Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities

Jingyue Li*
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

Sindre Beba
Norwegian University of Science and Technology
Trondheim, Norway
sindrbeb@alumni.ntnu.no

Magnus Melseth Karlsen
Norwegian University of Science and Technology
Trondheim, Norway
magnumk@alumni.ntnu.no

ABSTRACT

Securing information systems has become a high priority as our reliance on them increases. Global multi-billion dollar companies have their critical information regularly exposed, costing them money and impairing their users' privacy. To defend against security breaches, IDE-integrated plugins to detect and remove security vulnerabilities in the first place are being used more frequently. More information about these plugins is needed in order to improve the state of the art within the field. Five open-source IDE plugins which can identify and report vulnerabilities are evaluated. We evaluate and compare how many categories of vulnerabilities the plugins can detect, how well the plugins detect the vulnerabilities, and how user-friendly the output of the plugin is to the developers. Our results show that certain vulnerabilities such as injection and broken access control are vastly covered by most plugins, while others have been completely ignored. A discrepancy between the claimed and actually confirmed coverage of the plugins is discovered, underlining the importance of this research. High false positive rate and obvious limitations in usability show that more work is needed before these plugins can be widely used and relied upon in a corporate setting.

CCS CONCEPTS

• **Security** → **Software security**;

KEYWORDS

Software security, static analysis tools, vulnerability detection, integrated development environment, plugin

ACM Reference Format:
Jingyue Li, Sindre Beba, and Magnus Melseth Karlsen. 2019. Evaluation of Open-Source IDE Plugins for Detecting Security Vulnerabilities. In *Evaluation and Assessment in Software Engineering (EASE '19), April 15–17, 2019, Copenhagen, Denmark.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3319008.3319011

*All authors contributed equally to this research.

EASE '19, April 15–17, 2019, Copenhagen, Denmark

ACM ISBN 978-1-4503-7145-2/19/04…$15.00
https://doi.org/10.1145/3319008.3319011

1 INTRODUCTION

As society gets more dependent on technology, the importance of securing information systems increases. A solution to reduce the number of security vulnerabilities is static analysis tools. With these tools included directly into the Integrated Development Environment (IDE) of the developer, they have never been easier to use. Unfortunately, not a lot of research has been made to compare and evaluate the existing IDE plugins, leaving the users to blindly accept the developer's claims of quality.

In this study, we aim to provide information about the state-of-the-art open-source IDE plugins. We report on their actual coverage of vulnerabilities, their performance, and their usability. This is achieved by evaluating them using a credible framework, i.e., Juliet Test Suite [12], for test cases and commonly used performance metrics, e.g., recall, precision, and discrimination rate. We select and evaluate five open-source plugins, namely ASIDE [14, 27, 28], ESVD [23, 24], LAPSE+ [2, 15], SpotBugs [8, 25], and Find Security Bugs (FindSecBugs) [6]. The plugins are evaluated on the most critical security vulnerabilities according to OWASP Top 10 [16], a reputable source for web application security. The evaluation looks for limitations of the existing plugins and is conducted with reproducibility in mind.

Our contribution comes in the form of two evaluations, i.e., a practical evaluation focused on coverage and performance as well as a theoretical evaluation of usability. Our results show that the evaluated plug-ins have clear limitations and a worrying discrepancy between what they claim and what they do. Coverage is focused around vulnerabilities connected to injection and access control, while other categories are left untouched. Several of the plugins also have a high false positive rate, which Christakis and Bird [4] identified as a leading cause of disuse. Many of the features, such as quick fixes, are not implemented adequately in the plugins. Insights we get from this study can help guide the open-source communities to improve their plugins further. For users of the plugins, our results can help them to understand better the strengths and weaknesses of the plugins they intend to use.

The rest of the paper is organized as follows. We introduce related studies in Section 2, and explain the design of the study in Section 3. Section 4 presents our evaluation results. We compare our results with related work in Section 5, and conclude our paper and discuss possible future work in Section 6.

2 RELATED WORK

Johnson et al. [9] looked at why static analysis tools, despite their benefits, are not widely used by developers. Through interviews with 20 software developers, they look at how current tools can

be improved. Results show that reasons for developers using static analysis tools are mainly because they automatically find bugs, they are already a part of the development environment, or for raising awareness of potential problems in a team setting. A reason for not wanting to use static analysis tools is poorly presented output. Another reason is large quantities of false positives, sometimes outweighing the true positives in volume.

Christakis and Bird [4] looked at what makes a static program analyzer attractive to developers through a broad survey of 375 Microsoft employees. They found that bad warning messages and slow speeds are reported pain points. It is interesting to note that while too many false positives is a largely reported pain point, too many false negatives is not. 90% of the participants are willing to accept a 5% false positive rate, while 47% of developers accept up to a 15% false positive rate. When forced to pick more bugs or fewer false positives, they typically choose fewer false positives. Developers also want the possibility of analyzing only part of the code, such as a file. Another feature that is often used and often requested is the possibility of suppressing warnings, preferably through code annotations.

Baset and Denning [1] collected information about available IDE plugins and compared them based on information provided by developers or manufacturers. They compared factors such as IDEs and languages supported, availability, scope of feedback, vulnerabilities covered and plugin uptake. The goal of their work was to synthesize the available information for future work within the field. In total, they gathered information about 17 different IDE plugins. Among them were both free and commercial tools as well as closed- and open-source. The vulnerability coverage comparison focused on nine input-validation related vulnerabilities. Of the nine vulnerabilities in question, only ESVD, FindBugs and LAPSE+ claimed to check for six or more. In addition, only ASIDE and Codepro AnalytiX provided *quick fixes* when reporting on vulnerabilities. Baset and Denning did not evaluate the plugins themselves, and all the information they collected came from other sources.

Oyetoyan et al. [19] looked into the capabilities of the freely available, open-source static analysis tools FindBugs, FindSecBugs, SonarQube, JLint, LAPSE+, and an undisclosed commercial tool. They tested the tools on all of the test cases in the Juliet Test Suite. Their results showed that FindSecBugs and LAPSE+ had the most true positives, and that FindSecBugs also had the highest recall and precision. FindSecBugs also had a good discrimination rate, while LAPSE+ was poor. The commercial tool ranked third of all the tools and had a poor precision. In addition to evaluating the efficiency, Oyetoyan et al. also interviewed six developers on their experience with using static analysis tools. They discovered developers had a generally positive attitude towards them, but that there were several barriers that stood in the way such as the need for multiple tools as well as poor performance.

There are multiple other related works. Charest et al. [3] compared the accuracy and precision of CodePro AnalytiX, JLint, FindBugs, and VisualCodeGrepper. The results were generally low, but Charest et al. argued that it was not too alarming as other studies had shown similar recall and precision for other tools, both open-source and commercial. Sadowski et al. [22] present a static analysis platform developed at Google and a philosophy on how such a platform should be created. Google's philosophy on program analysis is to have no false positives, allow the users to contribute with their own detections, reducing confusing tool output by accepting user feedback, and that the program should be analyzed while the user is changing or compiling the code.

3 RESEARCH DESIGN AND IMPLEMENTATION

Existing comparisons of vulnerability detection plugins lack vigorous testing on Java code for many of the free and open-source plugins. Some plugins claim to be developing new detection methods that would be superior to previous tools, but do not confirm this through testing afterwards. Not knowing what vulnerabilities each plugin covers, and its accuracy and performance when scanning for security vulnerabilities, can result in reduced usage or misuse of such tools. The focus on comparing existing free and open-source static analysis tools in regards to usability is non-existent. Considering the usability aspect of such tools plays a significant role in how many developers may choose to use these tools. The lacking amount of comparisons is problematic.

Our research aims to evaluate the vulnerability coverage, performance, and usability of current free and open-source IDE plugins utilizing static analysis on Java applications. For a large amount of free and open-source IDE plugins, such an evaluation does not exist today. We believe that an evaluation like this could bring useful information to developers without deep financial pockets, where buying expensive vulnerability scanners is too costly.

We formulate the following four criteria for selecting the plugins:

(1) The IDE plugins must detect security vulnerabilities, not just code bugs.
(2) The IDE plugins must be freely available and open-source.
(3) The IDE plugins must report detected vulnerabilities inside the IDE without the need for external software.
(4) The IDE plugins must detect vulnerabilities in Java code.

We want to evaluate plugins that are able to mitigate security vulnerabilities in code and can be used by both individuals, development teams, and large corporations. In order to lessen the work it is to adopt a static analysis tool, we want to contribute with information on how to improve an IDE plugin that requires minimal effort to install, learn, and use on a daily basis. This is the reasoning for the first three specifications listed above. To limit the study scope, we focus only on plugins analyzing Java code, as Java is the most used programming language according to TIOBE [26].

3.1 Research Questions

Based on the criteria presented, we formulate three research questions. For current open-source IDE plugins used to identify security vulnerabilities in Java using static code analysis:

RQ1 What is the coverage?
RQ2 How good is the performance?
RQ3 How good is the usability?

3.2 Plugins to be Tested and Test Cases

Based on information in [1], we decide upon the plugins in Table 1. All the plugins are available for the Eclipse IDE, and Eclipse is used for the evaluation. SpotBugs is not covered in [1], but its spiritual predecessor FindBugs is. We decide to focus on SpotBugs instead of FindBugs since SpotBugs is still actively worked on, while FindBugs has not been updated in years. FindSecBugs is not an IDE plugin in the same sense as the others. Instead, it is a plugin for SpotBugs (and FindBugs). That means it requires SpotBugs to run and its purpose is to expand the capabilities of SpotBugs further.

Plugins that are the result of academic work are especially interesting to us as they have more documentation and give more insights into their implementation. Both ASIDE and LAPSE+ are included due to this fact even though they are no longer being supported. Of the five plugins we have selected, only FindBugs (SpotBugs) frequently occurs in previous studies, such as [3], which compare static analysis tools. In other words, this evaluation will also provide new useful information about their coverage, performance, and usability.

Table 1: Information about the selected plugins.

IDE Plugin	Downloaded From	Version	Date
ASIDE	GitHub [28]	1.0.0	Feb 2013
ESVD	GitHub [23]	0.4.2	Jul 2016
LAPSE+	GitHub [2]	2.8.1	Jun 2013
SpotBugs	Eclipse Marketplace [25]	3.1.11	Jan 2019
FindSecBugs	Project Webpage [6]	1.8.0	Jun 2018

ASIDE is created with early detection in mind and with the goal to educate the user in secure programming. However, the static analysis tool does not offer any control- or data-flow analysis. Neither does LAPSE+, but it gives the user the opportunity to track variables manually through its provenance tracker. ESVD is inspired by ASIDE's focus on early detection and improves upon it by including data-flow analysis. It also claims to use inter-procedural analysis. SpotBugs and FindSecBugs analyze the Java bytecode instead of the raw source code like the others. Both of them utilize control-flow, data-flow, and inter-procedural analysis.

In order to test the IDE plugins in Table 1, we need test cases containing security vulnerabilities. Many comparisons of static analysis tools, such as the one by Rutar et al. [21], test the tools on the source code of actual software which can be referred to as *natural code*. As explained by NSA [13], this has both advantages and drawbacks. Two major issues are:

- Identifying false negatives. In natural code, it is often problematic to know how many vulnerabilities there are in the code. Without knowing this, it is impossible to calculate the number of false negatives.
- It is also problematic to know which types of security vulnerabilities are present. Thus, natural code cannot confidently prove which vulnerabilities an IDE plugin covers.

Delaitre et al. [5] also compared different types of test cases and which performance metrics they were applicable to test for. Table 2 shows the applicability of natural and artificial code as according to [5].

Table 2: Metric applicability for natural and artificial code according to Delaitre et al. [5].

Metric	Natural Code	Artificial Code
Coverage	Limited	Applicable
Recall	Not applicable	Applicable
Precision	Applicable	Applicable
Discrimination	Not applicable	Applicable

Given the purpose of our evaluation, it is apparent that using artificial code is beneficial to get the most accurate results. To choose the artificial code to compare the plugins, we consider four possible frameworks with vulnerabilities deliberately inserted: WebGoat [18], SecuriBench Micro [10], OWASP Benchmark [17], and Juliet Test Suite [12, 13]. We conclude that WebGoat is not suitable to test static analysis tools and that both SecuriBench Micro and OWASP Benchmark did not have test cases for enough vulnerabilities. Juliet Test Suite, on the other hand, fulfills both of these criteria.

Juliet Test Suite is a collection of intentionally vulnerable artificial code. Its purpose is to serve as a testing platform for static analysis tools. It consists of over 28,000 test cases which are categorized into 112 different CWE entries [12]. Each test case includes exactly one vulnerability as well as at least one non-flawed construct meant to represent a potential false positive. A single CWE entry can have thousands of test cases spanning over simple cases, control-flow cases and data-flow cases, where the later cases are more difficult to detect. All of this is intuitively incorporated in the naming of the test cases and its containing methods, which are further explained in its documentation [13]. It is still maintained and also used in previous research such as [3] and [19].

3.3 Measurement Metrics

We collect both quantitative and qualitative data to answer our research questions. For RQ1, we need to decide what *test case coverage* we want. Test case coverage is here defined as which security vulnerabilities we will test for. We decide to cover the vulnerabilities in OWASP Top 10 [16] because it is a reputable source for the most common and important web application vulnerabilities.

For RQ2, we use the performance metrics defined by Delaitre et al. [5]. The metrics are: 1) Recall, which is the percentage of vulnerabilities detected out of the total amount; 2) Precision, which is the percentage of alleged vulnerabilities the plugin reports that are indeed true vulnerabilities; 3) Discrimination Rate, which is the percentage of test cases the plugin discriminates. A plugin discriminates a vulnerability if it only reports a true positive on it, but no false positives.

$$\text{Discrimination Rate} = \frac{\text{Number of Discriminations}}{\text{Number of Test Cases}}$$

For RQ3, we decide to perform a qualitative evaluation. The usability metrics are taken from [4], [9], and [22]. In detail, we compare:

- **Tool Output**. Johnson et al. [9] and Sadowski et al. [22] found that poorly and confusing presented analysis output

was a largely reported problem when trying to figure out why software developers do not use static analysis tools. Developers want to know what the problem is, why it is a problem, and what should be done differently [9]. The tool output will be analyzed qualitatively. The satisfaction requirements are listed below, and must be present for a large portion of the vulnerabilities:

What is the problem? The output clearly states where the problem is in the form of line number and file name. The output also clearly states the vulnerability category of the problem.

Why is it a problem? The output clearly states why the reported vulnerability is a problem through text or an example. This must include the consequences of the vulnerability being in the code.

How to fix the problem? The output clearly states how to fix the problem through text or an example. Quick fixes also satisfy the requirement if all presented quick fixes are valid solutions to the problem, but not in the case where the tool does not try to make a distinction between relevant and irrelevant quick fixes.

- **False Positive Rate**. Christakis and Bird [4] found that over half the participants in their study do not accept a false positive rate above 15%. To present the false positive rate as clearly as possible, we will use two different methods to generate the percentage. The first method is calculating the false positive rate for each CWE, and then averaging the rate into what we call *averaged false positive rate*. The other method is adding all the true and false positives for each CWE into a total true positives and total false positives, and then calculating the false positive rate of the total, which we call *false positive rate of total result*.
- **Prioritized Output**. A high false positive rate might be counteracted by a prioritized output [7]. This metric will look at if the reported vulnerabilities are sorted by priority.
- **Quick Fixes**. Johnson et al. [9] and Sadowski et al. [22] discuss the need for quick fixes which can automatically fix the vulnerability. For each plugin, we investigate if quick fixes are presented to the user in a way that allows it to be automatically applied.
- **Early or Late Detection**. When looking at what the developers preferred, Johnson et al. [9] found that there was little agreement. Some developers preferred the tool to run in the background and immediately notify them of any bugs. Others preferred to integrate it with the compiler or sometimes later in the workflow. We look at whether the plugins utilize either early detection, late detection, or both.
- **Warning Suppression**. We check whether the plugin provides features to suppress the warnings. Both Johnson et al. [9] and Christakis and Bird [4] found that developers wanted the possibility of suppressing specific warnings, preferably through code annotation.
- **Environment Integration**. Developers are more likely to use the static analysis tool if it is already part of the development environment [9]. Sadowski et al. [22] found that the use of static analysis tools dropped when the developer was forced to run the analyzer as a stand-alone binary. This metric will be two binary results of whether the tool can integrate with the Eclipse IDE, and whether the plugin is available through the Eclipse Marketplace.
- **Immediate or Negotiated Interruptions**. Robertson et al. [20] describe the different styles of alerting mechanisms to investigate the impact of different interruption styles on the user. *Negotiated-style interruptions* are interruptions that inform the user of a pending alert without forcing them to acknowledge it at once. *Immediate-style interruptions* are alerts that immediately require the attention of users. Only negotiated-style interruptions were shown to have advantages [20].
- **Extendability**. The extendability metric will look at if the tools can be legally modified or extended. We will consider a tool to be legally modifiable if the source code is freely available, and under a software license where the code can freely and without cost be modified and redistributed. We will consider a tool easily extendable if it has a free API that allows any developer to modify and extend the original tool.
- **Granularity of Analysis**. When Christakis and Bird [4] asked to what level of granularity the developers would like to direct a program analyzer, file level (35%) or method level (46%) were chosen by the majority of participants. We will check if any of the tools support either file or method level granularity.

3.4 Research Implementation

Before conducting the evaluation, we need to assure ourselves that each IDE plugin is available and function properly. This proved not to be the case for ASIDE as the web page dedicated to the ASIDE project consists of mostly dead links to the plugin. This means we have to download the source code and build the plugin ourselves. We also have to change a couple of lines of code to correctly import external libraries as these are sets with an absolute path to the developer's computer. In addition, all plugins run in the newest version of Eclipse at the time, Eclipse Photon, except for LAPSE+ which we only manage to run in the older Eclipse Helios.

In order to automate the process of testing the IDE plugins, we create parsers that take the raw output data of the IDE plugins and transform it into the finished results of true positives, false positives, false negatives, and our performance metrics. This requires us to change the code that outputs the results of ASIDE, ESVD, and LAPSE+ as the information they provide is not adequate. We are particularly careful to only change the textual output from the plugins, without changing any logic that could alter the detection algorithms. The parsers are written in Python and with modularity in mind. The modified source code of the plugins and the source code of the parsers are available at https://github.com/Beba-and-Karlsen/. There are four different versions of the parsers, namely *aside.py*, *esvd.py*, *lapseplus.py* and *spotbugs.py*. As FindSecBugs is a plugin for

SpotBugs, they use the same version. All of these versions include the plugin-specific code for interpreting the bug reports. Then, they send the results to *plugincommon.py* that does the rest. In other words, they are similar in function, but with some implementation differences. They can all be explained as executing the following steps:

(1) **Read bug report** - The parser runs through the file containing the bug reports, extracting relevant information such as file name, CWE ID, vulnerability category, and in which test case it is detected.
(2) **Filter bug report** - The results are sent to *plugincommon.py* that checks whether each vulnerability is a true positive, false positive, or irrelevant. It then adds it to the respective list. Whether it is a true or false positive is based on the method it is detected in. All flawed methods in the Juliet Test Suite are named in a particular way. It is deemed relevant if the vulnerability category corresponds to the CWE entry of the test case. The output of the parsers is compared with the vulnerability categories provided by the plugin developers or documents.
(3) **Calculate test results** - With the filtering done, the parser then calculates the number of true positives, false positives, false negatives, recall, precision, and discrimination rate.
(4) **Print and log results** - In the end, the parser prints the results to screen and logs all results to a log file.

Juliet Test Suite v1.3 covers over 28,000 test cases spanning 112 different CWE entries. To run the plugins directly on the whole project would be very time-consuming and memory demanding. Because of this, we manually pick out the CWE entries we intend to test. Doing this manually is made easier by the fact that each test case in the Juliet Test Suite is categorized and sorted under its corresponding CWE entry. Selecting these CWE entries are based on the external references in OWASP Top 10 [16] and the CWE-1000 Research Concepts [11]. With the help of these resources, we are able to select the relevant CWE entries for each vulnerability category in OWASP Top 10.

However, not all CWE entries have a test case in the Juliet Test Suite. The 14 relevant CWE entries not included are listed in Table 3. In Table 5, all selected CWE entries that are included in the Juliet Test Suite are listed. In total, the plugins are tested on 8,675 test cases. As shown in Table 3, the Juliet Test Suite has no test cases for the CWE entries from categories A4 XML External Entities (XXE), A8 Insecure Deserialization, or A10 Insufficient Logging & Monitoring. In addition, no CWE entry at all matches A9 Using Components with Known Vulnerabilities. This is because this is a category that is more abstract, and is not a concrete weakness. This cannot be tested for by static analysis tools.

Table 3: The relevant CWE entries missing from the Juliet Test Suite.

OWASP Category	CWE Entries not in Juliet Test Suite	
	ID	Name
A1	564	Hibernate Injection
	917	Expression Language Injection
A2	384	Session Fixation
A3	220	Exposure of sens. info through data queries
	326	Weak Encryption
	359	Exposure of Private Information
A4	611	Improper Restriction of XXE
A5	284	Improper Access Control (Authorization)
	285	Improper Authorization
A6	2	Environmental Security Flaws
	16	Configuration
A8	502	Deserialization of Untrusted Data
A10	223	Omission of Security-relevant Information
	778	Insufficient Logging

4 RESEARCH RESULTS

4.1 RQ1: What is the coverage?

For RQ1, the coverage is defined by which and how many vulnerabilities the IDE plugin can detect. To answer what the coverage is of current plugins, we can look at both what they claim to detect and what our evaluation confirms they detect. We are looking at coverage for vulnerabilities in OWASP Top 10 included in the Juliet Test Suite. The results of our evaluation are shown in Table 4 and 5. The data in Table 5 show the number of true positives and false positives for each plugin divided into groups by CWEs. The hyphens (-) indicate that the CWE is not claimed to be covered by the plugin. The CWE is listed with its unique ID, name and its total amount of test cases. Based on the results of Table 5, we made a summary of the confirmed and claimed coverage for each plugin, which is shown in Table 4. The percentages correspond to the number of covered CWE entries covered by the plugins divided by the total number of all the vulnerabilities included in the Juliet Test Suite, which has 29 vulnerability categories.

Note that the results from FindSecBugs do not include the detections from SpotBugs, as we would like the coverage and performance of FindSecBugs to speak for itself. This allows us to evaluate the techniques of FindSecBugs independent from those of SpotBugs. In a typical use case, SpotBugs would by default be concurrently running when using the FindSecBugs extension.

Table 4: Confirmed and claimed coverage of the IDE plugins. Full coverage corresponds to covering all 29 vulnerability categories.

Tools	Confirmed Coverage		Claimed Coverage	
ASIDE	12	41%	12	41%
ESVD	5	17%	13	45%
LAPSE+	8	28%	11	38%
SpotBugs	8	28%	8	28%
FindSecBugs	18	62%	19	66%

The results in Table 4 and 5 show that the claimed coverage of the plugins is not great, with FindSecBugs being the only one with a claimed coverage higher than 50%. The confirmed coverage of our evaluation is even worse. Especially ESVD and LAPSE+ have several vulnerabilities they claim to cover, where our evaluation has shown that they in fact do not. It is worth noting that this could be due to the fact that the plugin does not detect the particular

Table 5: Detailed coverage data, showing the number of true and false positives. A hyphen (-) indicates that the plugin does not cover the CWE. The CWE is listed with its unique ID and its total number of test cases.

CWE			IDE-Integrated Static Analysis Tools									
ID	Name		ASIDE		ESVD		LAPSE+		SpotBugs		FindSecBugs	
A1 Injection		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
78	OS Command Injection	444	185	0	49	0	444	624	-	-	378	50
89	SQL Injection	2220	3[a]	3[a]	1440	2280	2220	3060	2220	3000	1900	300
90	LDAP Injection	444	185	0	0	0	0	0	-	-	379	50
113	HTTP Response Splitting	1332	555	795	0	0	0	0	57	0	989	0
134	Use of Externally-Controlled Format String	666	148	212	-	-	-	-	-	-	462	0
643	Xpath Injection	444	185	265	0	0	444	1248	-	-	379	49
A2 Broken Authentication		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
256	Unprotected Storage of Credentials	37	-	-	-	-	-	-	-	-	-	-
259	Use of Hard-coded Password	111	-	-	-	-	-	-	15	0	48	0
321	Use of Hard-coded Cryptographic Key	37	-	-	-	-	-	-	-	-	16	0
523	Unprotected Transport of Credentials	17	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	17	-	-	-	-	-	-	-	-	-	-
A3 Sensitive Data Exposure		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
315	Cleartext Storage of Sensitive Information in a Cookie	37	-	-	-	-	-	-	-	-	0	0
319	Cleartext Transmission of Sensitive Information	370	-	-	-	-	-	-	-	-	259	369
325	Missing Required Cryptographic Step	34	-	-	-	-	-	-	-	-	-	-
327	Use of a Broken or Risky Cryptographic Algorithm	34	-	-	-	-	-	-	-	-	17	0
328	Reversible One-Way Hash	51	-	-	-	-	-	-	-	-	51	0
329	Not Using a Random IV with CBC Mode	17	-	-	-	-	-	-	-	-	17	0
614	Sensitive Cookie in HTTPS Session Without 'Secure' Attribute	17	-	-	-	-	-	-	-	-	16	0
759	Use of a One-Way Hash without a Salt	17	-	-	-	-	-	-	-	-	-	-
760	Use of a One-Way Hash with a Predictable Salt	17	-	-	-	-	-	-	-	-	-	-
A5 Broken Access Control		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
23	Relative Path Traversal	444	108	0	0	0	444	624	19	0	378	52
36	Absolute Path Traversal	444	108	0	0	0	444	624	16	0	378	49
566	Auth. Bypass Through User-Controlled SQL Primary Key	37	36	0	-	-	37	0	-	-	-	-
A6 Security Misconfiguration		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
395	NullPointerException Catch to Detect NULL Pointer Deference	17	-	-	0	0	-	-	-	-	-	-
396	Declaration of Catch for Generic Exception	34	-	-	0	0	-	-	-	-	-	-
397	Declaration of Throws for Generic Exception	4	-	-	0	0	-	-	-	-	-	-
A7 Cross-Site Scripting		**Total**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
80	Basic XSS	666	642	900	28	0	666	936	19	0	666	76
81	Improper Neutralization of Script in an Error Message	333	321	450	14	0	0	0	19	0	333	38
83	Improper Neutralization of Script in Attributes in a Web Page	333	108	0	14	0	333	468	19	0	333	38

[a] ASIDE generates an exception when running on these test cases.

case that Juliet Test Suite has implemented while detecting others. However, it is still a flaw in the plugin implementation. We find this discrepancy rather alarming. It gives a false sense of security to the user and discredits the integrity of the plugins. It is also worth noting that the distribution of coverage over the different categories is uneven. Injection vulnerabilities, broken access control, and cross-site scripting are heavily covered, while the others are less represented in the plugins' coverage.

4.2 RQ2: How good is the performance?

The calculations of recall, precision, and discrimination rates are based on the results of true positives and false positives shown in Table 5. The calculated performance metrics are presented in Table 6. There are two important notes about the evaluation that may have affected the results:

(1) Because of imprecise detection classification, ASIDE's true positives might be higher than what it actually deserves. It uses only two vulnerability categories which are very general, *input validation vulnerability* and *output encoding vulnerability*. This means we cannot implement proper relevant category checking for ASIDE. In other words, we cannot be confident whether reported vulnerabilities by ASIDE are relevant or not.

(2) LAPSE+ is supposed to be used with a substantial amount of manual effort to complete backward propagation on each result, which cannot be automated. It reports both sources and sinks and requires the user to check whether the data was sanitized between these points. Our results are based on only the automatic results without the manual effort after. This might be the cause of the high amount of false positives LAPSE+ reports.

Table 6: Detailed performance metrics data, showing recall, precision, and discrimination rate. CWE names are slightly shortened, see Table 5 for the full names. A hyphen (-) indicates that the plugin does not cover the CWE.

CWE		Tools														
ID	Name	ASIDE			ESVD			LAPSE+			SpotBugs			FindSecBugs		
A1	**Injection**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
78	OS Command Injection	42%	100%	42%	11%	100%	11%	100%	42%	0%	-	-	-	85%	88%	74%
89	SQL Injection	0%	50%	0%	65%	39%	0%	100%	42%	0%	100%	43%	0%	86%	86%	72%
90	LDAP Injection	42%	100%	42%	0%	N/A	N/A	0%	N/A	N/A	-	-	-	85%	88%	74%
113	HTTP Response Splitting	42%	41%	0%	0%	N/A	N/A	0%	N/A	N/A	4%	100%	4%	74%	100%	74%
134	Externally-Controlled Format String	22%	41%	0%	-	-	-	-	-	-	-	-	-	69%	100%	69%
643	Xpath Injection	42%	41%	0%	0%	N/A	N/A	100%	26%	0%	-	-	-	85%	89%	74%
A2	**Broken Authentication**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
256	Unprotected Credentials Storage	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
259	Hard-coded Password	-	-	-	-	-	-	-	-	-	14%	100%	14%	43%	100%	43%
321	Hard-coded Cryptographic Key	-	-	-	-	-	-	-	-	-	-	-	-	43%	100%	43%
523	Unprotected Credentials Transport	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
549	Missing Password Field Masking	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A3	**Sensitive Data Exposure**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
315	Cleartext Sensitive Info in Cookie	-	-	-	-	-	-	-	-	-	-	-	-	0%	N/A	N/A
319	Sensitive Cleartext Transmission	-	-	-	-	-	-	-	-	-	-	-	-	70%	41%	0%
325	Missing Required Crypto. Step	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
327	Broken/Risky Crypto. Alg.	-	-	-	-	-	-	-	-	-	-	-	-	50%	100%	50%
328	Reversible One-Way Hash	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
329	Not Random IV in CBC Mode	-	-	-	-	-	-	-	-	-	-	-	-	100%	100%	100%
614	Missing 'Secure' in HTTPS Cookie	-	-	-	-	-	-	-	-	-	-	-	-	94%	100%	94%
759	One-Way Hash, no Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
760	One-Way Hash, Predictable Salt	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
A5	**Broken Access Control**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
23	Relative Path Traversal	24%	100%	24%	0%	N/A	N/A	100%	42%	0%	4%	100%	4%	85%	88%	74%
36	Absolute Path Traversal	24%	100%	24%	0%	N/A	N/A	100%	42%	0%	4%	100%	4%	85%	89%	74%
566	SQL PK Auth. Bypass	97%	100%	97%	-	-	-	100%	100%	100%	-	-	-	-	-	-
A6	**Security Misconfiguration**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
395	Catching NULL Pointer Deference	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
396	Catch for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
397	Throws for Generic Exception	-	-	-	0%	N/A	N/A	-	-	-	-	-	-	-	-	-
A7	**Cross-Site Scripting**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**	**Rec.**	**Pre.**	**Disc.**
80	Basic XSS	96%	42%	2%	4%	100%	4%	100%	42%	0%	3%	100%	3%	100%	90%	89%
81	Script in Error Message	32%	100%	32%	4%	100%	4%	0%	N/A	N/A	6%	100%	6%	100%	90%	89%
83	Script in Attributes in a Web Page	96%	42%	2%	4%	100%	4%	100%	42%	0%	6%	100%	6%	100%	90%	89%

The calculation of **recall** shows that both ESVD and SpotBugs have poor results when it comes to recall with their respective highest score except for CWE-89 SQL Injection being 65% and 100%. The low scores mean ESVD and SpotBugs cannot reliably find all vulnerabilities in a piece of code, which is a severe limitation. ASIDE has generally low scores as well, but with some exceptions where it is close to full score. LAPSE+ reports all vulnerabilities for a number of CWE entries, but with some exceptions where it reports none. This binary behavior of LAPSE+ may point to it generally performing very well, but it might not cover all that it claims. FindSecBugs, which had the highest coverage, also show very good results, with most CWE entries over 50% and some with full score.

Contrary to their poor results for recall, ESVD and SpotBugs have a high **precision**. This might be due to their implementation as static analysis tools which always have to deal with a trade-off between recall and precision. Another example of this trade-off is LAPSE+ which has a great recall, but has very low precision. FindSecBugs has a precision over 85% for all but one.

Discrimination rate combines recall and precision and will always be a number between zero and the respective recall. Due to this, ESVD, LAPSE+, and SpotBugs all have a low discrimination rate. ESVD and SpotBugs' low discrimination rates are due to their low recall, and LAPSE+ because of its poor precision. ASIDE is not convincing either and only does well on CWE-566, i.e., Authorization Bypass Through User-Controlled SQL Primary Key. FindSecBugs, which has both a good recall and precision, also has a high discrimination rate and proves to be the best solution of these five IDE plugins.

The performance results are varying with clear signs of trade-offs between recall and precision. ESVD and SpotBugs prioritize precision, while LAPSE+ prioritizes recall. This leads to a low discrimination rate for all of the mentioned plugins. Only FindSecBugs has good results for all three performance metrics.

4.3 RQ3: How good is the usability?

All the plugins have one or more CWEs where the number of false positives far outnumbers the number of true positives. As can be seen in Table 7, some tools produce a surprising amount of false

Table 7: Summary of the usability results.

		ASIDE	ESVD	LAPSE+	SpotBugs	FindSecBugs
FP rate	Averaged false positive rate	29%	12%	53%	7%	9%
	False positive rate of total result	50%	60%	60%	56%	13%
Detailed information	What is the problem	×	✓	✓	✓	✓
	Why is it a problem	N/A	×	×	✓	✓
	How to fix the problem	N/A	×	×	×	✓
	Prioritized output	×	✓	×	✓	✓
	Quick fixes	✓	✓	×	×	×
	(E)arly or (L)ate detection	E	E	L	E/L	E/L
	Can suppress warnings	✓	✓	×	×	×
	Eclipse Environment integration	✓	✓	✓	✓	✓
	Available on Eclipse Marketplace	×	×	×	✓	×
	(I)mmediate or (N)egotiated interruptions	N	N	N	N	N
	Easily extendable	×	×	×	✓	×
	Possible to analyze single file only	×	×	×	✓	✓
	Possible to analyze single method only	×	×	×	×	×

positives. Looking closer at the false positive rates of individual CWE categories, all tools have some cases where the rate is close to 60%. The highest false positive rate by all is the one produced by LAPSE+ for the CWE-643 vulnerability class, which is 74%. These numbers are not within the acceptable range of false positives. These high rates can lead to developers not wanting to use such tools in their work. It is important to notice that some tools' false positive rate is generally low except for a few CWEs. SpotBugs has a false positive rate of zero percent with the exception of CWE-89, where it has a false positive rate of 58%. This also shows the importance of the two different measurements of a false positive rate presented in this paper, where one of them tries to balance out this effect by also averaging the false positive rate of each CWE.

The quality of the detection output varies. ASIDE gives no information, and when trying to get more information, it opens a web page where the domain no longer exists. ESVD and LAPSE+ provides a description of what the problem is, but never explains why it is a problem nor how to fix it. SpotBugs and FindSecBugs clearly tell the user why the detected vulnerability is a problem, and FindSecBugs also provides examples of how such vulnerabilities can be fixed.

ESVD, SpotBugs, and FindSecBugs all allow their output to be sorted by priority. ESVD ranks each detected vulnerability based on a number. SpotBugs and FindSecBugs use words, symbols, and colors to show the vulnerability priority. They use words like "scary" with a red bug icon for high priority, and "troubling" with a yellow bug icon next to it for medium priority.

The only two tools that provide quick fixes are ASIDE and ESVD. Both of these give the option of multiple quick fixes, where some of them are not relevant to the current vulnerability at all. Examples of these are "HTML Encoder", "JavaScript Encoder", and "CSS Encoder". Both ASIDE and ESVD seem to produce the exact same quick fixes. These are very simple quick fixes which surround the code with methods from the OWASP Enterprise Security API (ESAPI).

ASIDE and ESVD utilizes early detection by continuously monitoring the workspace for changes. The static code analysis is executed incrementally on small parts of the code while it is being written to ensure quick feedback to the developers. LAPSE+ is using late detection by definition, as it does not automatically scan code. The user has to execute the vulnerability search manually. LAPSE+ will scan each file as if it was the first time, without remembering previous results. SpotBugs, and therefore also the SpotBugs plugin FindSecBugs, utilizes early detection by allowing the user to scan files when they are saved automatically. This can also be turned off so that scans have to be manually executed. That means that SpotBugs supports both early and late detection. SpotBugs can also be integrated into other tools than Eclipse, so that such scans can happen very late if the developer wants that, e.g., right before committing the code to the code repository.

ASIDE and ESVD are the only two tools that can suppress warnings in our evaluation. The result is that the vulnerability warning disappears, with no way of getting it back. This might be a negative experience for the developer if the vulnerability is suppressed by accident.

All of the static analysis tools integrates into the Eclipse IDE. However, the effort to integrate the tools into Eclipse varies. ASIDE is neither in the Eclipse Marketplace nor has any executable ready to be installed in Eclipse. It had to be compiled from source code. ESVD is available on the Eclipse Marketplace, but an error prevents us from downloading it. It seems that the Marketplace is attempting to download the executable from one of the ESVD authors' webpage without success. Therefore the plugin needs to be manually installed into Eclipse. LAPSE+ is not available through the Eclipse Marketplace either, but has an executable that can be downloaded and manually installed. SpotBugs is easily installed through Eclipse Marketplace, i.e., a few clicks are all that is needed. Adding the SpotBugs plugin FindSecBugs requires the user to download the executable from the developer's website. It is then added through the settings of the SpotBugs plugin.

Every tool uses negotiated-style interruptions. None of the tools gives the option of immediate-style interruptions.

All the tools are open-source. This makes it possible for anyone to look at the code. All of the software licenses allow for modification and redistribution of the code. Although all the tools can legally be changed and redistributed, it is difficult to change for most of them. SpotBugs is the only tool that allows other plugins to directly extend the functionality of itself. This can be done through the public SpotBugs API.

SpotBugs and FindSecBugs are the only two tools that allow developers to analyze a single file at a time. ASIDE, ESVD, and LAPSE+ only allow the user to scan the entire project. A finer granularity than file-level analysis is not supported by any of the plugins. Table 7 shows a summary of the usability evaluation results.

5 DISCUSSION

5.1 Comparison with Related Work

The discrepancy we found between claimed and confirmed coverage proves why it is important to test the capabilities of the plugins ourselves and not rely solely on the information provided by the developer. A possible horror situation could be a company which uses an IDE plugin to detect vulnerabilities in their developed code. Being confident in the capabilities of the plugin, they believe they ship software without any severe security vulnerability. However, the plugin does not detect all of the vulnerabilities it claims and the software is shipped with vulnerabilities that cause a security breach and cost the company a lot of money.

Baset and Denning [1] compare the plugins at a purely informational level and use the claimed coverage. Our research contributes with useful and new information about the actual coverage of the plugins that can assist the users in knowing which plugin to use and what to expect from it. While Charest et al. [3] look at coverage for the plugins they compared, they only look for coverage on four different vulnerabilities. In contrast, we look at 29 different CWE entries. Oyetoyan et al. [19] look at all of the 112 CWE entries in the Juliet Test suite, but they did not report the results per CWE entry. Instead, they aggregated the results into categories making it impossible to know which plugin covers which vulnerability. In addition, we tested plugins that have not been covered a lot in previous research. This makes the results of ASIDE, ESVD, LAPSE+, and SpotBugs especially interesting as it is a new contribution to the field. Oyetoyan et al. [19] perform their comparison on all of the 112 CWE entries in the Juliet Test Suite. While it gives their research a wider approach and provides more information, our narrower approach also has an advantage. By narrowing our evaluation down to only vulnerabilities found in OWASP Top 10, we make sure that all of the results are relevant. We only test on vulnerabilities that are considered more important. By testing on all test cases, Oyetoyan et al. [19] may open up for uncommon vulnerabilities to skew their final results. A plugin that does well for uncommon vulnerabilities, but cannot detect common ones, may not be a useful plugin even though it might be overall performing well.

By utilizing the performance metrics used in [5], we generate results that can be compared with other research as we consider these performance metrics the closest to an industry standard. The test cases of Juliet Test Suite are created by the National Security Agency (NSA) and the National Institute of Standards and Technology (NIST) which are trustworthy sources and we believe this enhances the credibility of our results. However, Juliet Test Suite consists of only artificial code. This is not necessarily a negative thing, but it does limit our research. The results indicate how the plugins perform on generated code which is a good indication of what it detects objectively, but it does not say anything about their performance in a real-life setting. Detecting vulnerabilities in natural code is a different matter and the distribution of occurrences by vulnerabilities are very different.

No existing papers are evaluating the usability of multiple static analysis plugins related to detecting security vulnerabilities in the way we did. Some of the plugins have themselves conducted usability evaluations of their tools. Xie et al. [27] conducted a usability evaluation of ASIDE with nine students, where each student used three hours to write code using the plugin. The ASIDE usability evaluation was aimed at evaluating functionality surrounding the plugin itself, without comparisons to other similar tools. Sampaio and Garcia [24] have also evaluated some usability aspects of their own plugin. They conducted an experiment looking at the difference early versus late detection does to a developer's motivation to address reported vulnerabilities. The experiment does not compare ESVD to any other similar tools. Christakis and Bird [4] look at what developers want and need from static analysis tools. Johnson et al. [9] look at why the number of developers using static analysis tools is so low. Sadowski et al. [22] present the guiding philosophy of Google regarding how static analysis tools should behave usability wise. Studies [4], [9], and [22] have all been used as a basis for our usability evaluation, but no direct comparison can be made, because these papers do not compare the usability of different static analysis tools. Given the lack of similar work in existing literature, our evaluation brings forth new information which developers will find useful when deciding which static analysis tool to choose for their own project. In addition, our evaluation gives insights into what existing static analysis tools for detecting security vulnerabilities offer, which in turn can help new tools develop features that existing tools do not currently have.

5.2 Threats to Validity

The main threat to internal validity is selection bias. The selection bias comes from which plugins, vulnerabilities, and metrics that were chosen. To reduce selection bias threat, we have done extensive research into each one in order to make the most sensible decision. We have looked at highly-cited literature and based our decisions on credible sources. Another threat to internal validity is experimenter bias related to the qualitative data analysis of some of the usability metrics. To reduce experimenter bias, the satisfaction requirements are taken from relevant literature regarding the usability of similar tools.

The main threat to external validity is generalizability. The most popular free and open-source static analysis tools were chosen for this evaluation. The limited number of tools that fit our selection criteria make these five tools representative. However, it is not possible to generalize the results to commercial static analysis tools as these are in a completely different category regarding development and research funding.

6 CONCLUSION AND FUTURE WORK

One possible approach to reduce software vulnerabilities is through IDE plugins which alert the developer whenever vulnerable code is written. This allows the vulnerability to be removed at once. There are several open-source vulnerability detection plugins available today. This study presents a coverage, performance, and usability evaluation of five plugins on vulnerability test cases from artificial code. The results of the study show that there are still many categories of vulnerabilities that are not covered by any of the plugins we evaluated. The coverage information published in the plugins' documentation may be misleading. Most plugins have a high false positive rate and are not user-friendly for developers.

To improve the plugins, we can focus on improving all those three aspects evaluated in this study. We believe that improving the coverage of the vulnerabilities, improving performance, and making the plugins more user-friendly will all contribute to more and better use of the plugins and will therefore reduce the number of vulnerabilities in the software code.

REFERENCES

[1] Aniqua Z. Baset and Tamara Denning. 2017. IDE Plugins for Detecting Input-Validation Vulnerabilities. In *2017 IEEE Security and Privacy Workshops (SPW)*. 143–146. https://doi.org/10.1109/SPW.2017.37

[2] Bernhard J. Berger. 2013. lapse-plus. (2013). https://github.com/bergerbd/lapse-plus/

[3] Thomas Charest, Nick Rodgers, and Yan Wu. 2016. Comparison of Static Analysis Tools for Java Using the Juliet Test Suite. In *Proceedings of the 11th International Conference on Cyber Warfare and Security, ICCWS 2016*. 431–438.

[4] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, New York, New York, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[5] Aurelien Delaitre, Bertrand Stivalet, Paul E. Black, Vadim Okun, Athos Ribeiro, and Terry S. Cohen. 2018. *SATE V Report: Ten Years of Static Analysis Tool Expositions*. Technical Report. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.500-326

[6] Find Security Bugs. 2018. Find Security Bugs - The SpotBugs plugin for security audits of Java web applications. (2018). https://find-sec-bugs.github.io

[7] Sarah Heckman and Laurie Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '08*. ACM Press, New York, New York, USA, 41. https://doi.org/10.1145/1414004.1414013

[8] David Hovemeyer and William Pugh. 2004. Finding Bugs is Easy. *SIGPLAN Not.* 39, 12 (2004), 92–106. https://doi.org/10.1145/1052883.1052895

[9] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, San Francisco, CA, USA, 672–681. https://dl.acm.org/citation.cfm?id=2486877

[10] Benjamin Livshits. 2006. Stanford SecuriBench Micro. (2006). https://suif.stanford.edu/

[11] MITRE. 2018. CWE VIEW: Research Concepts. (2018). https://cwe.mitre.org/data/definitions/1000.html

[12] NIST. 2017. Test Suites. (2017). https://samate.nist.gov/SRD/testsuite.php

[13] NSA. 2012. Juliet Test Suite v1.2 for Java User Guide. (2012).

[14] OWASP. 2016. OWASP ASIDE Project. (2016). https://www.owasp.org/index.php/OWASP_ASIDE_Project

[15] OWASP. 2017. OWASP LAPSE Project. (2017). https://www.owasp.org/index.php/OWASP_LAPSE_Project

[16] OWASP. 2017. OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks. (2017).

[17] OWASP. 2018. OWASP Benchmark Project. (2018). https://www.owasp.org/index.php/Benchmark

[18] OWASP. 2018. OWASP WebGoat Project. (2018). https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

[19] Tosin D. Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela S. Cruzes. 2018. Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital. In *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 86–103. https://doi.org/10.1007/978-3-319-91602-6_6

[20] T. J. Robertson, Shrinu Prabhakararao, Margaret Burnett, Curtis Cook, Joseph R. Ruthruff, Laura Beckwith, and Amit Phalgune. 2004. Impact of interruption style on end-user debugging. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, Vol. 6. ACM Press, New York, New York, USA, 287–294. https://doi.org/10.1145/985692.985729

[21] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. *15th International Symposium on Software Reliability Engineering* (2004), 245–256. https://doi.org/10.1109/ISSRE.2004.1

[22] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608. https://doi.org/10.1109/ICSE.2015.76

[23] Luciano Sampaio. 2016. TCM_Plugin. (2016). https://github.com/lsampaioweb/TCM_Plugin

[24] Luciano Sampaio and Alessandro Garcia. 2016. Exploring context-sensitive data flow analysis for early vulnerability detection. *Journal of Systems and Software* 113 (2016), 337–361. https://doi.org/10.1016/j.jss.2015.12.021

[25] SpotBugs Team. 2018. SpotBugs Eclipse plugin. (2018). https://marketplace.eclipse.org/content/spotbugs-eclipse-plugin

[26] TIOBE. 2018. TIOBE Index for November 2018. (2018). https://www.tiobe.com/tiobe-index/

[27] Jing Xie, Bill Chu, Heather R. Lipford, and John T. Melton. 2011. ASIDE: IDE Support for Web Application Security. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC '11)*. ACM, New York, NY, USA, 267–276. https://doi.org/10.1145/2076732.2076770

[28] Jun Zhu. 2013. ASIDE-Education. (2013). https://github.com/JunZhuSecurity/ASIDE-Education