Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools

Areej Algaith[1], Paulo Nunes[2], José Fonseca[2], Ilir Gashi[1], Marco Vieira[3]
[1]Centre for Software Reliability, City, University of London, UK
[2]CISUC, University of Coimbra, UDI, Polytechnic Institute of Guarda, Portugal
[3]University of Coimbra, Portugal
areej.algaith.1@city.ac.uk; pnunes@ipg.pt; josefonseca@ipg.pt; ilir.gashi.1@city.ac.uk; mvieira@dei.uc.pt

***Abstract*—The use of Static Analysis Tools (SATs) is mandatory when developing secure software and searching for vulnerabilities in legacy software. However, the performance of the various SATs concerning the detection of vulnerabilities and false alarm rate is usually unknown and depends on many factors. The simultaneous use of several tools should increase the detection capabilities, but also the number of false alarms. In this paper, we study the problem of combining several SATs to best meet the developer needs. We present results of analyzing the performance of diverse static analysis tools, based on a previously published dataset that resulted from the use of five diverse SATs to find two types of vulnerabilities, namely SQL Injections (SQLi) and Cross-Site Scripting (XSS), in 132 plugins of the WordPress Content Management System (CMS). We present the results based on well-established measures for binary classifiers, namely sensitivity and specificity for all possible diverse combinations that can be constructed using these 5 SAT tools. We then provide empirically supported guidance on which combinations of SAT tools provide the most benefits for detecting vulnerabilities with low false positive rates.**

Keywords— diversity analysis; security analysis; quantitative assessment; static analysis tools

I. INTRODUCTION

Static analysis tools are used to inspect software looking for vulnerabilities, without executing the code. Since they can cover all the source code effectively, they are a valuable tool to help security researchers to automate the task of vulnerability discovery. However, as with any other binary decision system, SATs also suffer from false negative (FN) errors (missing vulnerabilities in the code they inspect) and false positive (FP) errors (incorrectly labelling code as containing a vulnerability when in fact it does not).

There are many SATs available, and each one has its own strengths and weaknesses. Rather than using just one tool, several diverse SATs can be used for finding vulnerabilities to reduce the probability of vulnerabilities remaining undetected. However, for diversity to be effective, the SATs should be diverse in their design. This way, a vulnerability undetected by one SAT should, with high probability, be detected by another one, while at the same time not increasing prohibitively the number of false positives.

The important questions are whether a specific set of SATs would improve vulnerability detection more than another set; quantifying these gains; and quantifying the false positives.

In this paper, we provide empirical results to help with the problem of deciding which combination of SATs to use. We present results of analyzing the performance of diverse SAT configurations based on a previously published dataset by three of the authors of this paper [19]. The dataset consists of five SATs that were individually used to find two types of vulnerabilities, namely SQL Injection (SQLi) and Cross-Site Scripting (XSS), in 134 plugins of the WordPress Content Management System (CMS). WordPress powers 30% of the web and represents 60% of all CMSs [1]. According to the Hacked Website Report, WordPress is the most infected CMS [9]: it accounted for 74% of all CMS infections in Q3 of 2016, and 83% of all CMS infections in 2017.

In this paper, we study all the possible diverse configurations that we can build with the five individual SATs: 10 diverse pairs, 10 diverse triplets, five diverse quadruples and one diverse quintet SAT system. We considered various configurations for the adjudicator: 1-out-of-N (raise an alarm for a vulnerability when any of N SATs in the diverse configuration does so); N-out-of-N (raise an alarm for a vulnerability only when all N SATs in the diverse configuration do so); and simple majority (raise an alarm for a vulnerability when the majority of the N SATs in a diverse configuration do so).

Results are presented using the well-established measures for binary classifiers: sensitivity (measures the performance of the SAT to find vulnerabilities) and specificity (measures the performance of the SAT to not raise false alarms). These measures capture well the main requirements of practitioners when selecting SATs: a tool that finds most vulnerabilities without raising too many false alarms.

In summary, the contributions of the paper are as follows:

- We analyzed the measures for all possible two-, three-, four- and five-SAT diverse configurations. We found that none of the SATs, or combinations of SATs, was able to find all the vulnerabilities in the target plugins. But, we found that some of the SATs exhibit considerable diversity in their ability to detect the vulnerabilities analyzed.
- We provide empirically supported guidance on which combination of SATs provide the most benefits in the ability to detect vulnerabilities, with a reduced false positive rate. Hence, this paper provides a significant new contribution compared with our previous work [19] on which we only analyzed 1-out-of-N configurations. 1-out-of-N systems raise an alarm as long as any one of the SATs in the system raises an alarm.
- One limitation of these configuration is the potential increase of FPs, which may be unacceptable in many situations. In the present work, we look at all the possible N-out-of-N and majority voting configurations. This way a security researcher has more evidence on the interplay between FPs and FNs in diverse SAT configurations.

The rest of the paper is organized as follows: Section II introduces background and related work. Section III describes

the dataset used to perform the diversity analysis. Section IV outlines the analysis methodology. Section V provides the main results of the diversity analysis. Finally, Section VI presents the conclusions, limitations and provisions for further work.

II. Background Related Work

Information is one of the most important assets in almost all organizations. Information security vulnerabilities are weaknesses that expose an organization to risk. The number of vulnerabilities in applications is increasing and the attackers are exploiting them faster than ever before [14]. Thus, security researchers need to be proactive about finding and removing vulnerabilities before attackers can find and exploit them.

The OWASP provides the top ten most critical web application security risks. SQL Injection (SQLi) and Cross-Site Scripting (XSS) vulnerabilities are in the list, and are particularly damaging [10], [5]. An SQLi vulnerability occurs when untrusted data flowing from *entry points* (EPs, e.g., user input) with inadequate validation (i.e., input validation: analyze the data against a predefined pattern; sanitizing: cleaning, filtering input data; escaping: stripping out unwanted data) [2] is used for constructing SQL queries. An attacker may explore these data flows and execute queries not expected by the application developer or may access sensitive data without proper authorization [5]. These vulnerabilities occur whenever data input coming into applications from untrusted EPs is not validated, sanitized or escaped and flows through the application reaching *sensitive sinks* (SSs) (see Fig. 1). A SS is a call of a function that exposes private data to external systems. An example for SQLi is the PHP `mysql_query` function, which executes a SQL query and returns the results. The PHP `print` function that outputs HTML/JavaScript to the browser is an example of a SS for XSS. XSS vulnerabilities occur whenever an application includes untrusted data in a new web page without proper validation, sanitization or escaping [5]. The attacker can exploit the vulnerabilities by injecting malicious scripts in the new web page. The browser renders the page and executes the scripts in the victim's machine as a trusted script which can hijack user sessions, deface web sites, or redirect the user to malicious sites [5].

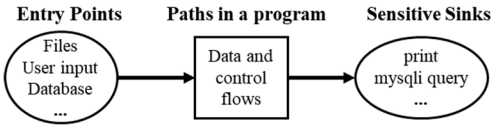

Figure 1 – Data flow vulnerabilities.

A. Examples of SQLi and XSS vulnerabilities

To explain how SQLi vulnerabilities occur, and how they can be exploited, we use a PHP script example (Fig. 2). The script inserts contacts data (name and phone) in a database without any validation. In this script, there are two EPs, (`$_POST` array at lines 1 and 2) and one SS (line 4). The data flowing from the EPs to the SS are not validated, so there is one SQLi vulnerability in line 4.

```
1 $name  = $_POST['name'];                                    EP
2 $phone = $_POST['phone'];                                   EP
3 $sql = "INSERT INTO Contacts (name, phone)
          VALUES ('$name', '$phone')";
4 $result = mysqli_query($connection, $sql);                  SS
```

Figure 2 –PHP code for inserting contacts in a database.

The PHP script in Fig. 3 shows an example of a XSS vulnerability. It searches contacts by name in the database and displays the results in an HTML page. The user provides the name (EP, line 1) to be searched through the `$_GET` array parameter. The script outputs the value of the parameter without any proper escaping (line 2). In this case, there is one *reflected* or *first order* XSS vulnerability. In reflected XSS the untrusted data coming from the user is immediately written back. The exploitation of this class of vulnerabilities requires some kind of social engineering by the attacker to convince the victim to click in the crafted URL. In line 3, the script makes use of the same parameter (`$name`) to build the SQL query to be sent to the database server (line 4). In this case, there is also one SQLi vulnerability. Similarly to the data flowing from the user input, the database is also a source of untrusted data due to inappropriate validation when inserted in the database, as shown in Fig. 2. In fact, any attacker can insert in the database malicious code instead of a valid contact name. Therefore, the PHP statement in line 7 is an EP in the application that retrieves untrusted data from the database. These data are outputted in lines 9 and 10 without any escaping, hence two *stored* or *second order* XSS vulnerabilities exist. This class of vulnerabilities is especially dangerous because it does not require any kind of social engineering to trick the victim and a single malicious code stored in the database can be executed in the browser of all users visiting the website.

```
1 $name = $_GET['name'];                                      EP
2 print("<h1>Your search for: $name</h1>");                    SS
3 $sql="SELECT * FROM Contacts where name like '%$name%'
4 $result=mysqli_query($connection, $sql);                    SS
5 echo '<table><tr><th>Name</th><th>Phone</th></tr>';
6 $n=0;
7 while($row = mysqli_fetch_array($result)) {                 EP
8   echo '<tr>';
9   echo '<td>' . $row[name] . '</td>';                       SS
10  echo "<td>{$row['phone']}</td>";                          SS
11  echo '<tr>'; $n++;
12 }
13 printf("Total records: %d", $n);                           SS
```

Figure 3–PHP code for searching contacts in a database.

For a given class of vulnerability one *line of code* (LOC) is potentially vulnerable if it contains an SS function call with at least one parameter [19]. A *vulnerable line of code* (VLOC) is a LOC with a SS and a variable with data coming from EPs without any validation. A *non-vulnerable line of code* (NVLOC) is a LOC with a SS where all variables are sanitized [13][21][7]. Lines 2, 4, 9 and 10 of the script in Fig. 3 are examples of VLOCs and line 13 is an example of a NVLOC.

B. Related Work

Rutar et al. [20] studied five well-known SATs on a small set of Java programs with different sizes and from various domains. They concluded that the results of each tool are highly correlated with the techniques used for finding bugs, and that no single tool can be considered the best to detect defects. They proposed a meta-tool to automatically combine and correlate SATs' outputs. This meta-tool is based on a set of scripts that combine the results of the various tools in a common format. The bugs found were not manually reviewed, thus, there is no distinction between True Positives (TP) and False Positives (FP). The metric used to evaluate and compare the tools was the number of bugs that each SAT found.

Meng et al. [17] proposed an approach to merge the results of multiple SATs. The user specifies the programs to be analyzed and chooses the classes of bugs to be scanned. After determining which tools can search for the specified class of bugs, the authors generated the necessary tools'

configurations, ran the tools, combined the outputs in a single report, and applied two prioritizing policies to rank the results. This approach has been used to shown that developers could benefit by using more than one SAT. However, SAT outputs were not classified as TP and FP.

Wang et al. [22] proposed an approach that combines multiple SATs in a simple Web Service. The user has the possibility to upload the source code and auxiliary information such as the programming language and the classes of bugs to be scanned. The tools perform the analysis of the source code and the results are merged in a way that the same defect is reported only once. The experiments had just a single Java test case, and the approach was evaluated in terms of the running time when combining two SATs, lacking the validation of the effectiveness of the vulnerability detection.rti

The NSA CAS specified a methodology, the CAS Static Analysis Tool Study Methodology, that measures and rates the effectiveness of SATs and combinations of SATs in a standard and repeatable manner [15]. The metrics used are precision, Recall, F-Score, and Discrimination Rate (DR). A discrimination occurs if a SAT reports a vulnerability in a vulnerable test case (TP) and keeps quiet in a non-vulnerable test case (TN). The CAS created a collection over 81,000 synthetic C/C++ and Java programs with known flaws, which was called the Juliet Test Suite [4]. Each test case is a slice of code having exactly one flaw and at least one non-flaw construct similar to the vulnerability. In 2011, the CAS conducted a study with the purpose of determining the capabilities of five SATs for C/C++ and Java [3]. In this study, they proposed the combination of two SATs to show that adding a second SAT might complement the first one. However, the evaluation of the combinations is limited because it is based on the Recall and DR metrics. The problem is that Recall does not consider the number of FP reported, and DR severely penalizes the SATs that report both many vulnerabilities and many FP. Also, from all possible SAT combinations, they only tested five, which is an important limitation of the study.

III. DATASET

A standard way to evaluate and compare the effectiveness of SATs is to make them search for vulnerabilities in a set of applications (i.e., the workload), followed by the computation of the evaluation metrics. The workload strongly determines the results, so it should be representative of all applications. Unfortunately, this is very hard to attain. To make the problem treatable, the workload can be built for a particular domain. However, the selection of a set of representative applications in a given domain is still a difficult task. Another difficulty is the characterization of the applications in the workload (especially if they are real applications) concerning the vulnerable (VLOC) and non-vulnerable (NVLOC) lines of code (LOC). Moreover, the computation of several evaluation metrics requires the outputs of all the tools to be classified into FP, FN (False Negatives), TP and TN (True Negatives). To compare the results of two or more SATs we need their outputs to be in a common format with detailed data about the vulnerabilities such as the LOC, the SS, the vulnerable variables, the chains of data/control dependencies of the vulnerable variables from the EPs to the SS to prove that the user input reaches the SS. Unfortunately, SATs report the vulnerabilities they find in different formats with varying degrees of detail. For example, some SATs report data in HTML pages and others in a GUI. Although these data are human readable, they need to be converted to a common format. To accomplish this in a seamlessly way we developed a tool able to automate the process.

In our work, we used a workload developed by three of the authors of the current paper in a previous work [19]. In that work, we proposed an approach to select applications based on public repositories of vulnerabilities that include confirmed vulnerabilities in real software. We applied this methodology to the domain of WordPress plugins and for SQLi and XSS vulnerabilities based on the online WPScan Vulnerability Database (WPVD) [6]. The workload is a set of 134 plugins composed of 4,975 PHP files, 1,339,427 LOC and where each plugin has at least one SQLi and/or one XSS VLOC (i.e., a LOC with at least one vulnerable SS). As identifying all VLOCs and NVLOCs (i.e., a LOC with all SSs non-vulnerable) in the workload is a hard task that requires a thorough review by security experts, our approach to find more VLOCs than those present in the WPVD was based on searching for further vulnerabilities in the workload with one or more SATs, followed by a manual review to confirm if they are TPs or FPs. The merge of all TPs with the vulnerabilities of the WPVD becomes the list of VLOCs in the workload (which is, nevertheless, a best-effort subset of all of them). Therefore, the list of NVLOCs is obtained from all LOCs with a SS with at least one variable, excluding those that were reported by the tools and confirmed manually as TP.

To detect the SQLi and XSS vulnerabilities in the plugins, the following five SATs were used: RIPS v0.55 [8], Pixy v3.03(2007) [12], phpSAFE [18], WAP v2.0.1 [16], and WeVerca v20150804 [11]. RIPS performs static taint analysis and string analysis. RIPS and Pixy are two of the most referenced PHP SATs in the literature, but they are not ready for Object Oriented Programming (OOP) analysis. RIPS has been developed as open source until 2014, and only its recently released commercial version is able to fully analyze OOP code. WAP, phpSAFE, and WeVerca are recent tools under active development and they are prepared for OOP code.

Static analysis is a complex task, and the tools may be unable to fully process some files of the workload. Overall, phpSAFE was unable to analyze 130 files, RIPS could not analyze 2179 files, Pixy did not process 1473 files, and WeVerca was not able to analyze a total of 20 files. To make the analysis comparable we consider only the results obtained from files that could be successfully analyzed by all five tools.

Overall, the plugins contain 713,456 LOC and 402,218 logical LOC (LLOC, i.e. commented and whitespace lines are excluded), as can be seen in Table I. The counting of the LOC and LLOC was performed using the *phploc* tool (https://phar.phpunit.de/phploc.phar). Since programming orientation may be relevant for the performance of the SATs, Table I also shows the LLOCs that have POP (Procedure Oriented Programming) code and those that have OOP (Object Oriented Programming) code.

TABLE I. PLUGIN INFORMATION.

SQLi				XSS			
		LLOC				LLOC	
Plug.	**Files**	**POP**	**OOP**	**Plug.**	**Files**	**POP**	**OOP**
117	2168	120917	46617	130	3401	175747	58937

Table II shows the VLOCs and NVLOCs for SQLi and XSS for the workload in Table I. In this, we can see more XSS than SQLi data, but that is also usual in real life applications.

TABLE II. DATASET: COUNTS AND PERCENTAGES (IN BRACKETS)

Vulnerability	Code Type	VLOC	NVLOC	Total
SQLi	POP	138 (18.6)	605 (81.4)	743
	OOP	509 (8.4)	5574 (91.6)	6083
	Total	**647 (9.5)**	**6179 (90.5)**	**6826**
XSS	POP	965 (41.3)	1370 (58.7)	2335
	OOP	3384 (15.4)	18525 (84.6)	21909
	Total	**4349 (17.9)**	**19895 (82.1)**	**24244**

IV. ANALYSIS METHODOLOGY

We can classify the decisions of a SAT into four classes (same as for any other binary decision system):

- For code that is not vulnerable:
  - *False Positive (FP)*: the SAT incorrectly determines that the code is vulnerable;
  - *True Negative (TN)*: the SAT correctly determines that the code is not vulnerable.
- For code that is vulnerable:
  - *False Negative (FN)*: the SAT incorrectly determines that the code is not vulnerable;
  - *True Positive (TP)*: the SAT correctly determines that the code is vulnerable.

The conventional statistical measures for the performance of a binary classification test that we have used, sensitivity and specificity, are summarized in Table III. There are many other measures available, however, we used these as they are the most widely used in literature on decision systems. Other measures can be derived either from these or directly from the FN, FP, TN, and TP counts.

TABLE III. THE FORMULAS AND DEFINITIONS FOR SENSITIVITY, AND SPECIFICITY MEASURES

Statistical Measure	Equation	Definition
Sensitivity (True Positive Rate TPR)	TP / (TP+FN)	The rate of detecting a vulnerability
Specificity SPC (True Negative Rate TNR)	TN / (TN+FP)	The rate of remaining silent when no vulnerability exists

Table IV presents the five SATs, the labels (in brackets) we use to refer to them in the rest of the paper, and the FP, TN, FN and TP counts respectively, for each of the two classes of vulnerabilities. Table V presents the Sensitivity, and Specificity measures for each SAT.

TABLE IV. THE 5 SATS AND THE FP, TN, FN AND TP COUNTS

SAT	SQLi				XSS			
	FP	TN	FN	TP	FP	TN	FN	TP
phpSAFE (A)	53	6126	268	379	213	19682	2293	2056
RIPS (B)	116	6063	465	182	454	19441	1469	2880
WAP (C)	0	6179	492	155	25	19870	3964	385
Pixy (D)	31	6148	583	64	172	19723	3313	1036
WeVerca (E)	4	6175	608	39	24	19871	3488	861

TABLE V. THE 5 SATS AND THE SENSITIVITY (SENS.) AND SPECIFICITY (SPEC.) MEASURES FOR EACH SAT

SAT	SQLi		XSS	
	Sens.	Spec.	Sens.	Spec.
phpSAFE (A)	0.586	0.991	0.473	0.989
RIPS (B)	0.281	0.981	0.662	0.977
WAP (C)	0.240	1.000	0.089	0.999
Pixy (D)	0.099	0.995	0.238	0.991
WeVerca (E)	0.060	0.999	0.198	0.999

In the present paper, we extend the analysis of [19] from the viewpoint of diversity. From the 5 SAT configurations, we can build a total of:

- 10 two-version combinations ($^5C_2$),
- 10 three-version combinations ($^5C_3$),
- 5 four-version combinations ($^5C_4$), and
- 1 five-version combination ($^5C_5$).

When using diverse SAT systems, the decision on whether to flag code as vulnerable or not depends on the adjudication of outputs from the individual SATs. We looked at three common configurations for adjudication:

- 1-out-of-N (abbreviated 1ooN): code is labelled as vulnerable as long as any one of N SATs has labelled it as vulnerable;
- Majority voting: code is labelled as vulnerable as long as the majority of SATs in a given configuration of N SATs (e.g. 2 out of 3, 3 out of 4, 3 out of 5 etc.) have labelled it as vulnerable;
- N-out-of-N (abbreviated NooN): code is labelled as vulnerable only if all N SATs in a given configuration of N SATs label the code as vulnerable.

Decision makers also frequently use the Receiver Operating Characteristic (ROC) curves for each system of interest when analyzing the decisions of a binary classifier. ROC curves are used to determine how a *threshold* should be set for a decision system to get an optimal configuration that maximizes the TP and minimizes the FP rates (what is "optimal" for a give system will inevitably depend on the relative cost that the decision maker assigns to the FP and FN failures). However, since the systems in our case are already pre-configured, the ROC plots show only a point for each system. By showing all the points for the single and diverse systems in the same plot, we can visualize which systems are configured most optimally for a given application.

In summary, in our analysis we performed the following steps for each application in the workload:

- We calculated the FP, FN, TP, and TN counts for each diverse configuration;
- We calculated the measures of interest (see Table III) for each diverse configuration;
- We generated the ROC plots showing all the diverse configurations and the individual SATs; and
- We calculated the differences in the measures of interest between diverse configurations and individual systems to measure the possible improvements or deteriorations from switching to a diverse system.

V. RESULTS

In this section, we present the results of our analysis of the diversity in the SAT tools.

A. Visualising Diversity

We begin our analysis with a simple visualization that shows the commonality and diversity of the tools on the vulnerable and non-vulnerable code. Figure 4 contains these plots for the two classes of vulnerabilities. We will use figure 1(a) to illustrate what each plot shows:

- The x-axis lists the five SAT tools;
- The y-axis lists the VLOCs (647 in total for SQLi);
- A green cell in the plot shows for each SAT whether they detected the vulnerable code as such (i.e. the green cells represent true alarms (TPs); the white cells represent no alarms (in this case, FNs)).

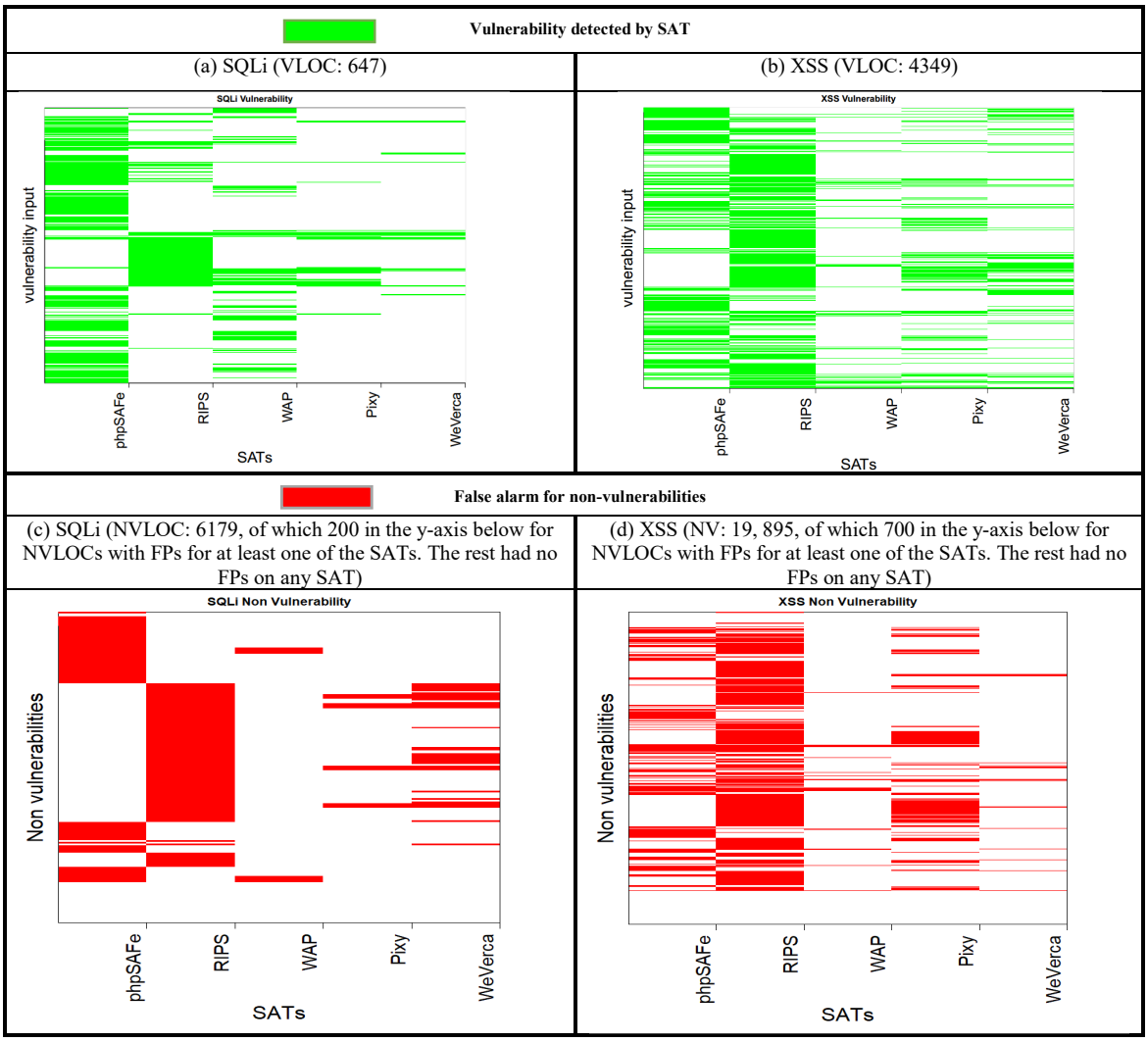

Figure 4 - Diversity between SATs for SQL Injection and Cross Site Scripting (XSS)

Figure 4(b) is the same but for the XSS vulnerabilities. Figures 4(c) and 4(d) are similar but in these plots we visualize the responses from SATs on code that was not vulnerable – NVLOC (hence an alarm is a false positive (FP) represented by a red colored cell; no alarms are again represented as white cells (in this case they are TNs)) for the SQLi and XSS vulnerabilities, respectively. In 4(c) and 4(d) we show only the NVLOCs on which at least one of the SATs reports an FP.

From these plots, we can observe that there is noticeable diversity between some of the SATs (e.g. considerable diversity for both SQLi and XSS between phpSAFE and RIPS, as is evident by the limited overlap in their alarms in the graphs).

B. Sensitivity, Specificity and ROCs for Diverse SATs

We then proceeded to calculate the sensitivity and specificity for each of the diverse combinations with the five SATs, for the three types of adjudication setups considered (namely 1ooN, simple Majority vote (2oo3, 3oo4 and 3oo5) and NooN). Table VI presents the results of this analysis for all the possible two-version, three-version, four-version and five version combinations for SQLi. Table VII shows the results for XSS.

From Tables VI and VII we can see some patterns emerging: the 1ooN systems are better at finding vulnerabilities (better sensitivity), compared with the best individual SATs; on the other hand, NooN systems are better at correctly labelling non-vulnerable code (higher specificity). This is to be expected since:

- 1ooN systems will in *all* cases perform:
  - *better or equal* to the best single SAT in the diverse combination N for *vulnerable code*, as any "alarm" from any of the N SATs systems will lead to an alarm in a 1ooN system;
  - *equal or worse* than the worst single SAT in the diverse combination N for *non-vulnerable code*, as any "alarm" from any single SAT will lead to this code being incorrectly labelled as vulnerable.
- NooN systems will in *all* cases perform:
  - *better or equal* to the best single SAT for *non-vulnerable code* as the NooN system only raises an "alarm" for non-vulnerable code if ALL the SATs in the diverse configuration do so;
  - *equal or worse* than the worst single SAT system in the diverse configuration N for *vulnerable code*, as the NooN system will only label code as vulnerable if ALL the single SATs in the diverse configuration do so.
- Majority voting setups usually balance out these extremes, as they are not as "trigger happy" as 1ooN setups in raising alarms, but also not as conservative as NooN setups in remaining silent.

What is important to understand is *how much* better, or *how much worse*, would a diverse configuration perform in these setups, and the results in Tables VI and VII provide us with some interesting observations.

TABLE VI. SENSITIVITY (SENS.) AND SPECIFICITY (SPEC) FOR THE 1OON, MAJORITY VOTE AND NOON CONFIGURATIONS FOR N BETWEEN 2 AND 5 FOR SQLI

SQLi	1ooN		Majority		NooN	
	Sens.	Spec.	Sens.	Spec.	Sens.	Spec.
(a, b)	0.782	0.976	-	-	0.085	0.996
(a, c)	0.770	0.991	-	-	0.056	1.000
(a, d)	0.655	0.987	-	-	0.029	0.999
(a, e)	0.624	0.991	-	-	0.022	1.000
(b, c)	0.444	0.981	-	-	0.077	1.000
(b, d)	0.289	0.981	-	-	0.091	0.995
(b, e)	0.297	0.981	-	-	0.045	0.999
(c, d)	0.280	0.995	-	-	0.059	1.000
(c, e)	0.277	0.999	-	-	0.023	1.000
(d, e)	0.111	0.995	-	-	0.048	0.999
(a, b, c)	0.901	0.976	0.193	0.999	0.012	1.000
(a, b, d)	0.787	0.976	0.153	0.998	0.026	0.999
(a, b, e)	0.796	0.976	0.111	0.994	0.020	1.000
(a, c, d)	0.784	0.987	0.138	0.999	0.003	1.000
(a, c, e)	0.787	0.991	0.097	0.999	0.002	1.000
(a, d, e)	0.668	0.987	0.056	0.998	0.022	1.000
(b, c, d)	0.447	0.981	0.119	0.998	0.045	1.000
(b, c, e)	0.457	0.981	0.102	0.994	0.022	1.000
(b, d, e)	0.301	0.981	0.094	0.994	0.045	0.999
(c, d, e)	0.292	0.995	0.083	0.998	0.023	1.000
(a, b, c, d)	0.901	0.976	0.087	1.000	0.003	1.000
(a, b, c, e)	0.913	0.976	0.051	1.000	0.002	1.000
(a, b, d, e)	0.799	0.976	0.053	0.998	0.020	1.000
(a, c, d, e)	0.796	0.987	0.045	1.000	0.002	1.000
(b, c, d, e)	0.459	0.981	0.079	0.998	0.020	1.000
(a, b, c, d, e)	0.913	0.976	0.094	0.998	0.000	1.000

TABLE VII. SENSITIVITY (SENS.) AND SPECIFICITY (SPEC) FOR THE 1OON, MAJORITY VOTE AND NOON CONFIGURATIONS FOR N BETWEEN 2 AND 5 FOR XSS

XSS	1ooN		Majority		NooN	
	Sens.	Spec.	Sens.	Spec.	Sens.	Spec.
(a, b)	0.963	0.970	-	-	0.172	0.9963
(a, c)	0.522	0.989	-	-	0.040	0.9991
(a, d)	0.631	0.982	-	-	0.080	0.9985
(a, e)	0.586	0.988	-	-	0.084	0.9997
(b, c)	0.687	0.977	-	-	0.064	0.9990
(b, d)	0.683	0.976	-	-	0.218	0.9921
(b, e)	0.733	0.977	-	-	0.127	0.9989
(c, d)	0.271	0.991	-	-	0.056	0.9995
(c, e)	0.251	0.998	-	-	0.035	0.9998
(d, e)	0.334	0.991	-	-	0.102	0.9994
(a, b, c)	0.981	0.970	0.208	0.996	0.034	0.9992
(a, b, d)	0.967	0.970	0.342	0.989	0.064	0.9991
(a, b, e)	0.986	0.970	0.309	0.995	0.037	0.9998
(a, c, d)	0.655	0.982	0.113	0.998	0.032	0.9997
(a, c, e)	0.613	0.988	0.133	0.999	0.013	0.9999
(a, d, e)	0.681	0.982	0.189	0.998	0.039	0.9998
(b, c, d)	0.702	0.976	0.236	0.991	0.051	0.9996
(b, c, e)	0.751	0.977	0.169	0.998	0.029	0.9998
(b, d, e)	0.747	0.976	0.257	0.991	0.095	0.9995
(c, d, e)	0.357	0.990	0.143	0.999	0.025	0.9999
(a, b, c ,d)	0.981	0.970	0.088	0.998	0.031	0.9997
(a, b, c, e)	0.998	0.970	0.073	0.999	0.013	0.9999
(a, b, d, e)	0.990	0.970	0.139	0.998	0.032	0.9999
(a, c, d, e)	0.696	0.982	0.071	0.999	0.012	1.0000
(b, c, d, e)	0.759	0.976	0.125	0.999	0.025	0.9999
(a, b, c, d, e)	0.998	0.970	0.154	0.998	0.003	1.0000

Sensitivity: Combining SATs phpSAFE (A), RIPS (B) and WAP (C) in a 1ooN setup (meaning we identify code as vulnerable as soon as any one of these tools identifies it as such) gives very large gains in sensitivity for both SQL Injection and XSS. Sensitivity for the best of these tools for SQL injection is 0.56. 1oo3 configuration of these three tools (as listed in the row (a,b,c)) is 0.9. Adding the remaining two SATs (Pixy and WeVerca) improves sensitivity a little bit more (to 0.91) in a 1oo5 setup (row (a,b,c,d,e)). For XSS, phpSAFE (A) and RIPS (B) in a 1oo2 setup have a sensitivity score of 0.96 (individually RIPS (B) had the best sensitivity at 0.66). Combining all 5 tools in a 1oo5 setup meant all the XSS vulnerabilities in the plugins we considered were detected. As we would expect, we see large deteriorations in sensitivity for NooN setups. We also observe poor sensitivity results for majority voting setups.

Specificity: We see gains in specificity in NooN setups (meaning we only label code as vulnerable if all N tools in the setup agree that the code is vulnerable). Many configurations never raise false alarms in these configurations. However, they also have very poor sensitivity values. As expected, majority voting setups do better for sensitivity compared with NooN, but worse for specificity.

ROC plots also help a decision maker to visualize these results and compare the performance of the different systems. Figure 5 shows the eight ROC plots, one for each vulnerability (SQLi and XSS), and for each configuration of N, $2 \leq N \leq 5$. In addition to the 1ooN, simple majority (1oo3, 3oo4 and 3oo5), and NooN setups that we showed in Tables VI and VII, we also calculated the remaining voting setups (2oo4, 2oo5, 4oo5) not shown in those tables.

The most optimal system in an ROC plot is one that appears on the top right-hand corner (i.e. one that has both sensitivity and specificity of 1, since it detects all vulnerabilities and never raises an alarm for code that does not contain vulnerabilities). We have no such system in the configurations in our examples. As we have seen from the results in Tables VI and VII, most of the results in our configurations have extremes of high sensitivity (1ooN) or high specificity (NooN). The ROC plots make it easier to identify configurations that lie between these extremes.

C. Averages for Different Diverse Setups

We conclude our analysis with a summary table (Table VIII) showing the average Sensitivity and Specificity for non-diverse setups (abbreviated "1v" in the first row of the table) compared with the averages for the different diverse configurations. These results confirm the observations we have shown so far:

- For 1ooN systems: more than 70% improvements in sensitivity on average in a 1oo2 setup compared with average individual SATs. More than three times the improvements in sensitivity on average on a 1oo5 setup compared with individual SATs. However, this comes at a correspondingly high deterioration in specificity.
- For NooN systems: almost perfect specificity can be achieved when using NooN setups (especially for configurations of N > 2). But this comes at a large deterioration in sensitivity.
- Simple majority voting setups on average lead to a deterioration in sensitivity (of between 30-65%) but with some improvements in specificity.

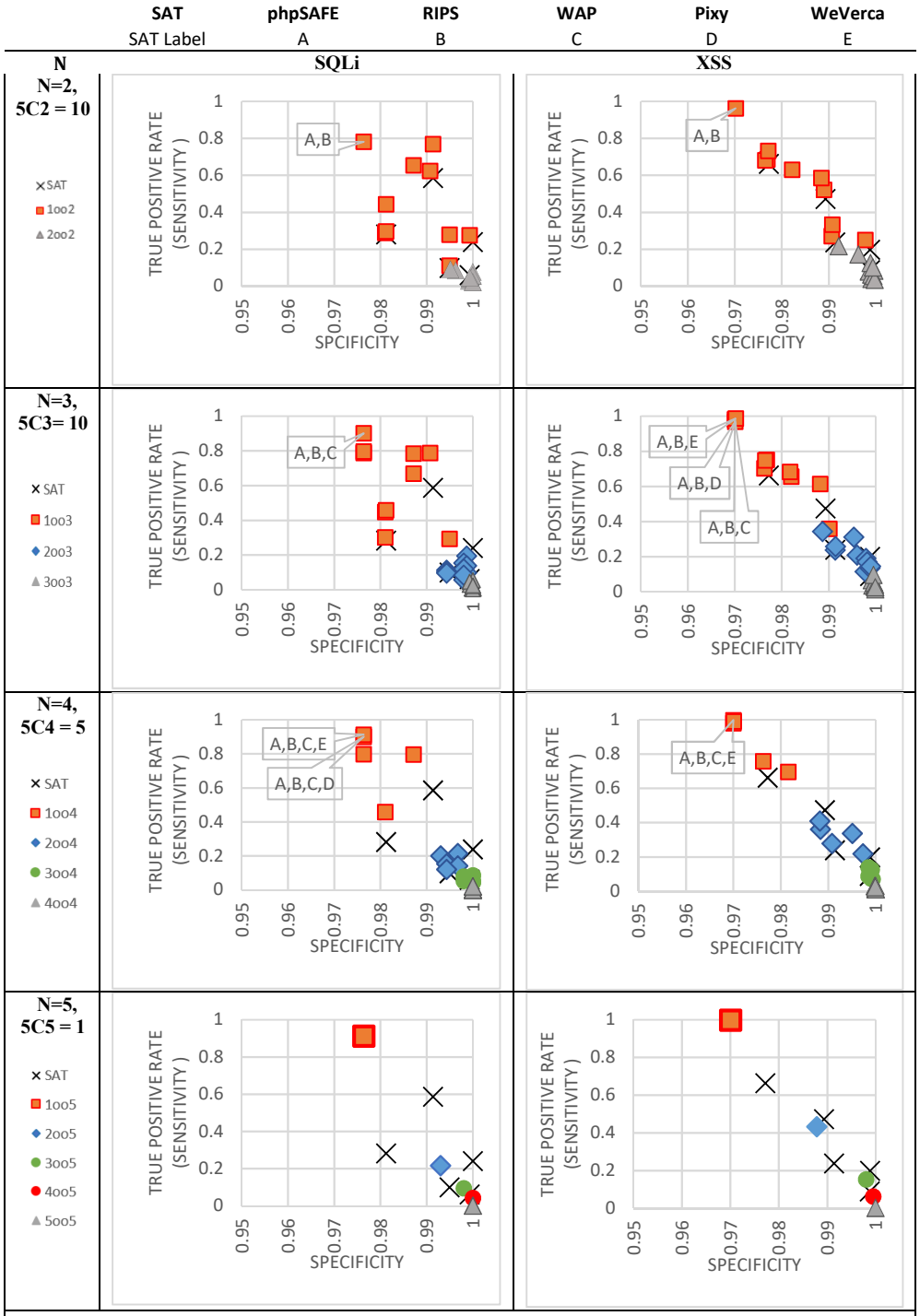

Figure 5 - ROC plots for the different diverse combinations and the two classes of vulnerabilities.

TABLE VIII. AVERAGE SENSITIVITY AND SPECIFICITY FOR EACH DIVERSE VERSION AND EACH CLASS OF VULNERABILITIES

	SQLi		XSS	
	Sens.	**Spec.**	**Sens.**	**Spec.**
1v	0.25	0.99	0.33	0.99
1oo2	0.45	0.99	0.57	0.98
1oo3	0.62	0.98	0.74	0.98
1oo4	0.77	0.98	0.89	0.97
1oo5	0.91	0.98	0.99	0.97
2oo2	0.05	0.99	0.10	0.99
3oo3	0.02	0.99	0.04	0.99
4oo4	0.01	1.00	0.02	0.99
5oo5	0.00	1.00	0.003	0.99
2oo3	0.11	0.99	0.21	0.99
2oo4	0.17	0.99	0.32	0.99
2oo5	0.21	0.99	0.43	0.99
3oo5	0.09	0.99	0.15	0.99
4oo5	0.04	1.00	0.06	0.99

VI. CONCLUSIONS AND FURTHER WORK

In this paper, we presented results of analyzing the performance of diverse Static Analysis Tools (SATs) configurations. The analysis is performed using a previously published dataset, where five SATs were used for finding two types of vulnerabilities, SQL Injections (SQLi) and Cross-Site Scripting (XSS), in 134 WordPress plugins. From the five individual SATs, we built 10 diverse pairs, 10 diverse triplets, 5 diverse quadruples and one diverse quintet SAT system. When analyzing the results, we considered various configurations of the adjudicator: 1ooN (raise an alarm for a vulnerability when any of N SATs in the diverse configuration do so); NooN (raise an alarm for a vulnerability only when all N SATs in the diverse configuration do so); and simple majority (raise an alarm for a vulnerability when the majority of the N SATs in a diverse configuration do so). We presented the results using the well-established measures for binary classifiers: sensitivity and specificity. The main conclusions from our analysis are:

- **For 1ooN systems**: improvements in sensitivity compared with individual SAT are from *70% on average* for 1oo2 systems, to more than 300% for 1oo5 systems, but come with a corresponding *specificity deterioration* on average. The largest improvements in sensitivity, with the least deterioration in specificity are from combining phpSAFE with WAP SATs in a diverse 1oo2 configuration.
- **For NooN systems**: specificity can be perfect in most setups, but with severe deterioration in sensitivity on average.
- **For simple majority voting setups**: average deterioration in sensitivity (of between 30-65%) but with some improvements in specificity.

For organizations primarily interested in detecting vulnerabilities (improved sensitivity) and that are willing to invest resources in sifting through alarms to separate out the false alarms from true alarms, diverse setups in a 1ooN adjudication setup can be very beneficial. In particular, phpSAFE, RIPS and WAP SATs exhibit considerable diversity in vulnerability detection.

We plan to investigate in more detail the types of vulnerabilities that are detected by the different tools so that we can provide more tailored advice to decision makers on ways in which they can configure the tools. We also plan to investigate optimal adjudication setups that allow us to improve both the sensitivity and specificity depending on types of code that is inspected by these tools. Optimal adjudicators are known to perform much better than conventional 1ooN, majority or NooN setups.

ACKNOWLEDGMENT

This work was supported in part by the EU H2020 framework projects DiSIEM and ATMOSPHERE (EU Cooperation Programme) and the UK EPSRC project D3S.

REFERENCES

[1] https://w3techs.com/technologies/overview/content_management/all.
[2] https://developer.wordpress.org/plugins/.
[3] https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf.
[4] NIST SARD Project. http://samate.nist.gov/SRD.
[5] OWASP Top 10 - 2017: The ten most critical web application security risks. Technical report, OWASP Foundation, 2017.
[6] WPScan Vulnerability Database. https://wpvulndb.com/, 2018-04-07.
[7] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *2017 IEEE EuroS&P*, pages 334–349. IEEE, April 2017.
[8] Johannes Dahse, G Horst, and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. (February):23–26, 2014.
[9] Sucuri Remediation Group, Incident Response Team, and Affected Open-source Cms. Hacked Website Report 2017. 2017.
[10] M. K. Gupta, M. C. Govil, and G. Singh. Static analysis approaches to detect sql injection and cross site scripting vulnerabilities in web applications: A survey. In *Int. Conf. on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, pages 1–5, May 2014.
[11] David Hauzar and Jan Kofron. Framework for Static Analysis of PHP Applications. In *29th European Conf. on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 689–711, 2015.
[12] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *IEEE Symp on Security and Privacy,* pages 6 pp.–263, May 2006.
[13] Adam Kieyzun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL Injection and cross-site scripting attacks. In *2009 IEEE 31st ICSE*, pages 199–209. IEEE, 2009.
[14] Thought Leadership and White Paper. Managing security risks and vulnerabilities. (January), 2014.
[15] Fort George Meade. https://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf.
[16] Iberia Medeiros, Nuno F. Neves, and Miguel Correia. Automatic Detection and Correction of Web Application Vulnerabilities Using Data Mining to Predict False Positives. In *Proc. of the 23rd Int. Conf. on World Wide Web*, WWW '14, pages 63–74, NY, USA, 2014. ACM.
[17] N. Meng, Q. Wang, Q. Wu, and H. Mei. An approach to merge results of multiple static analysis tools (short paper). In *2008 The Eighth Int. Conf. on Quality Software*, pages 169–174, Aug 2008.
[18] Paulo Nunes, José Fonseca, and Marco Vieira. phpSAFE: A Security Analysis Tool for OOP Web Application Plugins. In *45th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25*, pages 299–306, 2015.
[19] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. On combining diverse static analysis tools for web security: An empirical study. In *IEEE 13th EDCC*, pages 121–128, Sept 2017.
[20] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. A comparison of bug finding tools for java. In *IEEE Proceedings of the 15th Int. Symp. on Software Reliability Engineering* (ISSRE '04), pages 245–256, Washington, DC, USA, 2004.
[21] J. Walden, M. Doyle, G. A. Welch, and M. Whelan. Security of open source web applications. In *2009 3rd Int. Symp. on Empirical Software Engineering and Measurement*, pages 545–553, Oct 2009.
[22] Q. Wang, N. Meng, Z. Zhou, J. Li, and H. Mei. Towards soa-based code defect analysis. In *IEEE Int. Symp. on Service-Oriented System Engineering, 2008. SOSE '08*, pages 269–274, Dec 2008.