IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 22 February 2024, accepted 29 March 2024, date of publication 16 April 2024, date of current version 24 April 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3389955

RESEARCH ARTICLE

Comprehensive Evaluation of Static Analysis Tools for Their Performance in Finding Vulnerabilities in Java Code

MIDYA ALQARADAGHI[1,2] AND TAMÁS KOZSIK[1]
[1]Department of Programming Languages and Compilers, Eötvös Loránd University (ELTE), 1053 Budapest, Hungary
[2]Technical Engineering College of Kirkuk, Northern Technical University, Kirkuk 36001, Iraq

Corresponding author: Midya Alqaradaghi (alqaradaghi.midya@inf.elte.hu)

This work was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development, and Innovation Fund under Project TKP2021-NVA-29 and Project TKP2021-NVA.

ABSTRACT Various static code analysis tools have been designed to automatically detect software faults and security vulnerabilities. This paper aims to 1) conduct an empirical evaluation to assess the performance of five free and state-of-the-art static analysis tools in detecting Java security vulnerabilities using a well-defined and repeatable approach; 2) report on the vulnerabilities that are best and worst detected by static Java analyzers. We used the Juliet benchmark test suite in a controlled experiment to assess the effectiveness of five widely used Java static analysis tools. The vulnerabilities were successfully detected by one, two, or three tools. Only one vulnerability has been detected by four tools. The tools missed 13% of the Java vulnerability categories appearing in our experiment. More critically, none of the five tools could identify all the vulnerabilities in our experiment. We conclude that, despite recent improvements in their methodologies, current state-of-the-art static analysis tools are still ineffective for identifying the security vulnerabilities occurring in a small-scale, artificial test suite.

INDEX TERMS Empirical evaluation, Java, Juliet test suite, performance metrics, static analysis tools, security vulnerabilities.

I. INTRODUCTION

Industries across various sectors, such as finance, e-commerce, transportation, energy, and healthcare, rely heavily on computer systems and networks. The rise in cyberattacks by enemies and organized crime groups is a result of the expansion of online commercial activity, connected operations, and the amount of sensitive data available online [1]. A 2020 report published in Cybercrime Magazine states that during the next five years, the costs associated with cybercrime would rise by fifteen percent a year, resulting in a total cost of USD 10.5 trillion per year by 2025, which indicates a significant rise from the approximate USD 3 trillion cost of cybercrime in 2015 [2].

The presence of programming errors reduces the quality of the software and increases the cost of development life cycle [3]. Poor program quality is a leading cause of security vulnerabilities. These vulnerabilities refer to security weaknesses that may arise from a system's requirements, design, implementation, or operation and can be exploited intentionally or unintentionally to cause a security breach [4]. A security failure generally occurs due to the presence of a vulnerability. This highlights the importance of incorporating security measures in the design and development of software systems. Moreover, it is important to improve verification and validation abilities to solve data security and protection issues. Empirical studies conducted previously [5], [6] have indicated that the usage of many techniques for detecting and preventing vulnerabilities is crucial throughout the entire software development cycle. Static analysis of source code (and bytecode) is a vulnerability detection technique that enables scalable code review for security early in the software development cycle. It does not require executable systems and can be applied to some specific portions of the code

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

base. Over the last decade, static analysis technologies have significantly expanded beyond basic lexical analysis to incorporate more sophisticated methodologies. However, static analysis in general cases hits the undecidability line, i.e., it is mathematically not feasible to build an algorithm that can provide accurate answers for all cases [7]. As a result, these techniques are either incapable of detecting every vulnerability in the source code, resulting in false negatives, and/or they may report security vulnerabilities where there are none, resulting in false positives. The ideal situation is when the tool can detect as many vulnerabilities as possible (i.e., true positives) with minimal false positives.

The purpose of this study is to empirically evaluate how well-known static analysis tools identify programming flaws and security vulnerabilities. Five cutting-edge, freely available static analysis tools form our list: Facebook Infer [8], SonarQube Community Edition [9], SpotBugs [10], Find Security Bugs [11], and PMD [12]. You can refer to Section III-A for more information about the process of selecting the tools.

In this study, we evaluate the capabilities of the tools for detecting artificial vulnerabilities. Artificial vulnerabilities can be supplied in huge numbers, allowing for more data collection than would otherwise be possible. The Juliet benchmark test suite, developed by the National Security Agency's (NSA) Center for Assured Software and is freely available [13], served as the basis for our study. It consists of multiple artificially constructed test cases, each of which only targets one type of vulnerability as defined by the Common Weakness Enumeration (CWE) [14]. After evaluating the Java test cases of the Juliet test suite, we selected a relevant subset of them, which covers different types of vulnerabilities. Then we could perform an automated evaluation, followed by the computation of performance measures for each tool.

Despite the advantages of working with vulnerabilities occurring in real-world code bases, such as being "real", more representative, and more informative, we decided not to use real-world programs as input because (i) they rarely contain a sufficient number of security vulnerabilities and (ii) it is hard to obtain a guarantee of the correctness of the found issues, especially since we target a huge number of weakness categories in our research.

The motivations of the paper are the following.

- The TIOBE index shows that Java is still one of the most widely used programming languages in the software industry, despite a small drop in popularity [15].
- Static analysis techniques are useful for finding code flaws and security issues. However, it is difficult for the IT industries to select one over the other due to the number of these tools. This paper aids in that goal.

The main contributions of the paper are:

- Evaluation and validation of the Juliet Test Suite.
- Individual and collective evaluation results of five well-known static analysis tools using common performance metrics.
- Presentation of a well-designed and repeatable approach for evaluating the performance of the tools.
- Report on the classes of vulnerabilities that are best and worst detected by static Java analyzers. This will help in finding future research directions.

The research questions that this work tries to answer are:

- How accurately can the state-of-the-art static analysis tools identify flaws?
- Which tool could detect more security vulnerabilities? Which tool had the highest detection rate[1]?
- Which tool is more precise in detecting vulnerabilities?
- What are the vulnerabilities that have been perfectly detected, i.e., with optimal performance, by at least one static analysis tool? What are those that have been completely overlooked by all the tools?

The remainder of the paper is structured as follows. An overview of the related concepts is given in Section II. The method of preparing Juliet for the experiment, along with the experimental design and execution, are explained in Section III. In Sections IV and V, we present and discuss results of our experiment, and answer the research questions. Section VI discusses the potential threats to the validity of the study. Section VII provides a literature review of previous studies on a similar topic, and finally, Section VIII concludes the paper.

II. BACKGROUND

This section provides background information on the Common Weakness Enumeration Taxonomy [14], the Juliet test suite [13], and the static analysis tools of the study.

A. COMMON WEAKNESS ENUMERATION (CWE)

This taxonomy aims to provide a comprehensive reference of software flaws and vulnerabilities. The U.S. Department of Homeland Security supports it, and MITRE Corporation is in charge of managing it [14]. Each CWE has a unique ID that identifies it concerning a certain category or type of vulnerability, such as "Information Leak Error" (CWE 209) and "Null Pointer Dereference" (CWE 476). The CWEs have a hierarchical structure, with more general categories at the top. Subcategories of these higher-level CWEs might exist. Every CWE has zero or more children, and one or more parents. The vulnerabilities get more specific as you proceed down the path. Many static code analysis techniques make explicit or implicit connections between CWEs and the warnings they generate.

B. JULIET TEST SUITE (1.3) FOR JAVA

Developed by the U.S. National Security Agency's Center of Assured Software (CAS), the Juliet Test Suite is an artificial benchmark consisting of 112 weakness categories

[1]Detection rate is the percentage of the number of actually detected vulnerabilities to the number of vulnerabilities that the tool's documentation claims to detect.

(CWEs). It has been used in many studies to evaluate different security assurance tools, including static analysis tools.

The test suite comprises various programming languages, including Java, with individual test cases designed to target particular types of flaws. Juliet incorporates the *flow variant* concept to categorize flaws and vulnerabilities depending on the type of analysis needed to detect them. The test cases in Juliet are designed to target a single CWE, and are generated with one of three possible flow variant types: baseline, control flow, or data flow. Those flow variant types also come with an ID for each.[2]

The naming conventions used by the test cases in the Juliet test suite indicate whether or not they have flaws; the identifier prefix ''bad'' is used to indicate the presence of faulty code constructs that harbor vulnerabilities, while the identifier prefix ''good'' is used to name similar but invulnerable code fragments. Therefore, benchmarking static analysis tools on them may be done automatically. Each Juliet test case comprises one ''bad'' method and one or more ''good'' methods.

C. STATIC ANALYSIS TOOLS OF THE STUDY

This section presents the five static analysis tools evaluated for their performance in detecting Java security vulnerabilities.

a: FACEBOOK INFER (INFER)

Infer can analyze Java, C, and Objective-C code. Facebook has applied Infer in the development of several of its main applications, including Facebook, Messenger, and Instagram, to validate the correctness of code modifications. One of the strengths of the tool is its ability to work on Android code, but it can analyze other types of Java and C code as well. Infer is mainly used to detect memory and resource leaks, null pointer dereference, and other similar issues that are common and critical in mobile application development [8]. One of the major limitations of this tool is its lack of a good user interface.

b: SONARQUBE COMMUNITY EDITION (SONAR)

Sonar is a free tool used for continuous code quality inspection, code analysis, and reporting. It can analyze code written in more than 25 programming languages, including Java, C++, C#, Python, Ruby, and JavaScript. Sonar checks the code for potential issues like bugs, vulnerabilities, and code smells, which are then reported to the developers. The tool uses a set of rules[3] to detect these issues, which can be customized by the developers to fit their specific needs. It can be integrated with modern development environments like Visual Studio, Eclipse, and IntelliJ, as well as with CI/CD tools such as Jenkins and GitLab for continuous code quality checks during development. One of the main features of Sonar is its ability to provide a holistic view of code quality across multiple projects. It can track the progress of code quality improvements over time with reports and dashboards for metrics like code coverage, complexity, and duplication [9].

c: SPOTBUGS (SB)

This is a free and open-source static analysis tool for Java that helps identify more than 400 potential bugs and security vulnerabilities. It works by analyzing the bytecode of Java applications and identifying patterns that may indicate issues such as null pointer dereference, array index out-of-bounds errors, and potential security vulnerabilities. SB can be used standalone and by providing a variety of plugins and integrations for popular development environments and build tools, such as Eclipse, IntelliJ, and Gradle. It also has a large community of contributors who actively maintain and improve it [10].

d: FIND SECURITY BUGS (FSB)

It is the SB security plugin for Java. It helps detect 141 different vulnerability types in Java applications, such as SQL injection, cross-site scripting (XSS), and buffer overflows, so its main focus is taint analysis-related issues. FSB can be integrated with different IDEs, provides integration with various build tools and continuous integration (CI) systems like Jenkins, Maven, and Ant, and can be easily integrated into development workflows. Additionally, it offers a range of configuration options that allow developers to customize and fine-tune the scanning process [11]. FSB is actively maintained by a community of contributors and receives regular updates to ensure that it is up-to-date with the latest security threats and best practices. Although it is a plug-in, we evaluated it as a standalone tool in this study.

e: PMD

The main focus of this tool is Java, although it supports other languages as well. PMD identifies common programming flaws and bad smells like unused variables, empty catch blocks, and unnecessary object creation. PMD is free, open-source, and has many built-in checks. It also supports an extensible architecture that allows developers to define their own custom rules to suit their specific needs. To use PMD in the best way and make it serve as a quality gate, it can be integrated into the build process. This will enforce a coding standard for the whole code base. PMD can be executed

[2]Test cases without additional control or data flow complexity with a flow variation of ''01'' represent the most basic version of the defects. The term ''Baseline'' refers to this group of test cases. More complex test cases are those that have a flow variant other than ''01''. The ''Control Flow'' test cases are those that have a flow variant ranging from ''02'' to ''22'' (inclusive) and cover a variety of control flow constructions. The ''Data Flow'' test cases are those that have a flow variation of ''31'' or above, which include a variety of data flow constructions.

[3]In this research, we have used the terms ''rule'' and ''checker'' interchangeably, referring to them by their respective terms in the documentation of each tool.

TABLE 1. Features of the tools.

Tool	Install	UI [a]	Techniques	Inspect
Infer	easy	CLI	taint flow, data flow, abstract interpretation formal verification	both
Sonar	hard	CLI/GUI	taint flow, syntax and data flow	both
SB	easy	CLI/GUI	syntax and data flow	bytecode
FSB	easy	CLI/GUI	syntax and data flow	bytecode
PMD	easy	CLI/GUI	syntax and data flow	source code

[a]UI Stands for User Interface, and there are 2 types; CLI = Command Line Interface, and GUI = Graphical User Interface.

through the command line as an Ant task, a Gradle task, or a Maven goal [12]. PMD is actively maintained by a community of contributors, and receives regular updates to ensure that it is up-to-date with the latest programming language standards and best practices.

Table 1 presents the features of the tools under study that have been extracted from the tools' documentation.

III. RESEARCH METHODOLOGY

In this section, we first describe the design of the experiment, then present the process of preparing the Juliet Test Suite for the experiment and the evaluation metrics used. Finally, the execution of the experiment is demonstrated. All the datasets of this section are presented in the first author's Zenodo repository [16].

A. EXPERIMENT DESIGN

Our goal is to evaluate how well five of the most advanced static code analysis tools detect security flaws in a variety of Java CWEs with a large number of test cases. The two factors of the experiment are (1) the static analysis tool; and (2) the vulnerability type being evaluated.

The levels of the first factor are the specific tools that have been used in the study. We examined a survey with twenty commercial and free tools before beginning the selection process. Subsequently, we employed the following criteria to reduce the number of selected tools: (a) be free and widely used; (b) specifically recognize security vulnerabilities as defined by The MITRE Corporation [14]; (c) support Java; (d) support the detection of multiple types of vulnerabilities; and (e) be capable of analyzing large software applications. Due to the existence of many researches on commercial tools [5], [6], [17], [18], [19], none of them were chosen. We preferred to research free tools, especially since they are believed to be of equal quality and importance as commercial ones [20]. Consequently, we selected Infer, Sonar, SB, FSB, and PMD.

In selecting the levels of the second factor, vulnerability type, we followed a well-defined approach for filtering and validating the 112 CWEs of Juliet, resulting in 70 CWEs. The process is discussed in Section III-B. The list of the CWEs that have been selected and used in our experiment is given in Table 8.

B. PREPARING JULIET TEST SUITE

Employing the Juliet test suite as input in an experimental approach offers several benefits, including (1) analyzing in a controlled environment; (2) measuring the tools' performance for a significant number of test cases that address diverse vulnerabilities; (3) the ability to assess true negatives (vulnerabilities overlooked by the tools) to reflect recall (the likelihood of detection) in addition to identifying true positives and false positives; and (4) facilitating reproducible empirical research. However, according to previous experience with Juliet [21], [22], it contains bugs. Moreover, not all of these CWEs are relevant to the tools of our study, i.e., each of the tools targets only specific CWEs. Therefore, we needed to select the relevant CWEs. To achieve that, we decided to validate and evaluate the CWEs and their test cases.[4] Consequently, we propose a well-defined approach for selecting, validating, and evaluating them, as follows.

1) Selecting relevant CWEs: First, we excluded from the study those CWEs that are deprecated according to the Open Worldwide Application Security Project (OWASP) [23]. Then, we excluded those not targeted by at least one of the five tools of the study (according to the tools' documentation).
2) Validating individual test cases: A valid test case is one that represents the specified vulnerability according to the vulnerability definition in OWASP. For each CWE category considered relevant in the previous step, we manually reviewed their individual test cases; if they were not valid, we excluded them.
3) Evaluating individual test cases: The last step is to evaluate the CWEs according to the following criteria: (1) it should include both positive and negative test cases; (2) the number of test cases should be at least five; and (3) among the test cases, there are at least two "flow variants" (baseline, control flow, and/or data flow; check section II for more information).

Figure 1 presents the above approach, which results in two lists: *excluded CWEs* consisting of 42 CWEs, presented in Table 7 (in the Appendix) along with the reason for the exclusion,[5] and *valid CWEs* presented in Table 8 (also in the Appendix), which consists of 70 CWEs and has been used in evaluating the five tools of this study.

C. EVALUATION METRICS

We used four response variables, namely recall, false alarm rate, precision, and F-score, to represent distinct aspects of the performance of static code analysis tools in identifying security vulnerabilities. For each tool and each CWE listed in Table 8, we computed the confusion matrix (i.e., the number of true positives (TP), false negatives (FN), false positives

[4]In general, validation is about verifying that a system meets its intended requirements and specifications, while evaluation is the process of measuring and comparing against predefined criteria.

[5]Despite that CWE476 is included in this table, it is also included in the table of valid CWEs, because we only excluded 17 test cases from this set.

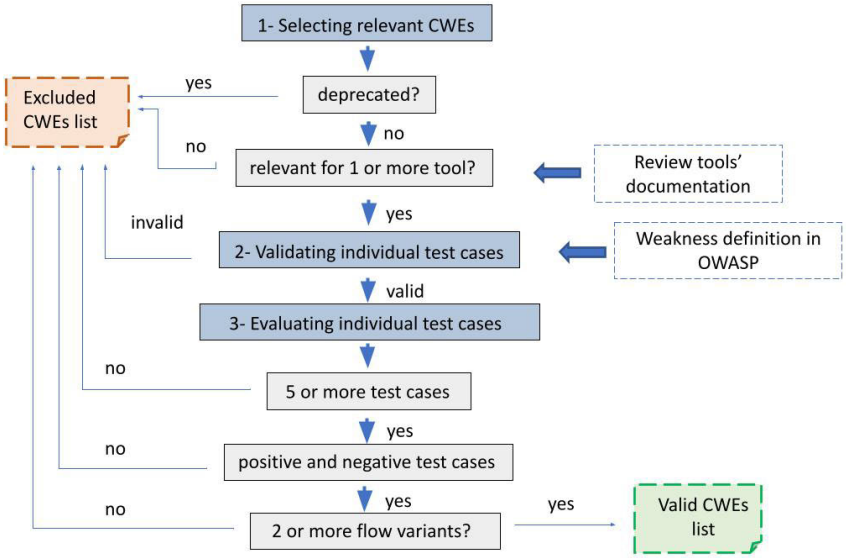


FIGURE 1. Juliet preparation process.

(FP), and true negatives (TN), and then we used them to calculate the following metrics.

$$Recall_i = \frac{TP_i}{TP_i + FN_i} \tag{1}$$

$$False\ alarm\ rate_i = \frac{FP_i}{FP_i + TN_i} \tag{2}$$

$$Precision_i = \frac{TP_i}{TP_i + FP_i} \tag{3}$$

$$F - Score_i = \frac{2 \cdot Recall_i \cdot Precision_i}{Recall_i + Precision_i} \tag{4}$$

The term *Recall*; which can also be referred to as the TP-rate, or probability of detection, is described by Eq. (1). It determines the likelihood of identifying vulnerabilities accurately (it is calculated by dividing the number of TP by the sum of TP and FN). The recall metric solely focuses on *bad* methods in Juliet, and it signifies the portion of these methods that a tool correctly identifies.

False alarm rate (also called FP-rate, or probability of false alarm), which is given with Eq. (2), measures the likelihood of misclassifying correct code as flawed, as it calculates the ratio of FP to the sum of FP and TN. The false alarm rate only considers Juliet's good methods; it represents the portion of good methods mistakenly classified as vulnerable.

Recall and false alarm rates evaluate the performance of static code analyzers on two distinct aspects. Recall evaluates the analyzer's ability to detect security flaws, making it an important indicator. However, since tools are imperfect, they may report some code as they contain vulnerabilities when they do not. This is called the false alarm rate. This latter metric is crucial because developers must manually review each alert to determine its accuracy. High false alarm rates (i.e., many correct methods reported as flawed) can result in substantial efforts being wasted. Higher performance is indicated by a high recall and a low false alarm rate, ranging from 0 to 1. In an ideal scenario, the recall would be 1, and the false alarm rate would be 0.

The precision relates to a tool's capability to detect errors. One of the main causes of static analysis tools' perceived uselessness is their lack of precision [24]. It is given with Equation (3), which provides the ratio of TP to all detections; i.e., the total of TP and FP. Therefore, precision is concerned with every test case that has been detected; it displays the ratio of the number of (flawed and unflawed) constructions reported by a tool to the proportion of flawed constructs that have been successfully discovered.

The harmonic mean of recall and precision is called F-score and given with Eq. (4), it integrates these two metrics in a single one. We used this metric to represent the tools' accuracy.

In addition to the previously explained metrics, we used two central tendency measurements, the mean, and the median, for presenting the overall evaluation of the tools (i.e., tools' evaluation across all the CWEs).

In our experiment, we used a measurement tool composed of multiple Python scripts, to enable the automatic collection of data and computation of response variables and their mean and median.

D. EXPERIMENT EXECUTION

The *valid CWEs list* of Juliet has been selected according to the approach presented in Section III-B and has been analyzed by each of the five tools of this study. Figure 2 presents the flowchart of the experiment execution.

a: STEP 1: FOR EACH OF THE FIVE TOOLS, REVIEW THE DOCUMENTATION, TO IDENTIFY AND ACTIVATE THE RELATED CHECKER(S)

We reviewed the documentation of all the tools for identifying the possible checkers included in each tool for detecting any

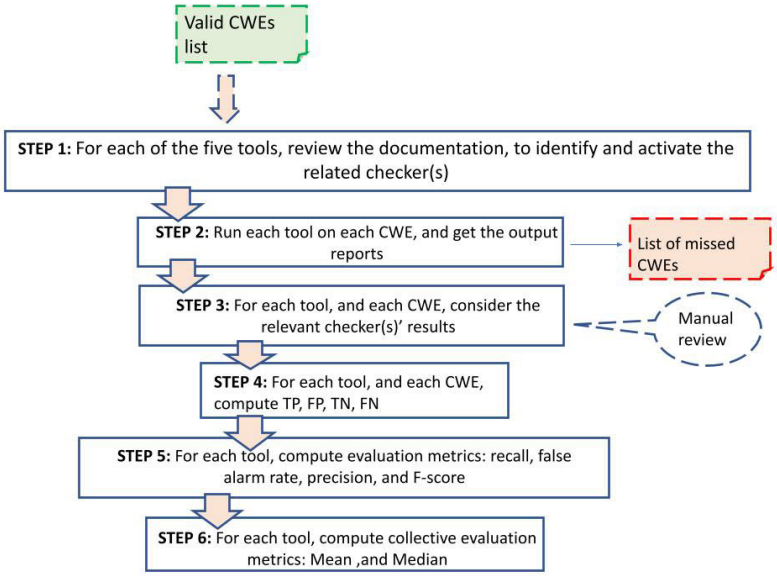


FIGURE 2. Experiment execution.

of the 70 CWEs of the Juliet test suite that have been selected according to the approach explained in Section III-B. This step was necessary to activate the required checkers if they were not active by default, and to exclude the unnecessary checkers of these tools (if possible) in the analyzing process.

b: STEP 2: RUN EACH TOOL ON EACH CWE_I, AND GET THE OUTPUT REPORTS

In this step, we analyzed each CWE of the *valid CWEs list* presented in Table 8 by running each tool and getting the output reports. Consequently, we could get the list of CWEs that have been detected by each tool, and a list of CWEs that have not been detected by any of the tools, despite the existence of relevant checker(s).[6]

c: STEP 3: FOR EACH TOOL, AND EACH CWE_I, CONSIDER THE RELEVANT CHECKER(S)' RESULTS

The output report obtained in the previous step may include the results of different checkers, sometimes for the same (Tool, CWE) pair.[7] However, these checkers differ in their detection quality; i.e., they result in different recall and false alarm rate values. Therefore, in this step, we manually reviewed some of the detected issues for each checker, studied their quality in detecting the specified vulnerability, and then decided which checker's output we wanted to consider.[8]

d: STEP 4: FOR EACH TOOL, AND EACH CWE_I, COMPUTE TP_I, FP_I, TN_I, AND FN_I

Using the output report of Step 2 and taking into consideration the relevant checker results from Step 3, the confusion matrix has been computed for each CWE and each tool of the study, as follows.

- It is a true positive (TP_i) if the tool detects the *bad method* of the specified test case.
- It is a false negative (FN_i) when the tool does not detect the *bad method* of a CWE.
- It is a false positive (FP_i) when the tool reports a *good method* as flawed.
- It is a true negative (TN_i) when the tool reports no warnings related to a *good method*.

To validate our computation of the confusion matrix values, we used the following as a sanity check:

- number of bad methods for CWE_i equals to $TP_i + FN_i$;
- number of good methods for CWE_i equals to $FP_i + TN_i$.

e: STEP 5: COMPUTE THE RESPONSE VARIABLES FOR EACH TOOL DETECTING EACH CWE

Using the TP_i, FN_i, FP_i, and TN_i that have been obtained for each tool detecting each CWE_i in Step 4, we calculated the response variables; i.e., recall, false alarm rate, precision, and F-score. The response variables are specified by Eqs. (1), (2), (3), and (4), respectively. Section IV-B provides a detailed report on the analysis.

f: STEP 6: FOR EACH TOOL, COMPUTE COLLECTIVE EVALUATION METRICS

We calculated the standard measures of central tendency (the mean and median) for each tool, given each performance metric; recall, false alarm rate, precision, and F-score. The results are explained in Section IV-C, and presented in Table 3.

IV. RESULTS

In this section, we present the performance evaluation of the five tools of the study. First, we present information

[6]List of missed CWEs are presented in Section IV.

[7]Multiple reports of the same CWE by various checkers have been mostly observed in the case of SB and FSB tools.

[8]Note that only the detections related to the investigated vulnerabilities have been considered. Incidental flaws – which are present in nearly all Juliet test cases – were also disregarded.

TABLE 2. Number of existing CWE checkers along with the number of detected CWEs; and detection rate for the tools.

Tool	CWE Checkers	Detected CWEs	Detection Rate
Infer	13	7	0.54
Sonar	51	32	0.63
SB	26	24	0.92
FSB	27	24	0.89
PMD	25	22	0.88

related to the missed CWEs and the tools' detection rate in Section IV-A, then we present the analysis results for the individual CWEs in Section IV-B, followed by the collective analysis results in Section IV-C.

A. LIST OF MISSED CWES AND THE DETECTION RATE OF THE TOOLS

Nine CWEs out of the 70 CWEs of the study have been missed by all the tools: CWEs 256, 325, 336, 379, 400, 459, 486, 526, and 759. Some of the tools do not have checkers for those CWEs, while others have checkers, but they were unable to identify the CWEs. Since the recall was 0 for these CWEs, we excluded them from the results' figures. Moreover, we excluded any CWE with Recall = 0 for any particular tool-CWE pair. The detection rate is calculated by dividing the number of detected CWEs in Juliet by the number of existing checkers for those CWEs. This has been presented in Table 2. We can see that SB had the highest detection rates (0.92), meaning that most of their checkers effectively detected the CWEs of the Juliet Test Suite. FSB and PMD are followed (with 0.89 and 0.88, respectively). Nevertheless, the detection rate for Infer is the smallest (0.54), indicating that only half of its checkers were effective in detecting the CWEs of the Juliet Test Suite.[9]

Despite its low detection rate, Sonar could identify the highest number of CWEs (32), while SB, FSB, and PMD identified similar numbers of CWEs (24, 24, and 22, respectively). Infer could detect only 7 CWEs.

B. ANALYSIS OF INDIVIDUAL CWES

The performance of the tools differs per bug type. To better represent our results, we classify CWEs into three groups according to the number of tools that were able to detect them. The bar graphs in Figures 3, 4, and 5 present the outcomes of the performance analysis of the individual tools for the 70 Java CWEs extracted from the Juliet test suite.

By comparing the performance of the five selected tools across all CWEs, the aim is to determine which tool or tools have superior abilities in identifying security vulnerabilities, resulting in a better F-score. The presented graphs depict recall, false alarm rate, precision, and F-score for individual CWEs.

[9]This happens because Infer was originally designed to target Android software, and hence it misses some bugs in non-Android Java code. Specifically, the six CWEs that Infer missed could be easily found by changing some configurations.

The recall values $Recall_i$ for each tool are displayed in subfigures *a* of each figure. Higher recall values indicate better performance. We had the following observations regarding recall from our results.

- 28 CWEs have been detected by only one tool.
- 19 CWEs have been detected by two tools.
- 13 CWEs have been detected by three tools.
- One CWE has been detected by four tools (CWE476).
- Each tool had a 0.0 recall for some specific CWEs. This can be observed from Table 2 in Section III-A. This also means that none of the CWEs have been detected by all five tools of the study.

These observations indicate that, despite the existence of many static analysis tools and the use of five state-of-the-art tools in this study, it is not easy to find a tool that targets a lot of existing vulnerabilities.

Subfigure *b* of each figure displays the values for the False alarm $rate_i$, which counts non-flawed structures that are being mistakenly identified as flawed. Smaller values here denote better performance. We see that different tools had better performance for different CWEs.

However, assessing the performance of a tool by looking at both the recall and the false alarm rate metrics provides a more meaningful evaluation. For instance, if a high recall rate is accompanied by a high false alarm rate for a particular CWE (as observed with tool PMD on CWE482 presented in Figure 3, subfigures *a* and *b*), it means that the tool may not make proper distinctions for that CWE. In other words, the tool may generate warning reports regardless of whether the vulnerability exists in the code.

The per-tool precision $Precision_i$ indicates the ability of the tool to avoid FPs. The precision values are presented in subfigures *c* of each figure. It is the ratio of correctly detected flawed constructs (TP) to the number of (flawed and unflawed) constructs reported by the tool (TP+FP). Higher values indicate higher performance. It can be observed from the figures that different tools are most precise for different CWEs. (For example, FSB is the most precise tool for detecting CWE89 – presented in Figure 5 –, while it is not as precise as Sonar regarding CWE614 – presented in Figure 4.)

As explained in Section III-C, the F-$score_i$ metric provides a way to assess both recall and precision in a single value, and it represents the accuracy of the tools. When the F-score values are near 1, it means that the tool can detect a particular CWE with little to no FPs. Upon reviewing subfigures *d* in all the figures, we can observe the following.

- Each tool has uniquely detected specific CWEs, as seen in Figure 3.
- Each tool had low F-scores for some CWEs.
- Sonar had better F-scores than the other tools on many CWEs (e.g., F-score = 1 on CWE477, 478, 481, 483, 484, 614, 584, 398, 252, 328, 338, 572, 586, and 597), but worse on other CWEs (e.g., $\text{F-score}_{CWE570,571}$ = 0.12).

- Some CWEs are easier than others to detect by static analysis (e.g., F-score is 1 for three tools in the case of CWE478, 481, 484, 572, 586, and 597), while CWE470 has been detected by only one tool, with a very low F-score of 0.05.

C. COLLECTIVE ANALYSIS OF CWES

According to the results presented in Section IV-B, the performance of a tool varies widely for different CWEs, which makes it difficult to draw broad judgments regarding the overall performance of the tool just by depending on the per-CWE evaluation. Consequently, in this section, we attempt to evaluate the *collective* effectiveness of each tool, aiming to determine if any tool outperforms the others. To do so, we calculated the standard measures of central tendency, namely the mean and the median.

For each of the five tools and each of the performance metrics presented in Section III-A, Table 3 shows the Mean and Median computed across all CWEs. Both have values between 0.0 and 1.0. Again, higher recall, precision, and F-score values indicate better results, while lower values indicate a better result in the case of a false alarm rate. From Table 3, we observe the following.

- FSB outperformed the other tools on the recall metric.
- Infer and SB had the best values for the false alarm rate.
- PMD had the highest result for the precision and F-score metrics, which means it was the most precise and accurate tool among all the tools.

V. DISCUSSION AND ANSWERS TO RESEARCH QUESTIONS

Software developers can benefit from the empirical research described in this paper by gaining a better knowledge of the capabilities of static code analysis as well as the degree of quality assurance it can provide. We initially address the research questions in this section. After that, we provide a summary of the key findings, evaluate how they stack up against related findings from earlier studies, and – above all – discuss the practical implications of these findings.

By looking at *Mean* and *Median* values for the F-score in Table 3, we can answer research question 1. *How accurately can the state-of-the-art static analysis tools identify flaws?* The Mean values range from 0.52 (for SB) to 0.83 (for PMD), and the Median values range from 0.49 (for SB) to 1.00 (for PMD). This indicates that the state-of-the-art static analysis tools had average accuracy when detecting 70 different weakness categories from artificial benchmarks.

To answer research question 2: *Which tool could detect more security vulnerabilities? Which tool had the highest detection rate?* we can look at Table 2. Sonar could detect the highest number of vulnerabilities (32). However, to answer the second part of the question, we look at the last column and see that SB had the highest detection rate (0.92), followed by FSB and PMD (0.89, and 0.88).

To answer research question 3: *Which tool is the most precise in detecting the vulnerabilities?* we can look at *Mean* and *Median* values for precision in Table 3. PMD had the highest *Mean* value (0.92), followed by SB (0.89) and Infer (0.88). Median values for all the tools were (1.00), except for FSB, which was (0.69), which indicates it was the least precise tool.

To answer research question 4: *What are the security vulnerabilities that have been perfectly detected, i.e., with optimal performance, by at least one tool? And those that have been completely overlooked by all the tools?* by looking at subfigure *d* of figures 3, 4, and 5, we can easily extract those vulnerabilities that have been detected with optimal F-score by at least one tool. They are twenty-one vulnerabilities (30% of the studied vulnerabilities) and are presented in Table 4.

However, the nine CWEs that have been missed by all five tools of the study are considered those that have been overlooked by state-of-the-art static analysis. They are: (CWEs 256, 325, 336, 379, 400, 459, 486, 526, and 759). The answer to this question opens the door to future research work.

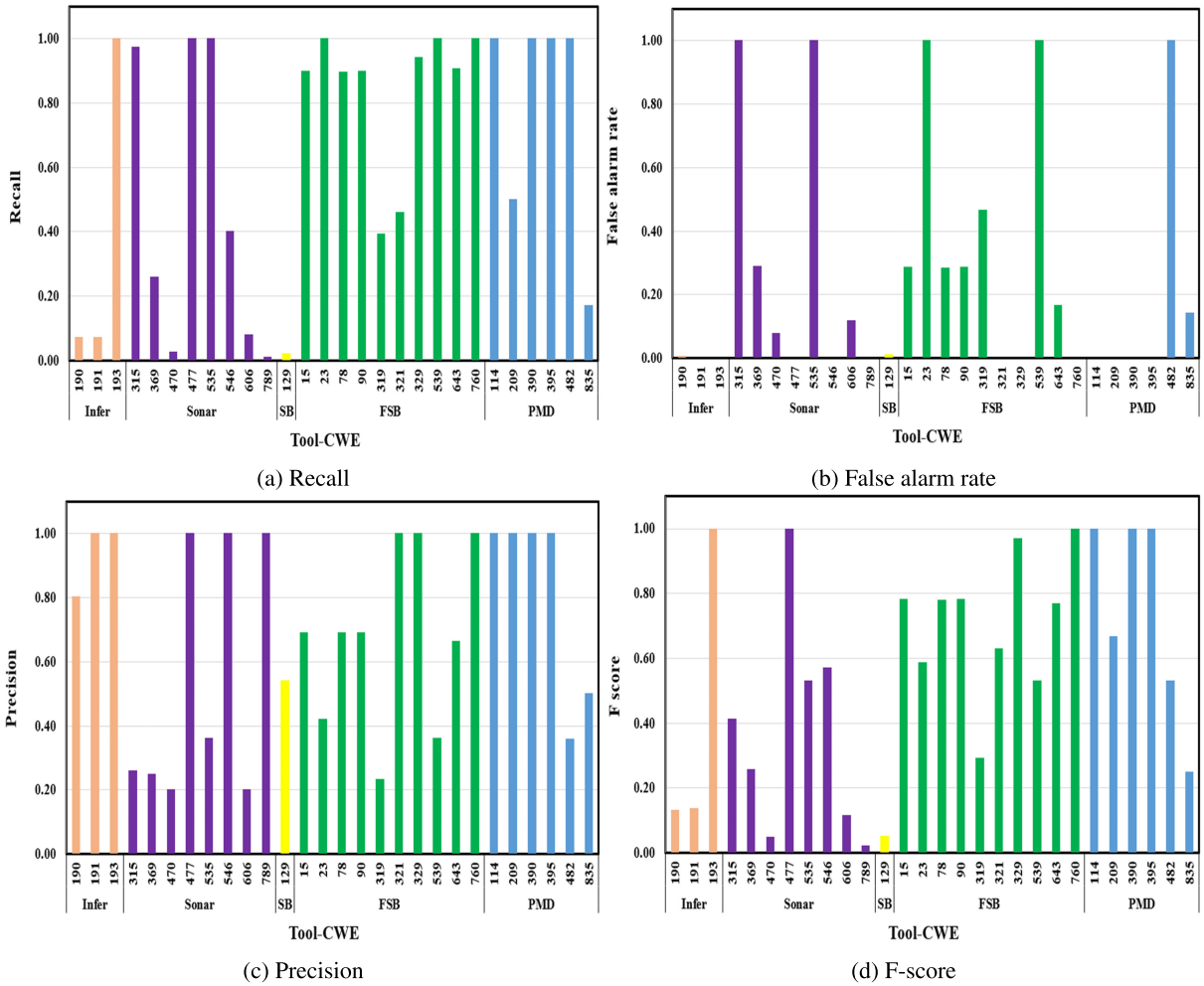
A. FALSE NEGATIVES

The capability of a static analysis tool used by a security assurance team to find as many vulnerabilities as possible, preferably all of them, is essential. Our primary observations about the effectiveness of the tools in this regard are as follows.

- None of the tools detected all the vulnerability categories (i.e., all CWEs of Juliet).
- Even though we only used those vulnerabilities that have been targeted by at least one tool of the five selected tools of the study, 13% of the vulnerabilities have been missed by all the tools.

The first result is consistent with the findings of all the related research that are covered in Section VII; some of these studies utilized a single tool for their experiments, while others used multiple tools. Furthermore, our second finding is similar to those in two studies [19], [25] that used an experimental strategy based on several static code analysis tools. In the first study [19], the researchers evaluated the ability of three tools to detect 22 C/C++ CWEs and 19 Juliet CWEs, knowing that these CWEs were not all targeted by all three tools. The results indicated that 11% of the Java vulnerabilities were not detected by any of the selected tools. The values of TP and FP produced by five static code analysis tools for ten categories of Juliet 1.0 C/C++ CWEs were published in NIST's 2013 report [25]. Each of the five tools had zero TP in some vulnerabilities.

Based on our current and previous studies, it is evident that even the most advanced static code analysis tools cannot completely ensure that software are free of vulnerabilities. Therefore, it is important to combine different static analysis techniques and/or combine other techniques (like, dynamic analysis and testing) with static analysis to detect vulnerabilities.



(a) Recall (b) False alarm rate

(c) Precision (d) F-score

FIGURE 3. CWEs detected by one tool.

TABLE 3. Performance metrics across collective CWEs.

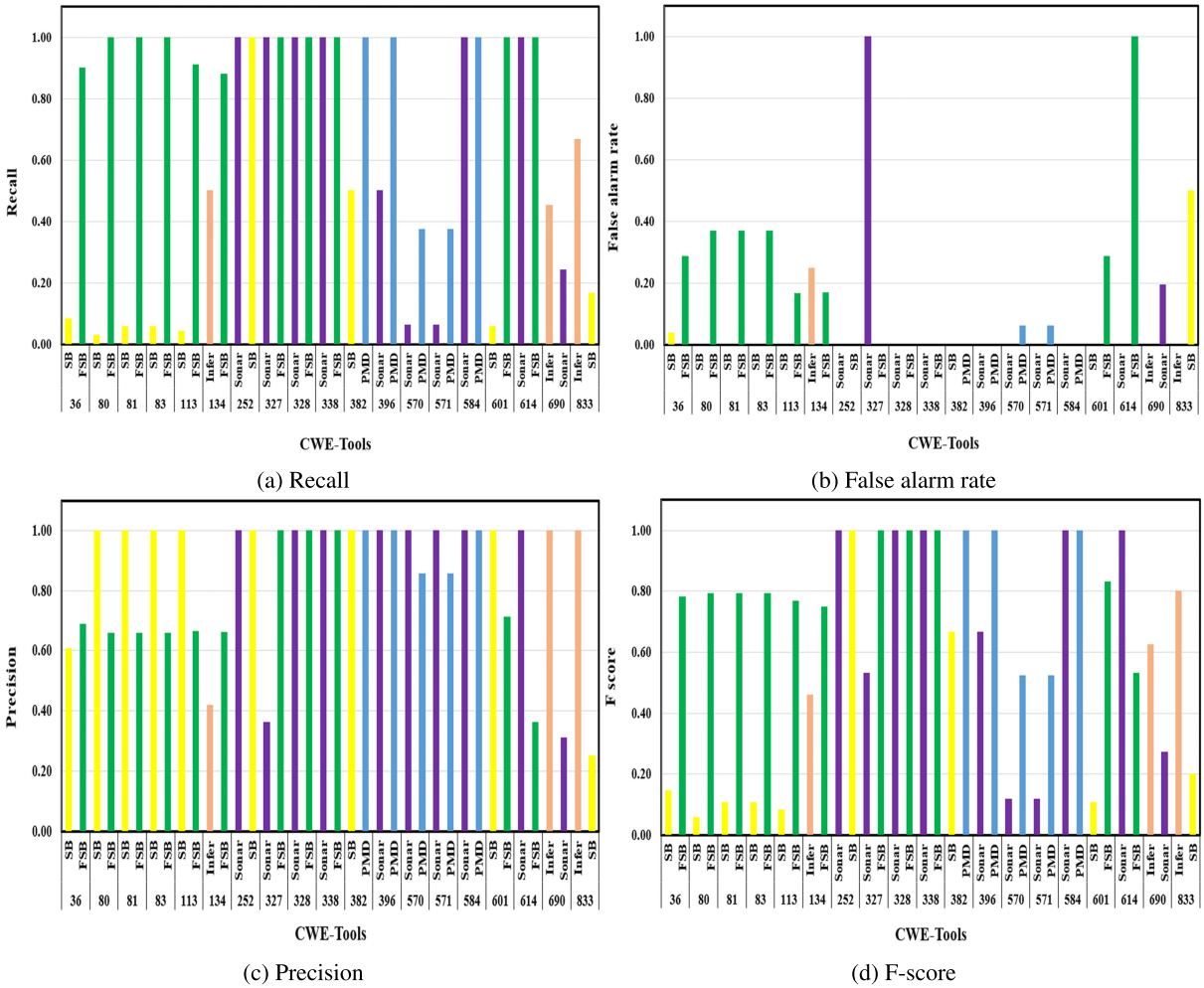
Metrics	Central Tendency Measurement	Infer	Sonar	SB	FSB	PMD
Recall	Mean	0.53	0.68	0.49	0.86	0.82
	Median	0.50	0.97	0.37	0.93	1.00
False alarm rate	Mean	0.04	0.16	0.04	0.28	0.07
	Median	0.00	0.00	0.00	0.23	0.00
Precision	Mean	0.88	0.78	0.89	0.73	0.92
	Median	1.00	1.00	1.00	0.69	1.00
F score	Mean	0.58	0.61	0.52	0.75	0.83
	Median	0.62	0.60	0.49	0.78	1.00

TABLE 4. Vulnerabilities detected perfectly (F-score = 1.0).

CWE ID	By tools
193	Infer
477, 614	Sonar
327, 760	FSB
114, 390, 395, 382, 396	PMD
252	Sonar, and SB
328, 338	Sonar, and FSB
483, 584	Sonar, and PMD
478, 481, 484, 572, 586, 597	Sonar, SB, and PMD

B. FALSE POSITIVES

In our experiments, the static code analysis tools generated a varying amount of FPs (an average rate ranging from 0.04 to 0.28), with Infer and SB exhibiting the best performance. High FP rates can give rise to imprecision; precision is defined as the ratio of TPs to the total number of detections (i.e., TPs plus FPs). Our study found that the collective precision of static code analysis tools ranged from 0.73 to 0.92, which indicates a fair precision level, at least compared to an earlier study [19] that evaluated three commercial tools reporting precision values from 0.34 to 0.56 for a subset of Java test cases of the Juliet benchmark. However, we should emphasize that our measured precision values still reflect the performance of the static code analysis tools on an artificial test suite. Research on the analysis of real software applications [6] reports lower precision values. This could be



(a) Recall

(b) False alarm rate

(c) Precision

(d) F-score

FIGURE 4. CWEs detected by two tools.

because real software applications contain a lot more non-flawed code than flawed code, which could lead to more FPs and ultimately lower precision.

In general, static code analysis tools tend to produce a high number of FPs [26], which can have various practical effects.

- It can lead to a major resource loss for large software products because a human review is necessary for the high number of FPs. Spending extra time on analysis to find more vulnerabilities may be appropriate for fragile products, but it could be costly for less sensitive ones.
- If the static code analysis tool is utilized as an early vulnerability detector during development instead of later in the life cycle, the high false alarm rate might be bearable.
- According to industrial experience, there are other issues to take into account besides the expense of inspecting FPs [17]. These difficulties include the following: (a) the necessity for actual project experience to choose particular static code analysis tools to reduce FPs; (b) the challenge for developers to recognize warnings as FPs; and (c) the possibility of identifying new vulnerabilities in the process of trying to "fix" the FPs that the developers misclassified as TPs.

Given the above, choosing a tool with a lower false alarm rate, assuming no difference in detection rate (recall) is highly advantageous. According to our results, Infer and SB performed very well in this respect, with the lowest collective false alarm rate (0.04).

C. ACCURACY OF TOOLS

In our study, the accuracy of tools has been characterized with the F-score, since the classical accuracy metric[10] is not applicable in the current experiment, as the positive and negative occurrences are imbalanced in the investigated Juliet Test Suite test cases. According to our results, the collective accuracy of the static analysis tools ranged from 0.52 (for SB) to 0.83 (for PMD) (see F-score values in Table 3). Goseva-Popstojanova and Perhinschi [19] considered the classical accuracy metric as the basis for finding the best

[10] Acc=(TN +TP)/(TN+FN+TP+FP)

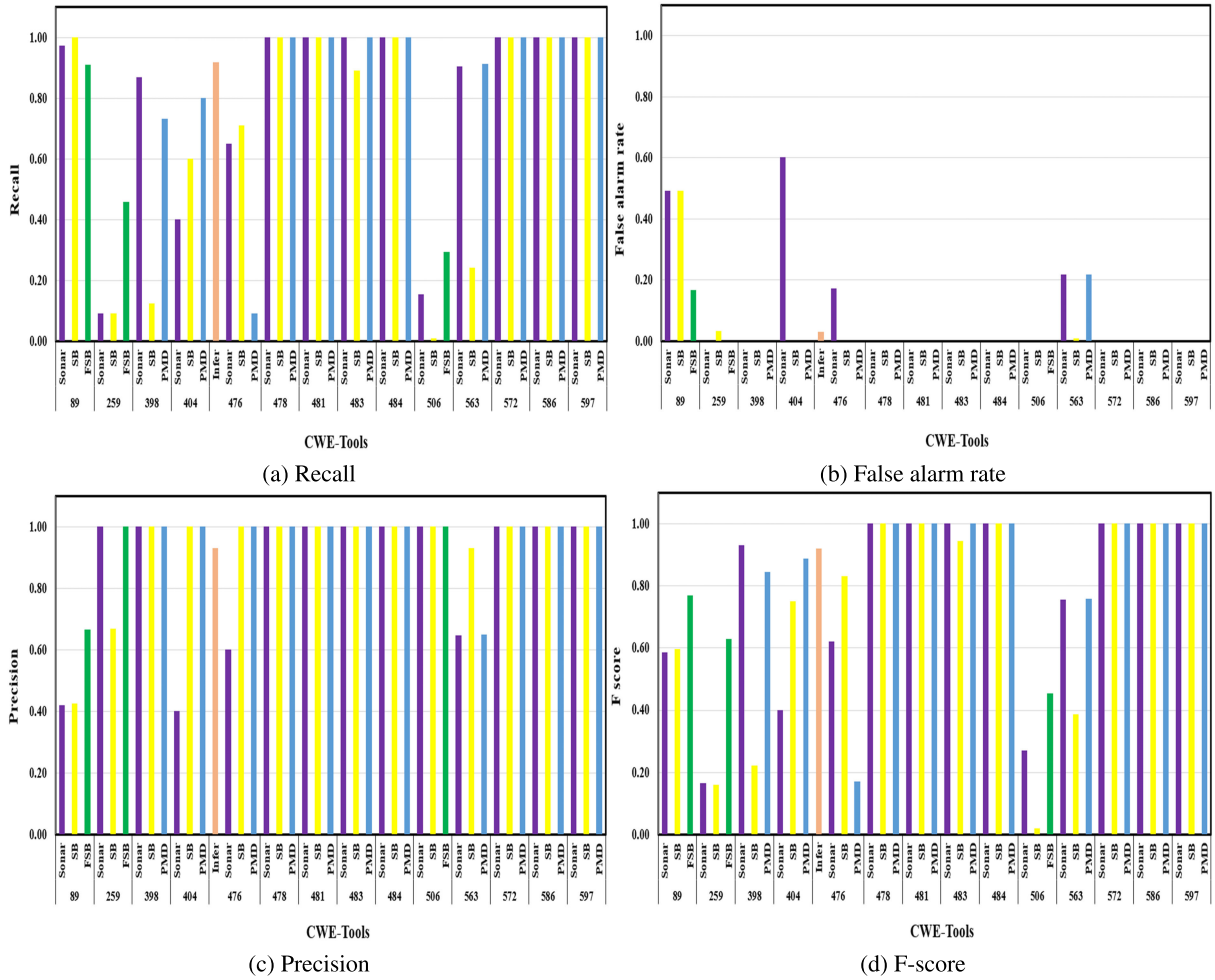


FIGURE 5. CWEs detected by three or four tools.

tool. According to their results, the tools' accuracy ranged from 0.67 to 0.73 (knowing that they had not mentioned the tools' names). However, we used the harmonic mean of recall and false alarm rate as the basis for accuracy in some of our previous studies [21], [27]. In the first study, Sonar was the most accurate tool, while in our second study, Infer had the best value.

D. MORE IMPROVEMENTS TO TECHNIQUES OF STATIC ANALYSIS TOOLS ARE REQUIRED

The use of static code analysis during software development can aid in addressing software security and quality concerns at an early stage. It can also handle incomplete systems and is scalable for large code bases in most cases. Nonetheless, the evaluation findings outlined in this study showed that, although utilizing these tools may enhance software security, it does not guarantee it and requires considerable manual effort to classify reported warnings. Therefore, further enhancements are needed in static code analysis techniques and tools to achieve more cost-effective and reliable identification of security vulnerabilities and improved software quality assurance.

VI. THREATS TO VALIDITY

The validity of any empirical study may be threatened by various factors. This section will discuss the threats that we believe could compromise the reliability of our findings and the actions we take to mitigate them.

A. CONSTRUCT VALIDITY

These threats must be considered to ensure that our evaluation accurately reflects our intended objectives. The first threat is the choice of static code analysis tools; there is a growing number of tools available, and our selection may not be inclusive of all of them. Since we have chosen five popular free tools that fall into several categories of static code analysis, we think our findings fairly represent the state-of-the-art in this field, which focuses on various types of static analysis.

The number of vulnerabilities that have been utilized in the tool evaluation process poses a second threat to validity. This

has an impact on the performance and output of the tools. We employed the Juliet test suite, which addresses a variety of vulnerabilities, to mitigate this threat. This makes it possible to evaluate the tools consistently and fairly.

A possible mismatch between the CWE that the tools report and the CWE that the Juliet test cases aim to represent adds a further threat. We have seen some cases where the tool detected the specified weakness in targeted methods, but still, it did not detect the exact line that had been disciplined by Juliet. In some of these cases, the tool gave zero or a significantly small number of FPs compared to the high number of detected TPs. So, we decided to consider these as valid detections for the specified tools. However, there were other cases when the tool was incorrectly detecting the vulnerabilities. In such cases, we considered them misidentifying, so it was a matter of human decision. We eliminate this threat by studying all available cases separately and comparing the detections of different tools for the same test cases.

The fourth threat is the version mismatch of the used software. Specifically speaking, in our evaluation, we used Java compiler version 14.0.1 for all of our analyses; but, the results of some analyzed CWEs were different when we used Java compiler version 8, with SB and FSB. This issue has been reported in some previous works [21], [28], and partially resolved by Beba and Karlsen [29]. The different settings for the static analysis tools are another threat. SB and FSB offer the user different levels of code issue severity, which are not available in other tools. This may affect the output evaluation for these tools since these results are not presented by these tools according to unified criteria. We eliminate these two threats by considering one severe level for all the detected CWEs by both SB and FSB.

Another threat to construct validity arises when relying on a single metric to give final results and research observations. To address this, we assessed various facets of static code analysis tools' capacity to identify security flaws and distinguish them from safe code using a variety of criteria. Recall, false alarm rate, precision, and F-score were employed to make sure that our findings were supported by a comprehensive assessment of the performance of the tools.

B. INTERNAL VALIDITY

This type of threat may occur when there are external factors that could alter the independent variables and/or measurements without the researchers' knowledge. Many warnings in static code analysis approaches need to be manually reviewed by human experts to separate TPs from FPs. This can lead to possible misclassification, and human classification errors pose a significant threat to the study's internal validity and the reliability of the results. However, using Juliet eliminated this threat by automatically classifying the warnings produced by the tools. Juliet identifies code sections with flaws (i.e., bad methods) and their counterparts without flaws (i.e., good methods), allowing for accurate classification of the warnings.

C. EXTERNAL VALIDITY

When determining the extent to which the results can be generalized, external threats to validity must be considered. In Section V, we address this threat by comparing our experimental results based on Juliet with other studies that evaluated static analysis tools (which may differ from the ones we used) on Juliet, another synthetic benchmark, and/or on natural code.

D. CONCLUSION VALIDITY

These threats are related to the ability to draw correct conclusions. The Juliet test suite includes different CWEs; some of them include a huge number of well-designed test cases, but some others do not. According to some research [19], small test cases and/or test cases with a single vulnerability of a specific type are considered limitations when choosing benchmarks. We eliminate this threat by removing the CWEs with a small number of test cases from the study. The validity of the test cases is another threat to the presented research. We eliminate it by validating the Juliet test cases before using them for evaluating the tools. Through this, we excluded the test cases that do not represent the targeted vulnerability by its definition in OWASP [30].

VII. RELATED WORK

In the past, several empirical static analysis comparison studies have been conducted, but they tended to overlook Java [31], [32], [33], [34] or included a wide range of programming language tools, leaving limited attention to Java-specific tools [19], [22], [35], [36], [37], [38]. Despite this, some studies that specifically targeted Java had either a restricted scope by evaluating only a few security vulnerability categories [20], [21], [27], [39], [40], [41], [42], [43], [44], [45], [46], [47] or concentrated solely on Android static analysis tools [48]. Now, let's take a closer look at these studies, categorizing them into two subsets: those utilizing only the Juliet Test Suite for evaluation and those that employed real-world programs, with or without using Juliet.

We begin with the first subset [21], [27], [31], [35], [37], [38], [40], [41], [42], [46]. These works are the most similar to ours. In our previous studies [21], [27], we had better experiences with Java static analysis tools. We evaluated and compared four free static analysis tools (PMD, SB, FSB, and Sonar) for their performance in detecting six Java vulnerabilities from Juliet Test Suite's official Top 25 list of CWE [21]. It turned out that the tools were able to only identify four vulnerabilities, with Sonar giving the highest accuracy. Later, we examined the ''null pointer dereference'' vulnerability in more detail, adding the Infer tool to the experiment, which achieved the best performance [27]. In the current research, we evaluated the same five tools using a larger number of vulnerabilities.

Desai and Jariwala [31] examined and compared 14 tools and determined their performance measures regarding

16 weakness categories. In their comparison, they focus on C programs, and their results indicate that static analyzers do effectively identify security vulnerabilities. The selected tools in our study resulted in precision scores ranging from 73% to 92%, which is close to the mentioned study. Nguyen-Duc et al. [35] conducted a case study of utilizing seven free static analysis tools for a humanistic security assessment in an e-government project. They have focused on many programming languages (Java, C/C++, Python, PHP, Javascript, Objective-C, and Rub), and twelve security weaknesses. A unique method of selecting, evaluating, and combining static analysis tools was presented and assessed through semi-structured interviews. The findings indicate that certain tools exhibit superior performance compared to others and that combining multiple tools can lead to improved performance. In our study, we have not considered combinations of tools, but we suggested it to give better performance.

Mahmood and Mahmoud [37] conducted a qualitative and quantitative review of four free Java static analysis tools: FindBugs, PMD, LAPSE+, and Yasca, using ten weakness categories of the Juliet. The review takes into account system requirements, efficacy, type and amount of found flaws, convenience of use, and support, such as web page updates and manuals. The study argues that none of the analyzed tools were perfect, nor were they intended to be, which is in line with one of our conclusions.

To assist companies in choosing an appropriate free and/or commercial static analysis tool, Díaz and Bermejo [38] presented unbiased data regarding these tools' capabilities. They have produced findings using state-of-the-art tools and a consistent, well-defined benchmark test suite for different programming languages (Java, C, C++, PHP, and J2EE). The methodology creates a rigid scale for the effectiveness of static analysis tools by applying well-known metrics based on rates of TP and FP and the vulnerability coverage degree of the tools. They showed how the evaluated tools cover distinct subsets of vulnerabilities and produce different answers for various vulnerability types. Their findings, however, indicated that some commercial tools outperformed free and open-source ones regarding performance, usability, and vulnerability coverage. Although, in our study, we only focused on free Java static analysis tools, we had similar conclusions regarding the tools' capabilities.

Amankwah et al. [40] assessed the performance of eight state-of-the-art Java tools, namely, Findbugs, PMD, YASCA, LAPSE+, JLint, Bandera, ESC/Java, and Java Pathfinder, concerning 20 Java weakness categories of Juliet. Several reliable metrics for performance were used in the study, such as the Youden index, precision, recall, and the OWASP web benchmark evaluation (WBE). The results showed that with a precision score of 90.7%, the Java Pathfinder tool showed the highest precision, followed by YASCA and Bandera, which had precision ratings of 88.7% and 83%, respectively. Furthermore, the Youden index of 0.8 was attained by Bandera, ESC/Java, and Java Pathfinder, indicating the efficiency of these tools in detecting security flaws in Java source code. Although, in our study, we only focused on free Java static analysis tools, using more vulnerabilities, we had similar conclusions regarding the best tool's precision (92% for PMD).

Amankwah et al. [41] address the two main problems with the existing static analysis techniques: processing time and false positives. They compared their tool to the other three tools on a subset of the Juliet Test Suite (3 weakness categories). Their results revealed that their tool was 66% better than the other evaluated tools in processing time and it also resulted in a drastic reduction in false positives.

Oyetoyan et al. [42] conducted a study to evaluate the performance of six free and commercial static analysis tools in detecting 13 weakness categories in Java Juliet. The tools were assessed based on their recall, precision, and discrimination rate. The results show that FSB achieved the highest recall of 18.4% with a precision of approximately 90%. It also had the highest discrimination rate, although slightly lower than its recall. LAPSE+ had a detection rate of 9.76%, but a poor discrimination rate of 0.41%. In our research, we also have used recall and precision for evaluation purposes. However, our results are different; the tools' recall and precision averages were (from 0.49 to 0.86) and (from 0.73 to 0.92), respectively.

Li et al. [46] evaluated and compared five free, and open-source Java IDE plugins for their ability to detect 29 weakness categories. The findings indicate that most plugins have detected certain vulnerabilities, like injection and broken access control, but other vulnerabilities have been missed. They claimed that more work is required before corporate reliance on these plugins due to usability limitations and a high FP-rate. In our study, we had a similar conclusion; 30% of the investigated weaknesses have been detected with optimal performance by many tools, and 13% of them have been totally overlooked by all the tools. Table 5 presents the summary of the related works in this category.

Now, we visit the studies in the second subset [19], [20], [22], [32], [33], [34], [36], [39], [43], [44], [45], [47], [48]. Goseva-Popstojanova and Perhinschi [19] investigated the ability of three – unspecified – commercial static code analysis tools to find Java and C/C++ security flaws. The evaluation includes a controlled experiment using 19 Java test case categories and 22 C/C++ test cases from Juliet Test Suite's benchmark, as well as case studies based on open-source programs with 44 known vulnerabilities. According to their results, static code analysis tools did not show a statistically significant difference in their ability to detect security vulnerabilities for both C/C++ and Java, and all the tools had average Median and Mean of the per CWE recall values, and overall recall across all CWEs close to or below 50%. More specifically, all three tools missed 27% of C/C++ vulnerabilities and 11% of Java vulnerabilities, and they all found 41% of C/C++ and 21% of Java vulnerabilities. In our study, we have neither considered commercial tools nor C/C++ tools. According to our results, static analysis tools totally missed 13% of the

TABLE 5. Summary of Related Work: First Category (using Juliet as a criterion for the evaluation).

Ref.	Targeted PL	Free/Commercial Tools	# Tools	# Vuln.
[21]	Java	free	4	6
[27]	Java	free	3	1
[31]	C/C++	both	14	16
[35]	Java, C/C++, Python, PhP, Javascript, Objective-C, and Ruby	free	7	12
[37]	Java and C/C++	free	7	10
[38]	Java, C,C++, PhP, and J2EE	both	9	22
[40]	Java	both	8	20
[41]	Java	free	4	3
[42]	Java	both	6	13
[46]	Java	free	5	29
current	Java	free	5	70

TABLE 6. Summary of Related Work: Second Category (using real-world software as a criterion for the evaluation, with or without using Juliet).

Ref.	Targeted PL	Free/Commercial tools	# Tools	# Vuln.
[19]	Java and C/C++	commercial	3	19 Java + 22 C/C++
[20]	Java	both	4	1
[22]	Java, C, and Objective C	both	5	10
[32]	C	both	6	21
[33]	C	commercial	3	no
[34]	Solidity-smart contract language	free	6	7
[36]	Java and C/C++	free	5	5 Java + 10 C/C++
[39]	Java	free	6	diff.
[43]	Java	free	3	1
[44]	Java	free	3	20
[45]	Java	free	5	12
[47]	Java	both	3	4
[48]	Java	commercial	14	42
current	Java	free	5	70

analyzed vulnerabilities. Moreover, despite that we consider Recall as one of the used metrics, we used F-score as the basis to evaluate the tool's performance.

In their study, Mamun et al. [20] assessed the effectiveness of four free, and commercial static analysis tools in detecting Java bug patterns and concurrency bugs. They collected 141 bug patterns from the tools and put them to the test. Through their evaluation, they classified 87 bug patterns as unique and observed that Jtest was the most efficient tool, with a defect detection ratio of 0.48. Meanwhile, the average defect detection ratio of all the tools evaluated was 0.25. These findings suggest that relying solely on static analysis tools may not be sufficient for identifying concurrency bugs, which is in line with one of our conclusions from the presented study.

Focusing on Java, C, and Objective C, Stikkelorum and Bruntink [22] conducted a thorough study of the performance of the Infer static analysis tool, running it on a wide range of open-source projects, and benchmarking it against the Juliet test suite to give developers and businesses additional information about it, by comparing it to four commercial and free static analysis tools. Although the results varied depending on the type of vulnerability and programming language, Infer's performance was generally found to be promising, with precision of up to 100% observed for numerous CWEs. Similar results to those obtained on the Juliet suite were also obtained when Infer was performed on industrial software. But as he also mentioned, getting Infer to work may be challenging, particularly on Java applications. In our study, we also evaluated Infer, but we got different results regarding its precision (0.88 for collective CWEs).

Lipp et al. [32] put out a methodology based on CVE reports for automatically assessing the performance of static analysis tools. They tested one commercial tool and five free and open-source ones against 27 C software projects with 1.15 million lines of code with 21 types of vulnerabilities. According to their findings, between 47% and 80% of the vulnerabilities in a benchmark set of real-world programs have been missed by the tools. Furthermore, their analysis reveals that by integrating the output of static analyzers, this false negative rate can be decreased to 30% to 69%, albeit at the expense of 15% more bugs being detected. As a result, many vulnerabilities have been missed, particularly those that go beyond the traditional memory-related security flaws. In our research, we neither evaluated C static analyzers nor used real-world software for the assessment. Rather, we focused on assessing Java tools against synthetic test cases.

Imparato et al. [33] performed a quantitative analysis – without focusing on specific vulnerabilities, using performance metrics and an alert classification model. They examined the AUTOSAR application software components for an instrument panel cluster's production code. Their comparison aims to determine the optimal set of commercial static analysis tools to minimize flaws in AUTOSAR software components. They have used three tools. The analysis results

TABLE 7. List of excluded CWEs along with the reasons for exclusion.

1-Selecting phase: Deprecated CWEs
(533) Info. Exposure Through Server Log Files
(534) Info. Exposure Through Debug Log Files
Irrelevant CWEs
(111) Unsafe JNI
(197) Numeric Truncation Error
(226) Sensitive Information Uncleared Before Release
(253) Incorrect Check of Function Return Value
(378) Temporary File Creation With Insecure Perms
(510) Trapdoor
(511) Logic Time Bomb
(523) Unprotected Cred Transport
(549) Missing Password Masking
(566) Authorization Bypass Through SQL Primary
(598) Information Exposure QueryString
(605) Multiple Binds Same Port
(613) Insufficient Session Expiration
(615) Info Exposure by Comment
(617) Reachable Assertion
(667) Improper Locking
(681) Incorrect Conversion Between Numeric Types
(698) Redirect Without Exit
(764) Multiple Locks
(765) Multiple Unlocks
(832) Unlock Not Locked
2-Validation phase: Invalid test cases
(476) Null Pointer Dereference (17 test cases)
3-Evaluation phase: Test cases less than five
(248) Uncaught Exception
(397) Throw Generic
(491) Object Hijack
(499) Sensitive Data Serializable
(500) Public Static Field Not Final
(561) Dead Code
(568) Finalize Without Super
(579) Non-Serializable in Session
(580) Clone without Super
(581) Object Model Violation
(582) Array Public Final static
(585) Empty Sync Block
(600) Uncaught Exception in Servlet
(607) Public Static Final Mutable
(609) Double Checked Locking
(674) Uncontrolled Recursion
(772) Missing Release of Resource
(775) Missing Release of File Descriptor or Handle
bad-only test cases
(383) Direct Use of Threads

have suggested some improvements in the static analysis tools. In our study, we assessed free tools against specific vulnerabilities.

To evaluate smart contracts with static analysis tools, Ghaleb and Pattabiraman [34] suggested an automated and systematic technique. The technique involves inserting bugs, sometimes known as code defects, into every possible spot within a smart contract to introduce specific security flaws. After that, it uses the tools to check the generated flawed contract and find the problems that the tools cannot discover (FN), as well as the incorrect bug reports (FP). Using a set of 50 contracts, and by injected more than nine thousand different defects, their technique has been used to assess six popular static analysis tools: Securify, Oyente, Mythril, Manticore, SmartCheck, and Slither. Despite the assessed tools claim to be able to discover such bugs, some of them were not able to find multiple bugs, and all of the tools indicate a high number of FPs. In our study, we had similar conclusions regarding the detected bugs, but the average FP of our tools was from 0.4 to 0.28, for the collective vulnerabilities.

Kaur and Nayyar [36] used the Juliet Test Suite and the code of Apache Tomcat to compare five static analysis tools (three C/C++, and two Java tools) in detecting 10 weakness categories for CC++, and five ones for Java source code. Their results showed that CPPCheck detected a higher number of vulnerability categories than RATS and Flawfinder. For Java, SB was found to detect more vulnerabilities than PMD. In conclusion, they noted – as we did in our study – that some vulnerabilities remain undetected by any of the evaluated tools.

Six free static analysis tools were evaluated by Valentina et al. [39] to examine 47 Java projects from the Qualitas Corpus dataset: PMD, Better Code Hub, CheckStyle, Findbugs, Coverity Scan, and Sonar. First, the researchers measured the precision of the tools; second, they compared the warnings detected by the tools and developed a taxonomy that can assist researchers and tool vendors in identifying critical warnings for refactoring; and third, they analyzed the agreement among the tools, which can provide information to tool vendors about the limitations of the current state-of-the-art tools. In our study, we also measured the tools' precision. However, we have used other metrics as well to compare the tools using different aspects.

An experiment and survey were set up by Manzoor et al. [43] to determine whether Java concurrency issues could be found using free, open-source, and cutting-edge static analysis tools. They have chosen RacerX, RELAY, and CheckThread for this purpose. They discovered that they lacked effectiveness in that regard. In our study, the tools' effectiveness ranged from 0.52 to 0.83.

In their case study, Wagner et al. [44] used multiple projects from an industrial environment to examine the interdependencies between them. The major conclusion is that while testing finds more flaws than bug-finding methods do, reviews uncover a subset of faults. On the other hand, more in-depth analysis is done on the detectable types. As a result, they suggested – like we did – combining many techniques to reduce the number of FPs.

Rutar et al. [45] evaluated five static analysis tools on a range of Java programs: Bandera, ESC/Java 2, FindBugs, JLint, and PMD. The outcomes of their experiments demonstrated that none of the tools completely replaces the others, and in fact, the tools frequently identify non-overlapping issues. They also discussed the methods that underpin each tool and offered suggestions for how different methods might impact the tools' results. Finally, they suggested creating a meta-tool that searches through the combined output of the tools for specific lines of code, methods, and classes that are flagged as dangerous by several tools. In our study, we also found that several vulnerabilities (40% of the investigated vulnerabilities) have been detected only by one of the evaluated tools.

TABLE 8. List of java CWEs included in our experiment with the number of associated bad and good methods.

CWE	Description	#bad	#good
(15)	External Control of System or Configuration Setting	444	624
(23)	Relative Path Traversal	444	624
(36)	Absolute Path Traversal	444	624
(78)	OS Command Injection	444	624
(80)	XSS	666	936
(81)	XSS Error Message	333	468
(83)	XSS Attribute	333	468
(89)	SQL Injection	2220	6120
(90)	LDAP Injection	444	624
(113)	HTTP Response Splitting	1332	3672
(114)	Process Control	17	30
(129)	Improper Validation of Array Index	2664	7344
(134)	Uncontrolled Format String	666	1836
(190)	Integer Overflow	4255	11730
(191)	Integer Underflow	3404	9384
(193)	Off by One Error	51	90
(209)	Information Leak Error	34	60
(252)	Unchecked Return Value	17	30
(256)	Plaintext Storage of Password	37	102
(259)	Hard Coded Password	111	156
(315)	Plaintext Storage in Cookie	37	102
(319)	Cleartext Tx sensitive Info	370	1020
(321)	Hard Coded Cryptographic Key	37	52
(325)	Missing Required Cryptographic Step	34	60
(327)	Use Broken Crypto	34	60
(328)	Reversible One Way Hash	51	90
(329)	Not Using Random IV with CBC Mode	17	30
(336)	Same Seed in PRNG	17	30
(338)	Weak PRNG	34	60
(369)	Divide by Zero	1850	5100
(379)	Temporary File Creation in Insecure Dir	17	30
(382)	Use of System Exit	34	60
(390)	Error Without Action	34	60
(395)	Catch NullPointerException	17	30
(396)	Catch Generic Exception	34	60
(398)	Poor Code Quality	137	241
(400)	Resource Exhaustion	1460	4008
(404)	Improper Resource Shutdown	5	5
(459)	Incomplete Cleanup	34	60
(470)	Unsafe Reflection	444	624
(476)	Null Pointer Dereference	181	466
(477)	Obsolete Functions	68	120
(478)	Missing Default Case in Switch	17	30
(481)	Assigning Instead of Comparing	17	30
(482)	Comparing Instead of Assigning	17	30
(483)	Incorrect Block Delimitation	19	32
(484)	Omitted Break Statement in Switch	17	30
(486)	Compare Classes by Name	17	30
(506)	Embedded Malicious Code	116	90
(526)	Info Exposure Environment Variables	34	60
(535)	Info Exposure Shell Error	17	30
(539)	Information Exposure Through Persistent Cookie	17	30

TABLE 8. *(Continued.)* List of java CWEs included in our experiment with the number of associated bad and good methods.

(546)	Suspicious Comment	85	150
(563)	Unused Variable	218	498
(570)	Expression Always False	16	16
(571)	Expression Always True	16	16
(572)	Call to Thread run Instead of start	17	30
(584)	Return in Finally Block	17	30
(586)	Explicit Call to Finalize	17	30
(597)	Wrong Operator String Comparison	17	30
(601)	Open Redirect	333	468
(606)	Unchecked Loop Condition	444	1224
(614)	Sensitive Cookie Without Secure	17	30
(643)	Xpath Injection	444	1224
(690)	Null Dereference From Return	296	816
(759)	Unsalted One Way Hash	17	30
(760)	Predictable Salt One-Way Hash	17	30
(789)	Uncontrolled Memory Allocation	1571	2214
(833)	Deadlock	6	6
(835)	Infinite Loop	6	7

A bounded model-checking tool named JBMC was created by Cordeiro et al. [47] to identify flaws in Java bytecodes. The tool has been built over the CPROVER framework. With the help of a model of the standard Java libraries and Java bytecode, JBMC processes and verifies several specifications. They compared their tool to two cutting-edge Java tools and found that it is superior to both and can accurately detect four Java vulnerability types.

An assessment of the effectiveness of 14 Java Android tools was conducted by Ranganath and Mitra [48] against 42 known vulnerabilities that were tracked by Ghera benchmarks. The study revealed that the evaluated tools have limitations and are not able to detect all known vulnerabilities. The evaluated tools could only detect 30 out of 42 known unique vulnerabilities. Therefore, developers should not rely solely on security analysis tools to create secure applications but must also put in additional effort to ensure app security, which is in line with one of our conclusions. Table 6 presents the summary of the related works in this category.

According to the reviewed literature, none of them have covered as many (i.e., 70) weakness categories in Java programs as in this study. Moreover, researchers mentioned that the Juliet Test Suite includes bugs [21], [22], [49], but to the best of our knowledge, this is the first study that evaluates and validates Juliet test cases before using them for evaluating the static analysis tools.

VIII. CONCLUSION

Software bugs are expensive and challenging to find and resolve. This study evaluates state-of-the-art static analysis tools for their ability to find security vulnerabilities in Java programs. For this, we employed an experimental strategy based on the Juliet benchmark test suite. This enabled us to: (1) statistically measure tool performance, both per weakness and across all weaknesses; and (2) automatically evaluate tool performance on many test cases covering a wide spectrum of Java vulnerabilities. Our results show that static code analyzers are still ineffective at detecting security flaws. Specifically:

- None of the five tools used in our evaluation aims to detect all the vulnerabilities.
- On the one hand, among the 70 security weaknesses that have been used to evaluate the tools, twenty-one (30%) have been detected with optimal performance (F score = 1) by at least one tool. while, nine (13%) were completely overlooked by all five tools (Recall = 0).
- Every tool uniquely detected specific vulnerabilities; in total 40% of the vulnerabilities have been only detected by one of the tools.

We argue that among all the tools of the study:

- Facebook Infer has the soundest static analysis engine; it results in the least number of false positives.
- SonarQube targets the highest number of weaknesses in the Juliet benchmark.
- PMD seems to be the most accurate tool, with a 0.83 collective F-score.

Our findings suggest that static code analysis alone cannot be relied on because it may miss a large number of vulnerabilities. Other techniques, like dynamic analysis and testing, may be useful to complement them. Moreover, a combination of two or more static analysis tools may be used to increase the tools' accuracy.

APPENDIX
LONG TABLES

See Tables 7 and 8.

REFERENCES

[1] The White House. (2009). *Cyberspace Policy Review: Assuring a Trusted and Resilient Information and Communications Infrastructure*. Accessed: Dec. 2023. [Online]. Available: https://irp.fas.org/eprint/cyber-review.pdf

[2] S. Morgan. (Nov. 13, 2025). *Cybercrime to Cost The World $10.5 Trillion Annually by 2025*. Cybercrime Mag. [Online]. Available: https://cybersecurityventures.com/cybercrime-damage-costs-10-trillion-by-2025/

[3] K. Phung, E. Ogunshile, and M. Aydin, "Error-type—A novel set of software metrics for software fault prediction," *IEEE Access*, vol. 11, pp. 30562–30574, 2023, doi: 10.1109/ACCESS.2023.3262411.

[4] P. E. Black, M. Kass, and M. Koo, "Source code security analysis tool functional specification version 1.0," Special Publication (NIST SP), Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. 500-268 1.1, May 2007, doi: 10.6028/NIST.SP.500-268.

[5] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, Apr. 2006, doi: 10.1109/TSE.2006.38.

[6] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1279–1288, Jul. 2013, doi: 10.1016/j.infsof.2012.11.007.

[7] B. Chess and J. West, *Secure Programming in Static Analysis*. Reading, MA, USA: Addison-Wesley, 2007, pp. 35–55.

[8] Facebook Infer. *Static Analysis Tool*. Accessed: Dec. 2023. [Online]. Available: https://fbinfer.com/

[9] SonarQube. *Code Quality and Code Security*. Accessed: Dec. 2023. [Online]. Available: https://www.sonarqube.org/

[10] SpotBugs. *Find Bugs in Java Programs*. Accessed: Dec. 2023. [Online]. Available: https://spotbugs.github.io/

[11] Find Security Bugs. *The SpotBugs Plugin for Security Audits of Java Web Applications*. Accessed: Dec. 2023. [Online]. Available: https://find-sec-bugs.github.io/

[12] PMD. *An Extensible Cross-language Static Code Analyzer*. Accessed: Dec. 2023. [Online]. Available: https://pmd.github.io/

[13] Juliet Test Suite. *Artificial Test Suite*. Accessed: Dec. 2023. [Online]. Available: https://samate.nist.gov/SARD/test-suites/111

[14] MITRE Corp. *The Common Weakness Enumeration Initiative.* Accessed: Dec. 2023. [Online]. Available: https://cwe.mitre.org/

[15] TIOBE Index. (2024). *TIOBE Index for February 2024*. Accessed: Feb. 2023. [Online]. Available: https://www.tiobe.com/tiobe-index

[16] M. Alqaradaghi, Apr. 14, 2024, "Static analysis evaluation experiment data," *Zenodo*, doi: 10.5281/zenodo.10969306.

[17] D. Baca, K. Petersen, B. Carlsson, and L. Lundberg, "Static code analysis to detect software security vulnerabilities—Does experience matter?" in *Proc. Int. Conf. Availability, Rel. Secur.*, Mar. 2009, pp. 804–810, doi: 10.1109/ARES.2009.163.

[18] D. Baca, B. Carlsson, K. Petersen, and L. Lundberg, "Improving software security with static automated code analysis in an industry setting," *Softw., Pract. Exper.*, vol. 43, no. 3, pp. 259–279, Feb. 2012, doi: 10.1002/spe.2109.

[19] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Inf. Softw. Technol.*, vol. 68, pp. 18–33, Dec. 2015, doi: 10.1016/j.infsof.2015.08.002.

[20] M. A. Al Mamun, A. Khanam, H. Grahn, and R. Feldt, "Comparing four static analysis tools for Java concurrency bugs," in *Proc. 3rd Swedish Workshop Multi-Core Comput. (MCC)*, 2010, pp. 18–19.

[21] M. Alqaradaghi, G. Morse, and T. Kozsik, "Detecting security vulnerabilities with static analysis—A case study," *Pollack Periodica*, vol. 17, no. 2, pp. 1–7, Jun. 2021, doi: 10.1556/606.2021.00454.

[22] W. Stikkelorum and M. Bruntink, "Challenges of using sound and complete static code analysis tools in industrial software," M.S. thesis, Dept. Math. Comp. Sci., Fac. Sci., Univ. Amsterdam, Amsterdam, The Netherlands, 2016.

[23] Common Weakness Enumeration. *Deprecated CWEs*. Accessed: Dec. 2023. [Online]. Available: https://cwe.mitre.org/data/definitions/604.html

[24] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, May 2013, pp. 672–681, doi: 10.1109/ICSE.2013.6606613.

[25] V. Okun, A. Delaitre, and P. Black, "Report on the static analysis tool exposition (SATE) IV," Special Publication (NIST SP), Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep., Dec. 2023, doi: 10.6028/NIST.SP.500-297.

[26] P. Hegedus and R. Ferenc, "Static code analysis alarms filtering reloaded: A new real-world dataset and its ML-based utilization," *IEEE Access*, vol. 10, pp. 55090–55101, 2022, doi: 10.1109/ACCESS.2022.3176865.

[27] M. Alqaradaghi and T. Kozsik, "Inferring the best static analysis tool for null pointer dereference in Java source code," in *Proc. 9th Workshop Softw. Quality Anal., Monitoring, Improvement Appl. (SQAMIA)*, vol. 3237, Sep. 2022, pp. 1–12. [Online]. Available: http://ceur-ws.org/Vol-3237/

[28] M. Karlsen. *Random Chance of Detection for Some Files in Juliet 1.3 CWE89 SQL Injection*. Accessed: Dec. 2023. [Online]. Available: http://github.com/find-sec-bugs/find-sec-bugs/issues/456

[29] S. Beba and M. M. Karlsen, "Implementation analysis of open-source static analysis tools for detecting security vulnerabilities," M.S. thesis, Dept. Comput. Sci., NTNU, Trondheim, Norway, 2019.

[30] The OWASP Foundation. *Vulnerabilities*. Accessed: Dec. 2023. [Online]. Available: https://owasp.org/www-community/vulnerabilities

[31] V. V. Desai and V. J. Jariwala, "Comprehensive empirical study of static code analysis tools for C language," *Int. J. Intell. Syst. Appl. Eng.*, vol. 10, no. 4, pp. 695–700, 2022.

[32] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static C code analyzers for vulnerability detection," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2022, pp. 544–555, doi: 10.1145/3533767.3534380.

[33] A. Imparato, R. R. Maietta, S. Scala, and V. Vacca, "A comparative study of static analysis tools for AUTOSAR automotive software components development," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops (ISSREW)*, Toulouse, France, Oct. 2017, pp. 65–68, doi: 10.1109/ISSREW.2017.21.

[34] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? Evaluating smart contract static analysis tools using bug injection," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2020, pp. 415–427, doi: 10.1145/3395363.3397385.

[35] A. Nguyen-Duc, M. Viet Do, Q. Luong Hong, and K. N. Khac, "On the combination of static analysis for software security assessment - a case study of an open-source e-government project," 2021, *arXiv:2103.08010*.

[36] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and Java source code," *Proc. Comput. Sci.*, vol. 171, pp. 2023–2029, Jan. 2020, doi: 10.1016/j.procs.2020.04.217.

[37] R. Mahmood and Q. H. Mahmoud, "Evaluation of static analysis tools for finding vulnerabilities in Java and C/C++ source code," 2018, *arXiv:1805.09040*.

[38] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests," *Inf. Softw. Technol.*, vol. 55, no. 8, pp. 1462–1476, Aug. 2013, doi: 10.1016/j.infsof.2013.02.005.

[39] V. Lenarduzzi, S. Lujan, N. Saarimaki, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," 2021, *arXiv:2101.08832*.

[40] R. Amankwah, J. Chen, H. Song, and P. K. Kudjo, "Bug detection in Java code: An extensive evaluation of static analysis tools using juliet test suites," *Softw., Pract. Exper.*, vol. 53, no. 5, pp. 1125–1143, Dec. 2022, doi: 10.1002/spe.3181.

[41] R. Amankwah, J. Chen, A. A. Amponsah, P. K. Kudjo, V. Ocran, and C. O. Anang, "Fast bug detection algorithm for identifying potential vulnerabilities in juliet test cases," in *Proc. IEEE 8th Int. Conf. Smart City Informatization (iSCI)*, Dec. 2020, pp. 89–94, doi: 10.1109/iSCI50694.2020.00021.

[42] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes, "Myths and facts about static application security testing tools: An action research at telenor digital," in *Agile Processes in Software Engineering and Extreme Programming* (Lecture Notes in Bus. Information Processing), vol. 314. Portugal: Springer, 2018, pp. 86–103, doi: 10.1007/978-3-319-91602-6.
[43] N. Manzoor, H. Munir, and M. Moayyed, "Comparison of static analysis tools for finding concurrency bugs," in *Proc. IEEE 23rd Int. Symp. Softw. Rel. Eng. Workshops*, Dallas, TX, USA, Nov. 2012, pp. 129–133, doi: 10.1109/ISSREW.2012.28.
[44] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," in *Proc. Test. Communicating Syst. TestCom*, in Lecture Notes in Computer Science, 2005, pp. 40–55, doi: 10.1007/11430230_4.
[45] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for Java," in *Proc. 15th IEEE Int. Symp. Softw. Rel. Eng.*, Saint-Malo, France, Nov. 2004, pp. 245–256, doi: 10.1109/ISSRE.2004.1.
[46] J. Li, S. Beba, and M. M. Karlsen, "Evaluation of open-source IDE plugins for detecting security vulnerabilities," in *Proc. Eval. Assessment Softw. Eng.*, Apr. 2019, pp. 200–209.
[47] R. Brenguier, L. Cordeiro, D. Kroening, and P. Schrammel, "JBMC: A bounded model checking tool for Java bytecode," 2023, *arXiv:2302.02381*.
[48] V.-P. Ranganath and J. Mitra, "Are free Android app security analysis tools effective in detecting known vulnerabilities?" *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 178–219, Jan. 2020, doi: 10.1007/s10664-019-09749-y.
[49] P. E. Black, "Juliet 1.3 test suite: Changes from 1.2," Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Rep., Jun. 2018, doi: 10.6028/NIST.TN.1995.

MIDYA ALQARADAGHI was born in Baghdad, Iraq. She received the B.Sc. degree in information technology from Middle Technical University, Baghdad, Iraq, in 2006, and the M.Sc. degree in computer science from Sam Higginbottom University, India, in 2015. She is now a Ph.D. candidate and an instructor of programming languages lab at the Department of Programming Languages and Compilers, ELTE, Eötvös Loránd University, Budapest, Hungary. From 2015 to 2019, she worked as a Teaching Assistant at Northern Technical University, Kirkuk, Iraq. Her research focuses on finding security vulnerabilities in Java code using static analysis tools.

TAMÁS KOZSIK is currently an Associate Professor with the Department of Programming Languages and Compilers, Eötvös Loránd University (ELTE). His research interests include formal verification, programming paradigms (i.e., functional programming, concurrent programming, and quantum computing), static analysis, refactoring, and domain-specific programming languages.