AKADÉMIAI KIADÓ

Pollack Periodica • An International Journal for Engineering and Information Sciences

17 (2022) 2, 1–7

DOI: 10.1556/606.2021.00454
© 2021 The Author(s)

ORIGINAL RESEARCH PAPER

Detecting security vulnerabilities with static analysis – A case study

Midya Alqaradaghi[1,2*], Gregory Morse[1,3] and Tamás Kozsik[1,3]

[1] Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Egyetem tér 1-3, 1053 Budapest, Hungary

[2] Technical College of Kirkuk, Northern Technical University, 36001 Kirkuk, Iraq

[3] Research Group, Eötvös Loránd University, Martonvásár, Hungary

Received: May 29, 2021 • Revised manuscript received: September 23, 2021 • Accepted: September 30, 2021
Published online: December 3, 2021

ABSTRACT

Many security vulnerabilities can be detected by static analysis. This paper is a case study and a performance comparison of four open-source static analysis tools and plugins (PMD, SpotBugs, Find Security Bugs, and SonarQube) on Java source code. Experiments have been conducted on the widely used Juliet Test Suite with respect to six selected weaknesses from the official Top 25 list of Common Weakness Enumeration. In this study, analysis metrics have been calculated for helping Java developers decide which tools can be used when checking their programs for security vulnerabilities. It turned out that particular weaknesses are best detected with particular tools.

KEYWORDS

static analysis tool, accuracy, security vulnerability, Java, common weakness enumeration, Juliet

1. INTRODUCTION

One of the most important responsibilities for a software company is to ensure the secure operation of their software products. People's lives depend more and more on software intensive systems (e.g., in self-driving cars, smart cities and homes, health, government, and financial sectors), hence security failures and vulnerabilities have an increasing effect. One reason behind many security vulnerabilities is the low quality of the source code [1, 2]. Vulnerability is a flaw present in one of the system components, which may result in security failure when triggered accidentally or exploited intentionally [3]. A software security failure can lead to a user getting unauthorized access and affecting badly its behavior and functionality.

For ensuring high quality, it is essential to perform vulnerability detection during the development and maintenance of the code. One way to achieve this is to perform static analysis, which can detect vulnerabilities in the early phases of the software development process. Several static analysis tools are available today for different programming languages, both proprietary and open source.

There are many research results to assess the capabilities of static analysis tools. In this paper, three tools for the Java programming language, namely PMD [4], SpotBugs (SB) [5], and SonarQube ™ Community Edition with its SonarScanner engine (SS) [6], as well as a tool extension, Find Security Bugs (FSB) [7] have been selected because they are free, open-source and widely used. They are evaluated according to their ability to detect six weakness categories (Integer Overflow, Resource Exhaustion, Null Pointer Dereference, Resource Leak, Command Injection, and SQL-Injection), which have been selected from the Common Weakness Enumeration ™ (CWE) [8]. CWE is a common taxonomy of software weaknesses

*Corresponding author.
E-mail: Alqaradaghi.Midya@inf.elte.hu

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

and vulnerabilities, and it covers a wide range of design and implementation flaws. The analyses were applied on test cases of the Juliet Test Suite [9], which is an artificial code base developed specifically for the evaluation of the performance of static analysis tools. Juliet contains six weakness categories from the *Top 25 Most Dangerous Software Weaknesses for 2019 and 2020* [10] – these six categories are used in the present research.

The goal of this paper is to evaluate the performance of some of the most commonly used open-source tools on some of the most dangerous weaknesses and on a test suite developed exactly for this purpose. This research therefore can only give estimation on how well these tools would perform on real-world software.

1.1. Related work

When performing similar comparisons on Java and other programming languages, researchers followed various approaches, often different from the present study. Some researchers used existing, real code bases; others relied on artificial codebases (like in this study), and some developed their own applications and have injected vulnerabilities into them for benchmarking purposes. Without completeness in mind, some of the related works are pointed out here.

Emanuelsson and Nilsson [11] survey the underlying technology for three commercial tools: Klocwork, Coverity, and FlexeLint. Their survey includes checks for security vulnerabilities. They also present practical experiences from evaluations at the telecom company Ericsson. This research focuses on free, rather than commercial, tools.

Kaur and Nayyar [12] made a comparison among three C/C++ tools and two Java tools based on the categories of vulnerabilities detected by those tools, and the ratio of False Positives (FPs) produced by each one of them. They showed that there are certain vulnerabilities that were not detected by any of the tools, and their results on the Java tools showed that SpotBugs was able to find more vulnerabilities than PMD, which is in accordance with the findings of this study. They present five vulnerabilities categories for both Java tools. Among those, there are two, which are addressed by this study as well (CWE89 and CWE476). Further analysis tools have been also investigated, and the comparison has been done on a different input.

Goseva-Popstojanovaa and Perhinschi [13] evaluated three commercial static analysis tools (which support both C/C++ and Java) using 19 CWEs, including two from the present research (CWE190 and CWE476). They work with three real-world open-source programs (Gzip, Dovecot, and Tomcat) and with the Juliet Test Suite. They experienced high false-negative rates, and they state that the state-of-the-art tools were not very effective in detecting security weaknesses. The present research focuses on mostly different CWEs and different tools, although merely on the Juliet Test Suite.

Beba et al. [14] present usability, coverage, and performance evaluation on five static analysis tools and plugins including SpotBugs and Find Security Bugs (but not PMD and SonarQube Community Edition). They also rely on the Juliet Test Suite, and they have two vulnerability categories in common with us (CWE78 and CWE89). Their study explains many interesting anomalies and proposes valuable improvements on Find Security Bugs.

Compared to the aforementioned related works, the present study focuses on a limited, but important set of weaknesses (ones in the Top 25 Most Dangerous Software Weaknesses for 2019 and 2020) and uses a more refined methodology yielding more detailed results.

2. METHODOLOGY

This paper evaluates free static analysis tools for the Java programming language. The three static analysis tools and the plug-in used in these experiments are PMD 6.34.0, SpotBugs 4.2.3, Find Security Bugs 1.11.0, and SonarQube 8.8.0.42792 (Community Edition) with SonarScanner 4.6.0.2311. Moreover, SpotBugs has been extended with FindBugs-Contrib library 7.4.7. The development kit for the presented experiments was Oracle Java 8 (1.8.0_281). Note that running the SonarQube server requires Java 14, but it is only used to present analysis results, while SonarScanner with Java 8 is used for the experiments. Find Security Bugs plug-in performance results have been considered separately in this paper, but they were actually obtained by adding it as a plug-in to SpotBugs. All the tools were used with default configuration except for SpotBugs and Find Security Bugs where medium (which also includes high) confidence settings were applied as justified in Section 3.

The study focuses on security vulnerabilities described in the CWE™, which is a list of common software and hardware weakness types that have security ramifications. Six vulnerability categories have been chosen, all present in the 2020 and 2019 *CWE Top 25 Most Dangerous Software Weaknesses*, as well as in the Juliet Test Suite (version 1.3) for Java, which is an extensive set of test cases organized around 112 CWE categories.

The tools of this study have a difficult time detecting the 6 CWE because all of them are better suited to dynamic analysis as they require data-flow tracing. This is likely why they are on the Top 25 list. Following is a brief description of each investigated CWE.

CWE190 (Integer Overflow) is about performing an integer operation that results in a value too large or too small to store in the specified representation. This can lead to a wraparound e.g., a negative number instead of an expected positive value. If the wrapping around is unexpected, it could lead to security problems, especially if the integer overflow is triggered using user inputs. The case becomes security-critical when the result is used to control looping, make a security decision, or determine the offset or size in behaviors like memory allocation, copying, concatenation, etc.

CWE400 (Uncontrolled Resource Consumption) is about triggering the allocation of some limited resources like memory, file system, and CPU by an unauthorized user.

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

This may lead to a denial-of-service attack that consumes all available resources, especially if the number or size of the resources is not controlled.

CWE476 (Null Pointer Dereference) occurs when the application dereferences a reference which is supposed to be valid but is actually *null*, and that will cause runtime exception in Java. This issue can occur as a result of several flaws, like race conditions, and simple programming errors.

CWE772 (Missing Release of Resource after Effective Lifetime) occurs when resources are not released after use. This may lead to denial-of-service attacks by causing the allocation of resources without triggering their release.

CWE78 (Improper Neutralization of Special Elements used in an OS Command/OS Command Injection) may result in the execution of unexpected and dangerous commands directly on the operating system, and that leads to a vulnerability in the environment which the attacker is not supposed to have direct access to, like in a web application.

CWE89 (Improper Neutralization of Special Elements used in an SQL Command/SQL Injection) is similar – it may allow attackers to force the execution of unwanted or malicious SQL commands. This flaw occurs when the program constructs SQL commands or queries by textually including the values received from untrusted sources, e.g., user input, without any validation or escaping.

2.1. Juliet Test Suite (version 1.3) for Java

The Juliet Test Suite – created and developed by the Center of Assured Software (CAS) of the National Security Agency (NSA) – is an artificial code base, which contains intentional flaws, and is created specifically to test the abilities of static analysis tools for detecting flaws and security vulnerabilities. This test suite contains test cases for different programming languages, including Java.

Each test case for the 6 CWEs that are selected for this study targets exactly one type of flaw. Since the flaw can ultimately occur in one method, the method with the flaw with rule criterion is enough to determine a Real Positive (P). Every non-trivial function except the one with the flaw is a Real Negative (N).

Juliet uses the concept of a flow variant [15]. Each test case [16] is generated with one of three possible flow variants: Baseline, Control flow, and Data flow, which refer to the required analysis to reveal a flaw. The comments in the test cases also document which flow variant the test case belongs to. The test cases themselves have file groups based on name, numbering, and suffix.

For test cases without a suffix and those with an alphabetic suffix "a", "b", etc., Juliet sometimes uses the concept of a data source and a data sink to divide the chain of events leading to a flaw into two parts, both of which must be flawed for a flaw to actually occur. The naming convention of Juliet uses `bad()` for the entry into a flaw, and this may, in turn, use a `badSource()` and `badSink()` pattern. The function `badSink()` is the P if present otherwise `bad()` is used. Every other function aside from trivial wrapper functions, which in this case is `good()` when it merely calls `good1()`, `good2()`, etc., is an N. Of four possible combinations with sources and sink, three are not flawed: a bad source to a good sink (`B2G`), a good source to a bad sink (`G2B`), and where source and sink are good (which Juliet does not bother testing). For test cases with suffixes "_bad" and "_good", the `action()` function in the former is the P.

In this context, the term *Raw FPs* has been used for the actual number of FP detections, while when referring to FPs this indicates the count based on the number of functions containing one or more FPs. This way, it is easier to reason about FPs and merely interprets the raw form in terms of its average per function. If multiple detections occur in a P function, one is a True Positive (TP), the remainder(s) are Raw FPs.

The six CWE categories and their representations in Juliet are shortly introduced here. Due to space limitations, the test cases and the different flow variants they follow are not detailed. The interested reader is referred to the Juliet CWE source code [9].

CWE190 test cases perform addition, increment, multiplication or squaring operations on byte, integer, Integer, short and long values without checking if they are less than their respective `MAX_VALUE`. A value can be read by `readLine()` from the console, or a Transmission Control Protocol (TCP) connection (socket), or from a database by `ResultSet.getString()`, from an environment variable, using `System.getenv()`, from a `File`, from cookies, from a query string using `getParameter()`, from a properties file, from a system property, and a Uniform Resource Locator (URL) connection, or a pseudo-random number generator. A static analyzer should detect the code line where the arithmetic operation is performed without checking the suitability of the input value.

CWE400 is very similar to CWE190 in that they are both checking boundary conditions for integer values, but this time those integer values are used to control the iteration count of loop constructs. Juliet provides test cases for values that are read similarly to those for CWE190. CWE400 takes non-validated input or maximal or minimal integer values and executes a loop, which contains operations including thread sleep, writing to a console, or file, and thus exhausting resources including CPU, IO, disk space, as well as process, thread, and other operating system handles. Resources exhausted are CPU (for_loop) write to file (write/Servlet), processes/threads/handles (sleep). A static analyzer should only detect the line of in a `bad()` method where there is a for loop using a non-validated variable in the loop condition.

CWE476 refers to dereferencing a variable of a reference type without checking for a null value. Technically, the test cases invoke the `length()` method on a String. When the value equals null, a NullPointerException is raised in the program. The three tested scenarios are: value dereferenced after null checking, value dereferenced without null checking or unnecessary check for null after dereferencing the value. The last one was missed by all the tools in this study because they are not built with actual null dereference. So it is not a correctness issue but a bad practice. The validation uses an

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

if-statement, while the dereference of null values is triggered when calling some methods. A static analyzer should detect the flawed method where a call is made on a null value. The `good()` methods are built either with `G2B:` setting data to a non-null value and not checking for nullity before dereferencing, or with `B2G:` setting data to null and then checking data for nullity before dereferencing. Neither should be reported by the static analysis tool.

CWE772: Java DataBase Connectivity (JDBC) is allocating system resources including a Connection object, a *PreparedStatement* object, and a *ResultSet* object, and though the test case is one test, there could be three or one TP depending on the way the tool handles the resource closing issues. Some ambiguity is present in these test cases where due to aggregation, the JDK requires any one of several objects to invoke a `close()` method, but this would not increase the Ps. A static analyzer should detect one of the following locations: the first or the last line of a function, variable declaration line, object allocation line, nearest try block (start or end line), the first line after catch block (in case of a missing finally block), or the first or last line of the finally-block.

CWE78: test cases of Juliet cover OS command injections that are caused by passing a non-validated input to *Runtime.exec (String)*. A part of the input string is read either from a TCP connection, the console, the result of a database query, an environment variable, a file, a cookie, a servlet query string, a properties file, a system property, or a *URLConnection*. In contrast, the Ns use a hardcoded String. None of the `good()` methods in the test cases perform real input validation because they are all taking data of good source (and of course bad sink). The analysis tool is expected to report the code line in `bad()` where an OS command with non-validated external input is executed.

CWE89: here, the ways data is read are the same as CWE78 test cases, except that it is using JDBC SqlConnection.executeQuery(String) to execute a database query. SQL injections may happen when the non-validated external input is concatenated to an SQL statement. The analyzer is expected to report where the execution of the SQL command containing non-validated parts takes place. Ns are built either with `G2B:` using a hardcoded String and data concatenating into SQL statement, or with `B2G:` get data from external sources and use prepared statement and execute properly. Neither should be reported by the analysis tool.

2.2. Comparison metrics

Ten different metrics have been used in this study and they are defined as follows:

$$TPR = \frac{TP}{P},\ FPR = \frac{FP}{N},\ Avg\,FP = \frac{Raw\,FP}{FP},$$

$$Gi = \frac{2\times TPR\times(1-FPR)}{TPR+1-FPR}, TPO = \frac{\sum_{i\in C} TPR_i}{|C|},$$

$$FPO = \frac{\sum_{i\in C} FPR_i}{|C|},\ GiO = \frac{\sum_{i\in C} Gi_i}{|C|},$$

$$TPSC_t = \frac{\sum_{i\in C} TPR_i}{|Ct|},$$

$$FPSC_t = \frac{\sum_{i\in C} FR_i}{|Ct|},\ GiSC_t = \frac{\sum_{i\in C} Gi_i}{|Ct|}.$$

True Positive Ratio (TPR, *recall*) is the ratio of the number of TP reports given by a tool to the sum of Ps for every CWE. Similarly, false positive ratio (*FPR, probability of false alarm*) provides the ratio of FP reports to the sum of Ns. *Avg FP* provides the average duplicity of FPs per function, recalling that *Raw FP* is FPs + duplicated TPs. *Gi* describes the accuracy of the analysis as the harmonic mean of TPR and 1 – FPR (where 1 – FPR is the ratio of true negatives and Ns). *C* stands for the set of identifiers of CWEs: C= {190, 400, 476, 772, 78, 89}, and t indicates a tool: $t \in$ {PMD, SB, FSB, SS}, C_{PMD} = {476, 772}, C_{SB} = {476, 772, 89}, C_{FSB} = {400, 78, 89}, C_{SS} = {190, 400, 476, 772, 89}.

GiO is the overall accuracy of the analysis given by dividing the total sum of the *Gi* by $|C|$, which is the total number of CWEs in this study), while $GiSC_t$ is the per tool accuracy given by dividing the total sum of *Gi* by $|C_t|$, which is the total number of CWE(s) that the specified tool has a rule(s) to detect. Similarly, overall *TP* (*TPO*), per tool *TP* ($TPSC_t$), and overall *FP* (*FPO*), per tool *FP* ($FPSC_t$), have been calculated regarding TPR and FPR, respectively. Accuracy has been characterized with *Gi*, since the classical accuracy metric $Acc = (TN + TP)/(TN + FN + TP + FP)$ is not applicable in the current experiment, as the positive and negative occurrences are imbalanced in the investigated Juliet Test Suite test cases.

3. RESULTS AND DISCUSSION

The experiment conducted a comparison of the tools with six categories of test cases of the *Juliet Test Suite (version 1.3)* for Java. Section 3.1 presents the results of running every tool on each CWE, Section 3.2 presents the results of analyzing CWE78 and CWE89 test cases with FSB using different Java compiler versions, while Section 3.3 presents overall tools' performance results.

SB and FSB classify the detected weaknesses by the likelihood of their veracity into low, medium, and high confidence. In this paper, only *medium and high confidence* detections are presented, for the following reasons:

1. CWE476 analysis results have been improved this way as this excludes all the FPs that are coming from duplicate detections by the rule NP_LOAD_OF_KNOWN_NULL_VALUE of Style category, which also gives a large number of FPs;
2. For CWE772 nothing will change;
3. SB and FSB are using taint analysis with string concatenation, which is the reason for giving high FPR in both

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

CWE78 and CWE89. They detect most of the lines that bind data to dynamic OS and SQL commands execution regardless of the source of the data. In other words, they detect all Ns that use G2B; so, when detections with low confidence are removed, the results are improved in the case of CWE78: most of the FP excluded, while only a few TP have been missed, which improves the accuracy score, *Gi*. In the case of CWE89 detections results by FSB with medium and high confidence decrease the *Gi* by 0.38, but on the other hand, it also decreases the #FPs from 1016 to 252, which is a significant improvement.

4. The results for CWE89 by SB do not change as they are all *medium and high confidence* detections.

3.1. Running tools on the six CWEs of Juliet Test Suite

Table 1 presents the results of running the tools on the six CWEs of the Juliet Test Suite. The number of *test cases* is presented in the second column for each CWE and was extracted from the Juliet documentation (and programmatically verified). The number of *real negatives* is presented in the third column deduced programmatically. CWE190 and CWE400 were not detected in any of the tools, because determining boundary values on integers requires having analysis, which can propagate boundary value information across any control flow units like loops while having perfect data flow analysis. The difficulty of making inferences based on the undecidability of static analysis [17] has a theoretical limit as well as many practical constraints e.g., with constructs as ordinary as loops whose invariants and post-conditions need to be deduced. For this reason, boundary value issues are largely not covered by static analysis tools, beyond some primitive heuristics to find specific patterns. In a certain regard, this applies to all of the six CWEs studied, except the others follow more common patterns where heuristics are broader while still being straightforward to implement.

In the case of CWE476, SS gave the highest accuracy metric 68.9. All test cases of flow variant Baseline and most of those of Control flow were detected perfectly without any FP. However, SS missed the test cases of flow variant Data flow. SB detects 102 test cases without FP, with the rule NP_ALWAYS_NULL. Those are flow variants of type Baseline, most of the control flow, and only one test case with data flow type. Two other rules of SB detected 26 of the chained test cases (NP_NULL_PARAM_DEREF_ALL_TARGETS_DANGEROUS), and one data flow variant (NP_NULL_PARAM_DEREF_NONVIRTUAL). However, these detections are reported where the null value is passed in method `bad()` rather than the location of receiving it in `badSink()`, hence according to this methodology, those are FP. The extra Raw FPs are coming from duplicate detection for some test cases by one of the *Style* rule detectors. PMD results in 46.3 accuracy due to the high FPR. The rule in PMD is not detected Null Pointer Dereference, it actually targets Null Assignment. FPs come from null assignments without dereferencing the null value. Moreover, it totally missed all the test cases that are named binary_if (baseline flow variant), deref_after_check, and null_check_after_deref (which are both control flow variants).

CWE772 is detected by PMD with an accuracy of 100.0. It gives five detections for the two test cases, one for each non-closed resource. The extra detections are also in the bad()method, and they are Raw FP according to the methodology. SB and SS could detect only one of the test cases. SB reports two FPs from the method `bad()`.

CWE78 is only detected by FSB because there are no available rules in the other tools to detect this weakness. However, the accuracy is very high, 90.5.

CWE89 is detected by FSB with an accuracy of 91.1, which is the highest among the tools. The FPs are coming from `G2B`methods in data flow variant test cases. However, the accuracies for SB and SS are smaller because they detect issues in `G2B`methods for all the flow variant test cases. For detecting SQL-Injection in a better way using SS, the programmer might use the vulnerability rule "Database queries should not be vulnerable to injection attacks" (s3649) of the Developer Edition.

The tool with the highest accuracy for CWE476 is SS, but the tool with the lowest FPR is SB (including only medium and high confidence). FSB plug-in has the highest accuracy for both OS command and SQL injections (CWE78 and CWE89), with an accuracy of 90.5 and 91.1 respectively. This is not surprising because these types of weaknesses are the focus of the tool. However, in the case of CWE89, the highest TPR is achieved by SB and SS, which had identical

Table 1. The results of running all the tools on four CWEs of Juliet (as CWE190 and 400 have not been included in the table because they have not been detected by any of the tools)

CWE	# Test Cases	# N	# Files	Tool	TP	TPR (%)	FP	FPR (%)	Gi	Raw FP	Avg. FP
476	198	952	293	PMD	76	38.4	398	41.8	46.3	558	1.4
				SB	102	51.5	27	2.8	67.3	44	1.63
				SS	110	55.6	88	9.2	68.9	88	1.00
772	2	2	2	PMD	2	100.0	0	0.0	100.0	3	-
				SB	1	50.0	0	0.0	66.7	2	-
				SS	1	50.0	0	0.0	66.7	0	-
78	444	1,704	720	FSB	376	84.7	48	2.8	90.5	48	1.00
89	2,220	13,020	3,660	SB	2,220	100.0	3,000	23.0	87.0	3,000	1.00
				FSB	1,889	85.1	252	1.9	91.1	252	1.00
				SS	2,220	100.0	3,000	23.0	87.0	3,060	1.02

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

values. Details on detectors and rules as well as the analysis workflow of this research can be found in [18].

3.2. CWE78 and CWE89 test cases with FSB using different Java compiler versions

The CWE test cases of this study have been investigated using Java Compiler versions 14.0.1 and 16.0.1 and different results have been observed for both CWE78 and CWE89 when analyzed with FSB. FSB with these Java versions yields total recall (100 TPR) but significantly increased false alarm rate (36.6 FPR) and so less accuracy (77.6 Gi) compared to results obtained with Java 8 (84.7 TPR, 2.8 FPR and 90.5 Gi). Similarly, analyzing CWE89 gives 100 TPR and 23.5 FPR, hence 86.7 Gi with higher Java versions compared to TPR FPR Gi obtained with Java 8. The extra FPs result from G2B methods. The reason is that Java compiler 8 is doing different string optimizations, which leads to minimizing the FPs in both cases. (It is well-known that Java byte code uses a constant pool, which deals with immutable objects like Strings). Those results cannot be further optimized by setting lower confidence since they are all of medium confidence. Similar inconsistencies across Java compiler versions were not found for SS or SB, and PMD is not relevant here as it analyzes source code rather than byte code.

FSB yields another type of inconsistency with CWE89 of Juliet. It gives different results when using batches of test case packages (s01, s02, s03, and s04) rather than processing each one separately. This issue has been reported already [19] and has been partially resolved [14].

From the above, it is concluded that the accuracy of the CWE78 and CWE89 analysis in FSB depends heavily on the used Java version so there is still room for improvement in the tool chain, i.e., with the collaboration of FSB and SB.

3.3. Performance of the tools

Figure 1 presents the overall performance of the tools of this study on analyzing the six CWEs. SS has the highest overall accuracy in detecting the six CWEs. PMD and SB have the highest per tool accuracy. However, PMD also has the highest per tool FP which is 20.9.

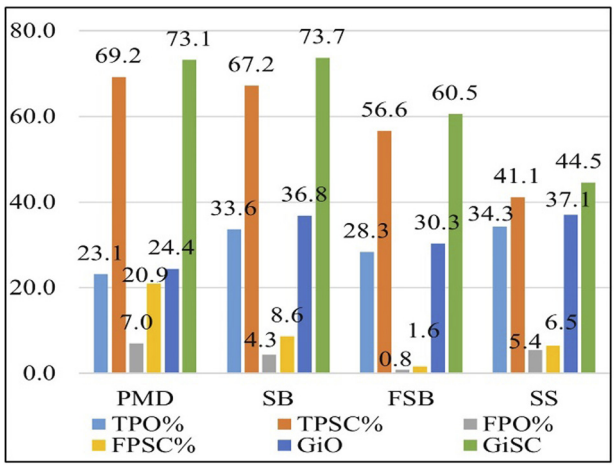

Fig. 1. Overall and per tool performance on the six CWEs

4. CONCLUSION

SonarQube gave the highest accuracy for all investigated CWEs, while PMD and SpotBugs result in the highest accuracy when the analysis is restricted to those CWEs, which the tools claim they can detect. As expected, test cases of Juliet with flow variant of type *data flow* were the most difficult for the tools to detect. All of the investigated tools need improvements with respect to all the investigated weaknesses. Juliet Test Suite would also benefit from a wider weakness spectrum (e.g., CWE400) as well as an increased number of test cases for some of the categories (e.g., CWE772, which only includes two test cases). Initial investigations on the combined use of the tools gave no significant benefits as opposed to using the best particular tool for a given CWE.

As for future work, the study would extend to cover all 112 test cases of the Juliet Test Suite, which would give a better judgment of the performance of the static analysis tools. Other (free and commercial) static analysis tools can also be involved in the research (like the Infer static analyzer from Facebook).

ACKNOWLEDGMENTS

Midya Alqaradaghi has been supported by the Stipendium Hungaricum program. The research of Gregory Morse and Tamás Kozsik has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

REFERENCES

[1] C. Kuang, Q. Miao, and H. U. A. Chen, "Analysis of software vulnerability," in *Proc. 5th WSEAS Int. Conf. Inf. Secur. Priv.*, Venice, Italy, Nov. 20–22, 2006, pp. 218–223.

[2] A. Bagnato, Ed., "Security in model-driven architecture," in *Proceedings on European Workshop on Security in Model Driven Architecture*, Enschede, The Netherlands, June 24, 2009.

[3] P. E. Black, M. J. Kass, and H. M. M. Koo, "Source code security Analysis tool functional specification, Version 1.0," Special Publication, National Institute of Standards and Technology, Gaithersburg, MD, USA, 2007, Paper no. 500–268.

[4] PMD An extensible cross-language static code analyzer. [Online]. Available: https://pmd.github.io/. Accessed: May 18, 2021.

[5] SpotBugs, Find bugs in Java Programs. [Online]. Available: https://spotbugs.github.io/. Accessed: May 18, 2021.

[6] SonarQqube. [Online]. Available: https://www.sonarqube.org/. Accessed: May 18, 2021.

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC

[7] Find Security Bugs. [Online]. Available: https://find-sec-bugs.github.io/. Accessed: May 18, 2021.
[8] Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/. Accessed: May 18, 2021.
[9] Juliet Test Suite, National Institute of Standard and Technology. [Online]. Available: https://samate.nist.gov/SRD/testsuite.php. Accessed: May 18, 2021.
[10] CWE, Common weakness enumeration, A community-developed list of software and hardware weakness types. [Online]. Available: https://cwe.mitre.org/data/definitions/1350.html. Accessed: May 16, 2021.
[11] P. Emanuelsson and U. Nilsson, "A comparative study of industrial static analysis tools," *Electron. Notes Theor. Comput. Sci.*, vol. 217, no. C, pp. 5–21, 2008.
[12] A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in C/C++ and JAVA source code," *Proced. Comput. Sci.*, vol. 171, pp. 2023–2029, 2020.
[13] 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, 2015.
[14] S. Beba and M. M. Karlsen, "Implementation analysis of open-source Static analysis tools for detecting security vulnerabilities," MSc Thesis, Norwegian University of Science and Technology, 2019.
[15] L. Kota and K. Jarmai, "Improving optimization using adaptive algorithms," *Pollack Period.*, vol. 16, no. 1, pp. 14–18, 2021.
[16] E. Ferencz and B. Goldschmidt, "A novel program synthesis approach in test driven software development," *Pollack Period.*, vol. 12, no. 2, pp. 3–15, 2017.
[17] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, 1992.
[18] Static-analysis-tools. [Online]. Available: https://github.com/Midya-ELTE/Static-analysis-tools. Accessed: June 01, 2021.
[19] Random chance of detection for some files in Juliet 1.3 CWE89 SQL injection. [Online]. Available: https://github.com/find-sec-bugs/find-sec-bugs/issues/456. Accessed: May 01, 2021.

Open Access. This is an open-access article distributed under the terms of the Creative Commons Attribution 4.0 International License (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited, a link to the CC License is provided, and changes – if any – are indicated. (SID_1)

Brought to you by Library and Information Centre of the Hungarian Academy of Sciences MTA | Unauthenticated | Downloaded 07/19/22 09:32 AM UTC