Critical Understanding of Security Vulnerability Detection Plugin Evaluation Reports

1st Sindre Beba
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
sindrbeb@alumni.ntnu.no

2nd Magnus Melseth Karlsen
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
magnumk@alumni.ntnu.no

3rd Jingyue Li
Department of Computer Science
Norwegian University of Science and Technology
Trondheim, Norway
jingyue.li@ntnu.no

4th Bing Zhang
School of Information Science and Engineering
Yanshan University
Qinhuangdao, China
bingzhang@ysu.edu.cn

***Abstract*—Integrated development environment (IDE) plugins aimed at detecting web application security vulnerabilities can help developers create secure applications in the first place. Most of such IDE plugins use static source code analysis approaches. Although several empirical studies evaluated the plugins and compared their precision and recall of detecting web application security, few follow-up studies tried to understand the evaluation results. We analyzed more than 20,000 vulnerability reports based on 7,215 distinct test cases spanning 11 categories of web application vulnerabilities to understand the evaluation results of three open-source IDE plugins, namely, SpotBugs, FindSecBugs, and Early Security Vulnerability Detector (ESVD), which aimed at detecting security vulnerabilities of Java-based web applications. Our results identify many factors besides the source code analysis approach that can dramatically bias the detection performance. Based on our insights, we improved the studied plugins. In addition, our study raises the alarm that, without solid root cause analyses, the evaluation and comparisons of security vulnerability detection approaches and tools could be misleading. Thus, we proposed a guideline on reporting the evaluation results of the security vulnerability detection approaches.**

***Index Terms*—Software security, Source code analysis, Vulnerability detection, Empirical study, Root cause analysis**

I. Introduction

As society grows more dependent on secure web applications, it is critical to reduce the number of security vulnerabilities. Using plugins in an Integrated Development Environment (IDE) to detect and inform developers about possible security issues in the code they are writing could be an efficient solution to help developers create secure code in the first place. IDE plugins have been developed for this purpose. Studies [1] [2] have empirically evaluated and compared how accurately IDE plugins, such as Application Security in IDE (ASIDE) [3]–[5], ESVD [6], LAPSE+ [7], [8], SpotBugs [9], [10], and Find Security Bugs (FindSecBugs) [11], can identify OWASP [12] vulnerabilities. [1] found that ESVD's precision in identifying SQL injection vulnerability is only 39%, and FindSecBugs' recall in identifying hard-coded passwords is only 43%. A high false-negative rate among these plugins will mislead the developers to interpret an insecure web application as a secure one, and a high false-positive rate will discourage developers from using the plugins [13]. Thus, it is critical to identify the reasons for the plugins' false negatives and false positives and to improve them to detect vulnerabilities precisely. We selected the three plugins evaluated in [1], i.e., ESVD, SpotBugs, and FindSecBugs, which are still actively maintained and evolved by the community, and manually examined each of their false-negative and false-positive results to understand why the plugins perform as they did in [1]. We performed the root cause analysis by first hypothesizing the possible reasons for the false results. Based on the hypothesis, we implemented fixes. If the fixes eliminated the false results, our hypotheses of the root causes of the false results are confirmed. The studies identified design and implementation weaknesses of the plugins, such as missing sources and sinks in taint analysis and incomplete data-flow and control-flow analysis. More importantly, we found that many other factors, which are not necessarily relevant to the detection approach and algorithm, can significantly bias the recall and precision of the evaluation. The factors include systematically biased test cases, missing test cases, inconsistent confidence ranking of the detected vulnerabilities, misclassification of the detected vulnerabilities, and missing reports of the vulnerabilities in the same line of code. To our knowledge, no similar studies have analyzed the root causes of so many (i.e., more than 20,000) vulnerability reports of vulnerability detection tools. The contributions of our study are threefold:

- First, we provide insights on issues and solutions of implementing taint analysis-based security vulnerability detection tools. Some of the insights can also be generalized to other types of static source code analysis-based vulnerability detectors.
- A more significant contribution is that our deep under-

standings of evaluation reports identified several novel bias factors of evaluating vulnerability detectors. Thus, we propose an evaluation guideline and encourage researchers to perform and report more thorough empirical evaluations of the vulnerability detectors. The guideline can also help the industrial practitioners who want to use these detectors to read and understand the evaluation reports more wisely.
- In addition, we identified weaknesses of the studied plugins and improved their precision and recall significantly.

The rest of the paper is organized as follows: We introduce the study design in Section 2, and Section 3 presents the study results. Section 4 presents the related work. We discuss our results in Section 5 and conclude and propose future work in Section 6.

II. Study design

Many studies use static code analysis [14] in general and taint analysis [15] in particular to detect web application vulnerabilities. IDE plugins, such as [16], use static code analysis approaches to help developers identify vulnerabilities when writing their code. Study [1] evaluated five open-source IDE plugins, which use static source code analysis approaches to detect vulnerabilities. The results of [1] show that the plugins have much higher false-negative and false-positive rates than the level (as explained in [17] [13] [18]) that developers are willing to accept. Similar empirical studies, e.g., [19], [20], and [2], also observe low recall and precision of IDE plugins in detecting many vulnerabilities. Thus, it is necessary to understand why those false negatives and false positives happen to help improve the recall and precision of the plugins to a satisfactory level. Our research question is: **what are the root causes of correct and false results of plugin evaluations?**

A. Our studied vulnerability detecting plugins and vulnerabilities

Although ASIDE, LAPSE+, ESVD, SpotBugs, and FindSecBugs are evaluated in [1], in this study, we limited our focuses to the plugins that are actively maintained. Thus, we excluded LAPSE+ because it must run in an older release of Eclipse from 2010, namely, Eclipse Helios. We also excluded ASIDE because it has not been updated since 2013. Detailed information about the selected plugins in this study can be found in Table I.

The results of [1] show that different plugins focus on detecting different categories of vulnerabilities. To get an in-depth and generalizable understanding of approaches to detect specific vulnerability categories, we decided to choose vulnerabilities that allow us to compare the results and implementations between multiple plugins. We chose to study vulnerabilities that at least two of the plugins in [1] claim to cover. The web security vulnerabilities we studied include OS Command Injection, SQL injection, LDAP injection, HTTP Response Splitting, XPath Injection, Hard-coded password, Relative Path Traversal, Absolute Path Traversal, Cross-Site Scripting (XSS), Script in Error Message, and Script in Attributes. As shown in Table II, our studied vulnerabilities cover categories A1 (Injection), A2 (Broken Authentication), A5 (Broken Access Control) and A7 (XSS) in the OWASP top 10 2017 [12]. The Common Weakness Enumeration (CWE) [21] IDs of the studied vulnerabilities are also listed in Table II.

TABLE I
INFORMATION ABOUT THE SELECTED PLUGINS.

Plugin	Download From	Version	Version Date
ESVD	[6]	0.4.2	Jul 2016
SpotBugs	[10]	3.1.11	Jan 2019
FindSecBugs	[11]	1.8.0	Jun 2018

B. Research methods

To understand the reasons for true positives, false positives, and false negatives of the detection results of the plugins [1], we first read the source code of the test cases, which are in the Juliet Test Suite, and plugins to understand their implementations. Then, we hypothesized why the vulnerabilities are detected correctly or incorrectly by each plugin. Based on our hypotheses, we modified the plugins' source code to eliminate the false detection results. If our modifications successfully improved plugins' vulnerability detection precision and recall, we knew that our hypotheses of the root causes of the false results are correct. The precision, recall, and discrimination rate of detecting the plugins before and after improvements are shown in Table II. The discrimination rate [22] measures how well a plugin reports true positive detection results without also reporting false positives. The source code of our improved plugins are available at https://github.com/Beba-and-Karlsen/ide-plugins-modified.

We analyzed 11 types of vulnerabilities, which are related to 7,215 distinct variations of test cases in the Juliet Test Suites. The numbers of test cases of each vulnerability are shown in Table II. Considering the fact that we are executing three different plugins on these 7,215 test cases, this resulted in 20,313 different vulnerability reports, excluding the 1,332 vulnerabilities which SpotBugs does not claim to cover.

III. Research results

Our studies first gave us insights into how the test cases in the Juliet Test Suite and the plugins are designed and implemented. The studies also gave critical understandings of the reasons for the evaluation results of the plugins using the Juliet Test Suite as the test bed.

A. Design and implementations of Juliet Test Suite

There are multiple test cases for each vulnerability in the Juliet Test Suite, and each of these test cases has a unique composition of which source is used and what control- and data-flow complexity is added. These will be referred to respectively as source variants and flow variants. A source variant is a variant of a test case using a specific source. The Juliet Test Suite consists of similar cases where the only difference is what source is used. This way, it is possible to

TABLE II
RECALL (REC.), PRECISION (PRE.), AND DISCRIMINATION (DISC.) RATE BEFORE AND AFTER IMPROVEMENT. AFTER IMPROVEMENT NUMBERS ARE IN PARENTHESES. GREEN INDICATES INCREASED PERFORMANCE AND RED INDICATES DECREASED PERFORMANCE.

Vulnerabilities		Plugins								
CWE ID	Name (Number of test cases)	SpotBugs			FindSecBugs			ESVD		
OWASP A1: Injection		Rec.	Pre.	Disc.	Rec.	Pre.	Disc.	Rec.	Pre.	Disc.
78	OS Command Injection (444)	- (-)	- (-)	- (-)	86% (89%)	86% (100%)	72% (86%)	11% (19%)	100% (100%)	11% (19%)
89	SQL Injection (2220)	100% (84%)	43% (70%)	0% (49%)	86% (89%)	86% (100%)	72% (86%)	65% (22%)	39% (100%)	0% (22%)
90	LDAP Injection (444)	- (-)	- (-)	- (-)	86% (89%)	86% (100%)	72% (86%)	0% (19%)	N/A (100%)	N/A (19%)
113	HTTP Response Splitting (1332)	4% (47%)	100% (100%)	4% (47%)	74% (89%)	100% (100%)	74% (86%)	0% (0%)	N/A (N/A)	N/A (N/A)
643	XPath Injection (444)	- (-)	- (-)	- (-)	86% (89%)	86% (100%)	72% (86%)	0% (0%)	N/A (N/A)	N/A (N/A)
OWASP A2: Broken Authentication		Rec.	Pre.	Disc.	Rec.	Pre.	Disc.	Rec.	Pre.	Disc.
259	Hard-coded Password (111)	14% (14%)	100% (100%)	14% (14%)	43% (43%)	100% (100%)	43% (43%)	18% (18%)	87% (87%)	16% (16%)
OWASP A5: Broken Access Control		Rec.	Pre.	Disc.	Rec.	Pre.	Disc.	Rec.	Pre.	Disc.
23	Relative Path Traversal (444)	4% (47%)	100% (100%)	4% (47%)	86% (100%)	86% (88%)	72% (86%)	11% (19%)	100% (100%)	11% (19%)
36	Absolute Path Traversal (444)	4% (40%)	100% (100%)	4% (40%)	86% (100%)	86% (88%)	72% (86%)	11% (19%)	100% (100%)	11% (19%)
OWASP A7: Cross-Site Scripting		Rec.	Pre.	Disc.	Rec.	Pre.	Disc.	Rec.	Pre.	Disc.
80	Basic XSS (666)	3% (46%)	100% (100%)	3% (46%)	100% (89%)	88% (100%)	86% (86%)	11% (19%)	100% (100%)	11% (19%)
81	Script in Error Message (333)	6% (46%)	100% (100%)	6% (46%)	100% (89%)	88% (100%)	86% (86%)	11% (19%)	100% (100%)	11% (19%)
83	Script in Attributes (333)	6% (46%)	100% (100%)	6% (46%)	100% (89%)	88% (100%)	86% (86%)	11% (19%)	100% (100%)	11% (19%)

discover whether a static code analysis tool is unable to detect a specific source. Figure 1 and Figure 2 are examples of the source variants Connect TCP and File. These lines of code retrieve potentially vulnerable input from a TCP connection and a File, respectively.

```
1 socket = new
 ↪  Socket("host.example.org", 39544);
2 reader = new
 ↪  InputStreamReader(socket.
getInputStream(), "UTF-8");
3 readerBuffered = new
 ↪  BufferedReader(reader);
4 data = readerBuffered.readLine();
```

Fig. 1. Simplified Connect TCP source variant.

```
1 file = new File("C:\\data.txt");
2 stream = new FileInputStream(file);
3 reader = new
 ↪  InputStreamReader(stream,
 ↪  "UTF-8");
4 readerBuffered = new
 ↪  BufferedReader(reader);
5 data = readerBuffered.readLine();
```

Fig. 2. Simplified File source variant.

A flow variant of a test case includes specific control-flow statements that add complexity to the test case. Every vulnerability category includes the same 37 flow variants. The first flow variant is called the baseline and includes no added complexity. It is the easiest of the test cases. The next group of flow variants is called control-flow cases. There are 18 control-flow cases, and they hide the source or sink within control-flow statements (e.g., if-then statements) to make them harder to detect. The third group of flow variants is called data-flow cases, e.g., vulnerable data are retrieved from the source in one class and used in a sink in another class.

B. Design and implementations of the plugins

1) Design and implementation of SpotBugs: SpotBugs utilizes *bug patterns* to discover vulnerabilities. A bug pattern is defined as "*a code idiom that is likely to be an error*" [9]. SpotBugs analyzes the Java bytecode using the Byte Code Engineering Library (BCEL). SpotBugs uses both control-flow and data-flow analysis, which it implements as part of an internal framework [9]. An advantage of analyzing bytecode is that the compiler will optimize parts of the code. A relevant example is that for `if`-statements that will always be `true`, the compiler will exclude the `if`-statement in the bytecode. However, if the `if`-statement can be either true or false, e.g., by using a variable, the `if`-statement remains in the bytecode. In the Juliet Test Suite, a total of eight control-flow variants are optimized by the Java compiler, which removes the control-flow statement. This way, these test cases will be similar to the baseline for SpotBugs, and it will detect the vulnerabilities in these control-flow variants if it detects them in the baseline. SpotBugs categorizes each vulnerability occurrence into a rank category and a confidence category. The rank indicates how severe the vulnerability is, and the default setting of SpotBugs is to report on all rank categories. The confidence indicates how confident SpotBugs is in the vulnerability's authenticity and can be low, normal, or high.

2) Design and implementation of FindSecBugs: FindSecBugs is similar to SpotBugs and consists of added vulnerability detectors based on the functionality provided by SpotBugs. In addition to the many new detectors, FindSecBugs also adds techniques for detecting new classes of vulnerabilities. Taint analysis is one of these new techniques and is applied by many detectors to find connections between sources and sinks where tainted data are handled. Many of the detectors in FindSecBugs use resource files to list their vulnerable sources and sinks, which makes it possible to easily add or update the detectors without changing a detector's code itself. Possible tainted data are given properties using tags. There are many different reasons a tag can be assigned to a possible taint. For

example, for SQL injection taint detection, a tag is assigned to data to say either the data have been adequately sanitized or the data contain apostrophes which have been properly encoded to not interfere with an SQL query's apostrophes. The `encodeForSQL(Codec, String)` method will sanitize the data and make them safe from SQL injections, which results in the sanitized data being tagged as safe from SQL injections. In addition to tags, possible tainted data have a state. The state can be tainted, unknown, safe, null, or invalid. A hard-coded integer will automatically be given a safe state, while the configuration file can be used to identify that data from a list of specific methods are tainted. For example, the output of `Cookie.getName()` will be automatically given a tainted state, as a malicious user can change the name of a cookie. When all invoked instructions have been compared to the list of vulnerable sinks, and the tags and states have been updated, the detector will rank the confidence as follows and choose what to report:

- **High confidence:** If the state is set to tainted, set the confidence to high;
- **Normal confidence:** If the state is set to unknown or invalid, set the confidence to normal;
- **Low confidence:** If the state is set to safe, set the confidence to low.

3) Design and implementation of ESVD: ESVD analyzes Java code instead of bytecode. ESVD uses taint analysis and context-sensitive data-flow analysis [23]. ESVD uses XML files containing a list of sources, sanitization points [24], and sinks. Sources and sanitization points are universal for all kinds of vulnerabilities, while sinks are unique for each type of vulnerability. The XML file lists include 75 sources, 141 sinks, and 52 sanitization points. In ESVD, a variable receiving input from a vulnerable source is considered tainted. By using what is called tainted propagation, every other element in contact with this tainted element is also marked as tainted. Elements flowing through containers, such as arrays or lists, are automatically marked as tainted if one of the other elements in the container is tainted.

C. Design and implementation weaknesses of the plugins

Although SpotBugs, FindSecBugs, and ESVD have slightly different implementations, they have applied taint analyses more or less. ESVD uses taint analysis for all of its detectors, FindSecBugs uses taint analysis for many of its detectors, and SpotBugs uses taint analysis in several detectors, such as SQL injection, path traversal, and XSS. Our root cause analyses revealed two common weaknesses of implementing the taint analysis in these plugins. The weaknesses are explained below using some examples from the studied plugins. The complete root cause analysis results can be found in [25].

1) Missing sources and sinks: Taint analysis relies heavily on correct and complete lists of sources and sinks. One of the recurring fixes we carried out was adding missing sources and sinks.

One major limitation of SpotBugs detector of HTTP Response Splitting vulnerability is that the sources it can track are few and cover few of the test cases in the Juliet Test Suite, which leads to a recall of 4%. Only three sources are defined in this detector, two of which exist in the Juliet Test Suite. SpotBugs identifies vulnerabilities in one source, namely `getQueryString()`, but not the other. Taint analysis fails to track the tainted value because it is immediately broken down into substrings. The HTTP response splitting detector does not understand that the data from the string tokenization originate from the tainted string but rather sees the data as originating from the tokenizer itself. We have analyzed possible additional sources and added them to the HTTP response splitting detector. The added sources result in an increase in the recall and discrimination rates from 4% to 47%, as shown in Table II.

All the injection detectors suffer from FindSecBugs' decision to exclude a set of vulnerable sources from its taint analysis. The authors of FindSecBugs have intentionally defined `System.getenv(String)` and `System.getProperty(String)` as safe sources. These two sources are used to retrieve tainted data from the Juliet Test Suite's source variants `Environment and Property`. For the SQL injection category in the Juliet Test Suite, this results in a 14% false-negative rate. Similar numbers of false negatives are seen in FindSecBugs' other injection detectors because of the two missing vulnerable sources. Both environment variables and system properties can be tampered with by a user with access to the system, by another application running on the same system, or by a different vulnerability in the same application. Assuming environment variables and system variables are safe is not very rational. Through redefining the `System.getenv(String)` and `System.getProperty(String)` as unsafe sources, the recall of SQL injection detector of FindSecBugs is increased from 86% to 89%. The recalls of other FindSecBugs' injection detectors are also increased, as show in Table II.

Study [1] shows that ESVD can detect OS command injection and SQL injection vulnerabilities. However, the LDAP injection, HTTP response splitting, and XPath injection get zero true and false positives. This is despite the fact that all of the detectors in ESVD use the same underlying algorithm. The reason for the low recall of the OS command injection detector is that five source variants are not included in ESVD's resource list. As mentioned in section III-B3, ESVD has a shared list of sources that every detector utilizes. If a source is not included in the list, the source will not be detected by any of the detectors. The LDAP injection detector and HTTP response splitting detector of ESVD are unable to detect any vulnerabilities in the Juliet Test Suite because the sink used in the Juliet Test Suite is not included in ESVD's resource lists. When the missing sinks are added, the LDAP injection detector is able to detect some vulnerabilities' occurrences and the recall is increase from 0% to 19%, as shown in Table II.

2) Incomplete control-flow and data-flow analysis: The XSS detector in SpotBugs has recall and discrimination rates of between 3% and 6% due to the small number of detected sources and a taint analysis that struggles to follow data. The

taint analysis of the XSS detector for SpotBugs could not follow data through the use of `String.replaceAll()`. Half, i.e., 333, of the XSS test cases in the Juliet Test Suite executed a `replaceAll()` on the data before they were sent to the sink. The data tracking capabilities of the XSS detector are unable to understand that the output of the `replaceAll()` method is tainted, resulting in half of the test cases not being detected. We improved the taint analysis so that it can track data through the `replaceAll()` method, and this change results in 333 new true positives of XSS and increased the recall rate of SpotBugs' detectors to 46%, as show in Table II.

SpotBugs only detects hard-coded database passwords for the Java method `java.sql.DriverManager.getConnection()`. If one method is found to invoke `getConnection()`, the detector checks if the password argument of `getConnection()` is hard-coded, i.e., a literal or a constant instantiated as a literal. If it is indeed hard-coded, the detector reports the vulnerability with normal confidence. SpotBugs has only a 14% recall in detecting a hard-coded password. The limitation of SpotBugs is that it does not utilize control-flow analysis. Even though it reports more than one true positive for the control-flow variants, most true positives are not due to control-flow analysis, but rather the Java compiler optimizing the code and removing the control-flow statement, as explained in section III-B1. For control-flow variants that the Java compiler cannot optimize, the detectors could not identify the hard-coded password. Another limitation of SpotBugs is its incomplete data-flow analysis in this detector. The detector implements some data-flow analysis and is able to detect the vulnerabilities on the first method call even though the data travel through several methods and classes. However, it only detects data-flow test cases when the data are transferred as an argument, not when a method returns a value. Nor does it detect the vulnerability if the data are hidden in a data structure such as an `array` or a `Map`.

A few false positives and false negatives that are produced by the injection detectors in FindSecBugs are located in five data-flow cases, where the taint analysis in FindSecBugs is unable to track the data. Two of the five problematic data-flow test cases pass the value between methods using a class field. Another two of the five data-flow test cases process and wrap the data into other data structures, such as a Java `Container` or by serializing and deserializing the data. The taint analysis framework in FindSecBugs loses track of all four of these test cases, and the tainted data are given an "unknown" state. The fifth test case in the Juliet Test Suite that is misidentified is due to FindSecBugs' taint analysis being unable to track the data in class-based inheritance, and the data are therefore given an "unknown" state. This results in five false positives because FindSecBugs' OS command, SQL, LDAP, and Xpath injection detectors will report a vulnerability where data with an unknown state reach a vulnerable sink. However, the HTTP response splitting detector of FindSecBugs will not report when data with unknown states reach a vulnerable sink, which results in zero false positives in these five difficult data-flow test cases. This leads to a total recall of 74% of HTTP response splitting instead of the 86% recall that can be seen for the other detectors of FindSecBugs, but also a higher precision of 100% compared to 86% for the other detectors of FindSecBugs.

For ESVD, the control-flow variants detected by the OS command injection detector use a `while`-loop and a `for`-loop. It turns out that ESVD's underlying algorithm is able to track the data through these two control-flow statements but struggles with `if`-statements and `switch`-statements, where it detects nothing. Since most of the control-flow variants in the Juliet Test Suite include `if`-statements, the detector only detects 11% of the control-flow variants. In addition, the ESVD OS command injection detector detects data-flow variants where data are sent between methods within the same class. However, when the data move between classes, i.e., when data-flow analysis is needed, the detector cannot detect anything. This is especially surprising as utilizing context-sensitive data-flow analysis was the biggest selling point of ESVD [24].

Instead of reporting a vulnerability when tainted data from a source reach a sink, the SQL injection detector of ESVD reports all cases where a concatenated string is used in conjunction with a sink. The only criterion to report a vulnerability is that the concatenated string stems from string variables and not literals, i.e., a string originating from user input. The detector does not look at the content of the string variable and whether it originates from a vulnerable source. Thus, the detector does not actually utilize data-flow analysis and is more similar to basic pattern matching. The discrimination rate of 0% is an indication that the detector cannot differentiate true positives from false positives and does not use data-flow analysis. We have another observation that all of the false positives are detected in test cases consisting of a safe source and an exploitable sink. This is because all the detectors in ESVD only report a vulnerability when they find an exploitable sink. A vulnerable source on its own triggers nothing. After removing the unique detection method for SQL injection, which is dependent on detecting concatenated strings, and let the SQL injection detector use the underlying algorithm of ESVD as other injection detectors, we got improved precision and discrimination rates but worse recall, as shown in Table II.

ESVD's hard-coded password detector has a unique detection algorithm. It is still partly based on ESVD's underlying algorithm. However, instead of tracking data from source to sink, the detector reports a vulnerability when literals are used as hard-coded passwords. This means the detector does not use the resource list for sources, but it does use a unique resource list for sinks. The only sink included is `Driver Manager`. The other two sinks, namely, `Kerberos Key` and `Password Authentication`, are not included in this detector's resource list. After we added the two missing sinks, the hard-coded password detector still could not detect the vulnerabilities. Both of the sinks use a `char` array as a parameter, and the `String` input must be converted into a

`char` array before being injected into the sinks. We found that the hard-coded password detector cannot track the input through the conversion from `String` to `char`, which leads to no vulnerabilities being detected. After we changed one of the test cases for both sinks to use a hard-coded `char` array, ESVD was able to detect it. This experiment confirms that it is the `String` to `char` conversion that the detector is unable to track that prevents ESVD from detecting the vulnerabilities.

D. Other factors that influenced plugin evaluation results

Besides the weaknesses of plugins' design and implementation, a few other factors also influenced the true positives, false positives, and false negatives of plugin evaluations. Without considering these factors, the evaluation results can be very misleading.

1) Systematically biased test cases: We found that the Juliet Test Suite contains systematically biased test cases that could lead to a misleading evaluation of the detection approaches. Although the SQL injection detector of SpotBugs has a 100% recall, it has a precision of 43% and a 0% discrimination rate. SpotBugs calculates the confidence of vulnerability by considering the content of the data used in the SQL injection sink. When a possible SQL injection has been identified, the confidence of the detected vulnerability is calculated as follows.

- **High confidence:** If an unsafe string concatenation has been used, the sink's data are tainted, and both an opening and a closing quotation mark is present in the SQL query.
- **Normal confidence:** If an unsafe string concatenation has been used, the sink's data are **not** tainted, and both an opening and a closing quotation mark is present in the SQL query.
- **Normal confidence:** If an unsafe string concatenation has been used, the sink's data are tainted, and a comma is present in the SQL query.
- **Low confidence:** If an unsafe string concatenation has been used, the sink's data are **not** tainted, and a comma is present in the SQL query.
- **Low confidence:** If string concatenation has not been used.

The main limitation of SpotBugs' approach is that the entire detection algorithm is dependent on identifying string concatenations. For example, the following code will concatenate three strings: `"insert into ..."`, `"foo"`, and `"'"`. The detector sees a string concatenation, and it sees the opening and closing quotation marks. The detector will mark the string concatenation as unsafe. However, this code does not have any unsafe string concatenations.

```
String data = "foo";
Boolean result =
  sqlStatement.execute("insert into
  users (status) values ('updated')
  where name='"+data+"'");
```

On the other hand, the example code below will allow a malicious user to execute arbitrary commands in the SQL database:

```
Socket socket = new
  Socket("example.org", 8081);

/* read input from socket */
InputStreamReader readerInputStream
  = new InputStreamReader(socket.
getInputStream(), "UTF-8");
BufferedReader readerBuffered = new
  BufferedReader(readerInputStream);

String data =
  readerBuffered.readLine();
Boolean result =
  sqlStatement.execute(data);
```

This code example is highly dangerous but is not reported by SpotBugs. SpotBugs can correctly detect the `data` variable as a tainted source. However, since no string concatenations are carried out, the detection of tainted data will not affect vulnerability confidence. The high recall shown in Table II can be considered as misleading. All test cases in the Juliet Test Suite contain string concatenations. Thus, SpotBugs produces a 100% recall.

2) Missing test cases: To detect hard-coded password vulnerability, FindSecBugs has implemented five different detectors for finding hard-coded passwords. However, the Juliet Test Suite only has test cases to cover one out of the five detectors. This means that the test cases of the Juliet Test Suite do not reflect the full detection competence of FindSecBugs with respect to finding hard-coded password vulnerabilities.

Another example is related to the XSS detectors of FindSecBugs. FindSecBugs contains three different XSS detectors. Out of the three XSS detectors, only the detector aimed at Java servlet XSS vulnerabilities is relevant to the vulnerabilities present in the Juliet Test Suite. The other two detectors cannot be evaluated on the Juliet Test Suite.

3) Inconsistent confidence ranking: No static analysis tool is both sound and complete [23]. Most source code analysis tools will have to apply prioritization to generate output that prefers high recall or high precision. In the plugins, uncertain detections are given a lower priority or confidence, while certain detections are assigned higher confidence. The key issue we have found in the plugins regarding prioritization of output is inconsistency when deciding which ranking to give to the detection result. For SpotBugs, it is possible to change the code to adjust the confidence ranking approach of the detectors to favor recall or precision. By reporting data with an unknown state, recall is favored, and vice versa. While missing developer guidelines is a limitation, the fact that the detectors use a different confidence rating is also a limitation. If the user adjusts the plugin settings to hide false positives from the SQL injection detector, that is, to hide both low- and

normal-confidence detections, it will then hide every single detection from the HTTP response splitting detector, as even the certain detections are only given normal confidence.

Applying different confidence rankings can lead to different detection results. We believed that many developers would prefer low false positives [13], [17], [18]. Therefore, we modified its confidence rating as follows.

- **High confidence:** If tainted data, unsafe append, and opening and closing quotation marks have been seen.
- **Normal confidence:** If tainted data have been seen, in addition to either an unsafe append or opening and closing quotation marks.
- **Low confidence:** If an unsafe append and opening and closing quotation marks have been seen, but no tainted data.

This change of confidence rating led to 360 fewer true positives, but 2,220 fewer false positives, an increase in SpotBug's SQL injection detection precision from 43% to 70%, and an increase in the discrimination rate from 0% to 49%. The results of the improved detector can be seen in Table II, which shows only the vulnerabilities with normal and high confidence.

4) Misclassification of vulnerability: Despite not stating it anywhere in its documentation, ESVD is able to detect hard-coded passwords. When analyzing the source code to identify why it was not reported, we discovered that instead of being labeled as a hard-coded password, it is labeled as a security misconfiguration. In [1], the authors reported that ESVD could not identify hard-coded passwords. After analyzing the root cause of the weaknesses, we realized that this is due to ESVD's mislabeling of results. Hard-coding passwords into source code is not a misconfiguration but rather an authentication problem.

All three of our studied plugins have problems with the classification of relative and absolute path traversal. For SpotBugs, determination of whether the vulnerability is an absolute or relative path traversal is based on the source of the data. If the source is `HttpServletRequest.getParameter(String)`, SpotBugs reports the vulnerability as an absolute path traversal, and any other source is determined to be a relative path traversal. One possible reason for SpotBugs to implement the detector this way lies in the inability of its taint analysis to track the data. In the case of relative path traversal, a common way to combine the tainted user input "`data`" and the pre-determined root path "`root`" is to concatenate the strings, e.g., `new File(root + data)`. The taint analysis is unable to follow the data through string concatenations, therefore losing track of the data source. The path traversal detector in FindSecBugs is based on the same taint analysis in the injection detectors, with the exception of different sinks and a slightly different confidence ranking algorithm. The detector does not detect whether the vulnerability is a relative or absolute path traversal vulnerability, but rather reports all vulnerabilities as simply path traversal vulnerabilities. For FindSecBugs, when the taint analysis loses track of the data source, but the data enter a vulnerable sink, a vulnerability is reported with normal confidence. When the taint analysis is able to track the data source and the data enter a vulnerable sink without sanitization, a vulnerability is reported with high confidence. The sinks for both types of path traversal are the same, and trying to separate the two path traversal types would not allow the detector to keep using the taint analysis framework that is provided by FindSecBugs, as this framework could not classify the path traversal type. The relative and absolute path traversal vulnerabilities might have somewhat different countermeasures. If they are misclassified, it could be possible for developers to apply a countermeasure that only works for one of the path traversal vulnerabilities but still tricks the static analysis tool into thinking the data have been sanitized for the other category.

5) Missing report of multiple vulnerabilities in the same line of code: When executing the HTTP response splitting detector with standard settings in ESVD, the detector will only report one of the three sinks used in the Juliet Test Suite. It reports on `addCookie`, but not on `addHeader` or `setHeader`, even if all sinks have already been added. By manually inspecting the test cases, we discovered that ESVD reports XSS on `addHeader` and `setHeader`. When we alter the settings and turn off the XSS detector, ESVD reports HTTP response splitting on these instead. This exposes a critical limitation in ESVD's implementation, as it does not report multiple vulnerabilities in a single line of code. If a false positive is first reported, in this case XSS, then a true positive, i.e., HTTP response splitting, may not be reported.

IV. Related work

Many studies used source code analysis methods to detect security vulnerabilities. Some studies, e.g., [26]–[45], analyzed the reasons for the false results. Most of these studies focused on analyzing the weaknesses of their approaches and tools, as our results showed in section III-C. For example, [29] concludes that "*Since our approach is mainly based on static analysis (for determining attack conditions), it inherits one of the intrinsic weaknesses of the latter: it cannot deal with calls to system and library functions for which the source code is not available.*" Another example [36] states that "*All false positives are based on the fact that our prototype is not able to detect path sensitive sanitization.*"

Some studies identified other factors that could influence the evaluation results. For example, [27] identifies that one possible reason for false positives is that it does not include the sanitation routines, such as htmlentities. [41] identifies similar issues due to concatenation as we explained in section III-D1. [34] indicates that configuration of the confidence of the tool can potentially influence the precision. Few studies analyzed false positives, false negatives, true positives, and the test suites to determine if any incompleteness or bias in the test suite can cause misleading conclusions. One exception is [37], which states that "*number of existing vulnerabilities in software is unknown*" and it is, therefore, difficult to measure the false negatives. The Juliet Test Suite has been used in

many studies, e.g., [1] [2] [46] [47], to evaluate and benchmark vulnerability detectors. Without thorough understandings of the limitations of the test cases, results of the evaluation can be misinterpreted.

Few studies analyzed root causes of true positives. For example, the authors of [37] state "*A true positive was counted for every vulnerable line of code. This means that a vulnerability inside a function was counted only once if the function was called in an exploitable context and not for every call. Sometimes, a valid report was counted even if the vulnerability is not exploitable. For example, if the same input is used in two differently constructed SQL queries, but the application exits after a SQL query fails, and it is not possible to craft an injection that fits both SQL queries, two valid reports were counted nonetheless.*" Insights from [37] are in line with our observations that without understanding the true positives, it is very likely that the positive evaluation results are also misleading.

V. Discussions

Our root cause analysis illustrates that many factors can dramatically influence the evaluation results of security vulnerability detection tools. The results give insights to scientists who propose new tools and to the industrial tool users.

A. Implication for scientists

A study proposing vulnerability detection approaches and tools should report the following evaluation results.

- The version of the dataset or test suite the evaluation is based on because different versions of the datasets or test suite can include different types and numbers of vulnerabilities.
- The configuration of the tool which implements the possible approach. The impact of the configuration on the detection results should also be explained. As we have shown in section III-D3, different configurations of the confidence ranking can lead to very different precision and recall results.
- The list of the vulnerabilities identified, published as an appendix or in an accessible URL. As we have shown in section III-D4, security vulnerabilities can be misclassified into the wrong category. For a new type of vulnerabilities, the misclassification can be a more frequent problem than the well-known ones.
- Explanations of the reasons for the true positives. If a proposed approach can identify a particular security vulnerability, there must be some reasons. Without explaining the reasons, true positives can be misleading. As we have shown in section III-D1, the true positives can be results of a combination of bad detection approach and biased test cases. The true positives can also be results of good detection approach but incomplete test cases.
- Explanations of the reasons for the false positives. As we have shown in section III-D4, false positives can be results of different confidence ranking setups. Explanations of the false positives can help users identify approaches or setups to limit false positives.
- Explanations of the reasons for the false negatives, if the false negatives can be identified. One main difficulty of evaluating the security vulnerability detection approach is that the numbers of vulnerabilities of benchmark datasets or test suites can be unknown. It is, therefore, difficult to report the false-negative rates precisely. We would recommend that the security communities make a joint effort to develop more benchmark applications and test suites with a known number of vulnerabilities. For researchers who focus on approaches to detect novel vulnerabilities, a small application with studied vulnerabilities inserted should be developed, used in the evaluation, and published for investigation by others.

If a study also wants to compare its approaches and tools with existing approaches, the study should also distinguish whether the different results are caused by the pros and cons of the approaches under comparison, or by the detailed implementations of the approaches. As we have shown in section III-C1, detailed implementation of an approach, rather than the approach itself, can dramatically influence the results of the detection. A good approach may not show better detection results than a bad one if the good approach is not implemented properly.

Our study also reveals critical weaknesses, e.g., missing and systematically biased test cases, of the Juliet Test Suite. We believe similar weaknesses may exist in many other security vulnerability test suites. Thus, we encourage researchers to work together to improve the quality of the test suites to avoid misleading evaluation results.

B. Implication for industry practitioners

Developers like to use vulnerability detectors which report high precision and recall [17] [13] [18]. Our results show that precision and recall reported by the tool vendors can be misleading without proper root cause analyses of the evaluation result. Results of section III-D1 show that confidence ranking can significantly influence false-positive rates. Thus, industry practitioners must understand the impact of the configurations of the vulnerability detection tool and the tool limitations before using them.

C. Threats to validity

One possible threat to internal validity is that the root cause analysis can be wrong. To address this threat, every vulnerability report analysis was analyzed and cross-validated by at least two researchers. One possible threat to external validity is that our study is limited to only three plugins and has studied only a limited number of vulnerability categories. The results of the study may not be generalizable to other tools and categories of vulnerabilities. However, we believe that many of the insights we got from analyzing these three plugins can be generalized to similar vulnerability detectors, in particular those using taint analysis and static code analysis.

The evaluation bias issues we have identified from this study are also generalizable.

VI. Conclusion and Future Work

A vulnerability detection tool that claims to cover a specific vulnerability has an inherited level of trust to fulfill. Our study showed that it is critical to perform follow-up studies to understand the evaluation results of approaches and tools. Otherwise, the evaluation results can be wrong or biased. Based on observations of this study, we recommended how to evaluate the security detection approaches more thoroughly and what to include in the evaluation report. In addition, our study provides essential information for improving the three plugins we studied and similar tools.

As discovered in this study, the Juliet Test Suite can still give very misleading results, even if the test suite has been developed for the purpose of evaluating security vulnerability detection approaches. To help people evaluate their vulnerability detection approach better, we plan to update or improve the Juliet Test Suites, and similar test suites, and to empirically assess their strengths and weaknesses.

References

[1] J. Li, S. Beba, and M. M. Karlsen, "Evaluation of open-source IDE plugins for detecting security vulnerabilities," in *Proc. of the Evaluation and Assessment on Software Engineering Conference 2019*, 2019, pp. 200–209. [Online]. Available: https://doi.org/10.1145/3319008.3319011

[2] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. S. Cruzes, "Myths and Facts About Static Application Security Testing Tools: An Action Research at Telenor Digital," in *Agile Processes in Software Engineering and Extreme Programming*. Springer International Publishing, 2018, pp. 86–103.

[3] OWASP, "OWASP ASIDE Project," 2016. [Online]. Available: https://www.owasp.org/index.php/OWASP_ASIDE_Project

[4] J. Xie, B. Chu, H. R. Lipford, and J. T. Melton, "ASIDE: IDE Support for Web Application Security," in *Pro. of the 27th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, 2011, pp. 267–276. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076770

[5] J. Zhu, "ASIDE-Education," 2013. [Online]. Available: https://github.com/JunZhuSecurity/ASIDE-Education

[6] L. Sampaio, "TCM_Plugin," 2016. [Online]. Available: https://github.com/lsampaioweb/TCM_Plugin

[7] B. J. Berger, "lapse-plus," 2013. [Online]. Available: https://github.com/bergerbd/lapse-plus/

[8] OWASP, "OWASP LAPSE Project," 2017. [Online]. Available: https://www.owasp.org/index.php/OWASP_LAPSE_Project

[9] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, 2004. [Online]. Available: http://doi.acm.org/10.1145/1052883.1052895

[10] SpotBugs, "SpotBugs Eclipse plugin," 2019. [Online]. Available: https://marketplace.eclipse.org/content/spotbugs-eclipse-plugin

[11] Find Security Bugs, "Find Security Bugs - The SpotBugs plugin for security audits of Java web applications," 2019. [Online]. Available: https://find-sec-bugs.github.io/

[12] OWASP, "OWASP Top 10 - 2017 The Ten Most Critical Web Application Security Risks," Tech. Rep., 2017.

[13] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proc. of the 31st IEEE/ACM International Conference on Automated Software Engineering*. New York, New York, USA: ACM Press, 2016, pp. 332–343. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2970276.2970347

[14] A. Amira, A. Ouadjaout, A. Derhab, and N. Badache, "Sound and static analysis of session fixation vulnerabilities in php web applications," in *Proc. of the 7th ACM on Conference on Data and Application Security and Privacy*. New York, NY, USA: ACM, 2017, pp. 139–141. [Online]. Available: http://doi.acm.org/10.1145/3029806.3029838

[15] X. Yan, H. Ma, and Q. Wang, "A static backward taint data analysis method for detecting web application vulnerabilities," in *Proc. of the IEEE 9th International Conference on Communication Software and Networks*, May 2017, pp. 1138–1141.

[16] FindBugs, "Find Bugs in Java Programs," 2019. [Online]. Available: http://findbugs.sourceforge.net/

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proc. of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press, 2013, pp. 672–681. [Online]. Available: https://dl.acm.org/citation.cfm?id=2486877

[18] E. B. Sørensen, E. K. Karlsen, and J. Li, "What norwegian developers want and need from security-directed program analysis tools: A survey," in *Proceedings of the Evaluation and Assessment in Software Engineering*, ser. EASE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 505–511. [Online]. Available: https://doi.org/10.1145/3383219.3383293

[19] T. Charest, N. Rodgers, and Y. Wu, "Comparison of Static Analysis Tools for Java Using the Juliet Test Suite," in *Proc. of the 11th International Conference on Cyber Warfare and Security*, 2016, pp. 431–438.

[20] A. Z. Baset and T. Denning, "IDE Plugins for Detecting Input-Validation Vulnerabilities," in *2017 IEEE Security and Privacy Workshops*, 2017, pp. 143–146.

[21] MITRE, "About CWE," 2018. [Online]. Available: https://cwe.mitre.org/about/index.html

[22] A. Delaitre, B. Stivalet, P. E. Black, V. Okun, A. Ribeiro, and T. S. Cohen, "SATE V Report: Ten Years of Static Analysis Tool Expositions," National Institute of Standards and Technology, Tech. Rep., 2018.

[23] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, no. C, pp. 5–21, 2008.

[24] L. Sampaio and A. Garcia, "Exploring context-sensitive data flow analysis for early vulnerability detection," *Journal of Systems and Software*, vol. 113, pp. 337–361, 2016. [Online]. Available: 10.1016/j.jss.2015.12.021

[25] S. Beba and M. M. Karlsen, "Implementation analysis of open-source static analysis tools for detecting security vulnerabilities," 2019. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2624972

[26] A. Doupe, B. Boe, C. Kruegel, and G. Vigna, "Fear the ear: Discovering and mitigating execution after redirect vulnerabilities," in *Proc. of the 18th ACM Conference on Computer Communications Security*, 2011, Conference Proceedings, pp. 251–261.

[27] X. X. Yan, H. T. Ma, and Q. X. Wang, "A static backward taint data analysis method for detecting web application vulnerabilities," in *Proc. of the IEEE 9th International Conference on Communication Software and Networks*, 2017, Conference Proceedings, pp. 1138–1141.

[28] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proc. of the 2008 IEEE Symposium on Security and Privacy*, 2008, Conference Proceedings, pp. 387–401. [Online]. Available: https://ieeexplore.ieee.org/ielx5/4531131/4531132/04531166.pdf?tp=&arnumber=4531166&isnumber=4531132

[29] J. Thom, x00E, L. K. Shar, D. Bianculli, and L. Briand, "An integrated approach for effective injection vulnerability analysis of web applications through security slicing and hybrid constraint solving," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018. [Online]. Available: https://ieeexplore.ieee.org/ielx7/32/4359463/08373739.pdf?tp=&arnumber=8373739&isnumber=4359463&ref=

[30] S. Anil, S. G. Manoj, L. Vijay, and C. Mauro, "You click, i steal: analyzing and detecting click hijacking attacks in web pages," *International Journal of Information Security*, 2018.

[31] O. Olivo, I. Dillig, and C. Lin, "Detecting and exploiting second order denial-of-service vulnerabilities in web applications," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, Conference Paper, pp. 616–628.

[32] I. Medeiros, N. Neves, and M. Correia, "Dekant: a static analysis tool that learns to detect web application vulnerabilities," in *Proc. of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, Conference Paper, pp. 1–11.

[33] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen, "Thaps: Automated vulnerability scanning of php applications," in *Nordic Conference*

on Secure IT Systems, A. Jøsang and B. Carlsson, Eds. Springer Berlin Heidelberg, 2012, Conference Proceedings, pp. 31–46.

[34] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. of the ACM/IEEE 30th international conference on Software engineering*. ACM, 2008, Conference Paper, pp. 171–180.

[35] Y. H. Zheng and X. Y. Zhang, "Path sensitive static analysis of web applications for remote code execution vulnerability detection," in *Proc. of the 35th ACM/IEEE International Conference on Software Engineering*, 2013, Conference Proceedings, pp. 652–661.

[36] J. Dahse and R.-U. B. Thorsten Holz, "Static detection of second-order vulnerabilities in web applications," in *Proc. of the 23rd USENIX Security Symposium.*, 2014, Conference Proceedings.

[37] J. Dahse, "Simulation of built-in php features for precise static code analysis," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2014, Conference Paper.

[38] L. K. Shar and H. B. K. Tan, "Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Information and Software Technology*, vol. 55, no. 10, pp. 1767–1780, 2013.

[39] F. S. L. X. Z. Su, "Static detection of access control vulnerabilities in web applications," *20th USENIX Security Symposium*, 2011.

[40] S. Son and V. Shmatikov, "Saferphp: Finding semantic vulnerabilities in php applications," in *Proc. of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. New York, NY, USA: ACM, 2011. [Online]. Available: https://doi.org/10.1145/2166956.2166964

[41] I. Medeiros, N. Neves, and M. Correia, "Detecting and removing web application vulnerabilities with static analysis and data mining," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 54–69, 2016. [Online]. Available: https://ieeexplore.ieee.org/ielx7/24/7422884/07206620.pdf?tp=&arnumber=7206620&isnumber=7422884

[42] X. Li, W. Yan, and Y. Xue, "Sentinel: securing database from logic flaws in web applications," in *Proc. of the 2nd ACM conference on Data and Application Security and Privacy*. ACM, 2012, Conference Paper, pp. 25–36.

[43] A. Møller and M. Schwarz, "Automated detection of client-state manipulation vulnerabilities," in *Proc. of the 34th International Conference on Software Engineering*, 2012, Conference Proceedings, pp. 749–759. [Online]. Available: https://ieeexplore.ieee.org/ielx5/6218989/6227015/06227143.pdf?tp=&arnumber=6227143&isnumber=6227015

[44] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, "Toward automated detection of logic vulnerabilities in web applications," in *Proc. of the 19th USENIX conference on Security*. 1929834: USENIX Association, 2010, Conference Paper, pp. 10–10.

[45] F. Spoto, E. Burato, M. D. Ernst, P. Ferrara, A. Lovato, D. Macedonio, and C. Spiridon, "Static identification of injection attacks in java," *ACM Trans. Program. Lang. Syst.*, vol. 41, no. 3, Jul. 2019.

[46] A. Wagner and J. Sametinger, "Using the juliet test suite to compare static security scanners," in *2014 11th International Conference on Security and Cryptography (SECRYPT)*, 2014, pp. 1–9.

[47] T. Charest, N. Rodgers, and Y. Wu, "Comparison of static analysis tools for java using the juliet test suite," in *International Conference on Cyber Warfare and Security*, 2016.