Inferring The Best Static Analysis Tool for Null Pointer Dereference in Java Source Code

Midya Alqaradaghi[1,2,*], Tamás Kozsik[1]

[1]Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, Budapest, Hungary
[2]Technical College of Kirkuk, Northern Technical University, Kirkuk, Iraq

Abstract
Finding software bugs and security vulnerabilities using static source code analysis is a viable method. Static source code analysis techniques are already sufficiently matured for industrial use, and numerous tools have been developed to aid in the automatic detection of software faults. In this paper, the capabilities of three static source code analysis tools are investigated with respect to identifying null pointer dereference in Java source code. Our research uses artificial test cases as a benchmark. The study reports performance results based on five metrics. The experiments show that Facebook Infer outperforms the other tools in identifying null pointer dereference.

Keywords
Static analysis, Facebook Infer, SonarQube, SpotBugs, null pointer dereference, CWE476

1. Introduction

Static analysis technologies uncover security flaws early in the software development phase, which saves time and cost and ensures quick feedback for the responsible programmer. The security vulnerabilities and bugs that can be identified using static analysis techniques are in wide ranges, starting from simple programming errors, and ending with more complex ones like access control issues [1].

Null pointer dereference is a memory access error [2]. It arises when a program follows a pointer that is supposed to refer to a valid object but is actually null; this results in a program crash or an exception. In the Java programming language, this issue causes a *NullPointerException* to be thrown. Null pointer dereferences typically originate from some faulty assumptions made by the programmer. The majority of null pointer problems result in general software reliability issues, but if a hacker can provoke it on purpose, they may be able to use the resulting exception to get around security checks or to make the application reveal debugging data that will be valuable for planning future attacks [3]. Null pointer dereference is considered one of the most common programming errors in the Java programming language [2].

There are many static analysis tools – with varying capabilities – which can detect null pointer dereference. It is known that static analysis for an "interesting" problem is undecidable,

SQAMIA 2022: Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, September 11–14, 2022, Novi Sad, Serbia
*Corresponding author.
alqaradaghi.midya@inf.elte.hu (M. Alqaradaghi); kto@inf.elte.hu (T. Kozsik)
0000-0001-9881-5854 (M. Alqaradaghi); 0000-0003-4484-9172 (T. Kozsik)

i.e., it is not possible to build an algorithm that produces an accurate answer in each and every case [1]. As a result, static analysis tools are prone to reporting findings in source code that are not vulnerabilities (*false positives*), and they may miss to detect some of the vulnerabilities (*false negatives*). Static analysis tools would, in an ideal world, uncover as many vulnerabilities as possible, optimally all, with as few false positives as possible, ideally none.

With the goal of better understanding their strengths and shortcomings, this paper focuses on an empirical assessment of three static analysis tools for their performance in identifying null pointer dereference in Java source code.

For this purpose, we used the *CWE476 Null Pointer Dereference* test cases of the *Juliet Test Suite* [4] to benchmark the Facebook Infer [5], SonarQube [6] and SpotBugs [7] tools.

The main contributions of this work are as follows.

- We present quantitative results on the performance of three well-known, free, and open-source static analysis tools in identifying null pointer dereference in Java source code, based on the Juliet Test Suite.
- We present the numbers and types of shared, unique, and missed detections of these tools.
- We report on five performance metrics – recall, false alarm rate, precision, G-score, and F-measure – for the analyzed test cases.

The main empirical observations of this work are the following.

- None of the used tools were able to detect all null pointer dereference errors in the test cases of the Juliet test suite, specifically 8% of the flawed constructs have been missed by the tools.
- 14% of the vulnerabilities were detected only by Facebook Infer – these have been completely missed by the other two tools. Moreover, Infer gave the highest calculated recall, G-score, and F-measure.
- SpotBugs gave the highest calculated precision and the lowest (best) false alarm rate.

The remainder of the paper is structured as follows. Information on the static analysis tools and the Juliet benchmark, as well as some preliminaries of the research, are presented in Section 2. Then, Section 3 introduces the applied research method. The results of analyzing null pointer dereference test cases by the three tools are given in Section 4. We discuss related work in Section 5. Finally, Section 6 draws the conclusions.

2. Background

Null pointer dereference is a major source of bugs and vulnerabilities in programming languages without static typing support for nullable and non-nullable references [8]. It is also the main memory-related vulnerability in Java, a strongly typed, garbage-collected language.

The Common Weakness Enumeration (CWE) "is a community-developed list of software and hardware weakness types. It serves as a common language, a measuring stick for security tools, and as a baseline for weakness identification, mitigation, and prevention efforts" [9]. It lists null pointer dereference under the identifier CWE476 [10].

Juliet Test Suite [4] is a set of artificial test cases with predetermined outcomes for evaluating the effectiveness of software-assurance techniques, including static analysis tools in discovering various software faults and vulnerabilities. These test cases were developed in order to enable the automatic evaluation of static analysis tools. The Java test cases are divided into 112 weakness categories, including *CWE476 Null Pointer Dereference*, which has positive test cases (flawed constructs with the word *bad* in their names; these are supposed to be reported) and negative test cases (unflawed constructs with the word *good* in their names; these are supposed to be not reported). These two groups of test cases are called *positives* and *negatives*, respectively, throughout the paper.

When a static analysis tool correctly reports one or more *positives*, this is referred to as a *true positive*. Consequently, when a static analysis tool mistakenly reports a *negative*, this is referred to as a *false positive*. A *false negative* is the case when the tool misses one of the *positives*. Finally, a *true negative* is the case when a *negative* is, correctly, not reported.

The versions of the tools used in this study are Infer 1.1.0, SonarQube 8.8.0.42792 (Community Edition) with its SonarScanner 4.6.0.2311, and Spotbugs 4.2.3. Moreover, we have used the most up-to-date version of Juliet Java, which is version 1.3.

3. Research Method

Let us first explain the design of the experiment, including the used metrics. Then we present the details of the implementation of the experiment.

3.1. Experiment Design

Our study follows a two-factorial experiment design aimed to explore the abilities of three selected static analysis tools in the detection of null pointer dereference in the Juliet Java test suite. The two factors (i.e., independent variables) are the following: (1) static analysis tool, and (2) type of vulnerability.

The levels of the first factor are the specific tools used for evaluation. We started the tool selection process by compiling a survey that consisted of twelve commercial tools and eight open-source tools. The main characteristics of each tool were extracted from their documentation. The criteria for making the selection were as follows: (1) it has to be free and widely used; (2) it should specifically identify null pointer dereference; (3) it should support Java. Therefore, we excluded all the commercial tools, and from the list of the eight open-source tools, we first excluded the tools that did not support Java and tools that did not include specific rules for identification of null pointer dereference. Hence we ended up choosing Facebook Infer, SonarQube, and SpotBugs.

For the second factor, there is a single level: we investigate null pointer dereference only, which is CWE476.

As response variables (i.e. dependent variables) we used five metrics: *recall*, *false alarm rate*, *precision*, *G-score*, and *F-measure*; these reflect several dimensions of the performance of the tools. To compute these response variables, we start by calculating *true positives* (TP) and *false positives* (FP) from the reports of each tool. These, along with the number of *positives* and *negatives* are then used to compute the above metrics.

$$Recall = \frac{TP}{Positives} \tag{1}$$

$$False\ alarm\ rate = \frac{FP}{Negatives} \tag{2}$$

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

$$G\ score = \frac{2 * Recall * Specificity}{Recall + Specificity} \tag{4}$$

$$F\ measure = \frac{2 * Recall * Precision}{Recall + Precision} \tag{5}$$

Recall describes the capability of accurately detecting vulnerabilities; it is defined as the ratio of *true positives* to the total number of *positives* (Eq. (1)). Hence, Recall is restricted to faulty constructs and shows the percentage of flawed constructs successfully recognized by a tool.

False Alarm Rate is exclusively concerned with unflawed test cases, and it reflects the percentage of unflawed test cases that have been misidentified as flawed ones. It is the ratio of *false positives* to the total number of *negatives*, or the ratio of mistakenly reporting unflawed constructions as flawed ones. It is given by Eq. (2).

Specificity (used in the computation of G-score) is directly related to false alarm rate. It is the ratio of *true negatives* to all *negatives* (hence it is equal to $1 - \textit{False Alarm Rate}$).

Precision, given with Eq. (3), provides the ratio of *true positives* to the sum of *true positives* and *false positives.* Hence Precision is concerned with all the reported test cases; it shows the percentage of successfully detected flawed constructs to the number of (flawed and unflawed) constructs reported by a tool.

Eq. (4) defines *G-score*, which is the harmonic mean of recall and specificity. It enables us to merge two key metrics into a single one. Similarly, the *F-measure* is the harmonic mean of recall and precision (Eq. (5)). They both describe the accuracy of the analysis.

High recall, precision, G-score, and F-measure, as well as low false alarm rate values, indicate higher performance, and they are all in the interval $[0, 1]$.

3.2. Experiment Execution

The execution of our experiment can be structured into six major steps, as described in the flowchart shown in Figure 1.

Step 1: Evaluate CWE476 test cases. According to previous research [11], Juliet's test cases are not perfect, they may contain some issues. Therefore, we manually reviewed all the *positive* and *negative* test cases of the CWE476 group of Juliet Java in order to evaluate and validate their effectiveness. Only the valid test cases were passed to the next step.

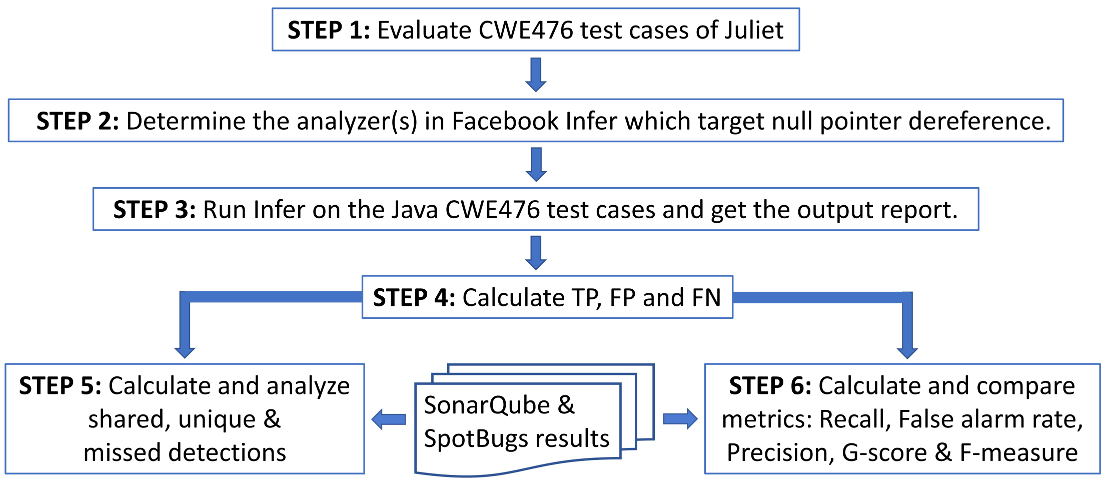
STEP 1: Evaluate CWE476 test cases of Juliet
STEP 2: Determine the analyzer(s) in Facebook Infer which target null pointer dereference.
STEP 3: Run Infer on the Java CWE476 test cases and get the output report.
STEP 4: Calculate TP, FP and FN
STEP 5: Calculate and analyze shared, unique & missed detections
SonarQube & SpotBugs results
STEP 6: Calculate and compare metrics: Recall, False alarm rate, Precision, G-score & F-measure

Figure 1: Flow chart of our experiment.

Step 2: Determine the analyzer(s) of Facebook Infer which target null pointer dereference. Based on the documentation of Infer, we determined the analyzers which target null pointer dereference. This step is important because there may be some checkers that are not turned on by default. The analyzers are *Pulse* and *Biabduction*. The latter works by default when running Infer, while *Pulse* should be explicitly activated. Note that SonarQube and SpotBugs related analyzers are all activated by default.

Step 3: Run Infer on the Java CWE476 test cases and get the output report. Both analyzers of Infer mentioned in Step 2 have been run on all the test cases that were compiled in Step 1. This resulted in an output report which contains all detections. Find more details about the used command and other technicalities in the author's GitHub account [12].

Step 4: Calculate TP, FP, and FN. True positives, false positives, and false negatives were counted for Facebook Infer, and the findings were verified with manual review.

- If the tool reported a positive test case (a bad method), it was counted as a true positive (TP). Only one TP was counted for each detected bad method, regardless of the number of reports on a single test case.
- If the tool reported a negative test case (a good method), then it was counted as a false positive (FP).
- If the tool did not report a positive test case (a bad method), then it was counted as a false negative (FN).

Table 1
TP and FP results of running the three tools on CWE476 of Juliet Test Suite.

Positives	Negatives	Tool	TP	FP
181	466	Infer	166	12
		SonarQube	118	80
		SpotBugs(h)	106	0
		SpotBugs(h&n)	129	0
		SpotBugs(all)	166	264

Step 5: Calculate and analyze shared, unique & missed detections. For calculating the shared, unique, and missed detections, we used the results obtained for Infer from Step 4, and we also brought here the analysis results for SonarQube and SpotBugs from our previous study [13] – with some adjustments (see Section 4).

Step 6: Calculate metrics: Recall, False alarm rate, Precision, G-score, and F-measure. We computed true positives and false positives in Step 3 as listed in Table 1. Those values together with positives and negatives from Step 1 are used in this step to compute the metrics mentioned in Section 3.1.

4. Results and Discussion

The quantitative results of the execution of the experiment described in Section 3 and their qualitative analysis are presented now.

In Step 1 of the experiment, the test cases in Juliet Java 1.3 were thoroughly investigated. It turned out that although documentation of Juliet mentions that the number of *positives* is 198 for CWE476, 17 of them are incorrectly labeled as positives (they do not result in a null pointer dereference). Consequently, we have excluded them from the positive test cases. Moreover, since the number of *negatives* is not mentioned in the documentation, we could simply count the number of “good” methods (which is 496) – however, after the manual review 30 of them turned out to be incorrectly labeled as good, and hence they must be excluded from *negatives*. The final number of *positives* and *negatives* used in this study are 181 and 466, respectively.

More details about the excluded test cases and the script used for counting negatives are provided in the author’s GitHub account [12].

4.1. True Positives and False Positives

The results of running the three tools on Juliet’s null pointer dereference test cases are shown in Table 1. The figures presented here for SonarQube and SpotBugs originate from our previous research [13], but they are adjusted to the slightly different methodology applied in this study; true positives and false positives have been recalculated for SonarQube and SpotBugs to ensure that a uniform data collection principle is followed for all three tools. More information can be found on this in the author’s GitHub account [12].

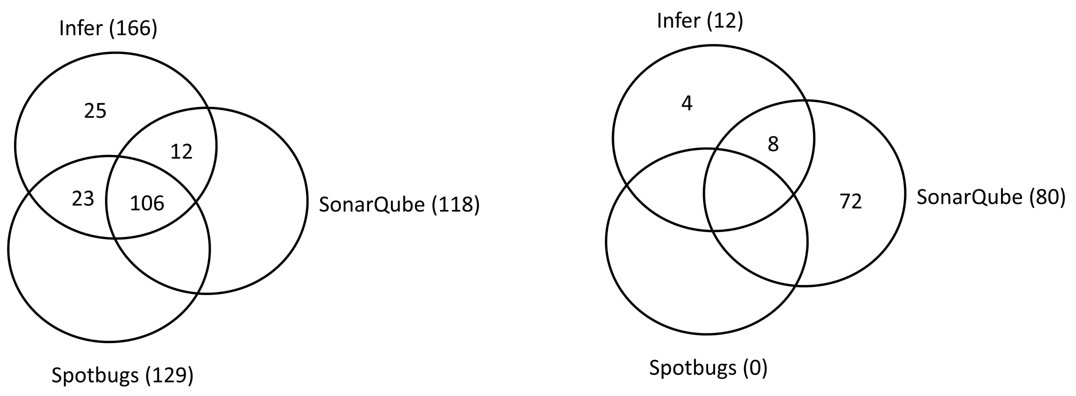

Figure 2: Shared and unique detections. Left: true positives; Right: false positives.

In SpotBugs, detected bugs have priorities and they can be shown at three levels: (i) *high*, (ii) *high & normal*, and (iii) *all* priorities. These priorities present confidence levels. They have been named as SpotBugs(h), SpotBugs(h&n), and SpotBugs(all) in Table 1. In our analysis below we rely on the SpotBugs(h&n) configuration because it gives the best results among the three options. The best result in this context means the one with a good balance of high true positives and low false positives.

We can see from Table 1 that currently, the best performing tool for this particular problem is Facebook Infer. This tool gave the highest number of true positives with a reasonably low number of false positives. The low rate of false positives in Infer is due to its ability to avoid infeasible control paths.

4.2. Shared, Unique and Missed Detections.

For a better understanding of how well the tools are performing in the detection of CWE476, it is important to further analyze the results of Section 4.1 with the help of a manual review, and to investigate the number and characteristics of detected test cases shared among some, or all, of the tools. It is also instructive to see those true and false positives which are detected by a single tool, and those cases which have been missed by all the tools (viz. *false negatives*). For more details see the author's GitHub account [12]).

Shared and Unique Detections. Figure 2 presents the number of shared and unique detections by the tools. The most important observations are the following:

1. There are 25 *positives* (15% of the 166 true positive detections, and 14% of all 181 *positives*) which were detected by Infer only, and not by the other tools. These test cases require data flow analysis among methods located in different classes and files (as an example, see Listing 1).
2. The three tools detected 106 *positives* in common (64% of the 166 true positive detections, and 59% of the 181 *positives*). These test cases of Juliet Test Suite are straightforward cases of null pointer dereference, and the dereference occurs in clearly feasible execution

paths. According to the terminology of the Juliet Test Suite, these test cases belong to the *baseline* and *control flow* category, and they require control flow and data flow analysis between different methods of the same class definition.

3. There is not a single tool that is able to detect all of the *positives*.
4. The tools have detected 92% of the *positives*. In particular, Infer was able to detect this 92% of *positives* itself, i.e. it could detect everything that other tools could detect (disregarding Spotbugs(all) here, as that configuration yields an unacceptable amount of *false positives*).
5. The tools have reported 92 false positives. Infer and SonarQube have 8 in common. These *negatives* rely on the value of a modifiable instance field; the tools behave conservatively by assuming that such a field might be changed somewhere. One can also observe that Spotbugs(h&n) performed extremely well in this respect by not giving any *false positives*.

Listing 1: An example for unique detections by Infer in Juliet Java 1.3

```java
public class CWE476_NULL_Pointer_Dereference__int_array_22a extends AbstractTestCase {
    /* The public static variable below is used to drive control flow in the sink function.
     * The public static variable mimics a global variable in the C/C++ language family. */
    public static boolean badPublicStatic = false;
    public void bad() throws Throwable {
        int [] data = null;
        data = null; /* POTENTIAL FLAW: data is null */
        badPublicStatic = true;
        (new CWE476_NULL_Pointer_Dereference__int_array_22b()).badSink(data );
    }
}

public class CWE476_NULL_Pointer_Dereference__int_array_22b {
    public void badSink(int [] data ) throws Throwable {
        if (CWE476_NULL_Pointer_Dereference__int_array_22a.badPublicStatic) {
            IO.writeLine("" + data.length); /* POTENTIAL FLAW: null dereference will occur
                if data is null */
        } else {
            /* INCIDENTAL: CWE 561 Dead Code, the code below will never run
             * but ensure data is initialized before the Sink to avoid compiler errors */
            data = null;
        }
    }
```

Missed Test Cases (False Negatives) We mentioned earlier that the tools of this study detected 92% of the *positives*. This means that only 8% of the *positives* was missed by Infer, and these have been missed by all the other tools as well. The missed test cases all rely on passing a data structure (an array, a Vector, or a LinkedList) storing a null value, or a serialized object having a field with a null value as a parameter to a method. The detection of vulnerabilities of this type may require built-in (lexical) knowledge about the standard library (the logic of data structures and the behavior of serialization), which – considering the volume of the Java standard library – is indeed a really demanding requirement.

Listing 2 presents a test case missed by Infer and the other two tools. The test case illustrates a vulnerability that involves more than one method. A static analyzer is expected to detect the bug either at line 10, or line 19, or both. Identifying this bug requires analysis that crosses method boundaries, which may be achieved by *global* analysis approaches. Infer can successfully detect such positive test cases in general. However, this test case requires maintaining (and passing around) information about an element of an array, which proved to be too sophisticated for Infer. Similar missed positives occur in Infer when an element of some other data structure is set to null.

Listing 2: An example of missed test cases in Juliet Java 1.3

```
1  import java.util.Vector;
2  public class CWE476_NULL_Pointer_Dereference__int_array_72a extends AbstractTestCase {
3      public void bad() throws Throwable {
4          int [] data;
5          data = null; /* POTENTIAL FLAW: data is null */
6          Vector<int []> dataVector = new Vector<int []>(5) ;
7          dataVector.add(0, data);
8          dataVector.add(1, data);
9          dataVector.add(2, data);
10         (new CWE476_NULL_Pointer_Dereference__int_array_72b()).badSink(dataVector );
11     }
12 }
13
14 import java.util.Vector;
15 public class CWE476_NULL_Pointer_Dereference__int_array_72b {
16     public void badSink(Vector<int []> dataVector ) throws Throwable {
17         int [] data = dataVector.remove(2);
18         /* POTENTIAL FLAW: null dereference will occur if data is null */
19         IO.writeLine("" + data.length);
20     }
21 }
```

4.3. Performance of the tools

The last step of our experiment described in Section 3 is to calculate the specified metrics. Figure 3 presents these for the three tools, characterizing their performance in the classification of the null pointer dereference test cases of the Juliet Test Suite. As mentioned in Section 4.1, in the case of SpotBugs, results of high & normal priority have been considered in this research (i.e., SpotBugs(h&n)).

The Recall values are shown in the first column. It represents how good the identification of the *positives* is, without taking into account the faulty reports of *negatives*. Higher recall indicates better performance. The findings revealed that SonarQube and SpotBugs had comparable recall values, whereas Infer had the best recall.

Column 2 displays the values for the false alarm rate. Smaller values here indicate better performance since the false alarm rate involves unflawed structures being mistakenly classified

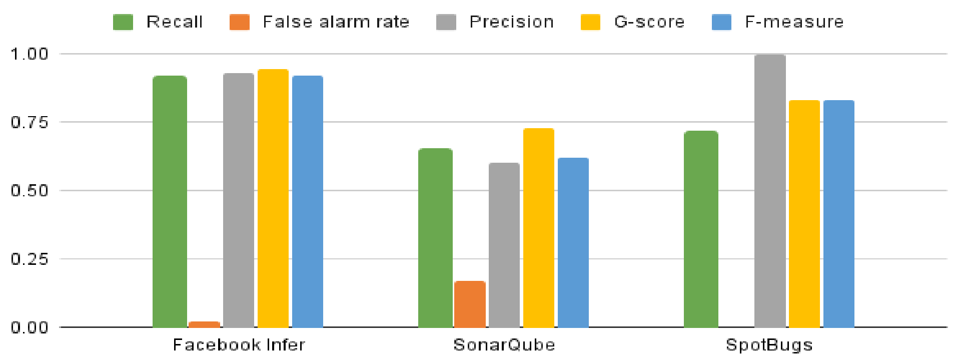

Figure 3: Calculated metrics for the tools.

as faulty. It can be observed from Figure 3 that SpotBugs had the lowest false alarm rate among the three tools, but Infer performed also quite well in this respect.

The third column in Figure 3 presents the precision of the three tools in identifying null pointer dereference. The precision measures how good the identification of faulty methods is compared to all the reports. Higher precision indicates better performance. SpotBugs has the highest performance for precision, followed closely by Infer.

The G-score and the F-measure enable us to combine two metrics into one, as discussed in Section 3. Recall & false alarm rate are combined for G-score and recall & precision are combined for F-measure. For both metrics, values close to 1 indicate that the tool can detect a specific weakness well, with no or very few false positives. This can be observed in the case of Infer (0.95 and 0.92, respectively). SpotBugs and SonarQube performed significantly worse: both G-score and F-measure for SpotBugs(h&n) are 0.83, while SonarQube yields 0.73 and 0.62, respectively.

5. Related Work

A variety of methodologies for evaluating static analysis tools are in use today. For benchmarking purposes, some researchers work with real code bases, while others rely on synthetic ones, or even both. Our study falls in the second category.

In a prior study [13], we compared the performance of four state-of-the-art static analysis tools (SonarQube, SpotBugs, Find Security Bugs, and PMD) in detecting six security vulnerabilities in Java source code, including *null pointer dereference*. The study found that SonarQube had the best aggregated performance and that all of the evaluated tools need to be improved to address better the highlighted weaknesses. The current research reveals the superiority of Facebook Infer for a very important vulnerability, null pointer dereference. We have excluded Find Security Bugs and PMD from this investigation because they provide no support for detecting null pointer dereference (although PMD can detect null pointer *assignments*).

Stephan Lipp and others [14] evaluated the vulnerability detection capabilities of six static C code analyzers, including Infer, against 192 real-world vulnerabilities in free and open-source programs. When applied to real-world software projects, the evaluated static analyzers were shown to be inefficient; most of the known vulnerabilities were missed. Our research suggests that Infer is fairly (while SpotBugs and SonarQube are moderately) efficient when analysing a particular weakness, null pointer dereference, on Java code. This may be due to a better analyzability of Java compared to C, or to the fact that we have carried out the experiment on a synthetic test suite.

Goseva-Popstojanova and Perhinschi [15] used 19 weakness categories, including null pointer dereference, to evaluate three tools with statistical methods. They used both some real-world programs and (a previous version of) the Juliet Test Suite (for Java and C/C++) to benchmark the tools. They also claim that the tools were ineffective at detecting security flaws. They measured approx. 0.48 and 0.64 as G-score for Juliet/Java/CWE476 using two (unspecified) tools, and their third tool was not able to detect CWE476 at all. Compared to our results, this suggests that the capabilities of static analysis tools improved significantly in the past 7 years.

Wouter Stikkelorum [11] provided a thorough assessment of Infer (version v0.8.1), which included benchmarking it against the Juliet test suite, running it against a large number of open-source projects, and against industrial code. According to his research, Infer performed well on industrial software, and the results are promising – which is in line with the opinion of other researchers [16, 17]. In particular, the results for the CWE476 category in the Java test cases of Juliet are quite similar to ours, although he used earlier versions of both Infer and Juliet: recall, precision and F-measure was measured 0.85, 0.95 and 0.90 in his experiment, and 0.92, 0.93 and 0.92. These values show a moderate increase in sensitivity, and a slight decrease in precision for Infer.

6. Conclusion

The results of this research revealed that Facebook Infer is quite reliable when used for identifying null pointer dereference in the Juliet Java test suite. We can also conclude that SpotBugs is fairly good in this vulnerability with the "high & normal" configuration, while SonarQube performed worst mostly due to the high number of false positives.

- 14% of the flawed constructs were detected only by Infer, and not the other tools.
- All the flawed constructs that have been detected by SonarQube and SpotBugs have also been detected by Infer, with a negligible number of false positives.
- SpotBugs is another very good static analysis tool, which, with well-chosen settings, gives highly accurate results.
- There is still room for improvement in static analysis tools.

Acknowledgments

Midya Alqaradaghi has been supported by the Stipendium Hungaricum program. The research of Tamás Kozsik has been supported by project no. TKP2021-NVA-29, which is implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development, and Innovation Fund, financed under the TKP2021-NVA funding scheme.

References

[1] B. Chess, J. West, Secure Programming with Static Analysis, Software Security Series, Addison-Wesley, 2007.

[2] D. Hovemeyer, J. Spacco, W. Pugh, Evaluating and tuning a static analysis to find null pointer bugs, SIGSOFT Softw. Eng. Notes 31 (2005) 13–19. doi:10.1145/1108768.1108798.

[3] The OWASP Foundation, Null Dereference, https://owasp.org/www-community/vulnerabilities/Null_Dereference, Accessed: July 2022.

[4] National Institute of Standards and Technology, Juliet Java 1.3, https://samate.nist.gov/SRD/test-suites/111, 2017.

[5] Facebook Infer, A tool to detect bugs in Java and C/C++/Objective-c code, https://fbinfer.com/., Accessed: July 2022.

[6] SonarQube, Code quality and code security, https://www.sonarqube.org/, Accessed: July 2022.

[7] SpotBugs, Find bugs in Java programs, https://spotbugs.github.io/, Accessed: July 2022.

[8] T. Hoare, Null references: The billion dollar mistake, Talk at QCon London, 2009.

[9] The MITRE Corporation, The Common Weakness Enumeration Initiative, https://cwe.mitre.org/, Accessed: July 2022.

[10] The MITRE Corporation, CWE476 Null Pointer Dereference, https://cwe.mitre.org/data/definitions/476.html, Accessed: July 2022.

[11] W. Stikkelorum, M. Bruntink, Challenges of using sound and complete static code analysis tools in industrial software, Master's thesis, University of Amsterdam, Faculty of Science, Mathematics and Computer Science, 2016.

[12] M. Alqaradaghi, Technical report of null pointer dereference analysis, https://github.com/Midya-ELTE/Technical_Report_of_NullPointerDereference_Analysis, Accessed: July 2022.

[13] M. Alqaradaghi, G. Morse, T. Kozsik, Detecting security vulnerabilities with static analysis – a case study, Pollack Periodica 17 (2021). doi:10.1556/606.2021.00454.

[14] S. Lipp, S. Banescu, A. Pretschner, An empirical study on the effectiveness of static C code analyzers for vulnerability detections, in: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), 2022, p. 12.

[15] K. Goseva-Popstojanova, A. Perhinschi, On the capability of static code analysis to detect security vulnerabilities, Information and Software Technology (2015). doi:10.1016/j.infsof.2015.08.002.

[16] D. Distefano, M. Fahndrich, F. Logozzo, P. W. O'Hearn, Scaling static analyses at Facebook, Communications of the ACM 62 (2019) 62–70. doi:10.1145/3338112.

[17] L. H. Newman, How Facebook catches bugs in its 100 million lines of code, Communications of the ACM (2019).