논문 2021-2-14 http://dx.doi.org/10.29056/jsav.2021.12.14

Static Analysis Tools Against Cross-site Scripting Vulnerabilities in Web Applications : An Analysis

Nurul Atiqah Abu Talib*, Kyung-Goo Doh*†

Abstract

Reports of rampant cross-site scripting (XSS) vulnerabilities raise growing concerns on the effectiveness of current Static Analysis Security Testing (SAST) tools as an internet security device. Attentive to these concerns, this study aims to examine seven open-source SAST tools in order to account for their capabilities in detecting XSS vulnerabilities in PHP applications and to determine their performance in terms of effectiveness and analysis runtime. The representative tools - categorized as either text-based or graph-based analysis tools - were all test-run using real-world PHP applications with known XSS vulnerabilities. The collected vulnerability detection reports of each tool were analyzed with the aid of PhpStorm's data flow analyzer. It is observed that the detection rates of the tools calculated from the total vulnerabilities in the applications can be as high as 0.968 and as low as 0.006. Furthermore, the tools took an average of less than a minute to complete an analysis. Notably, their runtime is independent of their analysis type.

keywords : Cross-site scripting, Open-source Static Analysis Security Testing Tools, Detection

1. Introduction

Cross-site scripting (XSS) is one of the most notorious threats to the internet world today. It occurs when the validation of user input in the web application is improper, thus resulting in the unintended interpretation of the inputs by the browser as scripts or code. The presence of these misinterpreted codes is a menace because it has an impact on causing disruptions to internet usage such as denial of service and information theft that may even contemptuously affect the users' privacy or the good name of their organization. In the 2017 OWASP report, XSS is ranked the third top threat, and remains practically unchanged from the year 2013 to 2017. The chart on XSS yearly vulnerability survey [1], has it that despite the number has somewhat dwindled from 1105 in 2014 to 495 in 2016, but a new surge to 855 in 2017 is an upset. In addition, other reports of rampant XSS vulnerabilities in web applications clearly show that they are an issue still yet to be dissolved [2].

There are several methods used to help prevent XSS in web applications. Among them is by performing automatic code review using static analysis security testing (SAST) tools

* Department of Computer Science and Engineering, Hanyang University ERICA
† Corresponding Author : Kyung-Goo Doh(email: doh@hanyang.ac.kr)
Submitted: 2021.11.30. Accepted: 2021.12.11. Confirmed: 2021.12.20.

on server-side code before the application is deployed. Albeit, getting an effective SAST tool for web applications remains an issue. By common rule, to reach an accurate result from a tool is to generally require more computational work.

However, having more computational work in a task would imply a longer time to complete a job of an analysis. For all it matters, a low total runtime of a web application is as critical as its effective security solutions that experts need to consider when designing a tool. A tool that lacks this specification on its performance would risk getting abandoned by the users.

It is, therefore, important that the issue of performance of a security tool be addressed accordingly through sound research. This is to say, the effectiveness of SAST tools and their runtime are pertinent connections to be viewed with scrutiny in a specific study. Taking up the issue, this study seeks to survey the applications of current SAST tools used to prevent XSS and to analyze the circumstances in which each technique and the size of the target application affects their performance.

This study is set on four main objectives: (1) to account for the current SAST tools to detect XSS vulnerabilities in PHP applications, (2) to determine the tools' sensitivity, (3) to determine the effectiveness (i.e., precision and recall) of each SAST tool running as a security device, (4) to compare the analysis time taken by each tool to complete a detection task, and (5) to relate the tools' effectiveness performance to the analysis time performance in respect to their vulnerability search type. More significantly, the research is an attempt to initiate a comprehensive analysis of SAST tools on the performance of XSS detection effectiveness and analysis run time.

In this study, we examine seven current open-source or free SAST tools to determine the extent of their ability to detect XSS vulnerabilities in PHP applications. These seven SAST tools are: phpcs-security-audit (PHPCS), PhpSAFE, Pixy, RIPS, VisualCodeGrepper (VCG), Web Application Protection (WAP), and Yet Another Source Code Analyzer (YASCA). Each of these tools would be run consecutively to firstly, determine the effectiveness (i.e., precision and recall) in XSS detection and to secondly, assess the runtime performance when analyzing the test cases. More specifically, the study is to answer 5 research questions: (1) do the tools' implementation decisions include sensitive-based analysis?, (2) are the tools' XSS vulnerability detections true or false?, (3) what is the rate that a tool may miss true vulnerabilities?, (4) how do the tools perform in terms of precision, recall and F-Score?, and (5) how long do the tools take to complete analyses on test cases?.

2. Open-source Static Analysis Security Testing Tools

Manual inspection of vulnerability present in web applications is known to be error-prone and time-consuming task. To overcome, the

use of automated technique of static analysis to validate server-side code before its output program is supplied to other entities such as the browser, is a way to go. Incidentally, the technique helps to scrutinize insufficient input validation in the code that causes XSS vulnerabilities.

This section provides a brief overview of the seven widely used open-source SAST tools of our study [6] to perform XSS vulnerability checks in PHP applications. Table 1 shows the tools that we categorize in terms of their main method (type) of vulnerability search, which we refer to as either *text-based* (TBA) or *graph-based* analysis (GBA) tools.

Table 1. Overview of Static Analysis Security Testing Tools For PHP Applications Under Study

Tool	Vuln. Search Type	Language Used	Year Created	Last Update
PHPCS	Text	PHP	2013	2017
VCG	Text	Visual Basic	2015	2016
YASCA	Text	PHP	2008	2010
PhpSAFE	Graph	PHP	2013	2016
Pixy	Graph	Java	2006	2014
RIPS	Graph	PHP	2010	2017
WAP	Graph	Java	2014	2015

phpcs-security-audit [8] is a tool that utilizes a set of PHP_CodeSniffer (PHPCS) [9] rules called "sniffs" to find vulnerabilities in PHP programs (including object-oriented programs). It uses PHP built-in functions to tokenize programs and can automatically corrects violated coding standards. *Visual Code Grepper (VCG)* [13] is a standalone automated code security review written in Visual Basic. It consists of a configuration file that aids the tool to perform checks for insecure functions. *Yet Another Source Code Analyzer (YASCA)* [15] is another PHP-based tool that leverages other static analyzers, such as PMD [16], to perform vulnerability scanning. YASCA can also be used to explore code quality, performance, and conformance to best practices.

PHP Security Analysis for Everyone or *phpSAFE* [10] is a PHP-based static code analyzer that identifies vulnerabilities for PHP plugins using OOP. It conducts lexical and semantic analysis on the Abstract Syntax Tree (AST) of a PHP program. Being a memory hog, phpSAFE is reportedly failing to parse files that contain large numbers of file-includes [10]. *Pixy* [11] is a Java-based analyzer that utilizes alias and literal analysis to obtain variable aliases and the literal values that each may hold at a single point in the program. Despite its ability to detect vulnerabilities concerning register_globals, Pixy fails to analyze object-oriented-featured PHP files [10]. *RIPS* [12] detects vulnerabilities by modeling the PHP program using Control Flow Graphs (CFG) and analyse whether or not potential vulnerable functions (sensitive sinks) are influenced by input sources using data-flow analysis. Although, this PHP-based tool is reportedly able to analyze programs that contain large numbers of file-includes, it fails to analyze object-oriented-featured PHP code [10]. *Web Application Protection (WAP)* [14] is a tool written in PHP that involves detecting

and correcting vulnerabilities in PHP programs using taint analysis and a data mining procedure.

All the above tools are designed to help developers automatically detect security flaws in programs. However, one of the properties of static analysis technique is detecting all possible program paths. As some of these paths may not be used during program execution, the tools utilizing this technique are likely to come up with false-positive results.

3. Methodology

This study is an experiment to evaluate the perfrormance of SAST tools involving TBA and GBA. The study procedures are discussed as the following.

3.1 Sampling

The experiment requires a set of 6 test cases to assess the effectiveness of each tool to detect XSS vulnerabilities. For this, we search for real-world vulnerable applications from the Common Vulnerabilities and Exposures (CVE) website [2]. From the many web application reports on the websites with XSS vulnerabilities, we refine our search of samples by picking only those that are PHP, open-source, and contain references to the proof of concept exploits in the application. From the refinement, we choose one version from each sample, thus, acquiring six PHP files from five different applications as test cases.

3.2 Vulnerable Test Cases

Next, to identify the types of vulnerabilities that exist in our test cases, we develop a PHP code that automatically collects all the sinks (i.e., `print`, `echo`, etc.) in the test cases. The code also categorizes benign sinks (containing hard-coded literals, such as strings and integers) as true-negatives. Following Kallin and Valbuena [17], we identify potentially vulnerable sinks according to 5 common injection contexts in which the user input may be inserted in the code within the application. The 5 contexts are HTML element content (**HTML**), HTML attribute value (**Attribute**), URL query value (**URL**), CSS value (**CSS**), and JavaScript value (**JS**). With the help of PhpStorm's data flow analyzer, we proceed to manually inspect each of the sinks individually to determine whether or not it carries parameters that originate from vulnerable sources (i.e., `GET`, `POST`, `REQUEST`, etc.). The inspection results to a total of 73 possible sinks or true-negatives, $T(tn)$, and a total of 82 true-positives, $T(tp)$, all of which are used as our benchmark in analyzing the effectiveness performance in the next section. Table 2 shows an overview of the test cases and the number of vulnerabilities in each as well as their injection contexts.

DW test case is a file from DokuWiki [18], an open-source wiki application software that offers built-in accesscontrols and authentication connectors for better security, thus, having an advantage over traditional wiki. This test case exhibits 2 vulnerabilities, both are in the

Table 2. An Overview of the Vulnerable Application Test Cases

ID	CVE	Application Name	Version	File Name	*LOC	Vuln. Sinks	Context
DW	CVE-2017-12583	DokuWiki	2017-02-19b	doku.php	591	2	HTML- 100.0%
PMW	CVE-2017-12984	PHPMyWind	5.3	admin/message_update.php	1,850	7	HTML - 42.9% Attribute - 57.1%
UPB-1	CVE-2015-2217	Ultimate PHP Board	2.2.7	profile.php	6,189	45	HTML - 48.9% Attribute - 20.0% URL - 17.8% CSS - 6.7% JS - 6.7%
UPB-2	CVE-2015-2217	Ultimate PHP Board	2.2.7	search.php	5,450	6	HTML - 50.0% Attribute - 16.7% URL - 33.3%
WP-1	CVE-2017-1002017	Wordpress gift-certificate-creator	1.0	giftcertificates.php	162	7	HTML - 28.6% Attribute - 71.4%
WP-2	CVE-2012-5229	Wordpress Slideshow Gallery 2	1.1.4	css/gallery-css.php	634	14	CSS 100.0%

*LOC covers include files

context of HTML. Meanwhile, PMW test case is a file from PHPMyWind [19], a popular application software from China that was built for easy website construction. Although a new version of the application is available, its solution for the vulnerabilities in CVE reports is unknown at the time of writing. In continuation, this test case contains 2,323 lines of code with 7 vulnerabilities that are in the contexts of HTML and attribute.

UPB-1 and UPB-2 test cases are from Ultimate PHP Board (UPB) [20], a simple-based forum application software that is ideal for small websites. It's a text-based forum that does not utilize MySQL databases with the hope to provide a faster experience for its users. UPB-1 is the largest test case with 5,591 lines of code. It exhibits the highest counts of 45 vulnerabilities in five common injection contexts. The other test case, UPB-2, contains 6 vulnerabilities which are in the injection contexts of HTML, attribute, and URL. The vulnerabilities in both test cases that were reported in CVE have not been fixed at the time of writing.

The next test cases are acquired from WordPress plugins. WP-1 is a file from Gift Certificate Creator [21], a plugin that allows users to manage gift-certificates in WordPress applications. This test case has 162 lines of code and 7 vulnerabilities. Two of the vulnerabilities is in the context of HTML, while

the other five are in the context of **attribute**. **WP-2** test case is a file from Slideshow Gallery Pro [22], a plugin to enable display of multiple galleries in the WordPress site. This test case consists of only 24 lines of code - the smallest file amongst all test cases. All the vulnerabilities in this test case is in the context of CSS value. The reported vulnerabilities in **WP-2** have not been fixed at the time of writing. Meanwhile, we could not obtain any information regarding the fixes for **WP-1**.

3.3 Characteristics of Static Analysis Security Testing Tools

Prior to performing the analysis, we first examine the characteristics of the tools based on the data-flow analysis methods. This is to ensure that they implement inter-procedural-based analysis, and conduct flow-, context- and path-sensitive analysis. Therefore, to determine this, we create several tests and examine the outputs from each tool.

The inter-procedural analysis allows operation between caller and callees throughout an entire program and in the correct sequence [23]. Therefore, the test is to check whether the analysis tools' operation can flow to and from calls in sequence, e.g, procedure calls. Context-sensitivity, on the other hand, allows the operation to recognize the dependent behavior of procedures based on the calling context i.e., call strings or assumption sets [24]. Therefore, the test is to ensure that the analysis tools' operation can differentiate between different calls to the same procedures in the correct order. We create and deploy two test cases for this purpose.

Next, flow-sensitivity allows the operation to regard the control-flow and to compute which statement each variable points to at every program point. The test for flow sensitivity would, thus, check that the analysis tools' operation regards the order of the executed statements that are being executed. We create four test cases for testing flow-sensitivity. Lastly, as path sensitivity analysis considers flow-sensitivity and computes information at every probable execution path, the test is to check that the analysis tools' operation can differentiate the information of variables between different conditional statements and identify improbable paths. We create two test cases for testing path-sensitivity.

Data-flow analysis operates on control-flow graphs of a program and does not lend itself to tools using text-based vulnerability search. Therefore, it is excluded from the sensitivity test. Only GBA tools are evaluated instead. The next section presents the results of this experiment.

3.4 Testing Sensitivity

We next continue with testing to determine the the sensitivity traits of the tools.

The flow-sensitive tests involve four test cases. The first one consists of a user-controlled value stored in the vulnerable `GET` method that is assigned to a variable, `$vuln_source`. Next is the variable

dreassigned to a new variable, $x. This new variable is later assigned an invulnerable static string value, which is then printed out using PHP's function, echo. The second is an object-oriented-structured PHP test that consists of three PHP classes named **Safe1**, **Vuln**, and **Safe2**. The Safe1 and Safe2 classes contain a function with the same name as the class that accesses the non-static property model and assigns invulnerable static strings to it. On the other hand, the function in the **Vuln** class assigns a vulnerable value from the GET method to the property model. Using the new keyword, instances of the three classes, **Safe1**, **Vuln**, and **Safe2**, are created and are stored in the variables $x, $y, and $z each, enabling them to access the same instance as the object that was assigned to it. These variables are then reassigned to the instantiated variables $y, $z, and $x, respectively, making the variables $x and $z instances of the **Vuln** class. The three variables are next printed out using PHP's function, echo.

The third test is similar to the previous test, except that it is not an object-oriented-structured code. Instead of functions embedded in classes, we created variables $x, $y, and $z and assigned them the values of 1, a GET method, and 2, respectively. Similar to the previous test, the variables are then reassigned to the values that are contained in the variables $y, $z, and $x and printed out. The last test contains a variable $x, that is assigned to the value of the vulnerable GET method. Then, based on if the $x variable is set, the variable is reassigned to the value of an empty string. Otherwise, the $x variable, which is now vulnerable, is first printed out and only then is reassigned to an empty string. Finally, the safe empty string is printed out.

As for the context-sensitive tests, there are two test cases involved. The first one starts with an inclusion of an external file, containing a function, id, that simply returns the parameter that is given to it. Then, the vulnerable GET method is assigned to the variable $vuln. The id function is next called twice, feeding it with the parameters $vuln and 1, each. The values that are simultaneously returned from the function are then assigned to the variables, $vuln_copy and $safe, each. Lastly, only the variable $safe is printed out. The second test is similar to the first test, except that the order of the function call is reversed.

For path-sensitive tests, there are two test cases. The first one consists of a variable $cond being assigned the boolean, true. Then, based on if the value of the variable $cond is true, the variable $y is assigned the static string safe and is assigned the value of the vulnerable GET method, otherwise. Based on the same condition, the variable, $y is printed out. The second test is different after the if and else clause, where the $cond variable is reassigned the value of false. Then, prints out $y if $cond is true and prints out an empty string, otherwise.

3.5 Testing Procedure

The Linux-compatible tools (i.e., PHPCS,

phpSAFE, Pixy, RIPS, and WAP) are installed on a 32-bit Ubuntu 16.04 LTS machine with a 1.9GB Memory and a 3.29 GHz Intel Core i5 650 processor. We use PHP version 7.0.29-1+ubuntu16.04.1+deb.sury.org+1, Apache version 2.4.18 server, and Mozilla Firefox version 59.0.2. Whereas, the Windows-compatible tools (i.e., VCG and YASCA) are installed on a 32-bit Windows 7 Enterprise machine with a 2.00 GB Memory and a 2.93 GHz Intel® CoreTM2 Duo processor. We use PHP version 7.1.8 and Apache version 2.4.27 (Win32) server.

Although various tools have different configurations to perform analyses, we choose the tools' default configuration with the following additional settings (if not already set by default): (1) analysis on a single PHP file and, (2) search for XSS vulnerabilities only. We also create a new folder for each test case inclusive of their file includes because; (1) YASCA can only perform analysis on directories, so as to imitate an analysis on a single file and, (2) to avoid bias for the tools that do not implement inter-procedural-based analysis (i.e., TBA tools) as well as for investigating the analysis run time for each tool. Additionally, we configure RIPS' verbosity level to "show secured +1,2" that shows all possible potentially vulnerable function calls. Furthermore, to analyze the analysis run time performance for each tool to complete a detection task per test case, we place a timer (if not already present), that reports the time taken in milliseconds rather than in seconds, at the start and end of the main analysis function of each analysis tool.

To proceed with the experiment, we feed, as inputs, the test cases' file path to the tools in turn. In the first phase, we collect, from each tool, the analysis results for each test case, the detected vulnerable sink, their respective line numbers, and the average analysis time, based on five runs, to complete the analysis.

Next, from these results, we count the total number of detections for each tool for all test cases. Some of the analysis tools, such as YASCA, report vulnerabilities through line numbers, and not the variables used in possible vulnerable sinks. Although this would disregard the ability of the other tools to detect potential harmful variable use in sinks, but for uniformity, we decide to count the number of detections for each tool based on the respective line numbers. We, then, compare the detections to our benchmark from the previous subsection and catalog them on whether they are true vulnerabilities (i.e., true-positives) or benign variables used as parameters to output functions (i.e., false-positives).

4. Analysis and Results

As static analysis evaluates all possible paths, we infer that when the size of the applications increases, so is the time to complete an a an analysis. In continuation, a decent analysis tool is measured based on its capability to precisely detect vulnerabilities within an acceptable time. Following is the discussion of our analysis results.

4.1 Sensitivity Traits

To address our first research question: do the tools' implementation include sensitive-based analysis?, we present the test results in Table 3. The analysis results of a tool's sensitivity traits, based on whether or not it performs inter-procedural, flow-, context- and path-sensitive analyses, is shown in. Table 3. A tool shows sensitivity for the correct vulnerability report (labeled with √) of potentially harmful variables to sinks.

Table 3. Sensitive Decisions For Each Tool

Analysis Tool	Tests for Sensitivity							
	F1	*F2*	*F3*	*F4*	*C1*	*C2*	*P1*	*P2*
PhpSAFE	√	√				√		√
Pixy	√	√	*	√	√	√		
RIPS	√	√	*	√	√	√		√
WAP	√	√			√	√		√

*Tool does not support analysis for object-oriented structure

Most of the tools show flow-sensitivity for *F1* and *F2,* but not for *F3.* phpSAFE sets function return values to `null` for those without a return statement, and consequently, would analyze them as safe. Similarly, WAP maintains the function to be untainted if no return statements exist at the end of the function. Moreover, WAP's operation does not include member access in objects or class instances (e.g., `$y->model`). Alternatively, Pixy and RIPS do not support object-oriented based programming structure, thus, we could assess their sensitivity for this trait. Next, for F4, phpSAFE and WAP again fail by incorrectly report vulnerabilities. This is because they consider the last assignment to the variables used in sinks.

Other than flow sensitivity, implementing path sensitivity in analysis tools is a complex procedure. Table 3 shows that all the tools fail for P1, but pass for P2, except Pixy. Many of the existing SASTs, including Pixy, would combine the output values of conditional control flows to report potential vulnerabilities and, therefore, may not be precise. On the other hand, Pixy fails for P2 because it mainly analyzes whether values originating from input sources are included in the sink of a program. Meanwhile, phpSAFE and WAP both report vulnerabilities for P1 due to the same reason they fail for F4. Interestingly, RIPS incorrectly reports vulnerabilities for P1, but not for F4 because, in the latter test, it would analyze the `if` block first, which would execute `$x=""` and then only analyze the `else` block, which prints out a safe variable.

Lastly, all but phpSAFE show context-sensitivity traits. For optimization and memory consumption purposes, phpSAFE parses functions only once during the function's first call, taking into account the context (parameters, global variables, scope, etc.) of the call. For this reason, it fails for C1. To clarify, because it analyzes the return value of the first call to the `id` function with the parameter `$vuln` as tainted, the next call to the same function with parameter, `1`, would also be analyzed as tainted.

4.2 Effectiveness

To answer the second research question: are the tools' XSS vulnerability detections true or false?, and the third research question: what is the probability of a tool missing true vulnerabilities?, we assess the effectiveness of each tool in detecting XSS vulnerabilities of each test case based on the total counts of: (1) detections, *C(d)*, or reports of potentially vulnerable sinks, (2) true-positives, *C(tp)*, or detected vulnerabilities that are real vulnerable sinks, (3) false-positives, *C(fp)*, or invulnerable sinks that are reported as vulnerable, and (4) false-negatives, *C(fn)*, or missed vulnerabilities.

To compute the probabilities of detection, *P(d)*, false-positive, *P(fp)*, and false-negative, *P(fn)*, for each tool, we use the following formulae:

$$P(d) = C(d) / T(d) \quad (1)$$

$$P(fp) = C(fp) / T(tn) \quad (2)$$

$$P(fn) = C(fn) / T(tp) \quad (3)$$

where, *T(d)* is the total number of possible detections for a test case. The results of the computation are as in the following.

The assessment of effective detections is carried out in reference to the production of two types of false reports: false-positives and false-negatives.

4.2.1 False-positives

Table 4 shows the probabilities of detection, true-positive and false-positive for each tool, the result of which is used to construct a graph similar to a Receiver Operating Characteristic (ROC) curve, that presents the performance of each tool in terms of the probability of detection, *P(d)*, against the false-positive, *P(fp)*, as shown in Fig. 1.

Table 4. The Probabilities of Detection and False-positive of the Tool

Analysis Tool	*P(d)*	*P(fp)*
PHPCS	0.484	0.080
VCG	0.355	0.091
YASCA	0.968	0.453
phpSAFE	0.123	0.105
Pixy	0.200	0.032
RIPS	0.245	0.000
WAP	0.006	0.000

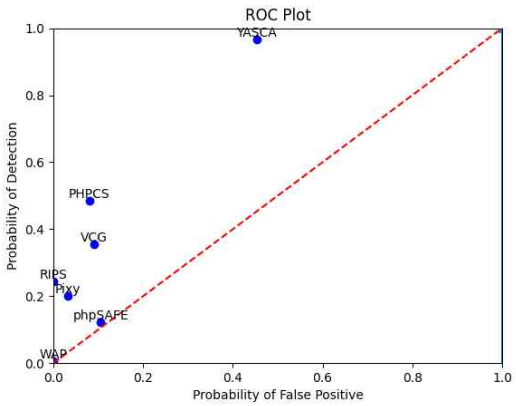

Fig. 1. The Detection Against False-positive Probabilities of the Tools

The diagonal line is the baseline representing a naïve analysis tool that randomly guesses the results of any performance by which *P(d)* = *P(f)* and *P(tp)* = *P(f)*. The target of a decent analysis tool is to either have an operating point in the

upper-left-hand corner, or above the diagonal line in the graph. As the figure shows, all analysis tools satisfy the latter and not the former target. By our observation, the TBA tools show higher detection probabilities compared to that of the GBA tools. YASCA charts the highest false-positive probability among the TBA tools. As for GBA tools, RIPS shows the highest detection probibility with nil false-positives, followed by Pixy, but with a slightly higher false-positive probability. Lastly the detection probability for phpSAFE and WAP are lesser and somewhat proportional to their false-positive probability.

In this part of our study, we analyze if a tool can possibly miss detecting a vulnerability. Table 5 shows the performance of each tool in detecting 82 true vulnerabilities present in all test cases. Interestingly, TBA tools show lower false-negative probabilities (0.00-0.40) compared to GBA tools (0.54-0.99). While, YASCA present nil false-negatives, PHPCS and VCG present a probability of more than 0.15 and less than 0.40, each. Pixy and RIPS of the GBA tools, appear with more than half false-negative probabilities, whereas phpSAFE and WAP both show high false-negative rates, with 0.79 and 0.99, each.

Table 5. Probabilities of the Tools Producing False Negatives

TBA Tools	PHPCS	VCG	YASCA	
P(fn)	0.159	0390	0.000	
GBA Tools	phpSAFE	Pixy	RIPS	WAP
P(fn)	0.792	0.634	0.537	0.988

4.2.2 Precision, Recall and F-Score

To answer the fourth research question: how do the tools perform in terms of precision, recall and F-Score?, we assess the performance of each tool as shown in Table 6. Precision measures exactness, which is the fraction of the detections classified as truly positive, whereas recall measures completeness, which is the fraction of detected valid vulnerabilities against all valid vulnerabilities. The F-score measures the harmonic balance between precision and recall. Below shows the corresponding formulae:

$$Precision = C(tp) / (C(tp) + C(fp)) \quad (4)$$

$$Recall = C(tp) / (C(tp) + C(fn)) \quad (5)$$

$$F = 2 * (Precision * Recall) / (Precision + Recall) \quad (6)$$

Table 6. Measures of F-Score For Each Tool

TBA Tools	PHPCS	VCG	YASCA	
F-score(%)	0.654	0.578	0.692	
GBA Tools	phpSAFE	Pixy	RIPS	WAP
F-score(%)	0.321	0.455	0.510	0.027

Fig. 2 shows a Precision-Recall (PR) space to indicate that all tools, except YASCA, chart an operating point above the diagonal line. For the TBA tools, PHPCS indicates high probability of showing both precision and recall, while VCG shows high precision but with a slighly lower recall, In contrast, YASCA shows high recall but low precision. In the GBA tools, PhpSAFE, Pixy, and RIPS show high precision with less than half the probability of recall. Lastly, WAP shows high

precision but with a low recall. Table 6 shows the F-score for each tool. YASCA shows the highest probability among all tools with about 0.7, while RIPS shows the highest probability (0.5) for GBA tools.

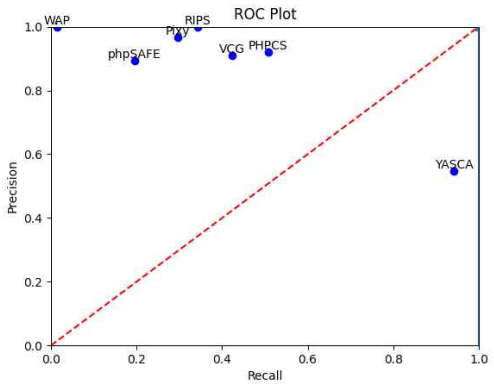

Fig. 2. The Probability of Precision Against Recall For Each Tool

Tools with high precision (i.e., GBA tools) can be used to minimize false alarms in the detection vulnerabilities. Alternatively, tools with high recall (i.e., YASCA) can be used when the detection of all possible vulnerabilities is of importance whereby the developers are willing to manually inspect those that may be benign. For developers that are searching for a tool with balanced precision and recall, we suggest YASCA, PHPCS, or combining TBA with GBA tools for a more optimal result.

4.3 Analysis Runtime

For our fifth research question:how long do the tools take to complete analyses on test cases?., we present the results of the time duration for each tool to complete analyses of the applications in sizes set in a descending order, as shown in Fig. 3.

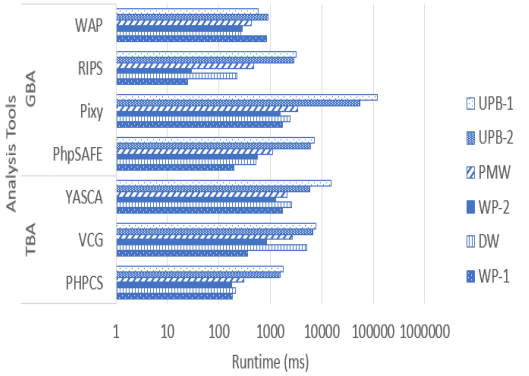

Fig. 3. The Average Analysis Runtime for Each Tool

In the large test case applications, namely, UPB-1, UPB2, and PMW, the TBA tools take an average of less than 10 seconds to complete analyses. Whereas, WAP of the GBA tools, takes less than a second to complete analyses on all test cases, to come up as the fastest tool, regardless of application size. RIPS and phpSAFE, on the other hand, take similar time as TBA tools to complete analyses for large test cases. However, although Pixy, takes an average time to analyze test case PMW, it requires significantly more time with more than a minute to analyze the other two larger test cases.

For the three smaller test cases, WP-2, DW, and WP-1, the TBA tools, VCG and PHPCS, less than a second to complete analyses, while YASCA takes more time with slightly more than a second. As for GBA tools, RIPS takes a

significantly short time to analyze these test cases, except for DW, with less than 100 milliseconds. It takes less than a second for RIPS to analyze test case DW, a similar time taken for WAP, phpSAFE, and the TBA tools to analyze the smaller test cases. Pixy, however, requires more than a second to complete analyses on these test cases.

To sum up, the detection probability of TBA tools are higher than that of GBA tools, but with higher probability of false-positives. In contrast, although GBA tools have a lower probability to produce false-positives, they have higher chances of producing false-negatives. In general, all tools show high precision (except YASCA), but TBA tools show higher recall compared to that of GBA tools. Additionally, text-based tools tend to have a balance between precision and recall compared to GBA tools. Meanwhile, RIPS is the only GBA tool to show harmonic balance between precision and recall. Lastly, the time duration to complete analyses is not dependant on a tool's vulnerability search type. However, all the tools complete analysis in an acceptable time (i.e., within one minute) for most of the test cases.

5. Evaluation and Discussion

Our comparative analysis results show that the relation of analysis tools between their detection probability is somewhat directly proportional to their false-positives probability. However, the detection probability is indirectly proportional to their false-negative rates.

In having a high detection probability, TBA tools would fare better in detecting possible vulnerable sinks. Contrastingly, GBA tools would fare better in detecting sinks that are truly vulnerable.

TBA tools, which typically use regular-expressions to parse data, have the advantage of expressiveness in detecting recurrent patterns and information [27], which may be the reason for their higher probability in vulnerability detection. Meanwhile, the performance for GBA tools are possibly dependent on their sensitivity trait. In having these traits would help them in detecting real vulnerabilities. In turn, their probability of false reports may be caused by imprecise approximation of temporal variable properties or runtime information manipulation or validation [28].

Next, the results also show that most analysis tools' goal is to achieve either precision or recall. This is related to the trade-offs between false-positive and false-negative rates [26]. From Eq. (4) and (5), precision is related to the proportion of detections obtained (i.e., consists of either true-positive or false-positive results) that are classified as truly positive, whereas recall is related to the proportion of all true vulnerabilities (i.e., consists of either true-positive and true-negative results) that are misclassified as invulnerable. A previous study discovered that based on both theoretical and empirical evidence, that, the trade-offs between precision and recall are inherent. An

improvement in precision would decrease the true-positive counts, having a negative effect on recall. At the same time, an improvement to recall would decrease the number of false-positive counts, thus lowering the precision [31].

Additionally, there is an indirect relationship between false-positive and false-negative rates. According to Rice's theorem, which states that any non-trivial property of the language recognized by a Turing machine is undecidable [29]. This suggests that it is impossible to decide any program property without solving the halting problem. Hence, in practice, SAST tools can only perform analysis through approximation, which ramifies that the analyses are, inevitably, either sound but incomplete or complete but unsound. An analysis tool is sound if, given a set of assumptions, it reports all errors, or in our case, vulnerabilities, thus, no false-negatives, but it is possible that the reports are false. An analysis tool is unsound if it tries to reduce false-positives at the cost of producing false-negatives [30]. This suggests that it may not be possible to create a perfect SAST tool, but, it is possible to create a useful one.

However, based on our results, it is also clear that it is not impossible to satisfy both measures, regardless of the types of analysis used, which is the case for RIPS. It has been previously stated that, although trade-off between precision and recall are inevitable [31], it is possible to improve both measures simultaneously [31]. This can be done by performing a multi-stage procedure, in which an improvement of one measure of an analysis can be done by using a retrieved set of a subsequent measure. Belyaev et al. implemented this idea by improving the precision of an analysis by filtering out false-positives after performing one stage of the analysis, thus having no impact on recall [32].

Most TBA tools, except for YASCA, show faster analysis time compared to GBA tools. TBA tools commonly implement regular-expression-based parsing, whereas GBA tools commonly implement context-free grammar-based parsing [33]. Context-free grammar parsing operates recursively, where it repeatedly invokes mechanisms and method calls. This can be expensive in both processor time and memory space. On the contrary, regular-expression parsing operates iteratively and normally occurs within a method so the overhead of repeated calls and extra memory assignment is omitted [27]. This supports our finding for some TBA tools (in exception to PHPSAFE and YASCA) which is able to perform faster compared to GBA tools in detecting XSS vulnerabilities.

Finally, based on the results of sensitivity decisions used by analysis tool authors, there may be a relation between flow- and context-sensitivity with precision and also context-sensitivity with analysis completion time. Several studies have suggested that flow- and context-sensitivity can be used to greatly improve precision [23]. However, our results cannot clearly explain the relationship between path-sensitivity and other measures.

6. Related Work

Past works on the use of open-source static software analysis have viewed on several issues. Torri et al., for example, evaluated open-source static software analysis in embedded softwares and discovered that a majority of the tools are not applicable to embedded systems due to their poor performance in detecting software bugs [39]. We take note of this evaluation as it provides a motivation to our research, whereby there are yet limitations that exist in current static code analysis tools. On the other hand, the work by Zitser et al. was to perform a comprehensive evaluation of SAST tools to detect buffer overflows in the source code. In a way, their use of probabilities of detection and false-positive as a performance measure for five SAST tools and present them in a kind of ROC curve in the work is interesting to us [25]. As their work is closest related to ours in evaluating SAST tools, we used the same data model in our research. The significance of our work from theirs is that we evaluate SAST tools that detect XSS rather than buffer overflow vulnerabilities and extend our performance measure to include precision, recall, and analysis runtime.

In yet another study, Baset and Denning reviewed the use of IDE plugins to detect security vulnerabilities in codes and found that most plugins lack information on vulnerability checks that may degrade their detection accuracy. They found that most plugins lack information on vulnerability checks and may degrade their detection accuracy is information of importance to our reference in our use of analysis [40]. Suto analyzed the accuracy and the runtime of web application scanners in detecting vulnerabilities [41]. His work is indeed helpful to our task of looking at the same issue in SAST tools that are capable of detecting XSS vulnerabilities in this study. The work of Alsaleh et al. that presented a comparative evaluation of on web application vulnerability scanners to assess their security features and performance does provide us with a key to our attempt at evaluating SAST tools. Nevertheless, we conducted an evaluation of the same key features of SAST tools rather than web scanners [42], specifically those that detect XSS vulnerabilities.

Antunes et al. compare the effectiveness of penetration testing tools and static code analysis tools to detect SQL injection vulnerabilities. Their results show that the latter outperforms the former in terms of vulnerability coverage [42]. Their following work surveys the challenges in vulnerability detection tools in Service-Oriented Architectures (SAOs). This study aims to provide service providers with potential ways to improve the detection of security vulnerabilities in SOAs using effective methodologies and tools [44]. Their work is helpful to our study of achieving a similar aim which is to help prevent XSS vulnerabilities.

7. Conclusion and Future Work

This paper presents our assessment of the performance of analysis tools in terms of their effectiveness to detect XSS vulnerabilities as well as their total time to complete an analysis on six test cases. We observed that PHPCS showed the highest detection counts of more than 50% of the total vulnerabilities. In the nutshell, its detection drew on 60% true-positive and about 20% false-positives. In terms of effectiveness, PHPCS recorded 80% precision and 50% recall, while YASCA recorded the highest measure of F-Score. Notably, text-based analysis tools took a shorter time to complete analyses compared to graph-based analysis tools. The limitations to our research that can be conveyed as future work are (1) we did not include evaluations for analysis tools that can correct vulnerable programs and, (2) we did not include evaluations for the type of injection contexts the analysis tools are able to detect. Our evaluation of SAST tools in detecting XSS vulnerabilities should provide more insight, especially to developers to help make improvements for a more effective and fast analysis tool.

References

[1] "Vulnerability distribution of cve security vulnerabilities by types", https://www.cvedetails.com/vulnerabilities-by-types.php, Accessed: Oct. 10, 2017.

[2] "CVE - Search Results", https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=XSS, Accessed: Oct. 10, 2017.

[6] OWASP, "Source Code Analysis Tools - OWASP", URL: https://www.owasp.org/index.php/Source_Code_Analysis_Tools, Accessed: Feb. 24, 2018.

[8] Floe, "Phpcs-security-audit", URL: https://github.com/FloeDesignTechnologies/phpcs-security-audit,

[9] Bob, "CodeSniffer Part 4: How does CodeSniffer Work | King Kludge", URL: http://www.kingkludge.net/2009/02/codesniffer-part-4-how-does-codesniffer-work/, Accessed: Feb. 19, 2018.

[10] Paulo Nunes, José Fonseca, and Marco Vieira, "PhpSAFE: A Security Analysis Tool for OOP Web Application Plugins", Proc. Int. Conf. Dependable Syst. Networks, vol. 2015-Septe, pp. 299-306, 2015. DOI: http://doi.org/10.1109/DSN.2015.16

[11] Nenad Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities", in 2006 IEEE Symp. Secur. Priv., 2006, pp. 6 pp. - 263. DOI: http://doi.org/10.1109/SP.2006.29

[12] Johannes Dahse, "RIPS-A static source code analyser for vulnerabilities in PHP scripts", Retrieved Febr., vol. 28, p. 2012, 2010.URL: http://www.nds.rub.de/media/nds/attachments/files/2010/09/rips-paper.pdf

[13] Nick Dunn and John Murray, "Visual Code Grepper".URL: https://github.com/nccgroup/VCG

[14] Ibéria Medeiros, Nuno F. Neves, and Miguel Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives", in Proc. 23rd Int. Conf. World wide web - WWW '14, 2014, pp. 63-74. DOI: http://doi.org/10.1145/2566486.2568024

[15] Michael V. Scovetta, "Yasca: Yet Another Source Code Analyzer".URL: http://scovetta.github.io/yasca/

[16] "PMD", URL: https://pmd.github.io/, Accessed: Feb. 19, 2018.
[17] Jakob Kallin and Irene Lobo Valbuena, "Excess XSS: A comprehensive tutorial on cross-site scripting", URL: https://excess-xss.com/, Accessed: Mar. 22, 2017.
[18] Andreas Gohr and DokuWiki, "DokuWiki", URL: https://github.com/splitbrain/dokuwiki
[19] "PHPMyWind",URL: http://phpmywind.com/
[20] PHP Outburst, "Ultimate PHP Board". URL: https://github.com/PHP-Outburst/MyUPB
[21] Bobcares, "Gift Certificate Creator", URL: https://wordpress.org/plugins/gift-certificate-creator/,
[22] Robot with Emotions, "Slideshow Gallery Pro - WordPress Plugins", URL: https://wordpress.org/plugins/slideshow-gallery-pro/, Accessed: Feb. 13, 2018.
[23] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools", 2006. ISBN: 978-0321486813, 2006.
[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin, "Principles of Program Analysis", Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. DOI: http://doi.org/10.1007/978-3-662-03811-6
[25] Misha Zitser, Richard Lippmann, and Tim Leek, "Testing static analysis tools using exploitable buffer overflows from open source code", ACM SIGSOFT Softw. Eng. Notes, vol. 29, no. 6, p. 97, 2004. DOI: http://doi.org/10.1145/1041685.1029911
[26] Nurul. Atiqah. A. Talib and Kyung-Goo Doh, "Assessment of dynamic open-source cross-site scripting filters for web application", KSII Trans. Internet Inf. Syst., vol. 15, no. 10, pp. 3750-3770, 2021. DOI: http://doi.org/10.3837/tiis.2021.10.015
[27] Davide Pasetto, Fabrizio Petrini, and Virat Agarwal, "Tools for very fast regular expression matching", Computer (Long. Beach. Calif)., vol. 43, no. 3, pp. 50-58, 2010. DOI: http://doi.org/10.1109/MC.2010.80
[28] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo, "Securing web application code by static analysis and runtime protection", Proc. 13th Conf. World Wide Web - WWW '04, p. 40, 2004. DOI: http://doi.org/10.1145/988672.988679
[29] H. G. Rice, "Classes of recursively enumerable sets and their decision problems", Trans. Am. Math. Soc., vol. 74, no. 2, pp. 358-358, 1953. DOI: http://doi.org/10.1090/S0002-9947-1953-0053041-6
[30] Brian V. Chess and Gary E. McGraw, "Static analysis for security", IEEE Secur. Priv., vol. 2, no. 6, pp. 76-79, 2004. DOI: http://doi.org/10.1109/MSP.2004.111
[31] Michael Buckland and Fredric Gey, "The relationship between Recall and Precision", J. Am. Soc. Inf. Sci., vol. 45, no. 1, pp. 12-19, Jan. 1994. DOI: http://doi.org/10.1002/(SICI)1097-4571(199401)45:1<12::AID-ASI2>3.0.CO;2-L
[32] Mikhail Belyaev and Vladimir Itsykson, "Fast and Safe Concrete Code Execution for Reinforcing Static Analysis and Verification", Model. Anal. Inf. Syst., vol. 22, no. 6, pp. 763-772, Jan. 2016. DOI: http://doi.org/10.18255/1818-1015-2015-6-763-772
[33] Görel Hedin, "Compiler Construction", vol. 9031, 2015. DOI: http://doi.org/10.1007/978-3-662-46663-6
[39] Lucas Torri, Guilherme Fachini, et al., "An evaluation of free/open source static analysis tools applied to embedded software", in 2010 11th Lat. Am. Test Work., Mar. 2010, pp. 1-6. DOI: http://doi.org/10.1109/LATW.2010.5550368
[40] Aniqua Z. Baset and Tamara Denning, "IDE Plugins for Detecting Input-Validation Vulnerabilities", 2017. DOI: http://doi.org/10.1109/SPW.2017.37
[41] Larry Suto, "Analyzing the Accuracy and Time Costs of Web Application Security

Scanners", 2010. Accessed: Sep. 22, 2017. URL: https://www.beyondtrust.com/wp-content/uploads/Analyzing-the-Accuracy-and-Time-Costs-of-Web-Application-Security-Scanners.pdf

[42] Mansour Alsaleh, Noura Alomar, Monirah Alshreef, Abdulrahman Alarifi, and AbdulMalik Al-Salman, "Performance-Based Comparative Assessment of Open Source Web Vulnerability Scanners", Secur. Commun. Networks, vol. 2017, pp. 1-14, 2017. DOI: http://doi.org/10.1155/2017/6158107

[43] Nuno Antunes and Marco Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services", in 2009 15th IEEE Pacific Rim Int. Symp. Dependable Comput., Nov. 2009, pp. 301-306. DOI: http://doi.org/10.1109/PRDC.2009.54

[44] Nuno Antunes and Marco Vieira, "Security Testing in SOAs: Techniques and Tools BT - Innovative Technologies for Dependable OTS-Based Critical Systems: Challenges and Achievements of the CRITICAL STEP Project", D. Cotroneo, Ed. Milano: Springer Milan, 2013, pp. 159-174. DOI: http://doi.org/10.1007/978-88-470-2772-5_12

Authors

Nurul Atiqah Abu Talib

2009-2013 BIT (Hons) in Computer System Security from Universiti Kuala Lumpur, Malaysian Institute of Information Technology (MIIT), Malaysia

2013.9-present Ph.D candidate in the Department of Computer Science and Engineering at Hanyang University ERICA, Gyeonggi-do, South Korea

<Research interests> Web Security, Machine Learning, Program Analysis

Kyung-Goo Doh

1980 B.S. degree, Industrial Engineering from Hanyang University

1987 M.S. degree, Computer Science from Iowa State University

1992 Ph.D. degree in Computer Science from Kansas State University

1993-1995 Assistant Professor, University of Aizu, Japan

present Professor, Department of Computer Science, Hanyang University ERICA

<Research interests> Programming Languages, Program Analysis, Software Engineering, Software Security