

Mind the Gap: Do Widely-Used SAST Tools Really Cover the Language-Specific Top CWEs?

Zhilin Li

Institute of Software, Chinese
Academy of Sciences
Technology Center of Software
Engineering
Beijing, China
University of Chinese Academy of
Sciences
Beijing, China
lizl@ios.ac.cn

Yiran Nie

University of Chinese Academy of
Sciences
Hangzhou Institute for Advanced
Study
Hangzhou, China
nieyiran25@mails.ucas.ac.cn

Xianglong Qi

Beijing University of Technology
Beijing, China
qixianglong@emails.bjut.edu.cn

Jiwei Yan*

Institute of Software, Chinese
Academy of Sciences
Technology Center of Software
Engineering
Beijing, China
University of Chinese Academy of
Sciences
Beijing, China
Beijing Key Laboratory of Intelligent
Software Engineering
Beijing, China
yanjiwei@otcaix.iscas.ac.cn

Jun Yan*

Institute of Software, Chinese
Academy of Sciences
Technology Center of Software
Engineering; Key Laboratory of
System Software
Beijing, China
University of Chinese Academy of
Sciences
Hangzhou Institute for Advanced
Study
Hangzhou, China
Beijing Key Laboratory of Intelligent
Software Engineering
Beijing, China
yanjun@ios.ac.cn

Abstract

Static Application Security Testing (SAST) tools play a key role in identifying vulnerabilities early in the development cycle. While these tools claim to detect a wide range of CWEs, their actual coverage is often unclear. In particular, due to language-specific characteristics, it is unclear to what extent SAST tools cover the CWEs most relevant to a given programming language, creating a gap between tool capabilities and developers' security needs. To address these questions, this experience paper presents a large-scale empirical investigation into the gap between *SAST Tools* and *Language-Specific Top CWEs*. Using a heuristic labeling approach on 197,016 CVEs reported over the past decade, we construct the **Top_{Lan} CWE Lists** for three widely used programming languages: C/C++, Java, and Python. Based on these datasets, we evaluate

nine popular SAST tools to identify coverage gaps and tool-specific performance characteristics.

Our study yields several key findings. **First**, the language-specific Top_{Lan} CWE lists differ a lot from the globally ranked CWE list, indicating that SAST tools need to focus on different types of vulnerabilities for different languages. **Second**, the whole SAST tool set now exhibits low coverage of the Top_{Lan} CWE list, highlighting the need for more targeted code checkers. **Third**, the covered Top_{Lan} CWE sets for different SAST tools vary substantially, underscoring the need to select and combine tools based on vulnerability types. **Finally**, for top CWEs shared across multiple languages, existing multi-language SAST tools may support detection in only one language, suggesting opportunities to enhance coverage by migrating existing checkers. These findings reveal fundamental challenges for static analysis in detecting certain types of vulnerabilities and provide insights for improving future SAST tool design. They offer practical guidance for practitioners in selecting and combining tools to detect language-specific vulnerabilities more effectively.

*Corresponding author.



CCS Concepts

• Security and privacy → Software security engineering.

Keywords

Static Analysis, Software Security, SAST Tool, CVE/CWE

ACM Reference Format:

Zhilin Li, Yiran Nie, Xianglong Qi, Jiwei Yan, and Jun Yan. 2026. Mind the Gap: Do Widely-Used SAST Tools Really Cover the Language-Specific Top CWEs?. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26)*, October 12–16, 2026, Munich, Germany. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3832783.3837435>

1 Introduction

Software security is a major concern, as vulnerabilities can lead to severe risks and security breaches [5, 23]. The Common Weakness Enumeration (CWE) [52] provides a global standard for categorizing software weaknesses, enabling researchers and developers to classify and analyze vulnerabilities across projects and platforms. In practice, Static Application Security Testing (SAST) tools are widely adopted to mitigate software vulnerabilities during development [7, 44] by automatically detecting CWE-defined weakness patterns. Thus, CWE serves as a knowledge base of vulnerability types, while SAST tools provide the primary approach for identifying these weaknesses in real-world systems.

Given the importance of SAST tools, recent studies have conducted realistic and comprehensive evaluations of their effectiveness [10, 17, 28, 29, 37]. While these studies assess the detection accuracy of supported CWEs, developers are more concerned with whether tools cover popular and project-relevant CWEs, how tools complement each other in coverage, and how detection capabilities can be improved for specific CWE types [16, 46, 49]. However, existing research provides limited insights into these practical concerns, lacking systematic evaluations of overall CWE coverage across tools. This limitation hinders effective tool selection and prevents researchers from identifying gaps in current detection approaches.

Another related question is which CWE types are the most severe and suitable for evaluation. Existing criteria often rely on general rankings, such as the CWE Top 25 [11] and OWASP Top 10 [20]. However, these rankings aggregate vulnerabilities across all programming languages and reflect overall prevalence rather than language-specific characteristics. Since different languages exhibit distinct vulnerability patterns, top CWEs vary by language. As SAST tools typically target specific languages, evaluating them with general CWE rankings may introduce mismatches between selected CWEs and language-specific weaknesses, reducing the accuracy and usefulness of evaluations for developers and researchers.

In this work, we present the first large-scale study on SAST tool coverage over language-specific top CWEs. We focus on three widely used languages: *C/C++*, *Java*, and *Python*. We first apply a heuristic language identification method to CVE entries and compute CWE frequencies for each language. Based on these results, we construct a practical CWE ranking, namely the **Top_{Lan} CWE List**, where $Lan \in \{C/C++, Java, Python\}$, and compare it with the general CWE Top 25 ranking. We then evaluate nine widely used SAST tools on the Top_{Lan} CWE lists, analyzing individual coverage and cross-tool/language coverage relationships. Finally, we provide practical recommendations for tool selection and future research. Our major findings are summarized as follows:

(1) The language-specific Top_{Lan} CWE lists differ significantly from the global CWE ranking. We construct Top_{Lan} CWE lists and compare them with the general CWE Top 25. The overlaps between Top_{C/C++} 25, Top_{Java} 25, Top_{Python} 25 and CWE Top 25 are only 0.17, 0.54, and 0.14, respectively. This indicates substantial deviations for C/C++ and Python, highlighting the need for language-specific SAST evaluation.

(2) Existing SAST tools provide limited coverage of Top_{Lan} CWEs. For C/C++, 43% of top CWEs remain uncovered. The uncovered cases fall into four categories: *dynamic data value dependence*, *project configuration and environment dependence*, *system timing and hardware interaction dependence*, and *complex input and sequence dependence*. Around 60% of these cases belong to the first two categories and may be addressed by combining static analysis with LLM-based or runtime-aware techniques, while the remaining cases remain challenging.

(3) SAST tools exhibit substantially different Top_{Lan} CWE coverage. CodeQL achieves the highest average coverage across languages (68.46%), which is 3.6 times higher than the lowest-performing tool. However, each tool covers unique vulnerabilities and none provides complete coverage, indicating the need for tool combination based on target vulnerability types.

(4) Multi-language SAST tools have opportunities to expand shared CWE coverage across languages. For shared top CWEs, some checkers support only one language. Our analysis shows that 47.7% of such checkers in CodeQL and 92.5% in Semgrep can potentially be migrated to additional languages, suggesting an effective way to improve multi-language CWE coverage.

In summary, this paper makes the following contributions:

- We performed a large-scale study to extract language-specific high-frequency CWEs for C/C++, Java, and Python, yielding the Top_{Lan}, i.e., Top_{C/C++}, Top_{Java}, and Top_{Python} CWE Lists. These results were derived by collecting 197,016 CVEs over the past ten years and analyzing 944 distinct CWE types, providing developers with guidance on selecting appropriate CWEs for evaluation.
- We compared the top 25 CWEs in the general ranking with those in the language-specific lists, revealing notable differences that can inform users' selection of relevant CWEs.
- We evaluated nine popular SAST tools using the Top_{Lan} CWE Lists to address three research questions (RQs), assessing the overall tool performance in terms of top CWE coverage, checker complementarity, and the potential for cross-language migration of checkers for shared CWEs.
- We summarized five major findings and provided practical recommendations for future tool development and usage. These insights highlight opportunities to improve SAST tools and assist developers and researchers in selecting the most effective approaches for detecting vulnerabilities across multiple programming languages.

2 Study Design

Analyzing SAST tool coverage of top CWEs involves three main steps. First, we identify the top CWEs relevant to each programming language and generate the corresponding Top_{Lan} CWE list. Second, we analyze the overall coverage of the Top_{Lan} CWE list by current SAST tools and investigate the reasons why certain CWEs remain

uncovered. Third, we examine coverage differences across tools and analyze how multi-language tools cover top CWEs shared across programming languages. Our research workflow aligns with these three components, each corresponding to one of our RQs.

2.1 Research Questions

To guide our empirical study, we formulate three RQs corresponding to the three key aspects of top CWE coverage by SAST tools:

- **RQ1:** What is the language-specific distribution of top CWEs in real-world vulnerabilities across C/C++, Java, and Python?
- **RQ2:** To what extent are top CWEs covered by existing SAST tool sets across different programming languages, and what causes any non-coverage?
- **RQ3:** For the coverable top CWEs, what differences exist in coverage across different SAST tools, and how do multi-language tools perform on top CWEs shared across programming languages?

In RQ1, we analyze CVE counts in real-world projects for each programming language, identify the top CWEs, and construct the corresponding Top_{Lan} CWE lists to analyze how language features and typical application scenarios shape their distribution. This analysis establishes a foundation for subsequent evaluation of SAST tools. In RQ2, we assess how SAST tools cover these top CWEs and investigate why some remain uncovered, providing insights into the limitations of current static analysis techniques. Together, RQ1 and RQ2 offer a comprehensive evaluation of SAST tool capabilities and inform strategies for tool improvement.

In RQ3, we focus on individual SAST tools to provide detailed guidance for users and developers. First, we compare different tools within the same language. We analyze differences in CWE coverage across tools based on their design and objectives. Second, for multi-language tools, we examine whether their coverage of top CWEs shared across languages is consistent, and whether transferring checkers for these CWEs can address coverage inconsistencies. The findings from RQ3 can guide users in selecting the most suitable tool and help developers improve tool coverage and effectiveness.

2.2 Evaluation Criteria

The CWE system organizes software weaknesses into a hierarchical structure. Each CWE may have multiple parent nodes, representing an inclusion relationship that abstracts properties such as causes, patterns, or triggers. CWE provides several views for different purposes. For example, View 700 [4] uses a broad conceptual category, but it does not support a systematic analysis of all CWEs. View 1340 [3] focuses on data protection and secure coding. View 1003 [2] is designed for human usability and only covers a small set of common weaknesses with a shallow structure. These views are not suitable for a complete evaluation of SAST tools because they lack coverage or depth. To solve this problem, we use CWE View 1000 [42] in our study. View 1000 includes 944 CWEs and organizes them by cause and pattern. Figure 1 shows a simplified version of this hierarchy. This structure allows us to systematically group and analyze related weaknesses. By following these relationships, we can group all CWEs into ten main categories. These ten root categories are listed in Table 1. Therefore, View 1000 is the best choice for a comprehensive and consistent evaluation of SAST tool coverage. In this study, we consider a CWE as covered if the tool

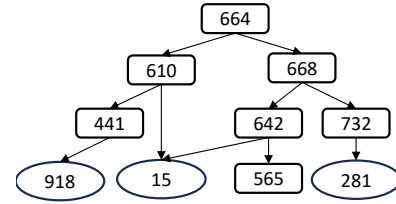


Figure 1: Part of Hierarchical Structure of CWE View 1000. Rectangular nodes denote non-leaf nodes, while elliptical nodes denote leaf nodes.

Table 1: Ten Root CWE Categories under CWE View 1000

CWE ID	CWE Name
CWE-284	Improper Access Control
CWE-435	Improper Interaction Between Multiple Correctly-Behaving Entities
CWE-664	Improper Control of a Resource Through its Lifetime
CWE-682	Incorrect Calculation
CWE-691	Insufficient Control Flow Management
CWE-693	Protection Mechanism Failure
CWE-697	Incorrect Comparison
CWE-703	Improper Check or Handling of Exceptional Conditions
CWE-707	Improper Neutralization
CWE-710	Improper Adherence to Coding Standards

provides at least one checker for it, and we also mark all parent and ancestor nodes in the CWE hierarchy as covered. Since a tool can hardly cover all instances of a CWE, our focus is on whether a CWE is addressed, emphasizing whether the tool recognizes or pays attention to it.

2.3 CVE Dataset

2.3.1 CVE Data Collection. We use CVE data to identify top CWEs. These CWEs require attention from SAST tools. MITRE maintains the CVE system and provides standardized IDs and descriptions for publicly known security vulnerabilities [52]. Many security studies use CVE because it links real-world bugs to specific CWE types. We use CVE frequency to estimate CWE frequency. High-frequency CWEs often correspond to the top CWEs for each language, as they affect more code in practice. Therefore, SAST tools should prioritize detecting these top CWEs.

CVE data and CWE labels come from two main sources: the MITRE Corporation CVE repository and the National Vulnerability Database (NVD) [43]. Both sources provide CWE labels for CVEs, but the number of labeled records differs significantly. For the period from 2016 to Nov. 10, 2025, the MITRE dataset contains 118,626 CVE entries with CWE labels, while the NVD dataset contains 197,016 entries with CWE labels. In addition to CWE annotations, NVD also provides more systematically curated vulnerability meta-data, including standardized CVSS severity scores [19] for a large portion of entries. Because NVD offers broader coverage and more structured records, we select it as the primary data source in this study. Using this dataset, we identify the top CWEs for each programming language, considering a CWE to be top if it appears more than ten times in the records for that language.

2.3.2 CVE Language Annotations. We observe that vulnerability distributions are influenced by both programming languages and application domains, while programming language serves as a more fundamental factor. This is because domain-specific characteristics are often implemented through language-specific ecosystems and frameworks, and SAST tools inherently operate

Table 2: Overview of nine SAST tools and their CWE coverage across different programming languages.

Type	Tool	Description	ID	Language	#Checker	#CWE
Multiple Language Tool	CodeQL	Datalog-based semantic engine for multi-language vulnerability detection.	4227dd7d733	C/C++	192	154
				Java	276	227
				Python	111	139
	Semgrep	Pattern-based engine using declarative rules for efficient vulnerability detection.	12f2eb5	C/C++	12	29
				Java	119	78
				Python	269	91
Single Language Tool	SonarQube	Continuous inspection platform for code quality and security analysis.	eee2f48	Java	79	143
				Python	65	93
	Cppcheck	Specialized analyzer for detecting C/C++ bugs and undefined behaviors.	554b98a	C/C++	90	54
	CSA	Symbolic execution-based tool for memory and state analysis.	2078da4	C/C++	14	25
	Flawfinder	Lexical scanner targeting risky API usage patterns.	56ffb9f	C/C++	71	44
	Insider	Detects common bug patterns in Java components.	20af38a	Java	71	46
	SpotBugs	Bytecode-level analysis for Java security vulnerabilities.	e38542b	Java	113	106
	Bandit	AST-based tool for detecting Python security issues.	e1ffdf6	Python	42	47

on language-specific code representations. However, existing CWE rankings [11, 20] are not organized by programming language. Therefore, it is necessary to identify the top CWEs for each language to better guide tool evaluation and improvement. A key challenge is that CVE records often do not explicitly specify the programming language used, and the large volume of CVEs makes manual annotation infeasible. To address this issue, we design a set of lightweight heuristic rules to automatically label each CVE with its corresponding implementation language. This approach enables efficient large-scale analysis of language-specific CWE distributions.

Our labeling process consists of three steps. First, we assign programming language labels using deterministic signals, including explicit file extensions mentioned in CVE descriptions. Second, we further identify languages based on explicit mentions of programming language names in the CVE text, which also serve as deterministic signals. These first two steps provide high-confidence, rule-based language assignments. Third, for CVEs not covered by the above deterministic signals, we apply a heuristic keyword-based analysis using 167 manually selected language-indicative terms derived from the 2,000 most frequent nouns in CVE descriptions.

Finally, we validate our detection process in two steps. First, we evaluate the accuracy of our labeling method. We randomly sample records from the already labeled dataset and manually review each sampled CVE by comparing our assigned CWE labels against evidence from the CVE description and associated references, thereby directly assessing the correctness of our labeling results. Second, we examine whether our labeling process preserves the distributional properties of the full dataset. We randomly sample records from the entire dataset and apply the same manual labeling procedure as in the first step. We then map the CWEs of CVEs across different programming languages to their corresponding ten root CWEs. Finally, we compare the distribution of languages and ten root CWEs

in the sampled dataset against those in the fully labeled dataset to assess whether our labeling method introduces systematic bias.

2.4 Studied Tools

Table 2 summarizes the SAST tools used in this study, the programming languages they support, the number of CWE checkers implemented, and the number of CWEs covered. We selected nine SAST tools that support C/C++, Java, and Python and represent commonly used static analysis approaches, including pattern matching, data-flow analysis, and symbolic execution. The tool selection was guided by three criteria. First, each tool must provide at least partial open-source access to its checker implementations, which is necessary to understand the underlying detection logic and support our analysis of coverage limitations and cross-language checker migration. Second, the tool must explicitly document or expose CWE annotations for its checkers to enable systematic comparison across tools. Third, we prioritized tools that are widely adopted in either academic studies or real-world development practice, based on prior literature [8, 10, 17, 24, 29, 37, 45, 47, 51, 53, 57]. Among the selected tools, SonarQube and Semgrep were used in their community editions, while SpotBugs was used with the FindSecBugs plugin. We do not include major commercial SAST tools because their internal checkers and rule definitions are not publicly accessible, which prevents systematic analysis of CWE coverage and checker-level behavior. Therefore, our results primarily reflect widely used open or partially open-source SAST tools. While tool documentation reflects the CWE coverage provided by each tool, it does not capture the gap between this coverage and the actual vulnerability needs observed in real-world software systems.

2.5 Manual Validation Protocol

To ensure rigor and reliability, we followed a structured annotation protocol. Two experienced developers and one domain expert first defined validation guidelines covering CVE language identification, criteria, and reasons for CWE coverage limitations, and rules for

Table 3: CVE Language Labeling and Validation Results

Language	CVE Count	Sample Distr.	Accuracy
C/C++	37,887 (40.47%)	113 (43.46%)	91.88%
Java	6,703 (7.16%)	37 (14.23%)	96.00%
Python	1,915 (2.05%)	6 (2.31%)	97.33%
Other	47,120 (50.33%)	104 (40.00%)	89.53%

checker migratability. For each instance, at least one developer independently reviewed and labeled the CVE, followed by verification from the domain expert. Any disagreements were resolved through joint discussion and majority voting among the three annotators.

3 Results of the Study

In this section, we present the results of our empirical study addressing the three research questions (RQs) defined in Section 2.1.

3.1 Language-Specific Distribution of Top CWEs

Validation of Heuristic Language Annotation. We first check the reliability of our method for labeling CVE languages. As shown in Table 3, our approach successfully labels 93,625 CVEs, accounting for 48.4% of the total dataset. Only 0.88% of all CVEs have conflicting labels. There are two main reasons for these conflicts. First, some CVE descriptions mention multiple languages. This often happens because the software uses a cross-language technology stack. In these cases, it is hard to decide which language is the primary one. Second, some software project names consist of several English words. Our tool may incorrectly split these names into separate keywords. This leads to conflict in language labels.

To verify whether the language distribution of our labeled CVEs is consistent with that of the overall dataset, we randomly selected 300 CVEs and manually labeled them. Our labeling process identifies programming languages using signals from CVE descriptions and associated references, including patches, commit diffs, bug reports, and repository metadata when available. Table 3 shows the results. CVEs labeled as belonging to closed-source projects are excluded, since there is insufficient information in these records to reliably identify the programming language. Overall, although the heuristic method leaves a larger proportion of CVEs unlabeled due to imperfect rules, the relative distribution across programming languages remains stable. In particular, C/C++ and other languages exhibit highly consistent patterns between the heuristic labels and the manually labeled sample, while Java shows a relatively lower proportion in the heuristic results, likely due to the diversity of Java project naming conventions that makes them harder to identify via file names or keywords.

We further examine whether our labeling process affects the CWE distribution within each programming language, as shown in Figure 2. The distributions of ten root CWE categories across the three main languages are largely consistent between the heuristic labeled dataset and the randomly sampled manually labeled subset. Although Python shows larger fluctuations due to its small sample size (only six CVEs), its dominant CWE peak remains unchanged. These results indicate that our labeling process has a limited impact on the underlying CWE distribution and introduces no substantial structural bias into the dataset.

Table 4: CVE Trends Across Languages (2016–Nov. 10, 2025)

Language	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025
C/C++	2696	4255	3884	3090	3081	4159	4328	3886	5407	3101
Java	392	334	465	709	669	748	921	856	738	871
Python	55	86	112	115	155	189	274	212	377	340
All	3143	4675	4461	3914	3905	5096	5523	4954	6522	4312

Finally, we evaluated the accuracy of our labeling method by randomly selecting 925 CVEs from the labeled dataset, ensuring balanced representation with at least 75 samples per language. Manual review, summarized in Table 3, shows that the method achieves over 89.5% accuracy for each language, with an overall average of 91.5%. These results demonstrate that our method is effective for automatic CVE labeling and supports large-scale analysis of CWEs across programming languages.

The reliability of these results is further supported by our stepwise design, in which deterministic labeling—based on file extensions and explicit language mentions—is applied first, followed by heuristic inference. These deterministic steps not only anchor the overall distribution but also contribute substantially to labeling accuracy, as they provide high-confidence assignments before introducing potentially noisier heuristics. Notably, deterministic labeling accounts for 52.17% of all labeled CVEs, forming a reliable foundation that mitigates potential bias and error propagation in the subsequent heuristic stage.

Finding 1: Our heuristic labeling method achieves high accuracy when mapping CVEs to programming languages, with an average accuracy exceeding 90%. The resulting labeled dataset preserves the distribution of both programming languages and their corresponding CWE patterns, enabling a reliable, large-scale analysis of language-specific top CWEs.

Language-Specific top CWEs. We first analyzed the ten-year trends of CVE counts for C/C++, Java, and Python, as shown in Table 4. Overall, reported CVEs are increasing, highlighting the growing importance of early vulnerability detection. The 2025 count appears lower because data collection ended on November 10. Among the three languages, C/C++ consistently exhibits the highest number of CVEs, reflecting its widespread use and vulnerability-prone nature. Java and Python also show steady growth over time. These trends underscore the need to strengthen SAST tools across all three languages. Figure 4 illustrates the distribution of CVEs across the ten root CWE categories in View 1000. Across C/C++, Java, and Python, two categories are consistently the most frequent: Improper Control of a Resource Through its Lifetime (CWE-664) and Improper Neutralization (CWE-707). The third most frequent category varies by language: for C/C++, it is Improper Adherence to Coding Standard (CWE-710), whereas for Java and Python, it is Improper Access Control (CWE-284).

Based on this CVE data, we generated the Top_{Lan} CWE list for each programming language, yielding 205 for C/C++, 123 for Java, and 86 for Python. To further analyze the most representative vulnerabilities, we extracted the Top 25 CWE lists for each language, denoted as $\text{Top}_{\text{C/C++}}$ 25, Top_{Java} 25, and $\text{Top}_{\text{Python}}$ 25. To quantitatively compare the language-specific CWE lists with the CWE Top

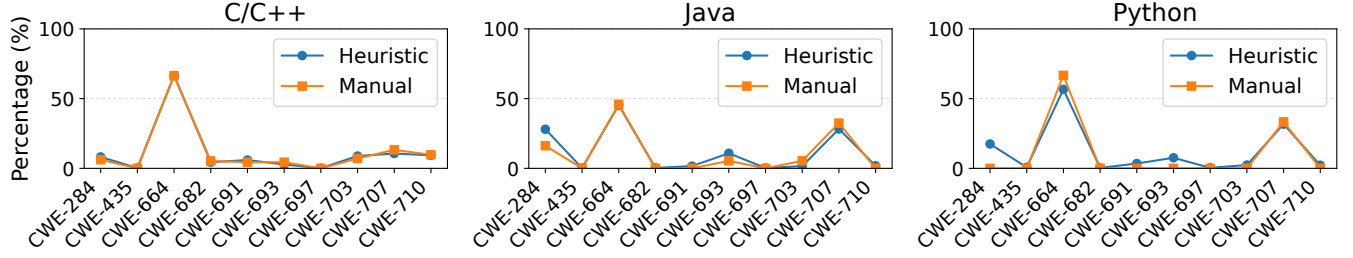


Figure 2: Distribution Alignment of Root CWEs Between Heuristic and Manual Labeling.

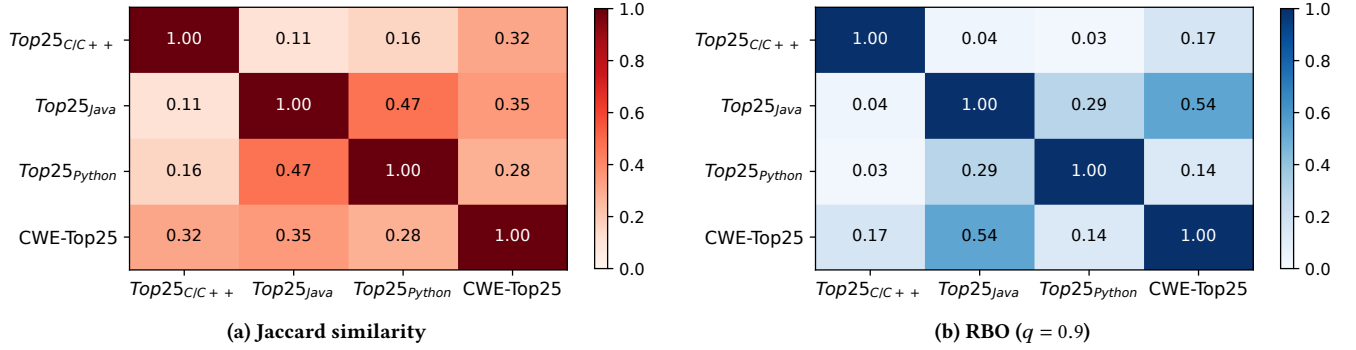
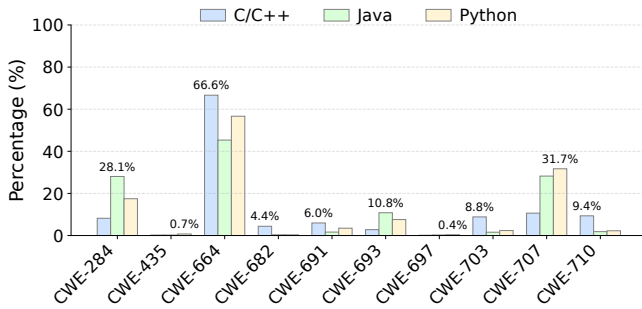
Figure 3: Comparison between language-specific Top 25 CWE lists and the CWE Top 25 benchmark. The Jaccard similarity (left) measures set-level overlap, while RBO ($q = 0.9$) (right) captures rank-aware similarity with emphasis on top-ranked CWEs.

Figure 4: Percentage of CVEs in ten Root CWE Categories for C/C++, Java, and Python

25 [52], we employ two complementary similarity metrics, namely Jaccard similarity and Rank-Biased Overlap (RBO).

The Jaccard similarity measures the overlap between two sets. Given two sets A and B , it is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad (1)$$

However, since both CWE lists are inherently ranked, we further adopt Rank-Biased Overlap (RBO) to account for top-weighted agreement between ranked lists. Given two ranked lists S and T , RBO is defined as:

$$\text{RBO}(S, T, q) = (1 - q) \sum_{d=1}^{\infty} q^{d-1} A_d \quad (2)$$

where $A_d = \frac{|S_d \cap T_d|}{d}$ denotes the overlap proportion at depth d , and $q \in (0, 1)$ is a persistence parameter that controls the weight assigned to deeper ranks.

In our study, we set $q = 0.9$, a commonly used value that emphasizes agreement at higher ranks [1, 12, 58].

As shown in Fig. 3, all language-specific Top 25 CWE lists overlap with the CWE Top 25, confirming that the benchmark partially captures vulnerability patterns across programming languages. However, when ranking is taken into account, Java shows much higher similarity to the CWE Top 25, suggesting that the benchmark ranking is largely influenced by Java. In contrast, C/C++ and Python show lower similarity, indicating that the benchmark does not well reflect their vulnerability distributions. This mismatch suggests that using a single CWE Top 25 list is insufficient to represent language-specific vulnerability characteristics and highlights the need to construct Top_{Lan} CWE lists.

From Figure 3, C/C++ shows clear differences from Java and Python, while Java and Python are more similar to each other, which is consistent with their language design.

In C/C++, vulnerabilities are dominated by low-level memory and resource management issues that rarely appear in managed languages. For example, Use-After-Free (CWE-416), Double Free (CWE-415), and Stack-based Buffer Overflow (CWE-121) highlight the risks introduced by manual memory management. These issues arise because pointer safety is not enforced, and object lifetimes

must be managed explicitly. As a result, small programming errors can lead to severe memory corruption and control flow hijacking.

In contrast, Java and Python share several vulnerabilities related to input handling and application logic. Deserialization of Untrusted Data (CWE-502) appears prominently in both languages due to their reliance on object serialization. Similarly, Server-Side Request Forgery (CWE-918) and Unrestricted Upload of File with Dangerous Type (CWE-434) reflect common risks in web-based systems where external inputs are not properly validated.

In Java, vulnerabilities are more closely related to the unsafe handling of external inputs and to framework features. Code Injection (CWE-94) and Improper Restriction of XML External Entity Reference (CWE-611) illustrate risks in processing dynamic content and external data. These issues show that vulnerabilities often arise from the misuse of high-level frameworks and data-processing mechanisms rather than from low-level errors.

For Python, the vulnerability profile reflects its dynamic features and scripting nature. Improper Encoding or Escaping of Output (CWE-116) highlights risks in string processing and input handling. In addition, Eval Injection (CWE-95) shows how dynamic execution features can introduce security risks. These issues are closely related to Python's flexibility and its common use in rapid development.

Overall, these distributions show that SAST tools must adopt language-specific strategies. For C/C++, tools should focus on memory safety and proper management of pointers and object lifetimes. For Java, they should pay attention to safe handling of external inputs and proper use of frameworks. For Python, detecting unsafe string handling and dynamic code execution is critical. These results demonstrate that a one-size-fits-all approach is insufficient; SAST tools should be designed and tailored according to the unique vulnerability characteristics of each programming language.

To further examine the impact of additional factors on the language-specific *TopLan* CWE lists, we incorporate CVSS-based severity into the ranking formulation using the same scoring methodology as that used in constructing the MITRE Top 25 CWE list [41]. This produces severity-weighted rankings that combine vulnerability prevalence and CVSS severity scores. A comparison between the frequency-based *TopLan* rankings and their severity-weighted counterparts shows that incorporating CVSS severity introduces observable shifts, while largely preserving the overall ranking structure within each programming language. The Jaccard similarity reaches 0.67 for C/C++, 0.56 for Java, and 0.56 for Python, while the RBO values are 0.55, 0.62, and 0.59, respectively. Further analysis shows that the average CVSS score of the high-frequency CWEs exceeds 4.98, suggesting that these vulnerabilities generally fall into the medium-to-high severity range. This confirms that high-frequency CWEs are also security-critical and warrant systematic attention in SAST tool design. Overall, the results suggest that different prioritization strategies can be flexibly adopted depending on practical needs. For example, frequency-based rankings are more suitable for guiding broad vulnerability coverage and maximizing detection impact in large-scale codebases, while severity-weighted rankings are more appropriate for risk-sensitive scenarios, such as security auditing or vulnerability triage, where both prevalence and potential impact need to be considered.

Finding 2: The fact that the *TopLan* CWE lists for different programming languages differ from the globally ranked CWE list, with RBOs of $\text{Top}_{C/C++}$ and Top_{Python} against CWE Top 25 not exceeding 0.2, underscores the necessity for SAST tools to adopt tailored detection strategies. Evaluations of these tools should also be based on language-specific top CWEs rather than a universal ranking.

3.2 Top CWE Coverage Gaps in SAST Tools

Figure 5 presents the coverage of the *TopLan* CWE lists by SAST tools for C/C++, Java, and Python. Overall coverage is 56.59% for C/C++, 78.86% for Java, and 86.05% for Python. Python's higher coverage is partly due to its smaller *TopLan* CWE lists. Nevertheless, coverage remains insufficient across all languages, highlighting the need to expand detection capabilities. Moreover, substantial differences in coverage among the three languages indicate that current SAST tools provide uneven protection, and the security assurance depends strongly on the programming language under analysis. When focusing further on the top three root CWE categories, C/C++ consistently shows the lowest coverage. This highlights a major gap in tool support for the most critical C/C++ vulnerabilities. Therefore, improving coverage for C/C++ must be a high priority.

The above analysis adopts a permissive definition of CWE coverage, where a CWE is considered covered as long as a tool provides at least one corresponding checker. Under this definition, the reported results provide a macro-level view of whether a given vulnerability type is within the scope of existing SAST tools, and further highlight important CWE categories that remain unsupported by current tooling. Beyond coverage, an important complementary question is how effectively these covered checkers perform in practice. Existing empirical studies provide evidence that static analysis tools can detect a subset of defects within the CWEs they are designed to support [6, 10, 15, 29, 34, 37, 45, 63]. However, these results also indicate that their detection capability in real-world settings remains limited. In particular, among the tools analyzed in our study, Flawfinder is reported as the best-performing tool for C/C++ in the latest study, achieving a detection rate of only 31% [10]. SpotBugs is the best-performing tool for Java [29], with a detection rate of 10%, and CodeQL is the best-performing tool for Python, achieving 11% [45]. These findings suggest that, even when considering the strongest tool within each study, SAST tools still exhibit substantially constrained effectiveness in real-world software systems. This discrepancy indicates that, although coverage enables the possibility of detection, the practical effectiveness of existing checkers remains limited in realistic development environments. Therefore, improving SAST tools requires both expanding CWE coverage and enhancing the robustness of existing detection logic to better generalize from benchmark settings to real-world code.

Finding 3: The whole SAST tool set now exhibits low coverage of the *TopLan* CWE list. C/C++ shows the lowest coverage among the three languages, with an overall average of 56.59%. These findings show that enhancing C/C++ bug detection must be a critical priority for tool developers.

We performed a manual analysis to understand why these top CWEs are not covered by SAST tools. In cases where both a CWE

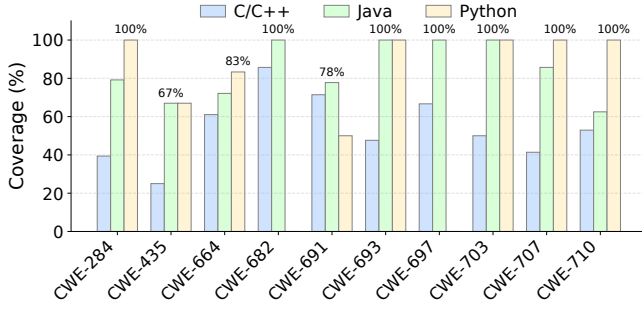


Figure 5: Coverage of Top CWE Categories under ten Root CWEs by SAST tools

and its parent nodes are uncovered, we only consider the parent node. We identified four main reasons for this lack of coverage:

- (1) *Dynamic Data Value Dependence*: Some vulnerabilities depend on runtime variable values, computation results, or memory states, which SAST tools cannot reliably predict across all execution paths, like Uncontrolled Recursion (CWE-674).
- (2) *Project Configuration and Environment Dependence*: Program security and correctness often depend on project-specific business logic, security policies, and unique factors such as industry standards, compliance rules, or system configurations. SAST tools struggle to define dedicated checkers for individual projects, such as Privilege Defined With Unsafe Actions (CWE-267).
- (3) *System Timing and Hardware Interaction Dependence*: Some vulnerabilities stem from dynamic execution factors, including thread scheduling, event ordering, and interactions among components. Static analysis cannot reliably model such physical or hardware states, such as Improper Control of Interaction Frequency (CWE-799).
- (4) *Complex Input and Sequence Dependence*: Certain vulnerabilities are triggered by specific external inputs or event sequences. Static analysis cannot exhaustively examine all input combinations, data encodings, or event orders, making it difficult to ensure correct program behavior and to prevent logic errors or code injection, such as Improper Handling of Values (CWE-229).

Table 5 summarizes the main reasons why top CWEs remain uncovered by SAST tools across programming languages. Among these, *Project Configuration and Environment Dependence* is the most significant blind spot, appearing consistently across all three languages and especially in C/C++. Most SAST tools focus on general code patterns, whereas vulnerabilities in this category often require an in-depth understanding of project-specific configurations or runtime environments. *Complex Input and Sequence Dependence* is another critical challenge, particularly for C/C++, where it is more prevalent than in Java or Python. Vulnerabilities often arise in contexts such as binary parsing or protocol handling. Since C/C++ is widely used for low-level parsers, drivers, and high-performance servers, programs frequently rely on the correct processing of specific input sequences, making such defects difficult for general-purpose static analyzers to detect. Finally, *Dynamic Data Value Dependence* is relatively limited in Python, despite its dynamic

Table 5: Uncovered Top CWEs: Reason Counts

Language	Dynamic	Project	System	Input
C/C++	10	31	7	17
Java	4	6	4	3
Python	3	1	3	2

typing, suggesting that existing Python analyzers already incorporate effective rules for handling certain runtime features. This also implies that Python vulnerabilities are less often dependent on complex memory states or deeply nested runtime values than those in C/C++.

Recent research leveraging Large Language Models (LLMs) offers promising solutions for *Project Configuration and Environment Dependence*, which arises from the unique rules and settings of each project. LLM-based systems can enhance tool understanding of project-specific details [21, 33, 36, 62]. Similarly, dynamic symbolic execution and LLM-augmented approaches address *Dynamic Data Value Dependence*, overcoming traditional static analysis limitations by more accurately predicting runtime program behavior [9, 31, 39]. In contrast, *System Timing and Hardware Interaction Dependence* and *Complex Input and Sequence Dependence* remain challenging, with current tools detecting a few isolated cases. These gaps highlight the need for further research and the development of systematic static analysis techniques to address these complex vulnerabilities.

Finding 4: Missed top CWEs occur primarily in C and C++. These gaps stem from *Project Configuration and Environment Dependence*, *Complex Input and Sequence Dependence*, and *Dynamic Data Value Dependence*. About 60% of these gaps can be addressed by hybrid methods using LLMs and runtime information, while *System Timing and Hardware Interaction Dependence* and *Complex Input and Sequence Dependence* remain largely unresolved challenges.

3.3 CWE Coverage Variation Across SAST Tools

While RQ2 assesses the overall capabilities of SAST tools, RQ3 offers a more detailed comparison of individual tools, examining CWE coverage from two specific perspectives:

Coverage Differences in the Same Language: We examine variations in top CWE coverage among tools for a single programming language, providing guidance for selecting the most appropriate static analysis tool for specific CWE types.

Coverage Differences in the Same Tool: For multi-language tools, we assess how shared top CWE coverage varies across languages, providing insight into the potential coverage capabilities of these tools.

3.3.1 Coverage Differences in the Same Language. As shown in Table 6, we summarize the coverage of top CWEs by SAST tools for C/C++, Java, and Python. For C/C++, the tools collectively cover 116 top CWEs. CodeQL achieves the highest coverage (52.68%), while other tools achieve substantially lower coverage: Flawfinder (19.51%), Cppcheck (16.59%), Semgrep (10.73%), and CSA (8.29%). For Java, the tools together cover 97 top CWEs. CodeQL again provides the highest coverage (74.80%), followed by SpotBugs (52.85%), SonarQube (49.59%), Semgrep (46.34%), and Insider (29.27%). For Python, the tools collectively cover 74 top CWEs. CodeQL covers 77.91%, followed by Semgrep (69.77%), SonarQube (54.65%), and

Table 6: Coverage and Tool-Only Coverage of Top CWEs by SAST Tools across Languages

Language	Tool	Covered CWEs	Covered Top CWEs	Tool-Only CWEs	Tool-Only Top CWEs
C/C++	CodeQL	156	108	78	45
	Semgrep	28	22	3	2
	Cppcheck	52	34	8	2
	CSA	24	17	5	1
	Flawfinder	47	40	4	3
Java	CodeQL	220	92	71	12
	Semgrep	77	57	2	0
	SonarQube	142	61	27	3
	Insider	44	36	1	0
	SpotBugs	106	65	17	1
Python	CodeQL	137	67	36	6
	Semgrep	89	60	15	4
	SonarQube	92	47	13	0
	Bandit	45	36	6	2

Bandit (41.86%). Across all three languages, no tool completely subsumes another’s coverage. Most tools can detect CWEs that other tools cannot, highlighting complementary coverage among them. This complementarity becomes even more evident when considering coverage of all CWEs. Overall, CodeQL provides the widest coverage, likely due to its support for global data-flow analysis and taint tracking, while other tools offer specialized coverage based on their analysis strategies. We next describe the specific features of each tool.

Five tools provide complementary coverage for C/C++ through different analysis paradigms. CodeQL uses a Datalog-based semantic engine for global data-flow reasoning and taint tracking, tracing attacker-controlled inputs across modules to detect multi-stage vulnerabilities such as Absolute Path Traversal (CWE-36). Semgrep performs rapid AST-based pattern matching to identify missing validations and characteristic code patterns, including Allocation of File Descriptors or Handles Without Limits or Throttling (CWE-774). Flawfinder applies lightweight lexical analysis by scanning risky APIs, efficiently detecting issues such as Inclusion of Functionality from Untrusted Control Sphere (CWE-829). Cppcheck uses localized value-flow tracking with path-sensitive reasoning to estimate variable states and expose low-level errors such as Signed-to-Unsigned Conversion Error (CWE-195). CSA leverages symbolic execution and constraint solving to analyze pointer interactions and boundary conditions, detecting issues such as Return of Pointer Value Outside of Expected Range (CWE-466). Accordingly, CodeQL is suitable for long-range data-flow analysis, Semgrep for local pattern enforcement, Flawfinder for rapid API misuse detection, Cppcheck for function-level resource analysis, and CSA for subtle pointer and boundary verification.

In the Java ecosystem, five tools address vulnerabilities by targeting language features, libraries, and runtime characteristics. CodeQL applies global data-flow analysis and taint tracking to trace attacker-controlled inputs across classes and modules, detecting issues such as Improper Restriction of XML External Entity Reference (CWE-611). Semgrep uses rule-driven AST pattern matching to identify structural vulnerability patterns without full compilation, including Improper Neutralization of Escape, Meta, or Control

Sequences (CWE-150). SonarQube combines AST analysis with localized data-flow reasoning to reveal risks such as Creation of Temporary File in Directory with Insecure Permissions (CWE-379). SpotBugs analyzes Java bytecode and simulates JVM operand stacks to uncover issues such as Use of Hard-coded Cryptographic Key (CWE-321). Insider targets mobile environments and detects Android configuration weaknesses, including Execution with Unnecessary Privileges (CWE-250). Accordingly, CodeQL is suitable for cross-class data-flow analysis, Semgrep for local pattern checking, SonarQube for code robustness analysis, SpotBugs for bytecode-level risks, and Insider for mobile security auditing.

For Python, four tools provide specialized analysis for its dynamic scripting environment. CodeQL builds comprehensive program representations and traces attacker-controlled inputs across modules to detect complex vulnerabilities such as Argument Injection (CWE-88). Semgrep applies pattern-based detection suited to Python’s dynamic nature, identifying structural misuse such as Static Code Injection (CWE-96). SonarQube extends semantic analysis to uncover robustness-related issues, including Generation of Error Message Containing Sensitive Information (CWE-209). Bandit uses Python AST parsing to scan dangerous APIs and insecure configurations, detecting issues such as Download of Code Without Integrity Check (CWE-494). Accordingly, CodeQL is suitable for cross-module data-flow analysis, Semgrep for structural vulnerability detection, SonarQube for information leak analysis, and Bandit for lightweight security checks.

Finding 5: The covered Top_{Lan} CWE sets across different SAST tools vary substantially, with the tool providing the most coverage achieving about 3.6 times the coverage of the tool with the least coverage. These differences are driven by diverse design principles. Since no single tool provides complete coverage of CWEs, practitioners should select or combine tools based on the target CWE categories to be detected.

3.3.2 Coverage Differences in the Same Tool. In the previous section, we discussed how SAST tools specialize in detecting specific CWE categories, largely due to differences in their technical implementations. At the same time, we observe the increasing presence of cross-language tools such as CodeQL and Semgrep, which apply similar detection techniques across multiple programming languages. In principle, adapting existing detection logic to a new language is often easier than developing entirely new checkers. Motivated by this observation, we analyze differences in the coverage of shared top CWEs across the languages of these tools. We further examine whether detection logic can be migrated between languages. A checker is considered migratable if its detection rules remain valid for the target language and all required code dependencies and libraries are available in that language’s analysis environment. To guide our study, we define four key concepts. First, a missing CWE is a top CWE that a tool does not cover in a target language but is covered in at least one other language. The corresponding detection checker is called a missing checker. Second, a migratable CWE is a missing CWE where the missing checker can be adapted to the target language. In these cases, the checker is called a migratable checker.

Table 7: Counts of Missing and Migratable CWEs and Checkers Across Languages for CodeQL and Semgrep

Language	CodeQL				Semgrep			
	Missing CWEs	Migr.	Missing Checkers	Migr.	Missing CWEs	Migr.	Missing Checkers	Migr.
C/C++	23	13	51	35	26	23	124	114
Java	6	3	22	5	5	5	9	9
Python	4	2	15	2	1	1	1	1

As shown in Table 7, C/C++ has the largest set of shared uncovered CWEs, followed by Java and Python, indicating substantial potential for coverage expansion through rule reuse. Both CodeQL and Semgrep demonstrate high overall transferability, confirming the feasibility of cross-language checker migration. Notably, Semgrep exhibits a higher transferability ratio than CodeQL, implying that pattern-based checkers are easier to transfer than semantically complex ones. These results suggest that checker migration is a practical way to enhance SAST coverage, with C/C++ benefiting the most.

To understand migration effort, we further analyze the migration characteristics of checkers and summarize the results in Table 8. At the rule level, about 52% of CodeQL rules are directly reusable across languages, requiring no semantic rewriting. For example, the `HeaderInjection.q1` rule relies on language-independent data flow logic that can be reused across different programming languages. In contrast, only around 32% of Semgrep rules can be migrated directly, as they are more syntactically concrete and often require partial re-modeling across languages, as illustrated by `java-pattern-from-string-parameter`. At the checker level, only around 38% and 36%, respectively, can be migrated with low cost through direct query adaptation or lightweight extension of language-specific API models. For example, `HeaderInjection.q1` can be migrated by mapping equivalent HTTP APIs, while more complex checkers such as `SensitiveDataExposureThroughErrorMessage.q1` require additional modeling of framework-specific behaviors such as exception handling and logging.

The remaining checkers are non-migratable for two reasons. First, some rely on platform-specific semantics that cannot be abstracted, such as `UnsafeAndroidAccess.q1`, which depends on Android `WebView` and `JavaScript` interaction models absent in C/C++. Second, others are prohibitively expensive to migrate. For instance, `UrlForward.q1` depends on servlet-based dispatch semantics and would require reconstructing comparable web routing abstractions in C/C++. Both categories account for a non-trivial proportion, with semantically non-migratable checkers being relatively more prevalent, representing approximately 66% of all non-migratable cases.

Overall, checker migration is more efficient than building rules from scratch and provides a practical way to extend CWE coverage. C/C++ shows the highest untapped potential, while Semgrep achieves higher migration rates than CodeQL due to its pattern-based design.

Finding 6: In multi-language tools, each language has top CWEs that lack local coverage but are supported by existing checkers in other languages. About 75% of checkers can be

Table 8: Fine-Grained Classification of Checker Migration Feasibility. Notes: Direct Rule Migr. (DR) denotes rule reuse without synonymous structural substitution; Query-only Migr. (QO) requires only rewriting queries without expanding libraries; Semantically Non-Migr. (SN) indicates language-exclusive semantic mismatches; Prohibitive Lib Migr. (PL) implies migration is blocked by the high costs of rebuilding underlying libraries.

Language	CodeQL				Semgrep			
	DR	QO	SN	PL	DR	QO	SN	PL
C/C++	21	15	7	9	39	43	8	2
Java	1	1	14	3	1	1	0	0
Python	0	0	8	5	0	1	0	0

migrated, indicating a possible direction for improvement. This is particularly true for C/C++, which has the most migratable checkers.

4 Threats To Validity

Our study has several threats to validity that may affect the accuracy of our findings. First, the language labels for the CVE dataset are incomplete and might be biased. We used a heuristic labeling method that could only assign a language to about half of the CVEs. Therefore, the coverage results in this study might differ slightly from those of the full dataset. This limitation could affect how well our findings represent all CVEs. To solve this in future work, we plan to use more advanced methods. For example, we could use LLMs for automated reasoning or extract more information from CVE references. These steps will help improve the completeness and accuracy of language labels.

Second, our top CWE evaluation relies only on CWE frequency and does not consider composite metrics such as CVSS [18]. Every CWE represents a real security vulnerability, so the minimum severity is guaranteed. Frequency helps SAST tools detect as many vulnerabilities as possible early. Future work may design new top CWE rankings based on different criteria, but the core idea of language-specific ranking remains crucial. Furthermore, as programming languages evolve with new security features, the frequency of specific CWEs will shift. Thus, future iterations of *Top_{Lan}* rankings must also dynamically adapt to this language evolution, maintaining their relevance over time.

Third, we used a relatively permissive evaluation strategy for both CWE coverage and checker transferability. For CWE coverage, we consider a target CWE covered if any checker detects it or one of its descendant nodes. This approach does not account for the actual depth or thoroughness of coverage in the target language. Similarly, for checker transferability, we only assessed whether a rule exists in the target language and whether its detection logic can be migrated. We did not analyze how widely or effectively the rule applies in practice. While this approach provides an initial estimate, it may overstate the actual coverage and transferability. In future work, we plan to refine our evaluation by developing more detailed metrics. These metrics will account for completeness, precision, and practicality of both CWE coverage and checker migration.

5 Implications

Based on our findings, we derive several implications that can inform the practical use and future development of SAST tools for vulnerability detection.

(1) Tailoring SAST Tools and their evaluations to language-specific top CWEs is necessary. CVE counts in Python and Java have steadily increased over the past decade, while C/C++ remains consistently high. Each language also exhibits distinct top CWEs, suggesting that tool developers should design checkers tailored to each language's top CWEs. Similarly, SAST tool evaluation should not rely on a universal CWE ranking but instead use language-specific top CWE lists to ensure fair and relevant assessment.

(2) Improving tool support for C/C++ is a high priority. SAST tools achieve the lowest top CWE coverage for C/C++, highlighting a critical gap in vulnerability detection. Given the consistently high number of C/C++ CVEs, tool developers should prioritize improving C/C++ SAST tool coverage. Practitioners should also be aware of these limitations and consider combining multiple tools or complementary techniques to mitigate coverage gaps.

(3) Combining static analysis with LLM-based and dynamic techniques is promising. SAST tools leave many top CWEs uncovered, primarily due to four types of limitations: *project configuration and environment dependence*, *complex input dependence*, *sequence dependence*, and *dynamic data value dependence*. LLM-based methods and dynamic symbolic execution can address some of these challenges, suggesting that combining static analysis with LLM reasoning and dynamic techniques is a promising direction. However, vulnerabilities related to system timing and highly complex inputs remain largely unresolved, highlighting the need for further research to improve detection in these difficult scenarios.

(4) Tool selection should be goal-driven instead of relying on one single tool. Since no single SAST tool achieves complete coverage, practitioners should establish a decision matrix based on our *Top_{Lan}* lists. For example, developers should first deploy a semantic engine like CSA for long-range control-flow tracking, and then integrate a lightweight pattern-matching tool like Cppcheck for enforcing local coding patterns and addressing configuration-related blind spots.

(5) Leveraging checker migration across languages can effectively improve tool coverage, especially for C/C++. To efficiently extend vulnerability coverage, particularly for C/C++, developers should execute a targeted migration strategy. First, prioritize checkers that support both direct rule reuse and simple query adaptation. Second, prioritize those requiring semantic-equivalent transformation but still allowing simple query adaptation. Third, consider checkers that support direct rule reuse but require additional component modeling. Fourth, handle checkers that require both semantic-equivalent transformation and additional component modeling. Finally, exclude checkers that depend on non-migratable platform-specific semantics or require prohibitively expensive architectural mappings.

6 Related Work

Traditional Static Analysis Enhancements. Prior research has improved the precision of static analysis through advanced pointer and taint analyses, hybrid methods, and domain-specific detection techniques. Key advances include unification-based pointer

analysis to mitigate memory oversharing [27] and the integration of dynamic execution data or heap snapshots to optimize accuracy [13, 22]. Other studies target specific behaviors, such as input usage via abstract interpretation [54], C++ sequence containers [35], exception handling [61], and smart pointer management [40]. For complex vulnerabilities, researchers leverage trust domains for supply-chain security [48], polymorphic call chains for deserialization detection [30], and type-based taint inference [26]. Unlike these specialized efforts, our study provides a holistic evaluation to identify the top CWEs that remain uncovered in each programming language, offering guidance for future tool improvements.

LLM-Enhanced Static Analysis. Recent research integrates Large Language Models (LLMs) into static analysis to improve precision, scalability, and coverage [14, 25, 33, 38, 50, 55, 56, 59, 60]. LLMs are employed to filter false positives through path feasibility analysis [14] and structured code-fact processing [25], or to perform compilation-free path verification via chain-of-thought guided dataflow analysis [56]. Specialized frameworks leverage LLMs to bridge semantic gaps in firmware [60], improve assembly-level taint tracking [38], and synthesize zero-day checkers from historical patches [59]. Additionally, LLMs facilitate project-level analysis by inferring global taint specifications [33], identifying API usage intentions [55], and decomposing complex smart contract logic [50]. Building on these advances, our work shows that most vulnerability categories currently missed by traditional tools require a hybrid LLM-static analysis for effective detection.

Empirical Studies of SAST tools. Empirical studies have evaluated SAST effectiveness across various programming languages [8, 10, 17, 24, 29, 37, 45, 47, 51, 53, 57]. For C/C++, researchers investigated open-source tool viability [17], benchmarked performance against the Toyota ITC suite [8], and revealed high false negative rates in real-world projects compared to synthetic benchmarks [37]. Others assessed the utility of a tool for prioritizing code reviews [10]. Comparative Java studies highlighted coverage gaps in null pointer exceptions [29, 53], while Python evaluations identified supply chain and environment-dependent defects [45, 47]. Further work explored multi-language taint analysis [57], incremental CodeQL analysis [51], and a comprehensive assessment of 246 security analyzers [24]. Building on these, our work provides a holistic evaluation across a large CWE set and generates language-specific top CWE lists.

7 Conclusion

This work investigates the gap between top CWEs and SAST tool coverage. We analyze ten years of CVE data across C/C++, Java, and Python to construct *Top_{Lan}* CWE lists and evaluate major SAST tools. Our results reveal a clear mismatch: C/C++ has the largest number of top CWEs but the lowest tool coverage. Most uncovered CWEs involve project configuration, environment dependencies, and complex input sequences, highlighting the limitations of traditional static analysis and the need for hybrid approaches combining static, dynamic, and LLM-assisted techniques. We further identify two practical directions: combining multiple tools to improve vulnerability coverage, and migrating shared CWE checkers across languages to reuse mature detection logic, especially for C/C++. This study provides insights into CWE distributions and SAST gaps, guiding future tool development and vulnerability management.

Data Availability

Our data is available in an open repository at <https://doi.org/10.5281/zenodo.19122421> [32]. Meanwhile, we have also presented our dataset and experimental results on an anonymous website at <https://language-specific-top-cwe.netlify.app/> for better accessibility.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions. This work is supported by the National Natural Science Foundation of China (NSFC) under grant No.62132020.

References

- [1] 2024. pyterrier-alpha 0.12.1. <https://pypi.org/project/pyterrier-alpha/0.12.1/>. Accessed: 2026-06-30.
- [2] 2025. CWE View 1003: Weaknesses for Simplified Mapping of Published Vulnerabilities. View page on the Common Weakness Enumeration (CWE) website. <https://cwe.mitre.org/data/slices/1003.html> <https://cwe.mitre.org/data/slices/1003.html>.
- [3] 2025. CWE View 1340: CISQ Data Protection Measures. View page on the Common Weakness Enumeration (CWE) website. <https://cwe.mitre.org/data/slices/1340.html> <https://cwe.mitre.org/data/slices/1340.html>.
- [4] 2025. CWE View 700: Seven Pernicious Kingdoms. View page on the Common Weakness Enumeration (CWE) website. <https://cwe.mitre.org/data/slices/700.html> <https://cwe.mitre.org/data/slices/700.html>.
- [5] Mamdouh Alenezi and Mohammad Zarour. 2020. On the Relationship between Software Complexity and Security. *CoRR abs/2002.07135* (2020). arXiv:2002.07135 <https://arxiv.org/abs/2002.07135>
- [6] Midya Alqaradaghi and Tamás Kozsik. 2024. Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code. *IEEE Access* 12 (2024), 55824–55842. doi:10.1109/ACCESS.2024.3389955
- [7] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2024. "False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 3979–3997. doi:10.1109/SP54263.2024.00019
- [8] Andrei Arusoae, Ștefan Ciobăcă, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2017. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2017, Timisoara, Romania, September 21-24, 2017*, Tudor Jelebelean, Viorel Negru, Dana Petcu, Daniela Zaharie, Tetsuo Ida, and Stephen M. Watt (Eds.). IEEE Computer Society, 161–168. doi:10.1109/SYNASC.2017.00035
- [9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
- [10] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An Empirical Study of Static Analysis Tools for Secure Code Review. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, Maria Christakis and Michael Pradel (Eds.). ACM, 691–703. doi:10.1145/3650212.3680313
- [11] MITRE Corporation. 2023. CWE Top 25 Most Dangerous Software Weaknesses. <https://cwe.mitre.org/top25/index.html>. Accessed: 2026-02.
- [12] Matteo Corsi and Julián Urbano. 2024. The Treatment of Ties in Rank-Biased Overlap. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2024, Washington DC, USA, July 14-18, 2024*, Grace Hui Yang, Hongning Wang, Sam Han, Claudia Hauff, Guido Zuccen, and Yi Zhang (Eds.). ACM, 251–260. doi:10.1145/3626772.3657700
- [13] Dipanjan Das, Priyanka Bose, Aravind Machiry, Sebastiano Mariani, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2022. Hybrid Pruning: Towards Precise Pointer and Taint Analysis. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 19th International Conference, DIMVA 2022, Cagliari, Italy, June 29 - July 1, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13358)*, Lorenzo Cavallaro, Daniel Gruss, Giancarlo Pellegrino, and Giorgio Giacinto (Eds.). Springer, 1–22. doi:10.1007/978-3-031-09484-2_1
- [14] Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, ZuoYu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. 2025. Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis. *CoRR abs/2506.10322* (2025). arXiv:2506.10322 doi:10.48550/ARXIV.2506.10322
- [15] Richard A. Dubniczky, Krisztofer Zoltán Horvát, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C. Cordeiro, and Norbert Tihanyi. 2025. CASTLE: Benchmarking Dataset for Static Code Analyzers and LLMs Towards CWE Detection. In *Theoretical Aspects of Software Engineering - 19th International Symposium, TASE 2025, Limassol, Cyprus, July 14-16, 2025, Proceedings (Lecture Notes in Computer Science, Vol. 15841)*, Philipp Rümmer and Zhilin Wu (Eds.). Springer, 253–272. doi:10.1007/978-3-031-98208-8_15
- [16] EricLambrecht and Contributors. 2019. *react/prop-types breaks when propTypes use spread operator #2195*. <https://github.com/jsx-eslint/eslint-plugin-react/issues/2195> GitHub issue on eslint-plugin-react discussing a bug related to react/prop-types and propTypes spread operators..
- [17] Valdeclébio Farrapo, Emanuel Rodrigues, José Maria Monteiro, and Javam C. Machado. 2025. Evaluating the Use of Open-Source and Standalone SAST Tools for Detecting Vulnerabilities in C/C++ Projects. In *Proceedings of the 27th International Conference on Enterprise Information Systems, ICEIS 2025, Porto, Portugal, April 4-6, 2025, Volume 1*, Joaquim Filipe, Michal Smialek, Alexander Brodsky, and Slimane Hammoudi (Eds.). SCITEPRESS, 394–401. doi:10.5220/0013483500003929
- [18] FIRST. 2019. Common Vulnerability Scoring System v3.1: Specification Document. <https://www.first.org/cvss/specification-document>. Accessed: 2026-03-27.
- [19] FIRST.org. 2019. Common Vulnerability Scoring System v3.1 Specification. <https://www.first.org/cvss/>
- [20] OWASP Foundation. 2021. OWASP Top 10: The Ten Most Critical Web Application Security Risks. <https://owasp.org/www-project-top-ten/>. Accessed: 2026-02.
- [21] Jiaqi Gao, Zijian Zhang, Yuqiang Sun, Ye Liu, Chengwei Liu, Han Liu, Yi Li, and Yang Liu. 2026. LogicScan: An LLM-driven Framework for Detecting Business Logic Vulnerabilities in Smart Contracts. *CoRR abs/2602.03271* (2026). arXiv:2602.03271 doi:10.48550/ARXIV.2602.03271
- [22] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the heap: ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 198–208. doi:10.1145/3213846.3213860
- [23] Assane Gueye, Carlos E. C. Galhardo, Irena Bojanova, and Peter Mell. 2021. A Decade of Reoccurring Software Weaknesses. *IEEE Secur. Priv.* 19, 6 (2021), 74–82. doi:10.1109/MSEC.2021.3082757
- [24] Kevin Hermann, Sven Peldszus, and Thorsten Berger. 2026. Many Tools, Few Exploitable Vulnerabilities: A Survey of 246 Static Code Analyzers for Security. *CoRR abs/2602.18270* (2026). arXiv:2602.18270 doi:10.48550/ARXIV.2602.18270
- [25] Mohsen Iranmanesh, Sina Moradi Sabet, Sina Marefat, Ali Javidi Ghasr, Allison Wilson, Iman Sharafaldin, and Mohammad A. Tayebi. 2025. ZeroFalse: Improving Precision in Static Analysis with LLMs. *CoRR abs/2510.02534* (2025). arXiv:2510.02534 doi:10.48550/ARXIV.2510.02534
- [26] Nima Karimipour, Kanak Das, Manu Sridharan, and Behnaz Hassanshahi. 2025. Practical Type-Based Taint Checking and Inference. In *39th European Conference on Object-Oriented Programming, ECOOP 2025, Bergen, Norway, June 30 - July 2, 2025 (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:25. doi:10.4230/LIPICS.ECOOP.2025.18
- [27] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. 2019. Unification-based Pointer Analysis without Oversharing. In *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, Clark W. Barrett and Jin Yang (Eds.). IEEE, 37–45. doi:10.23919/FMCAD.2019.8894275
- [28] Elizaveta Kuzmina, Shahbaz Pervez Chattha, Seyed Ebrahim Hosseini, Muazma Shahbaz, and Adnan Akhuzada. 2025. Spring Framework Benchmarking Utility for Static Application Security Testing (SAST) Tools. *IEEE Internet Things J.* 12, 22 (2025), 46863–46877. doi:10.1109/JIOT.2025.3598235
- [29] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 921–933. doi:10.1145/3611643.3616262
- [30] Weicheng Li, Hui Lu, Yanbin Sun, Shen Su, Jing Qiu, and Zhihong Tian. 2023. Improving Precision of Detecting Deserialization Vulnerabilities with Bytecode Analysis. In *31st IEEE/ACM International Symposium on Quality of Service, IWQoS 2023, Orlando, FL, USA, June 19-21, 2023*. IEEE, 1–2. doi:10.1109/IWQoS57198.2023.10188756
- [31] Yihe Li, Ruijie Meng, and Gregory J. Duck. 2025. Large Language Model Powered Symbolic Execution. *Proc. ACM Program. Lang.* 9, OOPSLA2 (2025), 3148–3176. doi:10.1145/3763163
- [32] Zhilin Li et al. 2026. Artifact for "Mind the Gap: Do Widely-Used SAST Tools Really Cover the Language-Specific Top CWEs?". doi:10.5281/zenodo.19122421
- [33] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net. <https://openreview.net/forum?id=9LdJDU7E91>

- [34] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Trans. Dependable Secur. Comput.* 21, 5 (2024), 4600–4618. doi:10.1109/TDSC.2024.3354789
- [35] Zhilin Li, Xutong Ma, Mengze Hu, and Jun Yan. 2024. Detecting Element Accessing Bugs in C++ Sequence Containers. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 – November 1, 2024*, Vladimir Filkov, Baishakhi Ray, and Minghui Zhou (Eds.). ACM, 883–894. doi:10.1145/3691620.3695471
- [36] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, and Tianyu Xu. 2023. Configuration Validation with Large Language Models. *CoRR* abs/2310.09690 (2023). arXiv:2310.09690 doi:10.48550/ARXIV.2310.09690
- [37] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 – 22, 2022*, Sukyoung Ryu and Yannis Smaragdakis (Eds.). ACM, 544–555. doi:10.1145/3533767.3534380
- [38] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhi Li, and Limin Sun. 2023. Harnessing the Power of LLM to Support Binary Taint Analysis. *CoRR* abs/2310.08275 (2023). arXiv:2310.08275 doi:10.48550/ARXIV.2310.08275
- [39] Zhengxiong Luo, Huan Zhao, Dylan Wolff, Cristian Cadar, and Abhik Roychoudhury. 2026. Agentic Concolic Execution. In *IEEE Symposium on Security and Privacy, SP 2026, San Francisco, CA, USA, May 18–21, 2026*. IEEE, 1673–1691. doi:10.1109/SP63933.2026.00003
- [40] Xutong Ma, Jiwei Yan, Wei Wang, Jun Yan, Jian Zhang, and Zongyan Qiu. 2021. Detecting Memory-Related Bugs by Tracking Heap Memory Management of C++ Smart Pointers. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15–19, 2021*. IEEE, 880–891. doi:10.1109/ASE51524.2021.9678836
- [41] MITRE Corporation. 2025. 2025 CWE Top 25 Most Dangerous Software Weaknesses: Methodology. https://cwe.mitre.org/top25/archive/2025/2025_methodology.html. Accessed: 2026-06-26.
- [42] MITRE CWE Team. 2025. CWE View 1000: Research Concepts. https://riskbasedprioritization.github.io/cwe/cwe_views/. Accessed: 2026-02-12.
- [43] National Institute of Standards and Technology (NIST). 2023. National Vulnerability Database (NVD). <https://nvd.nist.gov/>. Accessed: 2026-02-12.
- [44] Anh Nguyen-Duc, Manh Viet Do, Quan Luong Hong, Kiem Nguyen Khac, and Anh Nguyen Quang. 2021. On the adoption of static analysis for software security assessment—A case study of an open-source e-government project. *Comput. Secur.* 111 (2021), 102470. doi:10.1016/J.COSE.2021.102470
- [45] Haowei Qian, Junjie Wang, Xinzhe Li, Terry Yue Zhuo, Xiao Chen, and Xiaoning Du. 2025. An Empirical Study of Vulnerabilities in Python Packages and Their Detection. *CoRR* abs/2509.04260 (2025). arXiv:2509.04260 doi:10.48550/ARXIV.2509.04260
- [46] Reddit User ared38 and Others. 2023. Can the compiler detect memory leaks? https://www.reddit.com/r/cpp/comments/16xha6g/can_the_compiler_detect_memory_leaks Reddit discussion on detecting memory leaks, with comments about static analysis and dynamic tools such as sanitizers and Valgrind..
- [47] Denini Silva, MohamadAli Farahat, and Marcelo d'Amorim. 2026. An Empirical Analysis of Cross-OS Portability Issues in Python Projects. (2026).
- [48] Raphael J. Sofaer, Yaniv David, Mingqing Kang, Jianjia Yu, Yinzhi Cao, Junfeng Yang, and Jason Nieh. 2024. RogueOne: Detecting Rogue Updates via Differential Data-flow Analysis Using Trust Domains. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 101:1–101:13. doi:10.1145/3597503.3639199
- [49] Stack Overflow Users. 2018. Detect deadlock between a group of goroutines. <https://stackoverflow.com/questions/48548928/detect-deadlock-between-a-group-of-goroutines> Stack Overflow question about detecting deadlock among goroutines in Go, with answers discussing pprof stack dumps..
- [50] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14–20, 2024*. ACM, 166:1–166:13. doi:10.1145/3597503.3639117
- [51] Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023*, Satish Chandra, Kelly Blincoe, and Paolo Tonella (Eds.). ACM, 1716–1726. doi:10.1145/3611643.3613860
- [52] The MITRE Corporation. 2025. CWE – Common Weakness Enumeration. <https://cwe.mitre.org/>. Accessed: 2026-02-11.
- [53] David A. Tomassi and Cindy Rubio-González. 2021. On the Real-World Effectiveness of Static Bug Detectors at Finding Null Pointer Exceptions. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15–19, 2021*. IEEE, 292–303. doi:10.1109/ASE51524.2021.9678535
- [54] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14–20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 683–710. doi:10.1007/978-3-319-89884-1_24
- [55] Chong Wang, Jianan Liu, Xin Peng, Yang Liu, and Yiling Lou. 2025. Boosting Static Resource Leak Detection via LLM-based Resource-Oriented Intention Inference. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 – May 6, 2025*. IEEE, 2905–2917. doi:10.1109/ICSE55347.2025.00131
- [56] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. LLMDFa: Analyzing Dataflow in Code with Large Language Models. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 – 15, 2024*, Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (Eds.). http://papers.nips.cc/paper_files/paper/2024/hash/ed9dcde1eb9c597f68c1d375bbecf3fc-Abstract-Conference.html
- [57] Yayi Wang, Shenao Wang, Jian Zhao, Shaosen Shi, Ting Li, Yan Cheng, Lizhong Bian, Kan Yu, Yanjie Zhao, and Haoyu Wang. 2026. YASA: Scalable Multi-Language Taint Analysis on the Unified AST at Ant Group. *CoRR* abs/2601.17390 (2026). arXiv:2601.17390 doi:10.48550/ARXIV.2601.17390
- [58] William Webber, Alistair Moffat, and Justin Zobel. 2010. A similarity measure for indefinite rankings. *ACM Trans. Inf. Syst.* 28, 4 (2010), 20:1–20:38. doi:10.1145/1852102.1852106
- [59] Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, and Lingming Zhang. 2025. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13–16, 2025*, Youjip Won, Youngjin Kwon, Ding Yuan, and Rebecca Isaacs (Eds.). ACM, 655–669. doi:10.1145/3731569.3764827
- [60] Junjian Ye, Xincheng Fei, Xavier de Carné de Carnavalet, Lianying Zhao, Lifa Wu, and Mengyuan Zhang. 2024. Detecting command injection vulnerabilities in Linux-based embedded firmware with LLM-based taint analysis of library functions. *Comput. Secur.* 144 (2024), 103971. doi:10.1016/J.COSE.2024.103971
- [61] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. 2023. Detecting Exception Handling Bugs in C++ Programs. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14–20, 2023*. IEEE, 1084–1095. doi:10.1109/ICSE48619.2023.00098
- [62] Yining Zheng, Haiyang Wei, Jiahao Lu, Linqi Yin, Yunke Zhang, Chengguo Xu, Hetao Cui, Tianxiang Sun, Shuang Chen, and Xipeng Qiu. 2026. Investigating effective LLM-based in-context tool use: what matters and how to improve. *Frontiers Comput. Sci.* 20, 7 (2026), 2007323. doi:10.1007/S11704-025-41365-6
- [63] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. 2024. Comparison of Static Application Security Testing Tools and Large Language Models for Repo-level Vulnerability Detection. *CoRR* abs/2407.16235 (2024). arXiv:2407.16235 doi:10.48550/ARXIV.2407.16235