On Combining Diverse Static Analysis Tools for Web Security: An Empirical Study

Paulo Nunes[1] Ibéria Medeiros[2] José Fonseca[1] Nuno Neves[2] Miguel Correia[3] Marco Vieira[4]
[1]CISUC, University of Coimbra, UDI, Polytechnic Institute of Guarda, Portugal,
[2]LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal,
[3]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Portugal, [4]CISUC, University of Coimbra, Portugal
pnunes@ipg.pt, imedeiros@di.fc.ul.pt, josefonseca@ipg.pt, nuno@di.fc.ul.pt, miguel.p.correia@ist.utl.pt, mvieira@dei.uc.pt

***Abstract*—Developers frequently rely on free static analysis tools to automatically detect vulnerabilities in the source code of their applications, but it is well-known that the performance of such tools is limited and varies from one software development scenario to another, both in terms of coverage and false positives. Diversity is an obvious direction to take to improve coverage, as different tools usually report distinct vulnerabilities, but this may come with an increase in the number of false alarms. In this paper, we study the problem of combining diverse static analysis tools to detect web vulnerabilities, considering four software development scenarios with different goals and constraints, ranging from low budget to high-end (e.g., business critical) applications. We conducted an experimental campaign with five free static analysis tools to detect vulnerabilities in a workload composed by 134 WordPress plugins. Results clearly show that the best solution depends on the development scenario. Furthermore, in some cases, a single tool performs better than the best combination of tools.**

Keywords—static analysis; vulnerability detection; XSS; SQLi.

I. INTRODUCTION

The pervasive use of web applications poses new security challenges to developers. These applications are commonly built in a short time and with tight budgets, often leading to the presence of bugs and security weaknesses. When deployed, such applications are instantly exposed to millions of Internet users, and may be exploited by hackers, putting at risk the valuable assets within. In fact, the number of attacks documented by different entities has been growing in the past years. The consequences of such attacks can be huge, including financial cost, liability issues, brand damage, and loss of market share [1].

Content Management Systems (CMS) are increasingly used to support the development of web applications, as they provide many built-in features, allowing rapid development [4]. Moreover, they allow new features to be easily added through modules, themes and plugins. WordPress is the most popular and widely used open-source CMS adopted by businesses of all sizes and everyday website owners [2]. It supports the development of websites based on PHP resources, with over 44,000 plugins that have been downloaded more than 1.25 billion times. However, some of these plugins have vulnerabilities and, since a single plugin may be used in thousands of websites, they are an attractive target for hackers. For instance, 22% of the 170,000 websites hacked in 2012 were attacked via vulnerable plugins [3]. Unfortunately, the situation has not been improving, as shown in a study conducted by *Sucuri* [4] in the first quarter of 2016, which reports that 25% of all compromised WordPress sites (89,000) were attacked by exploiting three vulnerable plugins used in all of them (TimThumb, RevSlider, and GravityForms).

Static analysis tools (SATs) inspect the source code of software, without executing it, to discover potential bugs that lead to security vulnerabilities [6]. Therefore, they provide a valuable support during the software development lifecycle by automating the task of searching for the location of candidate vulnerabilities [7] [8]. However, SATs have recurring limitations, such as missing some of the vulnerabilities (*false negatives* (*FN*)) and generating many false alarms (*false positives* (*FP*)). These limitations arise from conceptual constraints of the static analysis process that mainly checks the structure of a program based on fixed rules and does not consider the user input data or the dynamic characteristics of the application [9]. Therefore, it is known that different SATs report distinct sets of security vulnerabilities, with some overlap [8][10][11].

The state-of-the-art SATs are, on average, able to detect about half of the existing security vulnerabilities [16]. To improve their overall detection capabilities, some researchers have proposed combining the results of multiple SATs to improve detection [17] [18] [19]. Of particular interest is the work of Diaz et al. [12], which compares the performance of nine SATs, most of them commercial tools, against the NIST SAMATE (Software Assurance Metrics And Tool Evaluation) Reference Dataset test suite [13]. Based on the results, the authors recommended the use of several tools with different designs and detection algorithms and/or heuristics to improve the analysis results. On the other hand, Beller et al. [14] investigated how common is the use of SATs in real-world, taking as reference the 122 most popular Open-Source Software projects. The results show that a single SAT was used in 41% of the projects, two SATs in 22% of the projects, and three SATs in 14% of the projects. This suggests that developers might not be aware of the benefits of using multiple tools and/or that the increase of false positives reported may lead developers to avoid using multiple SATs [15]. However, **existing works are limited in several aspects**: the workloads are synthetic or just small sets of applications, the evaluation metrics used are too simple (e.g., number of true positives (TP)), and the analysis does not consider the specificities of the development scenarios were the tools can be used, which may vary both in terms of development time and resources.

In this paper, we argue that the use of multiple SATs might be helpful, as more vulnerabilities may be reported, however, the drawback is that the number of false positives may at the same time increase. Furthermore, we also claim that the acceptable/expected outcome of the static analysis process (in terms of coverage and FPs) depends on the scenario. A scenario is a realistic situation of vulnerability detection that depends on the criticality of the application being developed and on the security

budget available. Antunes et al. [20] proposed four scenarios with increasing requirements: min-effort, best-effort, heightened-critical, and business-critical. For example, while the min-effort scenario has a very tight budget, so a high number of FPs is unacceptable due the cost associated with the excessive manual verification needed, in a business-critical scenario the objective is to detect the most vulnerabilities possible, even if that requires spending budget analyzing FPs. Thus, **it is no longer evident that combining more SATs is better in every case**.

This work studies the potential of combining the outputs of multiple SATs as a way to improve the performance of vulnerability detection across different realistic development scenarios. In practice, we formulate the following four hypotheses:

- H_1: *The number of vulnerabilities detected always increases as the number of combined SATs increases.*
- H_2: *The number of false positives always increases as the number of combined SATs increases.*
- H_3: *The best combination of SATs is the same across development scenarios.*
- H_4: *The best combination of SATs is the same across different classes of vulnerabilities.*

Although the response to the hypotheses above may seem obvious, empirical evidences are missing in the literature to better understand the advantages and limitations of different combinations of SATS, when considering representative vulnerability detection scenarios. For example, a less informed researcher or developer could easily state that *the number of vulnerabilities detected increases as the number of combined SATs increases*, however, knowing in which scenarios that is true, to which amount that happens for different types of vulnerabilities, and what is the impact in terms of false positives, are aspects that require more detailed studies, as the one presented here.

Our study is based on the results of benchmarking five free SATs to detect SQL injection (SQLi) and cross-site scripting (XSS) vulnerabilities in a workload of 134 WordPress plugins, organized in four vulnerability detection scenarios. We focus on free SATs as both occasional developers and professional software houses wanting to speed up the development process and reduce cost tend to use free tools as much as possible. Furthermore, such tools are easily available for research and results can be published without infringing licensing agreements. As for the workload, it is important to note that, according to WhiteSecurity statistics, from 42,000+ WordPress Websites in Alexa Top 1 Million, more than 73% of the installations have vulnerabilities, which could be potentially detected using free automated tools [21]. SQLi and XSS are information flow vulnerabilities that are on the top three of web security vulnerabilities [5], and are also quite common in WordPress plugins [27].

Results show that different combinations of tools achieve different performance, both in terms of vulnerabilities detected and false positives. Also, there are tradeoffs that should be considered when combining tools, which may influence decisions depending on the scenario where the tools will be used. There are even cases where using a single tool provides better results than combining multiple tools. These observations and others discussed later emphasize the need for a process to study the effectiveness of diverse tools, as the one proposed in this paper.

The outline of the paper is as follows. Next section presents the dataset used in the study. Section 3 discusses the metrics and Section 4 presents the process for combining the outputs of multiple SATs. Section 5 presents and discusses the results. Section 6 discusses related work and Section 7 concludes the paper.

II. DATASET

The dataset used is based on the results of benchmarking five free SATs looking for SQLi and XSS vulnerabilities in 134 WordPress plugins. Four criticality levels representing realistic scenarios were considered (we adapted the names of the scenarios defined by Antunes et al. [20] to better represent their requirements, but maintained their scope):

1. **Highest-quality** - every vulnerability missed may be a big problem due to the high criticality of the application;
2. **High-quality** - a few non-trivial vulnerabilities may be missed given that there are not many FPs;
3. **Medium-quality** - vulnerabilities may be missed at the cost of reducing the FPs;
4. **Lowest-quality** - every FP is an important cause of concern due to budget restrictions.

SQLi and XSS vulnerabilities occur when input data are not properly validated. A SQLi attack is based on the injection of code that changes the SQL query sent to the database and an XSS attack consists in the injection of malicious JavaScript in the input of a vulnerable web page. Both SQLi and XSS attacks are very serious to enterprises and individuals, since they may allow security violations such as manipulating unauthorized database data and causing denial of service.

For detecting the SQLi and XSS vulnerabilities in the WordPress plugins we used the following SATs: RIPS v0.55 [8], Pixy v3.03 [7], phpSAFE [10], WAP v2.0.1 [11], and WeVerca v20150804 [24]. RIPS performs static taint analysis and string analysis. RIPS and Pixy are the two most referenced PHP SATs in the literature, but they are not ready for OOP analysis. Pixy has not been updated since 2007 and RIPS has only been developed as open source until 2014. RIPS has recently released a commercial version able to fully analyze OOP code [25]. WAP, phpSAFE, and WeVerca are recent tools under active development, and they are prepared for OOP code.

A. Selecting the plugins

The online WPScan Vulnerability Database (WPVD) [27] provides a list of WordPress plugins with known vulnerabilities including, for most of them, proofs of concept (PoC) and other details, like the CVE identifier. From this list (on 2015/10/16) we selected all the plugins with SQLi and/or XSS vulnerabilities. The result is a list of 134 plugins with 152 SQLi (84 with PoC) and 67 XSS (13 with PoC) vulnerabilities registered. In practice, the dataset consists of 103 object-oriented programming (OOP) plugins and 31 procedural (POP) plugins having a total of 4,990 PHP files and 1,023,081 lines of code (LOC), where 57% is OOP, 32% is POP, and 11% are mix of both. The workload contains 466,164 LLOC (Logical LOC), where 39.5% are POP, 47.8% are OOP, and 12.7% is a mix of both.

B. Assigning plugins to scenarios

To compose the workload, we need to assign a representative

TABLE I – MAPPING OF SOFTWARE PRODUCT PROPERTIES TO ISO/IEC SUB-CHARACTERISTICS OF MAINTAINABILITY.

Maintainability sub-characteristics	Software Product Properties: Duplication	Unit complexi	Unit size	Module coupling	Class complexity	Unit interfaci	Class inter-	Unit testing	Rating average
Ratings example	5	4	3	4	3	3	2	3	
Analyzability	×	×	×						4.0
Changeability	×	×		×	×				4.0
Stability						×	×	×	2.6
Testability		×	×	×				×	3.5
Maintainability rating (average: ★★★★)									3.5

group of vulnerable plugins to each scenario. Since this is very hard to achieve (e.g., real business-critical software is often kept secret) and has an associated level of subjectivity (e.g., there are different interpretations of what constitutes critical software), we propose a general procedure to classify applications based on code quality. Generically, the assumption is that *scenarios that are more stringent normally run software with better quality*. Therefore, we should assign applications with better *quality* to scenarios with more *criticality*. In practice, this means that a given set of plugins can be used in a scenario if their source code has a sufficient level of quality and admissible artefacts.

The process for assigning plugins to scenarios has two steps. The first is based on the approach proposed by Baggen et al. [22] for rating the maintainability of the source code of applications (from 0.5 to 5.5 stars). The Baggen's approach uses a standardized measurement model based on the ISO/IEC 9126 definition of maintainability and a small set of source code metrics (SCMs) (e.g., Cyclomatic Complexity Number (CCN2) [23]). These SCMs are used to measure the Software Product Properties (SPPs) (e.g., *Unit Complexity*) of the software. Table I outlines the SPPs and their relationship with the maintainability sub-characteristics. The table also includes an example of assigning a rating to an application. The ratings of the sub-characteristics are obtained by averaging the ratings of the properties where a '×' is present in the sub-characteristic's line in the table. The final rating is obtained by summing the average ratings and dividing by 4 (in the example: *(4.0+4.0+2.6+3.5)/4 = 3.5 stars*). The Baggen's method is applied by the Software Improvement Group (SIG) to annually re-calibrate the SIG quality model [19], which forms the basis of the evaluation and certification of software maintainability conducted by SIG [20] and TÜViT [18].

The second step of the process is based on a simple schema for mapping the ratings in scenarios. Since the ratings vary from 0.5 to 5.5 in ascendant quality order and the scenarios from 1 to 4 in descendent level of criticality, we used the following mapping *rating-scenario*: [4.5, 5.5[- Scenario 1 (*highest-quality*); [3.5,4.5[- Scenario 2 (*high-quality*); [2.5, 3.5[- Scenario 3 (*medium quality*); and [0.5, 2.5[- Scenario 4 (*lowest-quality*). As shown, we used intervals of 1 for mapping the ratings into the scenarios, trying to respect Baggen's approach, except for the less stringent (and probably less relevant) scenario, which accommodates all the ratings below 2.5 (code of lower quality).

The results of applying the proposed process are presented in Table II, which shows the plugins that compose the workload, distributed over the four scenarios. Scenario 1 has a lower number of plugins compared with 2 and 3. This is realistic as finding code with very high quality is not trivial. At the same time, the WordPress plugins considered have been download and used so many times that a given level of quality is expected (they would not be used that much if that was not the case). Thus, the number of high- and medium-quality plugins is also much higher than the number of low-quality plugins. In terms of LOCs, the average size of the plugins tends to increase as the quality decreases (it is more difficult to assure quality for larger pieces of code).

C. Characterizing vulnerable LOCs

To evaluate a SAT, we need to know which LOCs are vulnerable (i.e., positive instances (*P*)) and which LOCs are not vulnerable (i.e., negative instances (*N*)). For large code bases, this is a hard task that requires a thorough review by security experts, and the result might not be completely accurate (as experts can also make mistakes).

A vulnerability may manifest in a restricted set of constructs (e.g., XSS in PHP's *echo*) of the programming language, named sensitive sinks (SS). Although the number of vulnerable LOCs in an application is limited to the lines that include such constructs, the number of vulnerabilities can potentially be greater than the number of LOCs, because one LOC can have several vulnerabilities. For example, the PHP LOC *echo "$name $category"*, may have two XSS vulnerabilities (due to the two variables used). SATs report results in different manners, for example, some SATs may report the sources of untrusted data (*Entry Points (EP)*), where others report the SS where that data is used in a risky way. Thus, in this work, we count the vulnerabilities at the level of the LOC. This means that a LOC with one or more vulnerabilities represents one positive instance.

Our approach to find vulnerable LOCs (VLOCs) in the workload is based on running the five SATs and on a manual review to confirm the results. Thus, to obtain the vulnerable LOCs, we ran the five SATs to scan for SQLi and XSS vulnerabilities in the workload. phpSAFE, RIPS, WAP and Pixy were configured by default for PHP entry points, sensitive sinks and sanitization functions (SF) (e.g., *htmlentities, mysql_real_escape_string*). WeVerca does not allow configuration and includes a programmed list of EPs, SSs, and SFs. Overall, phpSAFE was unable to analyze 18 plugins. WAP analyzed all plugins, but seven of them only partially. RIPS and Pixy analyzed partially 76 and 103 plugins, respectively. WeVerca was not able to analyze a total of 20 source files of 14 plugins. In practice, the tools could not to fully analyze some plugin/files, reporting runtime errors or taking a very long time without any results. This is potentially due to the size/complexity of some files and/or limitations of the static analysis tools used.

The outputs of the tools were combined and each candidate vulnerability was manually reviewed to determinate if it was a TP (i.e., vulnerable LOC or P) or a FP (i.e., non-vulnerable LOC or N). The union of all TPs became part of the set of positive instances (P) and the union of all FPs become part of the list of non-vulnerable LOCs. As tools may miss vulnerabilities (FNs) and, even when using multiple SATs, some vulnerabilities may

TABLE II – PLUGIN BACKGROUND INFORMATION.

Scenarios	OOP	POP	Total	%Tot.	Files	LLOC	%LLOC
Highest-quality	10	2	12	9.0	352	19,542	4.2
High-quality	39	17	56	41.8	1,687	122,835	26.4
Medium-quality	40	11	51	38.1	2,223	211,297	45.3
Low-quality	14	1	15	11.2	728	112,490	24.1
Total	**103**	**31**	**134**	**100.0**	**4,990**	**466,164**	**100.0**

remain undetected [26], the number of vulnerable LOCs that we found is likely less than the real number, which may introduce bias in the evaluation metrics. To minimize this issue, we also added to the list of non-vulnerable LOCs all the vulnerabilities registered in the public online WPScan Vulnerability Database [4] for the target plugins (see Table III).

D. Characterizing non-vulnerable LOCs

Data coming from untrusted user inputs can be propagated to one or more variables through data flows, control flows, function calls and return statements. From all LOCs in a program, only LOCs with SSs outputting at least one variable are susceptible to be vulnerable. Thus, LOCs with SSs outputting fixed data (i.e., without any output variable) are not vulnerable. The remaining LOCs (non-SSs) certainly are not vulnerable.

To obtain the list of NVLOCs, we developed a PHP script for gathering all SS function calls of the source code files based on their Abstract Syntax Tree (AST). From this list, we removed the items that were already labeled as VLOC (previous section). The script is very simple and is executed individually for each file. A manual check of random samples has been performed to increase trust on the accuracy of the NVLOCs identified.

E. Summary of the vulnerable and non-vulnerable LOCs

Table III reports the results of the *processes for characterizing VLOCs and NVLOCs* in the workload. The table depicts the number of TPs and FPs reported by the SATs, followed by the number of vulnerabilities registered in the online WPScan Vulnerability Database (column VD) [27]. The column NVLOC shows the NVLOCs obtained from the *process for characterizing non-vulnerable LOCs*. The two last columns show the total number of positive instances (the distinct VLOCs), and the total number of negative instances (the combination of the FPs with the list of NVLOC). These two last columns are the ones that will be used to calculate the metrics.

III. Metrics

Different tools may report different sets of vulnerabilities and false positives. When combining the results of several tools both the number of correctly detected vulnerabilities and the number of FPs are likely to increase. Although the detection of more vulnerabilities is the objective, particularly in critical scenarios, a high number of FPs may be undesirable in some cases.

For evaluating the SATs we propose the use of metrics that are adequate to the vulnerability detection scenario. As mentioned before, our approach is based on the scenarios defined by Antunes et al. [20]. For each scenario, Antunes et al. also proposed one main metric to rank the tools and a tiebreaker metric used only when there is among between tools (see Table IV).

TABLE III – DISTRIBUTION OF VULNERABILITIES AND NON-VULNERABILITIES BY SCENARIOS AND VD

Scenario		Tools		VD	NVLOC	Total	
		TP	*FP*	*P*		*P*	*N*
SQLi	1	65	5	17	82	**75**	**87**
	2	318	62	35	1053	**346**	**1115**
	3	251	163	22	2051	**267**	**2214**
	4	41	32	10	1105	**50**	**1137**
	Total	**675**	**262**	**84**	**4291**	**738**	**4553**
XSS	1	165	45	3	945	**168**	**990**
	2	1841	224	1	5601	**1842**	**5825**
	3	2386	680	4	9289	**2389**	**9969**
	4	543	117	5	3477	**545**	**3594**
	Total	**4935**	**1066**	**13**	**19312**	**4944**	**20378**
Total		5610	1328	97	23603	5682	24931

TABLE IV – SUMMARY OF METRICS BY SCENARIO

Scenario	Antunes et al. Scenario	Metric	Tiebreaker
1 - Highest-quality	Business-critical	Recall	Precision
2 - High-quality	Heightened-critical	Informedness	Recall
3 - Medium-quality	Best-effort	F-Measure	Recall
4 - Low-quality	Min-effort	Markedness	Precision

In practice, the metrics depend on the goals of the detection, which are related with the amount of available resources to fix the vulnerabilities (see Table V). Therefore, there is one main metric to rank the tools in each scenario. For example, for the highest-quality scenario the goal is to find the highest number vulnerabilities at any cost. Therefore, the metric recall is used to measure this global information. However, it ignores the precision (FPs) of the results. Only in the case of a tie, the *precision* metric is used to rank first the tool that reports less FPs.

The metrics *recall*=TP/ (TP+FN), *precision*=TP/(TP+FP) and *F-Measure*=2×TP/ (2×TP+FP +FN) are well known and widely used. Next, we define the remaining ones (see also [28]):

Informedness = Recall + Inverse Recall – 1 = TP/(TP+FN)+TN/(FP+TN)–1. Requires knowing both the overall number of positive and negative instances. Therefore, every TP increases the metric in the proportion 1/P and every FP decreases the metric in the proportion 1/N.

Markedness = Precision + Inverse Precision – 1 = TP/(TP+FP)+TN/(FN+TN)–1. Considers only the number of TPs and PFs reported. The metric sums the proportions of the positives and the negatives that are correctly identified as such.

IV. Combining the Results of Multiple SATs

For testing our hypotheses, all combinations of tools should be considered for each scenario and class of vulnerabilities. The process proposed to calculate the combined results for two or more tools is based on a set of automated steps (see Fig. 1):

1) **Calculate the number of P and N in the workload** - using the lists of VLOCs and NVLOCs, and the distribution of the plugins per scenario, calculate the number of P and the number of N for each scenario and for each class of vulnerability, as described in Section II (and summarized in Table III).

2) **Combine results of SATs** - for each scenario, class of vulnerability and possible combination of the SATs, merge the outputs of the tools discarding duplicated TPs and FPs.

TABLE V – GOALS OF VULNERABILITY DETECTION BY SCENARIO

Scenario	TP	FP	Resources to fix
1 - Highest-quality	highest	many	all that are needed
2 - High-quality	highest	not many	balanced appropriately
3 - Medium-quality	high	low	limit, redirected to fix
4 - Low-quality	high	lowest	very low

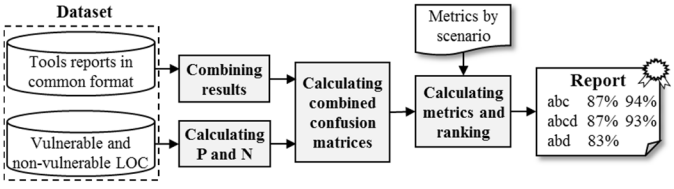

Fig. 1 – Overall process of combining the results of multiple tools.

3) **Calculate the combined confusion matrices** - with the outputs from 1) and 2) calculate, for each scenario, class of vulnerability and combination of tools, the corresponding confusion matrix (i.e., TP, FP, FN, and TN).

4) **Calculate the metrics and rank** - with the results from 3), compute the metrics recommended for each scenario (see Table V) and rank the combinations of tools.

V. Results and Discussion

The results for each combination of the five SATs, organized by scenario and type of vulnerability, are presented in Table VI (SQLi on the left-side and XSS on the right-side). Columns *TP*, *FP*, *FN*, and *TN* show the confusion matrix for the corresponding combination. The table shows only the TOP 7 (of 32) combinations of SATs for each scenario and type of vulnerability, due space limitations (detailed results can be found at [29]). The table also includes the ranking of the individual tools, as reference. The data are firstly ordered by the main metric (column *Metric*), secondly by the tiebreak metric (column *Tiebreaker*), and finally by the number of SATs in the combination.

To simplify the presentation of the results we assigned a character to each tool: *a - phpSAFE*, *b - RIPS*, *c - WAP*, *d-Pixy*, and *e - WeVerca*. Note that **vulnerability detection using static analysis is a deterministic process**, thus running the same tool several times always leads to the same results.

A. Results for SQLi vulnerabilities

The goal in the **highest-quality scenario** is to find the highest number of vulnerabilities, even if reporting many FPs. Therefore, the best solution is *ac*, as it has the least number of SATs (Table VI (a)). We see that RIPS, Pixy and WeVerca did not find vulnerabilities and did not report FPs. In this scenario, there are 2 POP plugins and 10 OOP (see Table II) plugins with 43% of POP code. Thus, the results of RIPS and Pixy are probably because they are not prepared to analyze OOP code.

In the **high-quality scenario**, the combination that ranks first is *acde*, with the highest number of vulnerabilities found (318). In 2nd place there is a similar solution (*abce*) with the same number of vulnerabilities, but one more FP. Individually, we see that phpSAFE found many vulnerabilities but also many FPs, while the others report less vulnerabilities and zero or few FPs.

In the **medium-quality scenario**, the combination of SATs that ranks first is *abce* with both the highest number of vulnerabilities found (251) and FPs (163). The solutions in the positions 2-8, have the same number of FPs and successively less vulnerabilities. In the 9th place there is a solution composed by three SATs (*acd*) with about 2/3 of the vulnerabilities found and 2/5 of FPs. In terms of vulnerabilities detected, this solution performs similarly to RIPS (*b*) individually. However, RIPS reports about the double of FPs. This means that, the best individual SAT (*a*) can be replaced by a combination of SATs (*acd*) that individually perform worse than it, but together perform better.

The best solution for the **low-quality scenario** is *bc*, with 6 TPs and zero FPs. Since the resources available for fixing vulnerabilities in this scenario are very limited, this solution fits very well in its goal. Note that, phpSAFE individually (SAT *a*) reports six times more vulnerabilities (36), but also many more FPs (32), which is not desirable in this case.

B. Results for XSS vulnerabilities

For the **highest-quality scenario**, and focusing on XSS vulnerabilities (see Table VI (b)), the best solution is the combination *ab*, which detected the highest number of vulnerabilities (165). In the 2nd position, there is a combination of three SATs (*abe*) that detected the same number of vulnerabilities and reported the same number of FPs (34). A key observation is that the combination *ab* detected the same vulnerabilities as WeVerca (*e*) individually, as well as other vulnerabilities. The solutions from the 3rd place to the 7th place also detected the same number of vulnerabilities of the two first solutions, but reported two more FPs. The number of SATs in these solutions varies from two to five. As shown, adding a tool to an existing solution does not always increase the number of vulnerabilities found.

In the **high-quality scenario**, the best combination of SATs is *abce*, which has the highest number of vulnerabilities detected (1841), but also has the highest number of FPs (224). In this case, every FP decreases the *informedness* metric by 0.02% (1/5825) while every TP (1/1842) increases the metric by 0.05%. Therefore, all the FPs reported decrease the metric only 3,56% (227/5825). For that reason, the best solution is the one with both the highest number of TPs and FPs.

The best combination of SATs to detect XSS vulnerabilities in the **medium-quality scenario** is also *abde*. We see that this combination of SATs is better than others because it has the highest number of TPs, but also the highest number of FPs. This is because the recommend metric, F-Measure, does not consider the TNs and it considers the TPs more important than the FPs.

SAT *c* (WAP) comes alone in the 1st position for the **low-quality scenario**. It detected 62 TPs and reported only 3 FPs. It means both high *precision* and *inverse precision*. Therefore, the resources will be consumed for fixing vulnerabilities, as desired for this scenario, instead of being wasted confirming FPs. However, from the 2nd to 8th positions there are combinations of SATs that detected about ten times more TPs but more than thirty times more FPs than SAT *c*. For example, the solution in the 2nd position, *abce*, detected 543 TPs but also reported over 39 times more of FPs (117) which is not acceptable for this scenario. Like for SQLi vulnerabilities in this same scenario, the SAT *a* (phpSAFE) reported both many vulnerabilities and many FPs. Finally, unlike for SQLi vulnerabilities, tool *b* (RIPS) reported both the highest number of TPs and FPs. This shows that tools may have different performance depending on the type of vulnerability being detected.

It is important to emphasize that, considering the top seven combinations presented for each scenario, SAT *a* (phpSAFE) is included in all cases excepted for the low-quality scenario, SAT *b* is included in five solutions, SAT *c* in seven, SAT *d* in one, and SAT *e* in four. SAT *c* is the tool that reported less FPs in almost all scenarios, despite it is not the SAT that found more vulnerabilities. Thus, the effectiveness of existing solutions has a high probability to improve when SAT *c* is added to these solutions. SATs *a* and *b* reports both many vulnerabilities and FPs. The individual ranking of the SAT *d* is always below the middle of the ranking and is the worst of all in 4 of 8 cases. This is probably because SAT *d* is old and not prepared for analyzing OOP code. However, despite being recent and prepared for OOP code, SAT *e* has a performance similar to *d*.

C. Testing the four hypotheses

Based on our findings, all hypotheses stated in the introduction are false. Hypothesis H_1 (*the number of vulnerabilities detected always increases as the number of combined SATs increases*) is false because we found many cases where adding a SAT to an existing combination of SATs, does not increase the number of vulnerabilities found (e.g., for the highest-quality scenario and XSS: *ab*, *abe*, *abce*). On the other hand, we also observed that the number of FPs does not always increase with the number of SATs in a combination (e.g., for the medium-quality scenario and SQLi: ab, abe, abde). Therefore, hypothesis H_2 (*the number of false positives always increases as the number of combined SATs increases*) is also false. As there is frequently an overlap between the FPs reported by different SATs, in some cases combinations with more tools can detect more vulnerabilities, while maintaining the same number of FPs. Also note that, none of the best combinations includes all SATs.

The best solution for vulnerability detection depends on the chosen scenario and on class of vulnerability. Therefore, hypotheses H_3 (*the best combination of SATs is the same across development scenarios*) and H_4 (*the best combination of SATs is the same across different classes of vulnerabilities*) are both false. In fact, the detection capabilities of the SATs are not uniform across the two classes of vulnerabilities. The same occurs for combinations of SATs. Moreover, in almost all cases the values of the metrics for XSS vulnerabilities are better than for the SQLi vulnerabilities.

In summary, the main advantage of combining the results of several SATs is the identification of more vulnerabilities. In fact, for several cases there are SATs that individually did not find any vulnerabilities or found few vulnerabilities in many plugins. Moreover, we observed that even using all the SATs some vulnerabilities remain undetected. A key remark is that combining many tools can be counterproductive in some cases as that will not lead to the detection of more vulnerabilities, but will increase the number of FPs reported, which then need to be verified manually by the developers. Finally, identifying the strengths and limitations of SATs, helps developers to determinate how such tools can be combined to provide a more thorough analysis of the software depending on the specificities of the scenario and on the class of vulnerability being analyzed.

D. Threats to validity

1) ***Workload*** - The workload across the various scenarios is unbalanced, which may affect the results in some cases. For example, in the low-quality scenario and for SQLi, only one SAT reported vulnerabilities, which may limit our study. Works using other tools are needed for improving the characterization of the vulnerable/non-vulnerable LOCs in the workload.

2) ***Vulnerabilities*** - There are limitations regarding the scope of the workload in this experiment, since it considers only WordPress plugins, SQLi, and XSS vulnerabilities. Future studies should expand the workload to include other kinds of applications and classes of vulnerabilities.

3) ***Free SATs*** - All SATs used in this study are free. Pixy is not updated since 2007 and RIPS has only been developed as open source until 2014. On the other hand, WAP, phpSAFE, and WeVerca are recent tools prepared do analyze OOP code. There are several commercial and other free SATs, thus, the results of this study are only valid for the tools used.

4) ***Tools configuration*** - The dataset used in this study was collected with all tools configured by default for PHP entry points, sensitive sinks and sanitization functions. The results of the tools may be improved (+TP and -FP) by adjusting their configuration settings for WordPress built-in database functions, sanitization and escaping routines.

We are obviously aware of the limitations of our study, but would like to emphasize that the main observations of the paper hold in many situations. Regardless of the threats, results show that the number of vulnerabilities detected does not always increase as the number of combined SATs increases (~H1), the number of false positives does not always increase as the number of combined SATs increases (~H2), the best combination of SATs does vary across development scenarios (~H3), and there are differences on the best combination of SATs across classes of vulnerabilities (~H4).

VI. Related Work

Rutar et al. [17] studied five well-known SATs on a small set of Java programs with different sizes 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 for automatically combining and correlating their 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 are not manually reviewed, thus, there is no distinction between TP and FP. The metric used to evaluate and compare the tools was the number of bugs found by each tool.

Meng et al. [19] 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. The approach then determines which tools can search for the specified class of bugs, generates the necessary configurations to run the tools, runs the tools, combines the outputs in a single report, and applies two prioritizing policies to rank the results. Meng et al. used their approach to conclude that developers could benefit from more than one SAT. Like the Rutar's approach, the results were not classified as TP and FP and the authors did not propose any metric to evaluate the approach. The workload was composed by a small Java program that is not representative of real applications. Therefore, with such limited validation it is very difficult to assess the strength and drawbacks of the solution.

Wang et al. [18] proposed an approach that combines multiple SATs in a simple Web Service. The user has the possibility to choose the classes of bugs to scan and 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 results are merged in a way that the same defect is only reported once. The combined results are sent back to the user. The approach was evaluated in terms of running time when combining two SATs, but the experiments were quite limited, having just a single Java test case. Therefore, the solution lacks the validation of the effectiveness of the vulnerability detection when using a combination of SATs.

The National Security Agency (NSA) Center for Assured

TABLE VI – RANKING OF COMBINED TOOLS (TOP 8) AND RANKING OF INDIVIDUAL TOOLS

(a) SQLi

Tools	*TP*	*FP*	*FN*	*TN*	*#Plug*	*Metric*	*Tiebreaker*
Highest-quality						***Recall***	***Precision***
ac	**65**	**5**	**10**	**82**	**9**	**0.867**	**0.929**
ace	65	5	10	82	9	0.867	0.929
abce	65	5	10	82	9	0.867	0.929
acde	65	5	10	82	9	0.867	0.929
abc	65	5	10	82	9	0.867	0.929
acd	65	5	10	82	9	0.867	0.929
abcd	65	5	10	82	9	0.867	0.929
Individual rank							
c	49	4	26	83	7	0.653	0.925
a	29	5	46	82	5	0.387	0.853
e	0	0	75	87	0	0.000	-
b	0	0	75	87	0	0.000	-
d	0	0	75	87	0	0.000	-
High-quality						***Informedness***	***Recall***
acde	**318**	**59**	**28**	**1056**	**36**	**0.866**	**0.919**
abce	318	60	28	1055	36	0.865	0.919
abcde	318	60	28	1055	36	0.865	0.919
ace	316	59	30	1056	36	0.860	0.913
acd	311	58	35	1057	35	0.847	0.899
abc	311	60	35	1055	35	0.845	0.899
abcd	311	60	35	1055	35	0.845	0.899
Individual rank							
a	274	58	72	1057	30	0.740	0.792
c	44	4	302	1111	12	0.124	0.127
b	43	2	303	1113	8	0.123	0.124
e	18	1	328	1114	6	0.051	0.052
d	16	0	330	1115	7	0.046	0.046
Medium-quality						***F-Measure***	***Recall***
abce	**251**	**163**	**16**	**2051**	**21**	**0.737**	**0.940**
abcde	251	163	16	2051	21	0.737	0.940
abc	250	163	17	2051	21	0.735	0.936
abcd	250	163	17	2051	21	0.735	0.936
abde	237	163	30	2051	19	0.711	0.888
abd	236	163	31	2051	19	0.709	0.884
abe	235	163	32	2051	19	0.707	0.880
Individual rank							
b	153	113	114	2101	6	0.574	0.573
a	99	50	168	2164	15	0.476	0.371
c	72	0	195	2214	11	0.425	0.270
d	54	13	213	2201	4	0.323	0.202
e	21	34	246	2180	3	0.130	0.079
Low-quality						***Markedness***	***Precision***
bc	**6**	**0**	**44**	**1137**	**2**	**0.963**	**1.000**
bce	6	0	44	1137	2	0.963	1.000
bcde	6	0	44	1137	2	0.963	1.000
bcd	6	0	44	1137	2	0.963	1.000
c	5	0	45	1137	2	0.962	1.000
ce	5	0	45	1137	2	0.962	1.000
cde	5	0	45	1137	2	0.962	1.000
Individual rank							
c	5	0	45	1137	2	0.962	1.000
b	1	0	49	1137	1	0.959	1.000
a	36	32	14	1105	7	0.517	0.529
e	0	0	50	1137	0	-	-
d	0	0	50	1137	0	-	-

(b) XSS

Tools	*TP*	*FP*	*FN*	*TN*	*#Plug*	*Metric*	*Tiebreaker*
Highest-quality						***Recall***	***Precision***
ab	**165**	**43**	**3**	**947**	**11**	**0.982**	**0.793**
abe	165	43	3	947	11	0.982	0.793
abc	165	45	3	945	11	0.982	0.786
abd	165	45	3	945	11	0.982	0.786
abce	165	45	3	945	11	0.982	0.786
abde	165	45	3	945	11	0.982	0.786
abcde	165	45	3	945	11	0.982	0.786
Individual rank							
b	113	29	55	961	10	0.673	0.796
a	102	18	66	972	8	0.607	0.850
d	69	14	99	976	7	0.411	0.831
e	44	5	124	985	7	0.262	0.898
c	23	6	145	984	3	0.137	0.793
High-quality						***Informedness***	***Recall***
abce	**1841**	**224**	**1**	**5601**	**51**	**0.961**	**1.000**
abcde	1841	224	1	5601	51	0.961	1.000
abe	1838	223	4	5602	51	0.960	0.998
abde	1838	223	4	5602	51	0.960	0.998
abc	1770	224	72	5601	51	0.923	0.961
abcd	1770	224	72	5601	51	0.923	0.961
abd	1767	223	75	5602	51	0.921	0.959
Individual rank							
a	1164	90	678	5735	46	0.617	0.632
b	1013	194	829	5631	46	0.517	0.550
e	436	50	1406	5775	25	0.228	0.237
d	453	148	1389	5677	28	0.221	0.246
c	219	55	1623	5770	18	0.110	0.119
Medium-quality						***F-Measure***	***Recall***
abce	**2386**	**652**	**3**	**9317**	**46**	**0.879**	**0.999**
abcde	2386	652	3	9317	46	0.879	0.999
abc	2383	652	6	9317	46	0.879	0.998
abcd	2383	652	6	9317	46	0.879	0.998
abde	2359	652	30	9317	46	0.874	0.987
abe	2345	652	44	9317	46	0.871	0.982
abd	2328	652	61	9317	46	0.867	0.975
Individual rank							
b	1812	490	577	9479	43	0.773	0.759
a	970	267	1419	9702	41	0.535	0.406
d	717	56	1672	9913	23	0.454	0.300
e	621	21	1768	9948	19	0.410	0.260
c	344	13	2045	9956	18	0.251	0.144
Low-quality						***Markedness***	***Precision***
c	**62**	**3**	**483**	**3591**	**6**	**0.835**	**0.954**
abce	543	117	2	3477	12	0.822	0.823
abcde	543	117	2	3477	12	0.822	0.823
abc	542	117	3	3477	12	0.822	0.823
abcd	542	117	3	3477	12	0.822	0.823
abde	534	116	11	3478	12	0.818	0.822
abe	533	116	12	3478	12	0.818	0.821
Individual rank							
c	62	3	483	3591	6	0.835	0.954
a	244	33	301	3561	10	0.803	0.881
e	73	8	472	3586	7	0.785	0.901
b	377	91	168	3503	10	0.760	0.806
d	51	7	494	3587	9	0.758	0.879

a - phpSAFE, b - RIPS, c - WAP, d – Pixy, e - WeVerca

#Plug - number of plugins where the SATs found vulnerabilities

Software (CAS) specified a methodology, the CAS Static Analysis Tool Study Methodology, that measures and rates the effectiveness of SATs in a standard and repeatable manner [26]. The main goal is to provide objective information to organizations that want to purchase commercial SATs or to use free ones. The metrics used are *precision*, *recall*, *F-Score* (i.e., *F-Measure*), and discrimination rate (*DR*). DR = *#Discriminations*/*#Bugs*, is the fraction of test cases where SATs report a discrimination. A discrimination occurs if a SAT reports a vulnerability in the vulnerable test case and keeps quiet in the non-vulnerable test case. Thus, for each pair of test cases, a discrimination occurs if a SAT reports a vulnerability (TP) present in the vulnerable test case and keep quiet in the non-vulnerable test case (i.e., the tool does not report FPs).

The CAS has created a collection over 81,000 synthetic C/C++ and Java programs with known flaws called as the *Juliet Test Suite,* which is available online [30]. Each test case is a slice of artificial 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 [31]. 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 metrics recall and DR. The metric recall does not consider the number of FPs reported, and the DR severely penalizes SATs that report many vulnerabilities but also reports FPs. Furthermore, they also evaluated the overall coverage (recall) of four combinations of SATs. The SATs were labeled with a number from 1 to 5. Then, the combination of SATs: 12, 123, 1234, and 12345, were evaluated across all the test cases. They concluded that the recall increases as the number of tools increases. However, like in the evaluation of combinations of two SATs this is limited as there are many combinations that were not considered.

Unlike the approaches above, that use a small workload or synthetic simple test cases, **our approach is based on a considerable number of real plugins and four representative vulnerability detection scenarios**. Moreover, the workload is characterized in terms of vulnerable and non-vulnerable LOCs for a more precise classification of the results produced by the tools with respect to TP and FP. Another difference is that our study uses a single main metric to evaluate the combinations of SATs in each scenario that take into consideration the goals of the vulnerability detection in that scenario. In fact, the approaches previously referred use a simple metric such as the number of bugs that each tool founds or several metrics. In this case, the user has the task of choosing the metric that seem most appropriate and use it for evaluating a single SAT or combinations of SATs, without any further guidance.

VII. Conclusion and Future Work

In this work, we addressed the problem of combining the output of several SATs searching for SQLi and XSS vulnerabilities in WordPress plugins. The workload is organized in four scenarios of increasing criticality. Each scenario uses different metrics to rank the tools. Our methodology is generic and can be used with other workloads and tools, either free or commercial.

The results show that combining the outputs of several free SATs does not always improve the vulnerability detection performance. Thus, the best solution can be a single tool or a combination of tools that may not include all the tools under evaluation. In principle, combining multiple static analysis tools has benefits due to the complementarity of the produced results. However, for solutions including SATs that report many FPs the overall performance is worse in some scenarios. In general, as the number of tools in a combination increases, both the new vulnerabilities found and the new FPs reported increase less and less and at different rates.

Future work will focus on two main directions. First, we plan to validate our findings on larger and balanced workloads using more SATs. Second, as static analysis tools have different strengths and weaknesses, we plan to research novel ways to automatically find superior combinations.

References

[1] "Annual consumer studies," http://www.ponemon.org/, Ponemon, 2015.

[2] "Imperva web application attack report," http://www.imperva.com/download.asp?id=509, November 2015.

[3] "WP template.com," http://www.wptemplate.com/tutorials/safety-and-security-of-wordpress-blog-infographic.html, 2016-05-01.

[4] "Website hacked trend report 2016-Q1," https://sucuri.net/website-security/Reports/Sucuri-Website-Hacked-Report-2016Q1.pdf, 2016.

[5] "OWASP code review guide project," https://www.owasp.org/.

[6] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," *14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. 2005, pp. 18–18.

[7] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, May 2006, pp. 6 pp.–263.

[8] J. Dahse, G. Horst, and T. Holz, "Simulation of built-in php features for precise static code analysis," no. February, pp. 23–26, 2014.

[9] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *NDSS*, vol. 2007, 2007, p. 12.

[10] P. Nunes, J. Fonseca, M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *45th IEEE/IFIP Intl. Conference on Dependable Systems and Networks, DSN 2015,* 2015, pp. 299–306.

[11] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proceedings of the 23rd International Conference on World Wide Web*, ser. WWW '14. Seoul, South Korea, 2014, pp. 63–74.

[12] G. Díaz and J. R. Bermejo, "Static analysis of source code security: Assessment of tools against SAMATE tests," *Information and Software Technology*, vol. 55, no. 8, pp. 1462–1476, Aug 2013.

[13] http://samate.nist.gov/, 2016-11-28.

[14] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, Mar 2016, pp. 470–481.

[15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" *35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013.

[16] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, "Effect of static analysis tools on software security: preliminary investigation," *2007 ACM workshop on Quality of protection*. ACM, 2007, pp. 1–5.

[17] N. Rutar, C. B. Almazan, and J. S. Foster, "A comparison of bug finding tools for java," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, ser. ISSRE '04, 2004, pp. 245–256.

[18] Q. Wang, N. Meng, Z. Zhou, J. Li, and H. Mei, "Towards soa-based code defect analysis," in *Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on*, 2008, pp. 269–274.

[19] 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 International Conference on Quality Software*, Aug 2008, pp. 169–174.

[20] N. Antunes and M. Vieira, "On the metrics for benchmarking vulnerability detection tools," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 505–516.

[21] https://colorlib.com/wp/is-wordpress-websites-secure/, 2017-03-09.

[22] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, no. 2, pp. 287–307, 2012.

[23] https://pdepend.org/, 2016-11-03.

[24] D. Hauzar and J. Kofron, "Framework for Static Analysis of PHP Applications," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, 2015, pp. 689–711.

[25] J. Dahse, N. Krein, and T. Holz, "Code reuse attacks in php: Automated pop chain generation," *2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14, 2014, pp. 42–53.

[26] F. G. G. Meade, "CAS static analysis tool study - methodology," https://samate.nist.gov/docs/CAS%202012%20Static%20Analysis%20Tool%20Study%20Methodology.pdf, 2012.

[27] "WPScan vulnerability database," https://wpvulndb.com/, 2015-10-26.

[28] D. M. Powers, "Evaluation a Monte Carlo study," *arXiv preprint arXiv:1504.00854*, 2015.

[29] https://github.com/pjcnunes/EDCC2017.

[30] http://samate.nist.gov/SRD/testsuite.php.

[31] https://media.blackhat.com/bh-us-11/Willis/BH_US_11_WillisBritton_Analyzing_Static_Analysis_Tools_WP.pdf