	6
Incorrect Provision of Specified Functionality	CWE-684 Incorrect Provision of Specified Functionality	9	27
Always-Incorrect Control Flow Implementation	CWE-670 Always-Incorrect Control Flow Implementation	5	7
Improper Validation of Integrity Check Value	CWE-354 Improper Validation of Integrity Check Value	5	8
Incorrect Comparison	CWE-697 Incorrect Comparison	4	10

Table 7 continued from previous page

Cluster Name	CWE Name	Commits	Functions
Incorrect Type Conversion or Cast	CWE-704 Incorrect Type Conversion or Cast	2	2
Improper Handling of Alternate Encoding	CWE-173 Improper Handling of Alternate Encoding	2	6
Improper Handling of Structural Elements	CWE-237 Improper Handling of Structural Elements	2	2
Business Logic Errors	CWE-840 Business Logic Errors	2	6
Acceptance of Extraneous Untrusted Data With Trusted Data	CWE-349 Acceptance of Extraneous Untrusted Data With Trusted Data	2	2
Access of Resource Using Incompatible Type ('Type Confusion')	CWE-843 Access of Resource Using Incompatible Type ('Type Confusion')	2	6
Unprotected Alternate Channel	CWE-420 Unprotected Alternate Channel	2	2
Undefined Behavior for Input to API	CWE-475 Undefined Behavior for Input to API	2	2
Prototype Pollution	CWE-1321 Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution')	1	1
Improper Output Neutralization for Logs	CWE-117 Improper Output Neutralization for Logs	1	2
Client-Side Enforcement of Server-Side Security	CWE-602 Client-Side Enforcement of Server-Side Security	1	2
Improper Restriction of Rendered UI Layers or Frames	CWE-1021 Improper Restriction of Rendered UI Layers or Frames	1	1
Unchecked Return Value	CWE-252 Unchecked Return Value	1	1
Initialization of a Resource with an Insecure Default	CWE-1188 Initialization of a Resource with an Insecure Default	1	1
Mutable Attestation or Measurement Reporting Data	CWE-1283 Mutable Attestation or Measurement Reporting Data	1	1
Function Call With Incorrect Order of Arguments	CWE-683 Function Call With Incorrect Order of Arguments	1	1
Interpretation Conflict	CWE-436 Interpretation Conflict	1	1
Improper Control of Dynamically-Managed Code Resources	CWE-913 Improper Control of Dynamically-Managed Code Resources	1	1
Unimplemented or Unsupported Feature in UI	CWE-447 Unimplemented or Unsupported Feature in UI	1	1