RESEARCH-ARTICLE

Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java

KAIXUAN LI, East China Normal University, Shanghai, China

Software security

SEN CHEN, Tianjin University, Tianjin, China

LINGLING FAN, Nankai University, Tianjin, China

RUITAO FENG, UNSW Sydney, Sydney, NSW, Australia

HAN LIU, East China Normal University, Shanghai, China

CHENGWEI LIU, Nanyang Technological University, Singapore City, Singapore

View all

Published: 30 November 2023

ESEC/FSE '23: 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering
December 3 - 9, 2023
CA, San Francisco, USA

Conference Sponsors:
SIGSOFT

Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java

Kaixuan Li*
East China Normal University
Shanghai, China
kaixuanli@stu.ecnu.edu.cn

Sen Chen*
College of Intelligence and Computing, Tianjin University
Tianjin, China
senchen@tju.edu.cn

Lingling Fan
College of Cyber Science, Nankai University
Tianjin, China
linglingfan@nankai.edu.cn

Ruitao Feng
University of New South Wales
Sydney, Australia
halertfeng@gmail.com

Han Liu
East China Normal University
Shanghai, China
hanliu@stu.ecnu.edu.cn

Chengwei Liu
Nanyang Technological University
Singapore, Singapore
chengwei001@e.ntu.edu.sg

Yang Liu
Nanyang Technological University
Singapore, Singapore
yangliu@ntu.edu.sg

Yixiang Chen†
East China Normal University
Shanghai, China
yxchen@sei.ecnu.edu.cn

ABSTRACT

Static application security testing (SAST) takes a significant role in the software development life cycle (SDLC). However, it is challenging to comprehensively evaluate the effectiveness of SAST tools to determine which is the better one for detecting vulnerabilities. In this paper, based on well-defined criteria, we first selected seven free or open-source SAST tools from 161 existing tools for further evaluation. Owing to the synthetic and newly-constructed real-world benchmarks, we evaluated and compared these SAST tools from different and comprehensive perspectives such as effectiveness, consistency, and performance. While SAST tools perform well on synthetic benchmarks, our results indicate that only 12.7% of real-world vulnerabilities can be detected by the selected tools. Even combining the detection capability of all tools, most vulnerabilities (70.9%) remain undetected, especially those beyond resource control and insufficiently neutralized input/output vulnerabilities. The fact is that although they have already built the corresponding detecting rules and integrated them into their capabilities, the detection result still did not meet the expectations. All useful findings unveiled in our comprehensive study indeed help to provide guidance on tool development, improvement, evaluation, and selection for developers, researchers, and potential users.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; • **Security and privacy → Software security engineering**; • **General and reference → Empirical studies**.

KEYWORDS

Static application security testing, Benchmarks, Empirical study

ACM Reference Format:
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 '23), December 3–9, 2023, San Francisco, CA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616262

1 INTRODUCTION

The early detection and handling of vulnerabilities in software code is a matter of concern for software development. In recent years, security vulnerabilities such as Log4Shell [66] and Spring4Shell [67] have raised alarm bells. Researchers have also proposed various methods to detect software vulnerabilities such as formal verification [4], static application security testing (SAST) [59], dynamic application security testing (DAST) [95], and interactive application security testing (IAST) [81]. Practically, SAST is the most popular technology due to its lower cost, faster operation, and stronger capability to detect bugs or vulnerabilities without executing a program. Hence, the development of SAST technology has obviously evolved, and the number of corresponding tools has rapidly grown in recent years [13, 44, 68–70, 74, 85, 90, 93, 107].

However, it is still challenging for users to objectively evaluate and select the appropriate SAST tools due to the following reasons. ***(1)*** Firstly, existing studies are mainly conducted on synthetic datasets [2, 58, 80, 101], where vulnerabilities are usually implemented and injected into programs manually. Compared to

*These authors contributed equally to this work.
†Corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616262

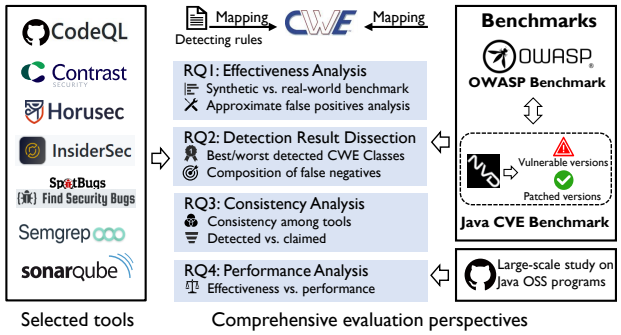


Figure 1: Overview of our study.

real-world vulnerabilities [8], they are much simpler in design and easier to be detected. Therefore, it is hard to objectively reflect the detection capability of tools in real-world programs. OpenSSF [76] also emphasized the importance of using real-world vulnerability data to evaluate the effectiveness of SAST tools and developed a benchmark [75] which contains over 200 CVEs (Common Vulnerabilities and Exposures) [14]. However, their benchmark only includes JavaScript and TypeScript CVEs. Some studies have also been exhibited to evaluate the effectiveness of SAST tools on open-source programs [1, 46, 97], in which the datasets used are often small in size and limited in the number and types of contained vulnerabilities. ***(2)*** Furthermore, the focus of existing studies [41, 51, 98, 100] concerns more on quality issues, e.g., code styles, performance, and bad practices, rather than security vulnerabilities. For example, Thung et al. [98] performed an evaluation on Java SAST tools, in which they explored to what extent the Java SAST tools detect real-world defects on three open-source Java programs, and analyzed five kinds of defects including code style and bad practices. ***(3)*** Thirdly, specifically for Java SAST tools, the shortage of knowledge on commonly detected types of vulnerabilities makes researchers even harder to gain deeper insights into the strengths and weaknesses of a given tool. Besides, the consistency of vulnerability types that actually reported in detection and claimed to support in documentation is also an interesting research question to explore.

Java is one of the most popular and well-developed programming languages, with a broad scope of application scenarios [99]. However, till now, there is still a lack of effort in evaluating SAST tools on real-world programs, especially for Java. In a concurrent work evaluating SAST tools, Lipp et al. [50] focused on SAST tools for C programs, in which they evaluated the effectiveness of six tools on 192 real-world vulnerability datasets using 27 open-source C projects. But the results on C SAST tools may be not feasible for Java SAST tools because of the different language constructs. Meanwhile, the corresponding SAST tools can differ in their usage, and it is a question that how to choose a Java SAST tool that is suitable for scanning speed or workflow integration besides the effectiveness of vulnerability detection.

As shown in Figure 1, to bridge these gaps, we evaluated 7 representative Java SAST tools filtered from 161 tools. Then, we used Common Weakness Enumeration (CWE) [25] as a reference to map the detecting rules of these tools and CVEs contained in our collected benchmark datasets to CWE, and automatically compared the effectiveness of each tool. We collected 2 types of benchmark datasets including a synthetic dataset (i.e., OWASP Benchmark) and a real-world benchmark (i.e., the Java CVE Benchmark). The latter includes 165 open-source Java programs with 165 unique CVEs. The dataset covers 37 unique vulnerability types (*CWE Weaknesses*), belonging to 8 *CWE Classes* in CWE-1000 [19]. For this, we evaluated the tools' effectiveness against the 2 benchmarks. Based on their poor effectiveness on the Java CVE Benchmark, we further dissected the composition of false negatives. Moreover, we performed a consistency evaluation on the vulnerabilities detected by these tools between the actually detected ones and what is claimed in the detecting rules. Finally, we performed a performance analysis on 1,049 representative Java open-source programs.

Our study unveils that the evaluation of SAST tools on synthetic datasets does not objectively reflect the detection capability of the tools. In particular, the selected tools overlooked most (87.3%) real-world vulnerabilities in the Java CVE Benchmark, while they have been shown to perform well on the OWASP Benchmark. Meanwhile, over 70% of vulnerabilities still remain undetected when combining the results of SAST tools, especially those beyond the scope of CWE-664 and CWE-707. For consistency analysis, we observed that these tools generally overstate their detection capabilities, even with 90.5% overstatement on our real dataset. Meanwhile, their analysis time increases sharply when the line of code (LoC) is over 50k. In particular, Insider [44] and Contrast [90] are the fastest, while Semgrep [85] and CodeQL [13] require the longest time.

In summary, we made the main contributions as follows:

- We constructed a real-world benchmark containing 165 open-source Java programs with 165 unique CVEs on the method level, which is considered the largest real-world vulnerability benchmark for Java. It costs 13.5 person-months for the construction.
- To fairly compare the 7 SAST tools' detecting rules, we mapped and grouped 1,801 rules of tools studied and vulnerability data in our 2 benchmarks into *CWE Classes*, and analyzed the detection consistency among tools, as well as that between the detected vulnerability types and those claimed in their detecting rules.
- We conducted a large-scale empirical evaluation of the selected tools from comprehensive perspectives, including effectiveness, consistency, and runtime performance. To this end, 43,519 (i.e., $7 \times (2,740 + 165 \times 2 + 1,049 \times 3)$) scanning tasks are conducted.
- Based on the evaluation results, we discussed the lessons learned and detailed the guidance on SAST tool development, improvement, evaluation, and selection for SAST tool developers, researchers, and potential users.

2 OVERVIEW

2.1 Tool Selection

We aim at gathering a representative set of SAST tools since it is infeasible to give a complete set of all existing tools. Therefore, we searched tool lists from recent scientific literature [1, 5, 40, 41, 51, 56, 80, 92, 98, 100] and snowballed from them, as they also recommend further lists. Eventually, we obtained several prominent websites [34, 35, 72, 73, 77, 78, 104] giving recommendations for SAST tools. This process resulted in a very substantial set of SAST tools [103], even after removing duplicates (192 out of 576). We designed the following criteria to select our evaluation subjects.

Table 1: Tool profile. Technology: Semantic (data-flow and control-flow analysis) or Syntactic (pattern-matching within the code). # Stars indicates GitHub stars.

Tool	Technology	# Stars	Version
CodeQL	Semantic	6.1k	v2.10.2
Contrast	Semantic	/	1.0.10
Horusec	Syntactic	903	v2.8.0
Insider	Syntactic	429	v3.0.0
SBwFSB	Semantic	3.1k	v4.7.0, v1.12.0
Semgrep	Semantic	8.3k	v0.108.0
SonarQube	Semantic	7.8k	v9.5.0, v4.1.0

① ***Java supported.*** First, we only included SAST tools that support Java programs and obtained 161 Java SAST tools in total.
② ***Free of charge.*** Second, the Java SAST tools must be free of charge. While commercial tools are indeed prevalent in the industry, they often entail substantial costs and do not disclose their internal rule implementations, thereby posing analytical limitations for our study. Thus, 54 commercial tools were excluded.
③ ***Being maintained.*** Third, we eliminated tools that were no longer maintained. Specifically, we manually checked whether the tool's open-source repository had been active for the last 2 years. After this step, 20 SAST tools were further removed.
④ ***Command-line interface (CLI).*** Moreover, we did not consider tools that have usage limits or purely operate through a graphical user interface, as we aim to conduct a large-scale experiment in this study. Therefore, we excluded 33 tools such as Reshift [88], HCL AppScan CodeSweep [42], and GitHub Code Scanning [36].
⑤ ***Security related.*** We try to select tools that identify generalized security vulnerabilities, rather than those aimed to detect specific vulnerabilities or code quality issues such as linters [105]. Initially, we selected tools that claim they can detect "vulnerabilities", "security issues", and other similar terms in their documentation. Furthermore, to facilitate the comparison and evaluation of tool effectiveness between synthetic and real-world benchmarks, it is required that tools should demonstrate an ability to detect vulnerabilities in synthetic benchmarks such as the OWASP Benchmark, i.e., be able to detect at least two vulnerability types. Finally, we excluded 6 tools including Error Prone [38], Facebook Infer [54], Checkstyle [9], PMD [83], google-java-format [39], and Mega-Linter [79].

⑥ ***Well-documented with detecting rules.*** Note that we intend to select SAST tools with well-documented detecting rules, which allows us to analyze the effectiveness of each tool by mapping them to CWE. Meanwhile, we explore whether the detecting capacity they claim in the rules is consistent with that in practice. After applying this criterion, we excluded 41 tools that did not provide publicly available documentation of their detecting rules.

Based on these criteria, we finally selected 7 tools: CodeQL, Contrast Codesec Scan (Contrast), Horusec [107], Insider, SpotBugs [70] with FindSecurityBugs [69] (SBwFSB), Semgrep, and SonarQube community edition (SonarQube) [93] (Table 1). We have uploaded the full candidate SAST tool list [102].

2.2 Benchmark Collection

2.2.1 OWASP Benchmark. For the synthetic dataset selection, we considered OWASP Benchmark [31], as it is consistently maintained and updated compared with other synthetic datasets such as the Juliet Test Suite Java [71] and OWASP Top 10 2020 Benchmark [30]. Although the vulnerabilities within it are synthetic, we can use them to draw preliminary conclusions about the detection capabilities of SAST tools. Each case within it has either a genuine, exploitable vulnerability (1,415 in total) or a non-vulnerable control instance mimicking a false positive (1,325 in total).

2.2.2 Java CVE Benchmark. In response to OpenSSF's call for real-world vulnerability data in SAST tool evaluation, we constructed a Java CVE Benchmark by involving four steps as follows:

- **Java programs collection:** We first searched Java open-source programs with disclosed CVEs and corresponding patch commits from advisory sources such as NVD [57], Debian [24], and Red Hat Bugzilla [87], initially obtaining a list of 680 programs.
- **Version range extraction and method-level locating:** We utilized SZZ [91] to extract the vulnerable version range of programs affected by each CVE, ensuring accurate identification of affected versions. Meanwhile, we employed Ctags [15] to locate method-level information for both vulnerable and patched versions, which is essential for a detailed analysis of the vulnerabilities.
- **Program packaging:** Since the tools under evaluation accept different types of input (e.g., source code and binaries), we further excluded the programs that failed to be packaged. We finally obtained 165 package-able programs [102].
- **Cross-validating:** To ensure the benchmark quality, we engaged three security experts from our industry partner. They verified the vulnerability locations identified by our automated process and cross-validated each other's work. Each expert thoroughly reviewed the details provided in the vulnerability and patch information obtained from advisory sources. After that, they cross-validated each other's results. If disagreements arose during the cross-validation, a majority voting [43] was used to make the decision. In cases where the votes were evenly split, a discussion was held to resolve the conflict. The vulnerability was then labeled with detailed information such as its location, the affected versions, and the specific methods where the vulnerabilities and patches were located.

Finally, we got 165 package-able open-source programs containing 165 unique CVEs, where each program owns a vulnerable version and a patched version, with the location of vulnerabilities and patches labeled at the method level. The entire process of collecting the Java CVE Benchmark took us 13.5 person months, with an additional 3 person months spent on cross-validating the vulnerability and patch locations. To the best of our knowledge, it is **the largest real-world Java vulnerability benchmark**.

2.3 Mapping Vulnerability Data in Benchmarks and Detecting Rules of Tools to CWE

Since SAST tools use different identifiers for the vulnerability types they support, e.g., Insider uses CWEs in the reported issues, while others introduce their own vulnerability identifiers. These different identifiers make it difficult to automatically determine whether a SAST tool hits a specific vulnerability type. CWE refers to a community-developed list of software and hardware weakness types, including security vulnerabilities, which is also used in CVE

Table 2: CWE mapped by our benchmarks and each SAST tool.

	# Vulnerabilities/Rules	# *CWE Weaknesses*	CWE-284	CWE-435	CWE-664	CWE-682	CWE-691	CWE-693	CWE-697	CWE-703	CWE-707	CWE-710
OWASP Benchmark	2,740	11	✗	✗	394	✗	✗	1,042	✗	✗	1,304	✗
Java CVE Benchmark	165	37	14	4	102	✗	7	15	✗	3	31	1
CodeQL	1,065	196	75	3	401	20	117	150	19	28	96	204
Contrast	46	41	5	1	15	✗	1	11	1	1	17	1
Horusec	216	47	11	✗	40	✗	3	59	✗	3	24	6
Insider	90	30	4	1	9	✗	1	8	✗	✗	6	3
SBwFSB	152	53	6	3	39	✗	3	31	✗	1	42	3
Semgrep	165	43	6	1	46	✗	12	67	✗	✗	21	3
SonarQube	66	37	10	1	21	✗	✗	21	✗	1	5	2

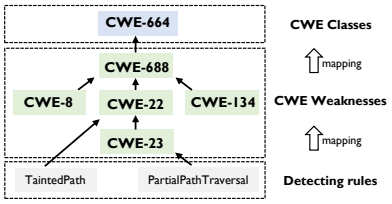


Figure 2: Example of mapping and grouping rules to CWE.

reports [18] and supported by many SAST tools [20]. In addition, all of these tools have mapped their own rules to CWE in their documentation or GitHub repository except for SpotBugs,[1] so we consider it a valid approach by evaluating them according to CWE.

In this study, we selected "CWE-1000: Research Concepts" as a reference, since it aims at facilitating research into weaknesses, including inter-dependencies among CWE entries when compared with the other two CWE Views [21]. Considering that direct mapping of rules to *CWE Weaknesses* poses some hierarchical inconsistencies, as shown in Figure 2, which may distinguish the effectiveness of the tools that map rules to different levels. Similarly, CWE also has hierarchical structure issues [40, 50]. We considered mapping detecting rules directly to the "Pillar" [22] level (hereafter denoted by *CWE Classes*) in CWE-1000 for the purpose of unifying them to the same level of CWE. Therefore, to enable us to automate the evaluation of the tools studied, we use CWE as a reference, with the vulnerability data in the two benchmarks and the tool's rules mapped to *CWE Classes* in CWE-1000, respectively.

Mapping vulnerability data to CWE. Since all of the vulnerabilities in our two benchmarks have been mapped to *CWE Weaknesses*, we thereby mapped them to *CWE Classes* according to CWE-1000.

Mapping detecting rules to CWE. Similarly, since these tools have mapped their detecting rules to *CWE Weaknesses* except for SpotBugs, we only need to map them to *CWE Classes* according to the hierarchy of CWE-1000. For SpotBugs, we manually mapped its rules to both *CWE Weaknesses* and *CWE Classes*. This process involved three co-authors independently performing the mapping. They consulted the rule documentation and the hierarchy of CWE-1000 during this process. Any conflicts in mapping results were resolved through "majority voting". Finally, we determined the support for *CWE Classes* by the tools. A *CWE Class* was considered supported by a tool if the rule documentation stated it implemented a check for at least one *CWE Weakness* within that class.

[1]We obtained the mapping documentation of Contrast from its technical support team.

Table 2 shows each class in CWE-1000 is included/supported or not (✗). The number of corresponding vulnerabilities/rules is further displayed if included/supported. Totally, 2,740 vulnerabilities included in OWASP Benchmark are grouped into 11 *CWE Weaknesses* and 3 *CWE Classes*, while our real-world benchmark owns more coverage than it does, i.e., including 37 *CWE Weaknesses* grouped into 8 *CWE Classes* except for CWE-682, and CWE-697.

3 COMPARISON AND EVALUATION

The evaluation aims to answer the following research questions:

- **RQ1: Effectiveness analysis.** How effective are these SAST tools in detecting vulnerabilities on OWASP Benchmark and our constructed Java CVE Benchmark?
- **RQ2: Detection result dissection.** What are the root causes of the detection results in RQ1?
- **RQ3: Consistency analysis.** Are the detection results consistent among these tools in terms of the detected vulnerability types? Are the detected vulnerability types consistent with what was claimed in each tool?
- **RQ4: Performance analysis.** How is the performance of these tools (i.e., the time cost of detection)?

3.1 RQ1: Effectiveness Analysis

3.1.1 Setup. We evaluated the effectiveness of the 7 tools on the two benchmarks. For the OWASP Benchmark, we compute Recall, Precision, and F1-score as the evaluation metrics. For the Java CVE Benchmark, we calculate the proportion of detected CVEs, denoted as CVE_R ($\frac{\#\ D_{vul}}{\#\ All\ CVEs\ in\ benchmark}$), and the proportion of CVEs still detected in the patched versions, denoted as CVE_R_{patch} ($\frac{\#\ (D_{vul} \cap D_{patch})}{\#\ D_{vul}}$). Here, D_{vul} and D_{patch} represent the detected CVEs in the vulnerable and patched versions, respectively. The latter metric approximates the rate of false positives. Inspired by previous works [50, 98], we evaluated the real-world detection capabilities of these tools with respect to *file-level* and *method-level*, and divided them into four different scenarios as follows:

- ***File-level Detection with Any CWE Class*** **(S_{F-A})**: A CVE is considered *detected* if ≥1 vulnerable file is hit by the tool, regardless of the *CWE Class* reported.
- ***File-level Detection with Correct CWE Class*** **(S_{F-C})**: A CVE is considered *detected* if ≥1 vulnerable file is hit by the tool, with the correct *CWE Class* reported.
- ***Method-level Detection with Any CWE Class*** **(S_{M-A})**: A CVE is considered *detected* if ≥1 vulnerable method is hit, regardless of the *CWE Class* reported.

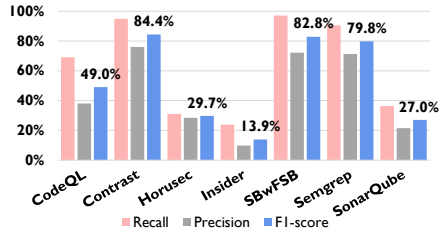


Figure 3: Effectiveness on OWASP Benchmark.

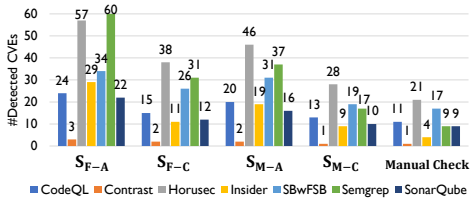


Figure 4: Number of detected CVEs in different scenarios.

- ***Method-level Detection with Correct CWE Class*** **(S_{M-C})**: A CVE is considered *detected* if ≥1 vulnerable method is hit with the correct *CWE Class* reported.

3.1.2 Results. The overall results on these two benchmarks are shown in Figure 3 and Figure 4, respectively.

Effectiveness on the OWASP Benchmark. Figure 3 shows that Contrast and SBwFSB can detect close to all synthetic vulnerabilities, with F1-score 84.4%, and 82.8%, respectively. However, Insider failed to detect most synthetic vulnerabilities with 23.9% Recall, 9.8% Precision, and 13.9% F1-score. As displayed in Figure 5, the effectiveness on three *CWE Classes* varies from tools. However, synthetic vulnerabilities belonging to CWE-693 are easier detected by tools, especially those involving insecure cryptographic algorithms or insufficiently random values. While those related to *Path Traversal* (CWE-22) and *Trust Boundary Violation* (CWE-501) are hardly detected by these tools. In particular, Horusec and Insider failed to detect all of these two types. For Insider, the number of its detecting rules is limited, with no strong coverage of diverse Java vulnerabilities, i.e., 90 in total, with only 9 rules related to CWE-664. Moreover, Insider has no rule supporting CWE-501 and CWE-22 although claiming to cover the OWASP Top 10. While for Horusec, CWE-501 is also not supported by its rules. However, we further noticed that CWE-22 is supported by its rules but not detected. More specifically, Horusec implements 3 related rules, but all of them are based on primitive regular expressions which only detect few related vulnerabilities related to the hard-coded use of either `@javax.ws.rs.PathParam()` or `@jakarta.ws.rs.PathParam()`.

Finding 1: SAST tools generally perform well on the synthetic dataset (i.e., the OWASP Benchmark), especially Contrast and SBwFSB, which both got an F1-score over 80%, while Insider showed the lowest detection rate (i.e., 13.9% F1-score), following SonarQube with an F1-score of 27.0%.

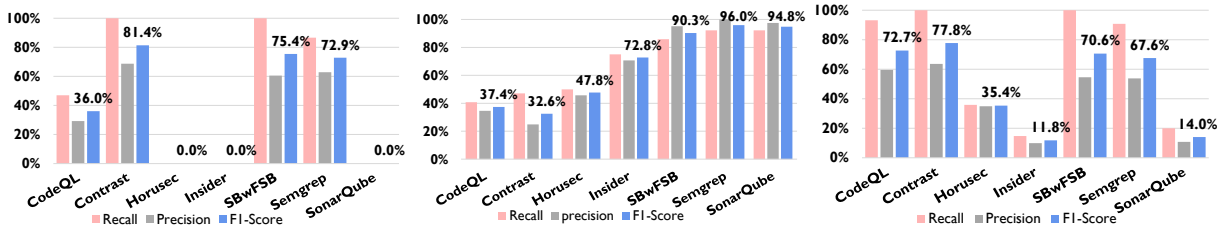
```
1 from  DataFlow::PathNode source, DataFlow::PathNode sink,
2        PathCreation p, Expr e, TaintedPathLocalConfig conf
3 where  e = sink.getNode().asExpr() and e = p.getAnInput()
       and
4        conf.hasFlowPath(source, sink) and not guarded(e)
5 select p, source, sink, "$@ flows to here and is used in a
       path.",
6        source.getNode()
```

Listing 1: TaintPathLocal.ql in CodeQL.

Effectiveness on the Java CVE Benchmark. As shown in Figure 4, the effectiveness of the 7 tools is analyzed in the 4 scenarios aforementioned. Note that there may exist some cases in S_{M-C} where some tools hit the true *CWE Class* of a CVE, but a wrong *CWE Weakness* was actually reported. To ensure the accuracy of our results, we further checked the *CWE Weaknesses* reported by the tools in S_{M-C}. The corrected results are presented as "Manual Check" in Figure 4 . Contrary to the effectiveness on OWASP Benchmark, it displays poor effectiveness on real-world vulnerabilities of these tools, reflecting that over 85% of CVEs were ignored by selected tools. Even the top-performing tool (Horusec) achieved a mere 12.7% *CVE_R*. Although Horusec is a syntax-based tool whose rules are implemented by regular expressions, it outperforms the three semantic-based tools (SBwFSB, CodeQL, and Semgrep). This unveils that the semantic analysis method is not always more effective than the less complex syntactic ones when in practice. Moreover, we observed that Horusec integrated with some other tools (e.g., GitLeaks [37] and Trivy [89]). It is worth noting that Horusec also integrates OWASP Dependency-Check [32] within it, a Software Composition Analysis (SCA) [33] tool, helping Horusec hit another 49 correct CVEs by scanning the vulnerable TPLs used in programs. Although it is not a SAST tool, such an approach may inspire us for detecting more vulnerabilities during DevSecOps [86].

Following Horusec, SBwFSB, and CodeQL detected 17 (10.3%) and 11 (6.7)% CVEs. Both tools are equipped with data-flow analysis (DFA) and control-flow analysis (CFA), especially taint analysis. To detect vulnerabilities, SBwFSB uses resource files to list and store vulnerable sources and sinks to search taint paths, which can limit its search scope in practice. CodeQL models source code as database records allowing its queries to search when scanning, which is considered a stronger technology than SBwFSB. But we found that the default rules within CodeQL are still simple which limits its effectiveness when detecting complex vulnerabilities such as CVEs. For instance, consider a CodeQL rule designed to detect CWE-22 vulnerabilities, as shown in Listing 1. It tracks tainted data from user input (source) to a file path creation (sink) (L3-4). However, it fails to detect indirect influences of user input on path creation. Additionally, it deems a variable as "guarded" if it undergoes any form of check or sanitization, which may not be sufficient. For example, simply replacing "../" in user input could not prevent an attacker from constructing a path traversal string such as "..././..././". This inadequate sanitization could still lead to an exploit, which this default rule would not detect.

Notably, Contrast, the most effective tool according to the OWASP Benchmark, nearly failed to detect all the CVEs in all 4 scenarios. Thus, the evaluation using synthetic datasets may yield discrepancies or even opposite conclusions from those on real-world ones.



(a) Effectiveness on CWE-664. (b) Effectiveness on CWE-693. (c) Effectiveness on CWE-707.

Figure 5: Effectiveness of each class on OWASP Benchmark.

Table 3: Detection results according to CVSS severity.

CVSS severity	# Detected	# Total
High	13	40
Medium	32	120
Low	3	5

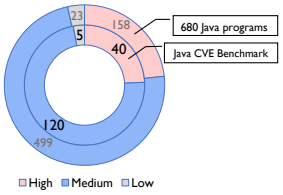


Figure 6: Severity distribution in the Java CVE Benchmark.

Table 4: Approximate false positives in S_{F-C} and S_{M-C}.

Tools	S_{F-C}		S_{M-C}	
	# D_{Vul}	# $(D_{vul} \cap D_{patch})$	# D_{Vul}	# $(D_{vul} \cap D_{patch})$
CodeQL	15	**9**	11	**5**
Contrast	2	**2**	1	**1**
Horusec	38	**23**	21	**6**
Insider	11	**11**	4	**4**
SBwFSB	26	**25**	17	**16**
Semgrep	31	**26**	9	**4**
SonarQube	12	**8**	9	**5**

Finding 2: These tools overlook more than 85% of CVEs (false negatives), although performing well against synthetic benchmarks. Horusec and SBwFSB perform better than the other tools, with a *CVE_R* of only 12.7%, and 10.3% respectively.

Effectiveness vs. CVSS severity. We try to explore the relationship between the effectiveness of SAST tools and the severity of vulnerabilities by leveraging the Common Vulnerability Scoring System (CVSS) [28] associated with each CVE. Due to the absence of CVSS V3 [17] for some CVEs, we finally used CVSS V2 [16] for a fair evaluation. As shown in Figure 6, the severity distribution of the Java CVE Benchmark aligns with that of the original 680 programs, suggesting that the distribution of our benchmark does not significantly skew the results. As outlined in Table 3, SAST tools detected 60% of low-severity CVEs. For medium-severity CVEs, the detection ratio dropped to 26.7% (32/120). Interestingly, the detection ratio for high-severity CVEs was slightly higher at 32.5%.

Meanwhile, we also observed distinct detection patterns within each severity level. For instance, all four high-severity injection-related vulnerabilities, which fall under the *Improper Neutralization* (CWE-707) class, were detected. However, no input-validation issues, also under CWE-707, were detected at this level. The discrepancy in detection rates could be due to the complexity of high-severity vulnerabilities or the nature of the vulnerability itself. For instance, injection vulnerabilities, which often involve improper input handling, might be easier to detect than deserialization vulnerabilities, which require complex object processing. Additionally, only 5% (2/40) of medium-severity *Deserialization of Untrusted Data* (CWE-502) vulnerabilities were detected. Out of these, 38 vulnerabilities were found in the widely used *jackson-databind* [53] project, yet none were detected, highlighting the importance of effective vulnerability detection in popular software components.

To analyze approximate false positives, we focused on S_{F-C} and S_{M-C} since there is no focus on *CWE Classes* in the other scenarios as mentioned before. As shown in Table 4, while most of the tools are generally poor in detecting real-world vulnerabilities, there are still rather the same vulnerabilities reported in the patched versions. Overall, it reflects that the selected tools are not "vulnerability-sensitive" enough since they are not sensitive enough to distinguish pieces of code before and after patching the vulnerability. Especially in S_{M-C}, it unveils that Horusec and CodeQL are more effective than the other tools on the patched versions, with a CVE_R_{patch} at 28.6% and 45.5%, respectively, while SBwFSB still reported 16/17 the same CVEs. It was observed that there exist minor differences between each vulnerable and patched version on the syntax level, but the patched version does fix the corresponding CVE. However, the detecting rules of these tools are coarse-grained. This results in these tools capturing only simple patterns of these CVEs on the syntax level, instead of capturing their exact patterns on the semantic level. Meanwhile, we observed that there are certain vulnerability types are more often labeled as false positives than others in the tools. For instance, *Incorrect Type Conversion or Cast* (CWE-704), which is mapped by HS-JAVA-143 by Horusec, is very likely to be false positives in our scope, especially on the *jackson-dataformats-binary* [27] (205/275, 74.5%). Therefore, identifying these kinds of vulnerability types and disabling corresponding rules probably contributes to reducing false positives, although a few true positives may be omitted.

Finding 3: These tools are not "vulnerability-sensitive" when performing on patched versions, which reflect tools' false positives. In particular, Horusec (6/21) and CodeQL (5/11) perform better than the others. While SBwFSB seems less sensitive (16/17) than the other 6 tools in S_{M-C}.

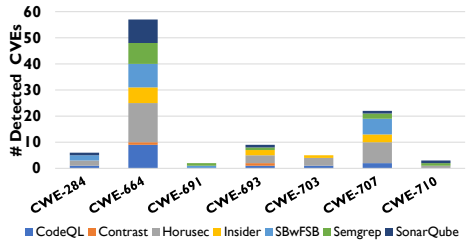


Figure 7: Distribution of detected CVEs in each *CWE Class*.

```
1 for (javax.servlet.http.Cookie theCookie : theCookies) {
2     if (theCookie.getName().equals("BenchmarkTest00002")) {
3         param = java.net.URLDecoder.decode(theCookie.
        getValue(), "UTF-8");
4         break;
5     }
6 }
7 fileName = org.owasp.benchmark.helpers.Utils.TESTFILES_DIR+
        param;
8 fos = new java.io.FileOutputStream(fileName, false);
```

Listing 2: Testcase 0002 (CWE-22) in the OWASP Benchmark.

3.2 RQ2: Detection Result Dissection

3.2.1 Setup. To analyze the root causes for the poor effectiveness of the selected tools, we manually reviewed the 48 detected CVEs and the 117 missing ones as well as the detecting rules' implementation. This process was conducted in three rounds involving three co-authors. Initially, each co-author individually analyzed all the CVEs and detection rules of each tool, checking for any rules mapped to *CWE Weaknesses* for undetected CVEs. Subsequently, they discussed their findings to reach a consensus, resolving any disagreements through further discussion. In the final round, 20% of the CVEs were randomly selected for review by all authors. Disagreements were resolved through discussion, potentially leading to updates on the root causes. Finally, we grouped the root causes into three main categories.

3.2.2 Results. ***Best vs. worst detected CWE Classes.*** Overall, Figure 7 reflects that among the vulnerabilities in our Java CVE Benchmark, **CWE-{664, 707}** are the easily detected *CWE Classes*. Specifically, CWE-664 refers to *Improper Control of a Resource Through its Lifetime*, which involves the management of system resources such as memory allocation and deallocation, and CWE-707 refers to *Improper Neutralization*, which includes vulnerabilities related to the improper handling of input or data. Interestingly, although vulnerabilities related to the two classes of CWE-691 and CWE-710 are theoretically supported by all the deployed tools, most of the associated CVEs remained undetected, except the SonarQube which lacks rules for CWE-691. It is due to their low proportion (4.8%) in our real-world benchmark. In particular, CVEs related to CWE-664 are detected by all the tools. After eliminating the overlapping in detected vulnerabilities, 32 unique CVEs (32/107) in this class were found. Especially, CVEs of {CWE-22, CWE-502, CWE-200, and CWE-611} are the most frequently detected types within this class, accounting for 62.5% (20/32). Moreover, 22 (14 unique) of the 31 CVEs related CWE-707 were detected by all tools except for Contrast. Within this class, 3 *OS Command Injection* (CWE-78), and 3 *Improper Input Validation* (CWE-20) were detected.

Finding 4: Real-world vulnerabilities related to CWE-664 and CWE-707 are more easily detected, especially those relevant to CWE-{22, 200, 502, 611}, CWE-{20, 78} are more effectively detected. However, these tools still missed 70.1% of the 107 CWE-664 vulnerabilities and 37.5% of the 29 CWE-707 ones.

Composition of missing vulnerabilities (false negatives). The 48 successfully detected CVEs can be grouped into seven *CWE Classes*: CWE-284 (3), CWE-664 (32), CWE-691 (2), CWE-693 (6), CWE-703 (2), CWE-707 (14), and CWE-710 (1). We observed that it is because their patterns are easy for SAST to spot, e.g., *CVE-2019-18393* [65] is a typical vulnerability related to CWE-22 caused by not checking the use of "\", which is a common path-traversal pattern for SAST tools to detect. As displayed in Table 5, the composition of overlooking the 117 CVEs can be summarized into three categories:
① **C1: No detecting rules supported by tools (2.6%-8.5%).** In this category, although only 3 (2.6%) CVEs are not supported by any tool's pre-defined rules, we found that each tool generally fails to support 2-10 (8.5%) CVEs which SAST is typically sufficient to detect but no rule was implemented. In particular, even CodeQL, which has the most rules among these tools, still failed to implement rules for 10 CVEs in the Java CVE Benchmark. For instance, *CVE-2015-2913* [62] is related to *Use of Insufficiently Random Values*, where SAST is generally sufficient to detect most relevant instances [6] although false negatives may occur if custom cryptography is used.
② **C2: Inadequate detection capabilities of tools (76.9%-82.9%).** 90-97 CVEs were undetected due to inadequate detection capacities of these tools, indicating that **the predefined rules in the tools are not sufficiently effective** in identifying real-world vulnerabilities. On the one hand, the primitive implementation of predefined rules, including source and sink lists, significantly impacts the tools' detection, e.g., 91.2% (52/57) of CVEs related to CWE-502 went undetected despite targeted rules. While a base search for `ObjectInputStream()` and `readObject()` in source code could detect some related vulnerabilities, most cases required additional DFA and CFA in rule implementation. On the other hand, code patterns in the CVEs were notably more complex than those in the synthetic cases, a finding that is also revealed in a concurrent study on Android [52]. For instance, despite owning rules targeting CWE-22, tools such as SBwFSB and CodeQL only detected 16.7% of related CVEs, even though they could detect all synthetic cases labeled with the same CWE in the OWASP Benchmark. As shown in Listing 2, the synthetic code pattern is relatively straightforward and follows a linear flow within the same scope.

Conversely, the code pattern in the Java CVE Benchmark example [94] (Listing 3) is more nuanced and encapsulated within a class structure. It arises from two separate methods for user-controlled input and usage respectively with an implicit connection of a class field. Although there is some normalization (L5), it is not sufficient to prevent path traversal sequences. The vulnerability manifests when another method `getURL` attempts to read the field `this.path` and resolve it (L11-17), and an attacker can exploit this by providing specially crafted paths. Detecting it requires the ability of inter-procedural DFA and insufficient validation detection, which is not required by Listing 2.

Table 5: Examples of missing CVEs by categories.

Categories	CVE ID	*CWE Weakness*	*CWE Class*
C1	CVE-2015-2913	CWE-330	CWE-693
	CVE-2013-5960	CWE-310	CWE-693
C2	CVE-2018-9159	CWE-22	CWE-664
	CVE-2021-20190	CWE-502	CWE-664
C3	CVE-2014-3651	CWE-400	CWE-664
	CVE-2018-1274	CWE-770	CWE-664

```
1  public ClassPathResource(String path, ClassLoader
       classLoader) {
2      Assert.notNull(path, "Path must not be null");
3      String pathToUse = StringUtils.cleanPath(path);
4      if (pathToUse.startsWith("/")) {
5          pathToUse = pathToUse.substring(1);
6      }
7      this.path = pathToUse;
8      this.classLoader = (classLoader != null ? classLoader :
         ClassUtils.getDefaultClassLoader());
9  }
10 @Override
11 public URL getURL() throws IOException {
12     URL url = this.classLoader.getResource(this.path);
13     if (url == null) {
14         throw new FileNotFoundException(this.path + "cannot
         be resolved to URL because it does not exist");
15     }
16     return url;
17 }
```

Listing 3: Simplified code snippet for *CVE-2018-9159*.

The marked distinction between these patterns highlights the need for SAST tools to excel in analyzing real-world code, especially when dealing with object interactions and method calls. While the OWASP Benchmark is useful for basic testing, it lacks the complexity present in real-world scenarios. Thus, using a real-world benchmark is vital for evaluating the practical effectiveness of tools.

③ **C3: Hard to be detected by SAST (14.5%).** In this case, we found that there are 24 CVEs difficult for SAST to detect. For example, *CVE-2014-3651* [61] is a vulnerability related to *Uncontrolled Resource Consumption* (CWE-400). However, SAST typically has limited utility in recognizing resource exhaustion problems, since determining boundary values on integers requires a strong capacity in propagating boundary value information across any control flow units including loops. Moreover, in addition to certain practical restrictions, there exists a theoretical limit when inferring based on the undecidability of SAST [47]. For instance, invariants and post-conditions are supposed to be deduced even for a loop.

Finding 5: Over 76.9% overlooked CVEs are caused by insufficient support of these tools, especially those mapped to CWE-22, CWE-502. While 14.5% are hard for SAST to detect, including those related to checking boundary value issues (e.g., CWE-400).

3.3 RQ3: Consistency Analysis

3.3.1 Setup. Inspired by findings on the Java CVE Benchmark as shown in RQ1, we further constructed two consistency analyses: ***(1)*** the consistency of detected CVEs among the tools, and ***(2)*** the consistency of detected CVEs between tools actually detect and

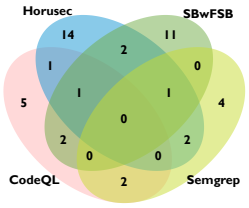


Figure 8: Tools combination in S_{M-C}.

what they claim to support. For the latter task, we try to explore whether these tools "keep their promises" based on the mapping results of detecting rules and CVEs. To weaken the impacts of CVEs that are hard for SAST to detect (C3 in Section 3.2.2), we place the scope on those CVEs that SAST technically has the ability to detect.

3.3.2 Results. The fact is that the number of detected CVEs in each *CWE Class* varies for each tool, as displayed in Figure 7.

Consistency among the tools. As indicated in Figure 8, there is no CVE that was detected simultaneously even by the four best-performing tools, which reflects these tools have different focuses. By comparing the detected vulnerabilities by all tools, we found that 11 unique CVEs are detected only by Horusec, with 11 and 4 ones only detected by SBwFSB, and Semgrep respectively. Specifically, the most detected CVEs by Horusec are in CWE-664, especially those related to CWE-611. Meanwhile, the most detected CVEs by SBwFSB are in CWE-707, including CWE-20. While CodeQL detected most CVEs related to CWE-693, e.g., the use of a broken or risky cryptography algorithm (CWE-327). However, there are 19 (39.6%) certain CVEs being detected by no less than two tools, such as *CVE-2018-17187* (CWE-295), *CVE-2018-20318* (CWE-611), and *CVE-2018-20227* (CWE-22), etc. It is observed that these CVEs' patterns are easy for SAST to detect. For instance, *CVE-2018-17187* [63], a vulnerability found in *The Apache Qpid Proton-J transport* [3], is related to *Improper Certificate Validation*, which even syntax-based tools can hit by searching for well-known dangerous sinks such as `X509TrustManager`, and `checkClientTrusted`.

Besides, although SBwFSB and CodeQL detected 3 CVEs related to CWE-22, we found that CodeQL even reported a more precise vulnerability type: *Path Traversal: '\\..\filename'* (CWE-29) for *CVE-2018-20227* [64]. Specifically, there are 20 rules for detecting related vulnerabilities covering both *absolute path traversal* and *relative path traversal*, including {CWE-22, CWE-23, CWE-29, CWE-36}, etc. It unveils that CodeQL has **more complete coverage and fined granularity** on path-traversal vulnerabilities.

Since none of the single tools performs well on the Java CVE Benchmark, and there are different focuses among tools, we try to analyze the effectiveness improvement by combining multiple SAST tools. Here, we selected and combined the SAST tools with the most CVEs found. A CVE is thereby considered found if at least one tool was able to detect it. As the combination of tools can also result in an increase in false positives, we selected those that contain the fewest SAST tools and also output the fewest false positives. The best combination of the 4 tools is **{CodeQL, Horusec, SBwFSB, Semgrep}**, which can cover 45 unique CVEs as shown in Figure 8. However, the CVE_R reaches 27.3% (45/165) and CVE_R_{patch} at 66.7% (30/45), which is an improvement of only

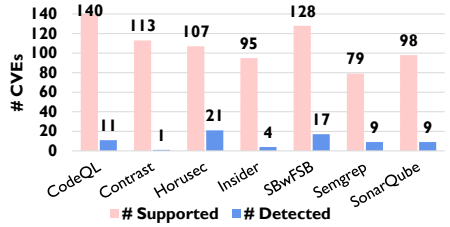


Figure 9: Consistency between the detected CVEs and those claimed to be supported by each tool.

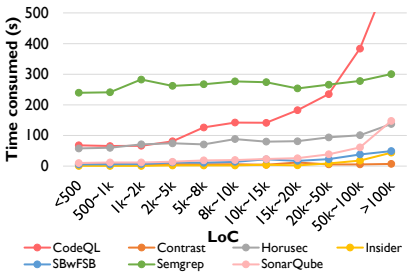


Figure 10: Average performance of SAST tools.

14.6% but 38.1% increase in false positives compared with Horusec. For a combination of three tools, the best one is **{CodeQL, Horusec, SBwFSB}**, with CVE_R at 24.8% and CVE_R_{patch} at 63.4%.

Finding 6: The combination of tools can improve vulnerability detection (45/165, 27.3%) but is not high as expected, which still fails to detect over 70% real-world vulnerabilities, with an approximate cost of a 63.4-66.7% increase in false positives.

Consistency between detected and claimed by each tool. As revealed in Table 2, by mapping their rules to CWE-1000, each tool is able to support a wide range of vulnerabilities but still misses some *CWE Classes*. In detail, CVEs belonging to CWE-{682, 697, 703, 710} are less supported by tools than those of the other *CWE Classes*, although CodeQL implements 204 rules supporting CWE-710. It unveils that it is generally consistent with the most detected *CWE Classes*, i.e., CWE-{664, 693, 707} as mentioned in Section 3.2.2.

However, vulnerabilities related to CWE-682 are only supported by CodeQL, while CodeQL and Contrast claim to support those belonging to CWE-697. These two classes did not appear in our benchmarking experiment, so we are not able to analyze their consistency. When it comes to the two *CWE Classes*, they concern vulnerabilities related to incorrect calculation and incorrect comparison, respectively. Through our analysis, we observed these vulnerabilities are only caused by security-critical calculations/comparisons, most likely causing security-unrelated issues including code smell, etc. Therefore, the influence of the lack of real-world vulnerabilities under the two *CWE Classes* has been weakened.

As unveiled in Figure 9, **there is much over-statement by these tools**. Specifically, these tools are generally over-claimed to support 90.5% vulnerabilities than their actual capacity in our real-world benchmark. It indicates that potential users should select tools cautiously, instead of only relying on tools' claims. Even the best performing tool, Horusec, overstates that 80.4% of CVEs can be detected by its support, whereas actually they are not. Moreover, CodeQL has the most default detecting rules (1,065), with support for unique 196 *CWE Weaknesses*, but it only detected 7.9% CVEs of those claimed to support. However, for specific *CWE Classes*, Horusec owns 40.4% (59/146) rules related to CWE-693, especially for CWE-295 (13), and CWE-798 (28), since it is integrated with GitLeaks. As a result, it also detected 20% CWE-693 related vulnerabilities, including *CVE-2013-2172*, which is caused by an XML signature cryptographic issue. But SBwFSB detected none of the related CVEs although owning (31/152) rules belonging to CWE-693.

Finding 7: These tools are generally over-stated in their capacity for vulnerability detection. Specifically, there are over 90% CVEs failing to be detected although they are claimed to be supported. Especially Contrast, it over-claims to support 99.1% (112/113) CVEs in the Java CVE Benchmark.

3.4 RQ4: Performance Analysis

3.4.1 Setup. ***(1)*** We first collected Java open-source programs from the repositories published in the package manager including *Maven* [29] since they are more likely to be packaged successfully. We got 3,500 programs as the initial list. ***(2)*** We then selected *representative* programs by setting two sub-criteria: ① each program's package should be relied on by at least one package, and ② there exist new packages relying on them within the last three years. Finally, we obtained 1,049 programs that can be packaged, of which the versions are all up-to-date till August 2022. To ensure robustness and consider potential infrastructure variability, we performed each performance measurement three times for each tool. The reported results represent the average of these trials.

3.4.2 Results. We analyzed their runtime performance based on the lines of code (LoC) of the programs.

Performance analysis. Figure 10 shows that the required time by these tools increases as LoC increases. Specifically, the performance does not vary considerably when LoC is less than 50k, but it increases significantly above 50k, particularly for CodeQL. Insider is the fastest among the tools studied, requiring on average less than 10 seconds when LoC is no more than 50k, and about 43.9 seconds to scan even for programs over 100k LoC. It is because Insider is a syntax-based SAST tool, by comparing the source code directly against the pre-defined keywords. Contrast is also efficient, surpassing Insider as the fastest tool when LoC is greater than 20k since its input must be a jar or war file of the Java programs, and the scan is performed by uploading the jar or war file to a cloud server. CodeQL requires more time than the other 6 tools when LoC is over 50k since it involves the aforementioned two steps: ***(1)*** first generates the codebase based on the given program; ***(2)*** then performs semantic analysis involving DFA and CFA on the codebase with queries. Moreover, some queries take a long time to scan, such as *"Taint Path" checking*.

Finding 8: The analysis time increases sharply on programs over 50k LoC. Insider and Contrast are the fastest, while Semgrep and CodeQL require the longest time to finish the scan, with an average of 267s, and 193s, respectively.

Effectiveness vs. performance. Syntax-based tools (Insider, Horusec) generally run faster than semantic-based ones (CodeQL, SBwFSB, SonarQube, and Semgrep), which coincides with our aforementioned assumption. Horusec is the slowest syntax-based tool, which generally takes longer than SonarQube to complete a scan. In particular, it takes an average of 83 seconds to complete a scan when the LoC is less than 50k, and 120 seconds when the LoC is over 50k. It is considered since ***(1)*** Horusec needs to copy the folder of programs to prepare for the scan, ***(2)*** integrates with various security tools including GitLeaks, Trivy, and OWASP Dependency-Check, and ***(3)*** is equipped with more complex detecting patterns than Insider. Meanwhile, Semgrep takes more time than others on each program when LoC is no more than 50k, generally taking 267 seconds on average. Interestingly, it is not affected much by the program size, with an average scan time of 230 seconds on LoC less than 500, while on LoC of 10k-15k, it takes an average of 274 seconds to complete the scan. Although both Semgrep and CodeQL perform semantic-based analysis when scanning, the performance of Semgrep was not influenced by LoC. We summarize the two reasons: ***(1)*** besides a combination of syntax analysis and semantic analysis, there are also some trade-offs between detecting capacity and scan speed in Semgrep, including limited intraprocedural DFA, no pointer or shape analysis, and individual elements in arrays or other data structures are not tracked, etc. It also results in the aforementioned limited effectiveness in vulnerability detection in Section 3.1.2. ***(2)*** Semgrep takes an optimization called *"Single-file analysis"* that directly links scanning with the number of rules, independent of LoC. Specifically, Semgrep slices and runs single files in a given program, which also deprives it of the ability to detect certain complex inter-procedural issues. These insights could guide the development and refactoring of SAST tools to handle continuously updated rules and increasingly complex software implementations, especially for those with over 100k LoC.

Finding 9: There is a trade-off between semantic-based analysis and the performance within Semgrep, which contributes to its scanning performance well. Meanwhile, its deployed file-slicing technology is considered useful when analyzing large programs.

4 DISCUSSION

4.1 Lessons Learned

4.1.1 For Java SAST Developers. ***(1)*** *Improve effectiveness with efficient rules.* Since detecting capacity is the foundation of SAST tools, developers should first ensure their effectiveness. ① Implement rules by extracting exact semantic patterns of vulnerabilities, e.g., to detect CWE-502 vulnerabilities, it is not enough to only search for common sinks such as `readObject()`, DFA and CFA should also be used to trace the tainted path (Section 3.2.2). ② Tools should excel in analyzing real-world code. This requires developers to observe and summarize the features of real-world vulnerabilities when designing rules, e.g., tools should be enhanced by analyzing vulnerabilities with object interactions and method calls (Section 3.2.2). ***(2)*** *Improve the scalability on large programs.* Since users would not consider a time-consuming SAST tool even though it could hit some vulnerabilities, developers should consider the performance when scanning large programs, e.g., the "single-file" analysis in Semgrep would be a useful inspiration (Section 3.4.2).

4.1.2 For Java SAST Researchers. ***(1)*** *A unified mapping reference is essential.* As mentioned in Section 2.3, it is desirable to use publicly available references such as CWE to map detecting rules and vulnerabilities, which would facilitate the evaluation of the effectiveness of SAST tools for various vulnerability types and further gives directions for improvement. ***(2)*** *Call for a more comprehensive and systematic real-world benchmark.* To better understand the actual effectiveness of tools, there is a need for constructing a real-world benchmark containing diverse vulnerability types according to existing references such as CWE. Despite our efforts to include as many CVEs as possible, the benchmark could be further diversified by incorporating more vulnerability types, particularly those belonging to CWE-682 and CWE-697.

4.1.3 For Java SAST Users. ***(1)*** *Select tools according to different application scenarios.* ① As mentioned in Section 3.3.2, Horusec performs better on detecting vulnerabilities related to CWE-611, with CodeQL better on CWE-327 and CWE-22, while SBwFSB outperforms on those related to CWE-20. ② Users are also recommended to choose different tools depending on various phases in SDLC, e.g., during the implementation phase of large programs, faster tools such as Semgrep would be better since the performance is not limited by the size of the program and not requiring the program to be compilable (Section 3.4.2). While for major phases of the SDLC, it may be necessary to choose tools that scan comprehensively and efficiently such as Horusec, which is more effective than others although using syntax-based analysis. ***(2)*** *Call for use of tools combination, and even other vulnerability detection tools.* In practice, we generally recommend using a combination of multiple SAST tools, even better also involving different types of tools such as SCA tools, to facilitate as much as possible shifting-left security during SDLC, e.g., Horusec integrates OWASP Dependency Check in it to detect vulnerable dependencies used (Section 3.1.2). ***(3)*** *Disable rules that are more likely labeled as false positives.* As mentioned in Section 3.1.2, this strategy helps triage through issues reported since certain issue types are more frequently labeled as false positives than others on a specific tool.

4.2 Threats to Validity

4.2.1 External Validity. ***(1)*** Our study's generalizability is the primary external threat. However, we have ensured diversity in our tool selection and comprehensiveness in our dataset to enhance the relevance of our findings. ***(2)*** Another threat relates to our SAST tool selection when focusing on "security-related" tools in Section 2.1. To mitigate this, we selected these tools systematically based on their documentation and proven ability to detect vulnerabilities in the OWASP Benchmark. This not only enabled us to compare tool effectiveness across synthetic and real-world benchmarks but also ensured a fair comparison across tools. Moreover, we required tools to have well-documented rules. While this requirement is

crucial for our analysis approach and consistency analysis (RQ3), we acknowledge that it may limit the variety of tools studied.

4.2.2 Internal Validity. ***(1)*** The first threat pertains to the mapping of rules to CWE. Since the studied tools have mapped their own rules to *CWE Weaknesses* except for SpotBugs, aiding our further mapping to *CWE Classes*. For SpotBugs, which lacks such mapping, we mitigated this threat by conducting a systematic, three-round mapping process involving three co-authors to minimize subjectivity. ***(2)*** Another threat is the validation of vulnerabilities in our benchmark. To mitigate this, we engaged three security experts from our industry partner in a rigorous cross-validation process (Section 2.2.2). This ensured the quality of our benchmark. ***(3)*** The last threat concerns the presence of undetected vulnerabilities in the Java CVE Benchmark. However, we focused on whether the selected tools could find known and existing vulnerabilities, so it is feasible to draw conclusions about their effectiveness.

5 RELATED WORK

5.1 Studies of SAST Tools

There are many existing studies evaluating SAST tools [2, 5, 7, 10, 26, 40, 41, 45, 50, 51, 58, 80, 98, 100, 101].

Most studies evaluating Java SAST tools use either synthetic benchmarks [2, 5, 49, 58] or real-world benchmarks that are limited in size and/or vulnerability types [40, 46, 97]. For instance, the benchmarks used in [2, 5, 49, 58] are synthetic and only consider partial vulnerability types, which could hinder a more comprehensive conclusion. While Kaur et al. [46] compared two SAST tools for Java on a real-world benchmark (Apache tomcat dataset), the vulnerability types of their benchmark are limited, i.e., only involving 5 CWE weaknesses, and they only evaluated the tools' false negatives. Similarly, Goseva-Popstojanova et al. [40] evaluated a commercial Java SAST tool using Tomcat, which contains 32 vulnerabilities grouped into only 4 CWE weaknesses. They considered the effectiveness of tools and their combination but without analysis of their rules mapping or efficiency. Meanwhile, Thung et al. [97] conducted an analysis of the false negatives of five Java SAST tools against three open-source programs over eight years ago. Their findings, which align with our study, revealed that the tools under examination exhibited weaknesses in detecting real-world vulnerabilities. However, their study was limited in terms of the number of tools evaluated and the range of vulnerability types included in their benchmarks. In contrast, our work provides a comprehensive evaluation of Java SAST tools, considering both synthetic and real-world benchmarks, and multiple evaluation perspectives including effectiveness, consistency, and efficiency. This distinguishes our work from previous studies and provides a more holistic understanding of the capabilities of Java SAST tools.

Several studies have evaluated SAST tools in other research areas, such as Android [10–12, 26, 52, 55, 82, 84, 96], C/C++ [1, 23, 50, 106], JavaScript [7], and PHP [60]. For instance, Chen et al. [10] evaluated 4 Android SAST tools on 2,157 security weaknesses of 693 banking apps and proposed a tool named AUSERA to identify data-related weaknesses. Pauck et al. [82] conducted an evaluation to explore whether Android taint analysis tools keep their promises, and proposed ReproDroid, a framework allowing accurate comparison. Mordahl et al. [55] explored the complexities of configuration spaces in Android static taint analysis tools. Similarly, studies such as [50] and [7] have included various real-world vulnerabilities to evaluate SAST tools for C/C++ and JavaScript, respectively. However, their results are not necessarily transferable to Java SAST tools due to the different language constructs. Yet, there is a lack of similar efforts in evaluating SAST tools for Java, particularly with real-world vulnerabilities. This gap in the literature underscores the novelty and importance of our work.

In summary, our work distinguishes it from the state of the art in terms of the considered ***(1)*** **programming languages** (Java SAST tools), ***(2)*** **benchmark types and diversity of vulnerabilities** (synthetic plus the largest real-world vulnerabilities), ***(3)*** **evaluation methodology** (mapping detecting rules and ground truth to CWE hierarchy), ***(4)*** **detection code granularity** (vulnerable *file-level* and *method-level*), and ***(5)*** **evaluation perspectives** (rules coverage, effectiveness, consistency among tools' focuses as well as their over-statements, and runtime performance analysis).

5.2 Studies of Other Analysis Tools

Some studies have been conducted on quality assurance tools, with a primary focus on code-quality issues [41, 48, 51, 98, 100]. For instance, Liu et al. [51] compared 6 Java quality assurance tools with 1,425 code-quality bugs and analyzed the effectiveness of bug warnings. Lenarduzzi et al. [48] performed a study of six tools including SonarQube, with a primary focus on syntax, design, and bad practices. They conducted an analysis of 47 Java projects, assessing the agreement and precision of the tools. Their study offers valuable insights into the overall capabilities of these tools, particularly in identifying low-quality code and improving it through the evaluation of 151 code smells. Contrary to the aforementioned studies, our research specifically concentrates on vulnerability detection, providing a more detailed analysis of SAST tools.

6 CONCLUSION

In this paper, we conducted a comprehensive study on seven SAST tools based on OWASP Benchmark and our constructed real-world benchmark, by evaluating them from effectiveness, consistency, and runtime performance analysis. The comparison and evaluation show that their detection capacity remained lower than expected. Many useful findings were unveiled to facilitate this important research direction, specifically, our work provides actionable guidance on SAST tool development, improvement, and selection for SAST developers, researchers, and potential users.

7 DATA AVAILABILITY

We have released all evaluation data and source code [102].

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2021ZD0114501), China Scholarship Council (202206140052, 202106 140088), and National Research Foundation, Singapore, and Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. 2019. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software* 158 (2019), 110427. https://doi.org/10.1016/j.jss.2019.110427

[2] Midya Alqaradaghi, Gregory Morse, and Tamás Kozsik. 2022. Detecting security vulnerabilities with static analysis - A case study. *Pollack Periodica* 17, 2 (2022), 1–7. https://doi.org/10.1556/606.2021.00454

[3] Apache. 2023. Home - Apache Qpid. https://qpid.apache.org/index.html. (Accessed on 31/01/2023).

[4] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of model checking*. MIT press.

[5] Sindre Beba and Magnus Melseth Karlsen. 2019. *Implementation analysis of open-source Static analysis tools for detecting security vulnerabilities*. Master's thesis. NTNU.

[6] Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. 2019. Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software. *IEEE Transactions on Reliability* 68, 4 (2019), 1384–1403. https://doi.org/10.1109/TR.2019.2937214

[7] Tiago Brito, Mafalda Ferreira, Miguel Monteiro, Pedro Lopes, Miguel Barros, José Fragoso Santos, and Nuno Santos. 2023. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node. js Packages. *arXiv preprint arXiv:2301.05097* (2023).

[8] Joshua Bundt, Andrew Fasano, Brendan Dolan-Gavitt, William Robertson, and Tim Leek. 2021. Evaluating Synthetic Bugs. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (Virtual Event, Hong Kong) *(ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 716–730. https://doi.org/10.1145/3433210.3453096

[9] Checkstyle. 2022. checkstyle – Checkstyle 10.6.0. https://checkstyle.sourceforge.io/. (Accessed on 31/01/2023).

[10] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. 2020. An empirical assessment of security risks of global Android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1310–1322.

[11] Sen Chen, Ting Su, Lingling Fan, Guozhu Meng, Minhui Xue, Yang Liu, and Lihua Xu. 2018. Are mobile banking apps secure? what can be improved?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 797–802.

[12] Sen Chen, Yuxin Zhang, Lingling Fan, Jiaming Li, and Yang Liu. 2022. Ausera: Automated security vulnerability detection for Android apps. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–5.

[13] CodeQL. 2022. CodeQL. https://codeql.github.com/docs/codeql-overview/about-codeql/ (Accessed on 31/01/2023).

[14] MITRE corporation. 2023. Common Vulnerabilities and Exposures. https://cve.mitre.org/. (Accessed on 31/01/2023).

[15] Ctags. 2023. Universal Ctags. https://ctags.io/. (Accessed on 31/01/2023).

[16] CVSS V2. 2023. CVSS v2 Complete Documentation. https://www.first.org/cvss/v2/guide. (Accessed on 16/06/2023).

[17] CVSS V3. 2023. CVSS v3.0 User Guide. https://www.first.org/cvss/v3.0/user-guide. (Accessed on 16/06/2023).

[18] CWE. 2022. CVE-CWE mapping guidance. https://cwe.mitre.org/documents/cwe_usage/guidance.html (Accessed on 31/01/2023).

[19] CWE. 2022. CWE-1000: Research Concepts. https://cwe.mitre.org/data/definitions/1000.html (Accessed on 31/01/2023).

[20] CWE. 2022. CWE-Compatible Products and Services. https://cwe.mitre.org/compatible/compatible.html (Accessed on 31/01/2023).

[21] CWE. 2023. CWE-View - CWE Glossary. https://cwe.mitre.org/documents/glossary/index.html#View. (Accessed on 31/01/2023).

[22] CWE. 2023. Pillar WeaknessCWE Glossary. https://cwe.mitre.org/documents/glossary/index.html. (Accessed on 31/01/2023).

[23] José D'Abruzzo Pereira and Marco Vieira. 2020. On the Use of Open-Source C/C++ Static Analysis Tools in Large Projects. In *2020 16th European Dependable Computing Conference (EDCC)*. 97–102.

[24] Debian. 2023. Debian – The Universal Operating System. https://www.debian.org/. (Accessed on 31/01/2023).

[25] Common Weakness Enumeration. 2022. Common Weakness Enumeration. https://cwe.mitre.org/index.html (Accessed on 31/01/2023).

[26] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering*. 408–419.

[27] FasterXML. 2020. jackson-dataformats-binary. https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformats-binary. (Accessed on 31/01/2023).

[28] Forum of Incident Response and Security Teams. 2023. Common Vulnerability Scoring System SIG. https://www.first.org/cvss/. (Accessed on 12/06/2023).

[29] The Apache Software Foundation. 2023. Maven – Welcome to Apache Maven. https://maven.apache.org/. (Accessed on 31/01/2023).

[30] The OWASP Foundation. 2020. OWASP-Top-Ten-Benchmark, 2020. https://github.com/jrbermh/OWASP-Top-Ten-Benchmark (Accessed on 31/01/2023).

[31] The OWASP Foundation. 2022. OWASP Benchmark. https://owasp.org/www-project-benchmark/ (Accessed on 31/01/2023).

[32] The OWASP Foundation. 2023. OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/. (Accessed on 31/01/2023).

[33] The OWASP Foundation. 2023. Software Component Analysis. https://owasp.org/www-community/Component_Analysis. (Accessed on 31/01/2023).

[34] GitHub. 2022. Awesome static analysis. https://github.com/mre/awesome-static-analysis#multiple-languages-1 (Accessed on 22/08/2022).

[35] GitHub. 2022. GitHub-analysis-tools-dev. https://github.com/analysis-tools-dev/static-analysis#java (Accessed on 22/08/2022).

[36] GitHub. 2023. GitHub code scanning. https://github.blog/2022-08-15-the-next-step-for-lgtm-com-github-code-scanning/. (Accessed on 31/01/2023).

[37] GitHub. 2023. Gitleaks. https://gitleaks.io/. (Accessed on 31/01/2023).

[38] Google. 2022. Error Prone. https://errorprone.info/. (Accessed on 31/01/2023).

[39] Google. 2023. Google-java-format. https://github.com/google/google-java-format. (Accessed on 31/01/2023).

[40] Katerina Goseva-Popstojanova and Andrei Perhinschi. 2015. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology* 68 (2015), 18–33. https://doi.org/10.1016/j.infsof.2015.08.002

[41] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/3238147.3238213

[42] HCL. 2023. HCL AppScan CodeSweep. https://marketplace.visualstudio.com/items?itemName=HCLTechnologies.hclappscancodesweep. (Accessed on 31/01/2023).

[43] Jerónimo Hernández-González, Daniel Rodriguez, Inaki Inza, Rachel Harrison, and Jose A Lozano. 2018. Learning to classify software defects from crowds: a novel approach. *Applied Soft Computing* 62 (2018), 579–591.

[44] Insidersec. 2022. Insider. https://github.com/insidersec/insider (Accessed on 31/01/2023).

[45] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting False Alarms from Automatic Static Analysis Tools: How Far Are We?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 698-709.

[46] Arvinder Kaur and Ruchikaa Nayyar. 2020. A Comparative Study of Static Code Analysis tools for Vulnerability Detection in C/C++ and JAVA Source Code. *Procedia Computer Science* 171 (2020), 2023–2029. Third International Conference on Computing and Network Communications (CoCoNet'19).

[47] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1, 4 (1992), 323–337.

[48] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575.

[49] Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of open-source IDE plugins for detecting security vulnerabilities. In *Proceedings of the Evaluation and Assessment on Software Engineering*. 200–209.

[50] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 544-555. https://doi.org/10.1145/3533767.3534380

[51] Han Liu, Sen Chen, Ruitao Feng, Chengwei Liu, Kaixuan Li, Zhengzi Xu, Liming Nie, Yang Liu, and Yixiang Chen. 2023. A Comprehensive Study on Quality Assurance Tools for Java. In *Proceedings of the 32st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA.

[52] Linghui Luo, Felix Pauck, Goran Piskachev, Manuel Benz, Ivan Pashchenko, Martin Mory, Eric Bodden, Ben Hermann, and Fabio Massacci. 2022. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering* 27 (2022), 1–41.

[53] Maven. 2023. Jackson Databind. https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind. (Accessed on 16/06/2023).

[54] Meta. 2023. Infer Static Analyzer. https://fbinfer.com/. (Accessed on 1/06/2023).

[55] Austin Mordahl and Shiyi Wei. 2021. The impact of tool configuration spaces on the evaluation of configurable taint analysis for android. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 466–477.

[56] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 532-543. https://doi.org/10.1145/3533767.3534374

[57] National Vulnerability Database. 2023. NVD-Home. https://nvd.nist.gov/. (Accessed on 31/01/2023).

[58] 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. *Computers & Security* 111 (2021), 102470.

[59] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. springer.

[60] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2019. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing* 101 (2019), 161–185.

[61] NVD. 2014. CVE-2014-3651. https://nvd.nist.gov/vuln/detail/CVE-2014-3651. (Accessed on 31/01/2023).

[62] NVD. 2015. CVE-2015-2913. https://nvd.nist.gov/vuln/detail/CVE-2015-2913 (Accessed on 31/01/2023).

[63] NVD. 2018. CVE-2018-17187. https://nvd.nist.gov/vuln/detail/CVE-2018-17187 (Accessed on 31/01/2023).

[64] NVD. 2018. CVE-2018-20227. https://nvd.nist.gov/vuln/detail/CVE-2018-20227 (Accessed on 31/01/2023).

[65] NVD. 2019. CVE-2019-18393. https://nvd.nist.gov/vuln/detail/CVE-2019-18393 (Accessed on 31/01/2023).

[66] NVD. 2021. Log4Shell: CVE-2021-44228. https://nvd.nist.gov/vuln/detail/CVE-2021-44228. (Accessed on 31/01/2023).

[67] NVD. 2022. Spring4Shell: CVE-2022-22965. https://nvd.nist.gov/vuln/detail/cve-2022-22965. (Accessed on 31/01/2023).

[68] The University of Maryland. 2022. FindBugs. http://findbugs.sourceforge.net/ (Accessed on 31/01/2023).

[69] The University of Maryland. 2022. FindSecurityBugs. https://find-sec-bugs.github.io/ (Accessed on 31/01/2023).

[70] The University of Maryland. 2022. SpotBugs. https://spotbugs.github.io/ (Accessed on 31/01/2023).

[71] National Institute of Standards and Technology. 2017. Juliet Test Suite. https://samate.nist.gov/SARD/test-suites (Accessed on 31/01/2023).

[72] National Institute of Standards and Technology. 2022. NIST: Free for Open Source Application Security Tools. https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers (Accessed on 22/08/2022).

[73] National Institute of Standards and Technology. 2022. SAMATE: Source Code Security Analyzers. https://www.nist.gov/itl/ssd/software-quality-group/source-code-security-analyzers (Accessed on 22/08/2022).

[74] Opensecurity. 2022. NodeJSScan. https://github.com/ajinabraham/nodejsscan (Accessed on 31/01/2023).

[75] OpenSSF. 2020. OpenSSF CVE Benchmark. https://github.com/ossf-cve-benchmark/ossf-cve-benchmark (Accessed on 31/01/2023).

[76] OpenSSF. 2022. Open Source Security Foundation. https://openssf.org/ (Accessed on 31/01/2023).

[77] OWASP. 2022. Free for Open Source Application Security Tools. https://owasp.org/www-community/Free_for_Open_Source_Application_Security_Tools (Accessed on 22/08/2022).

[78] OWASP. 2022. Source Code Analysis Tools. https://owasp.org/www-community/Source_Code_Analysis_Tools (Accessed on 22/08/2022).

[79] oxsecurity. 2023. Megalinter. https://github.com/oxsecurity/megalinter. (Accessed on 31/01/2023).

[80] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. 2018. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming*, Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar (Eds.). Springer International Publishing, Cham, 86–103.

[81] Yuanyuan Pan. 2019. Interactive application security testing. In *2019 International Conference on Smart Grid and Electrical Automation (ICSGEA)*. IEEE, 558–561.

[82] Felix Pauck, Eric Bodden, and Heike Wehrheim. 2018. Do Android Taint Analysis Tools Keep Their Promises?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 331–341.

[83] PMD. 2023. PMD. https://pmd.github.io/. (Accessed on 31/01/2023).

[84] Lina Qiu, Yingying Wang, and Julia Rubin. 2018. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 176–186.

[85] R2C. 2022. Semgrep. https://www.semgrep.dev/ (Accessed on 31/01/2023).

[86] RedHat. 2018. What is DevSecOps? https://www.redhat.com/en/topics/devops/what-is-devsecops. (Accessed on 31/01/2023).

[87] RedHat. 2023. Red Hat Bugzilla Main Page. https://bugzilla.redhat.com/. (Accessed on 31/05/2023).

[88] Reshift. 2023. Reshift. https://www.softwaresecured.com/. (Accessed on 31/01/2023).

[89] Aqua Security. 2023. Trivy. https://trivy.dev/. (Accessed on 31/01/2023).

[90] Contrast Security. 2022. Contrast Security. https://www.contrastsecurity.com/ (Accessed on 31/01/2023).

[91] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[92] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *Proceedings of the Sixteenth USENIX Conference on Usable Privacy and Security (SOUPS'20)*. USENIX Association, USA, Article 13, 18 pages.

[93] SonarSource. 2022. SonarQube. https://www.sonarqube.org/ (Accessed on 31/01/2023).

[94] Spark. 2018. spark/src/main/java/spark/resource/ClassPathResource.java at 27236534e90bd2bfe339fd65fe6ddd6a9f0304e1. https://github.com/perwendel/spark/blob/030e9d00125cbd1ad759668f85488aba1019c668~1/src/main/java/spark/resource/ClassPathResource.java. (Accessed on 31/01/2023).

[95] Martin R Stytz and Sheila B Banks. 2006. Dynamic software security testing. *IEEE security & privacy* 4, 3 (2006), 77–79.

[96] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why my app crashes? understanding and benchmarking framework-specific exceptions of Android apps. *IEEE Transactions on Software Engineering* 48, 4 (2020), 1115–1137.

[97] Ferdian Thung, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T Devanbu. 2015. To what extent could we detect field defects? An extended empirical study of false negatives in static bug-finding tools. *Automated Software Engineering* 22 (2015), 561–602. https://doi.org/10.1007/s10515-014-0169-8

[98] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. Association for Computing Machinery, New York, NY, USA, 50–59. https://doi.org/10.1145/2351676.2351685

[99] TIOBE. 2023. The Java Programming Language-TIOBE. https://www.tiobe.com/tiobe-index/java/. (Accessed on 31/01/2023).

[100] David A. Tomassi. 2018. Bugs in the Wild: Examining the Effectiveness of Static Analyzers at Finding Real-World Bugs. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 980–982. https://doi.org/10.1145/3236024.3275439

[101] Andreas Wagner. and Johannes Sametinger. 2014. Using the Juliet Test Suite to Compare Static Security Scanners. In *Proceedings of the 11th International Conference on Security and Cryptography - SECRYPT, (ICETE 2014)*. INSTICC, SciTePress, 244–252. https://doi.org/10.5220/0005032902440252

[102] Website of This Study. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. https://sites.google.com/view/java-sast-study/home (Accessed on 31/01/2023).

[103] Website of This Study. 2023. Tools Selection. https://sites.google.com/view/java-sast-study/tool-selection (Accessed on 31/01/2023).

[104] Wikipedia. 2022. List of tools for static code analysis. https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis (Accessed on 22/08/2022).

[105] Wikipedia. 2023. Linter-Wikipedia. https://en.wikipedia.org/wiki/Lint_(software). (Accessed on 22/06/2023).

[106] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, and M.A. Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32, 4 (2006), 240–253.

[107] Zupit. 2022. Horusec. https://docs.horusec.io/docs/overview/ (Accessed on 31/01/2023).

Received 2023-02-02; accepted 2023-07-27