Benchmarking Static Analysis Tools for Web Security

Paulo Nunes, Ibéria Medeiros, *Member, IEEE*, José C. Fonseca, Nuno Neves,
Miguel Correia, *Senior Member, IEEE*, and Marco Vieira, *Member, IEEE*

***Abstract*—Static analysis tools are recurrently used by developers to search for vulnerabilities in the source code of web applications. However, distinct tools provide different results depending on factors such as the complexity of the code under analysis and the application scenario; thus, missing some of the vulnerabilities while reporting false problems. Benchmarks can be used to assess and compare different systems or components, however, existing benchmarks have strong representativeness limitations, disregarding the specificities of the environment, where the tools under benchmarking will be used. In this paper, we propose a benchmark for assessing and comparing static analysis tools in terms of their capability to detect security vulnerabilities. The benchmark considers four real-world development scenarios, including workloads composed of real web applications with different goals and constraints, ranging from low budget to high-end applications. Our benchmark was implemented and assessed experimentally using a set of 134 WordPress plugins, which served as the basis for the evaluation of five free PHP static analysis tools. Results clearly show that the best solution depends on the deployment scenario and class of vulnerability being detected; therefore, highlighting the importance of these aspects in the design of the benchmark and of future static analysis tools.**

***Index Terms*—Benchmarking, security metrics, static analysis tools (SATs), vulnerability detection.**

Abbreviation and Acronyms

AST	Abstract syntax tree.
BAS	Benchmark accuracy score.
BSA	Benchmark for security automation.
Bugtraq	Electronic mailing list dedicated to issues about computer security.
CCN	Cyclomatic complexity number.
CCN2	Extended cyclomatic complexity number.
CVE	Common vulnerability enumeration.
DR	Discrimination rate.
FN	False negative.
FP	False positive.
FPR	False positive rate.
LOC	Lines of code.
LLOC	Logical lines of code.
N	Negative instances.
NIST	National Institute of Standards and Technology.
NVLOC	Nonvulnerable LOC.
OOP	Object-oriented programming.
OWASP	Open Web Application Security Project.
P	Positive instances.
PoC	Proof of concept.
POP	Procedure-oriented programming.
SAMATE	Software Assurance Metrics and Tool Evaluation.
SAT	Static analysis tool.
SCM	Source code metrics.
SIG	Software Improvement Group.
SPP	Software product properties.
SQLi	SQL injection.
SS	Sensitive sink.
SwMM	Software measures and metrics.
RSV	Reduce security vulnerabilities.
TN	True negative.
TP	True positive.
TPR	True positive rate.
TViT	TV Informationstechnik GmbH.
VLOC	Vulnerable LOC.
WPVD	WordPress vulnerability database.
XSS	Cross site scripting.

Manuscript received August 2, 2017; revised January 8, 2018 and May 5, 2018; accepted May 17, 2018. Date of publication June 28, 2018; date of current version August 30, 2018. Associate Editor: H. Waeselynck. *(Corresponding author: Paulo Nunes.)*

P. Nunes and J. Fonseca are with the Research Unit for the Development of the Interior, Polytechnic Institute of Guarda, Guarda 6300-035, Portugal, and the Centre of Informatics and Systems, University of Coimbra, Coimbra 3004-531, Portugal (e-mail: pnunes@ipg.pt; josefonseca@ipg.pt).

I. Medeiros and N. Neves are with the LaSIGE, Faculdade de Ciências, Universidade of Lisboa, Lisboa 1749-016, Portugal (e-mail: imedeiros@di.fc.ul.pt; nuno@di.fc.ul.pt).

M. Correia is with the INESC-ID, Instituto Superior Técnico, Universidade of Lisboa, Lisboa 1749-016, Portugal (e-mail: miguel.p.correia@ist.utl.pt).

M. Vieira is with the Department of Informatics Engineering, University of Coimbra, Coimbra 3004-531, Portugal (e-mail: mvieira@dei.uc.pt).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2018.2839339

I. Introduction

WEB applications have the remarkable ability to be quickly deployable and instantly accessible to the millions of users. They bring competitive benefits to most business areas, and therefore, the demand for new web applications with complex features is increasing fast. Since their development is often carried out under tight schedules, this can lead to many bugs and security problems. Thus, with no surprise, an Acunetix survey states that 60% of the vulnerabilities found are in web applications [1]. When exploited, their impact may have severe consequences for organizations, including financial losses, liability problems, brand damage, and the loss of market share [2].

Static analysis is one of the most important activities to discover bugs in the early stages of the software development life cycle [3]. In fact, there are estimates saying that static analysis tools (SAT) could detect about half of the existing security vulnerabilities [4]. However, despite being able to cover 100% of the code, static analysis has limitations, such as raising false alarms and missing some application flaws, as the level of their success depends on the factors, such as the complexity of the code, the employed programming constructs, and the included third-party components [5]. Consequently, different tools tend to return quite different results, and the selection of the SAT that best fits a specific project is a challenging task.

Benchmarking could assist in the selection of alternative SATs by comparing their behavior while testing relevant applications. However, the currently available SAT benchmarks are very limited, being the most well-known efforts the Software Assurance Metrics and Tool Evaluation (SAMATE) Project from NIST [6] and the OWASP Benchmark for security automation (BSA) [7]. Besides not producing true to life results, these benchmarks also lack the ability to be tailored to a specific context (e.g., critical or non-critical applications), which may affect the relevance of the results.

This paper proposes an approach to design benchmarks for the evaluation of SATs that detect vulnerabilities in web applications considering different levels of criticality. Contrasting with SAMATE and BSA, we propose the use of *workloads* composed by real applications that have known vulnerabilities (used to exercise the SATs, thus supporting their evaluation). This assures that SATs are tested considering the need to address both the complexity and the way real code is built, instead of processing much simpler synthetic code samples or test cases (as done by SAMATE and BSA). In fact, research shows that SATs perform better with synthetic test cases than with real software [8]. Additionally, by exploring the notion of *application scenarios* (a scenario is a realistic situation of vulnerability detection that depends on the criticality of the application being tested and on the security budget available), our approach allows a better match of its outcomes with the environmental requirements for the SAT operation. In particular, we consider four representative real-world usage scenarios, ranging from the development of business-critical to lower quality applications.

The use of application scenarios in the benchmark raises two fundamental challenges: *how should the SATs be ranked*? and, *how should the workload be created*? To rank the SATs we need several metrics, because no single metric is suitable to quantify all aspects of the performance of SATs in distinct scenarios [9]. Our approach relies on one main metric and a tiebreaker metric for each scenario, where the first is used to rank the tools and the second to decide eventual ties between two or more tools. To compose the workload, we consider a representative group of vulnerable applications for each scenario. Since this is very hard to attain (e.g., 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 standard procedure to assign the applications to scenarios based on their *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 higher criticality. The quality of the applications is measured using a quality model based on the ISO/IEC 9126 standard, relying on a set of SCM (e.g., the cyclomatic complexity), which are related to nonfunctional requirements and can be obtained without running the applications.

To demonstrate our approach, we designed a *benchmark to rank SATs for WordPress plugins*. As WordPress is by far the most popular content management system in use on the Web [10], its plugins are responsible for a huge number of vulnerabilities, e.g., 22% of the 170K sites hacked in 2012 were via vulnerable plugins [11]. Moreover, 25% of all compromised WordPress sites (89 000) analyzed by *Sucuri* [12] in the first quarter of 2016 were due to just three vulnerable plugins (TimThumb, RevSlider, and GravityForms). Our benchmark includes 134 off-the-shelf WordPress plugins, organized in four vulnerability detection scenarios, covering SQL Injection (SQLi) and Cross Site Scripting (XSS) vulnerabilities, which are among the most critical and frequent web application vulnerabilities [13].

The benchmark was used to evaluate five open-source PHP SATs (RIPS, Pixy, phpSAFE, WAP, and WeVerca). Results show that it can be used to rank the SATs, and that different tools have distinct vulnerability detection capabilities, with some performing very poorly in some cases. Moreover, we observed that no tool is the most appropriate for all scenarios, which confirms the relevance of adapting the ranking metrics and the workload considering the characteristics of the scenario where the tools are going to be used. By comparing our results with the SAMATE and BSA benchmarks, we show the relevance of using application scenarios considering specific metrics and tailored workloads.

The contributions of this paper can be summarized as follows:

1) A general approach to design benchmarks for the evaluation of SATs able to detect software vulnerabilities, considering a) workloads that include real vulnerable applications representative of scenarios with different levels of criticality, and b) different ranking metrics.
2) A process to build workloads by collecting vulnerable applications, characterizing them in terms of vulnerable and nonvulnerable lines of code (LOC), and assigning them to scenarios.
3) A process for assigning the applications to scenarios based on their software quality.
4) A concrete instantiation of the general approach to demonstrate its feasibility, evaluating five SATs on the detection of SQLi and XSS vulnerabilities in a workload composed of 134 WordPress plugins organized in four scenarios and using different metrics for each scenario.
5) A comparative evaluation of the ranking obtained using our instantiation with the SAMATE methodology and the OWASP's BSA benchmark.

The outline of this paper is as follows. Section II introduces background concepts and related work. Section III details the proposed approach for benchmarking SATs for vulnerability detection. Section IV describes an instantiation using WordPress plugins as workload. Section V presents the experimental results and Section VI discusses the main properties and limitations of the proposed benchmark. Section VII concludes the paper.

II. Background and Related Work

This section presents the background on benchmarking procedures, a review of SATs for the vulnerability detection and how they can be compared by means of existing benchmarks. An overview of software quality models is also included, as they are important to define the workload for each scenario.

A. Benchmarks

The most common method to assess and compare the performance of alternative tools is to run them with a set of representative test cases and compare the results. A standard process for doing this task is called a *benchmark* [14] [15] and typically includes three main components [16]:

1) Workload, which is a set of representative test cases for the tools under benchmarking.
2) Metrics to compare how the tools under benchmarking fit their purpose.
3) Procedures and rules for the benchmark execution.

The workload is the component most influenced by the benchmarking domain and strongly determines the results. Thus, the workload should ensure the following properties [16]:

Representativeness: The workload should be typical of the domain in which the benchmark will be applied. This is influenced by the size and diversity of the test cases [8]. The benchmark results should provide relevant information to the users in the context of their planned use.

Comprehensiveness: The workload should be able to exercise all the important features typically used in the target domain. Features should be balanced according to usage in real cases.

Focus: The workload should be centered on characterizing the targets under benchmarking. Three criteria should be considered: *coverage* (the need for applications with a broad and complete range of tests), *relevance* (the importance of the load in the context of the benchmark domain), and *ground truth* (ideally, to know the expected result from the execution of the load).

Configurability: Users should be able to customize the workload considering their requirements (scenario).

Scalability: The workload should increase or decrease in number and complexity of test cases preserving the relation with the real application scenario.

B. SATs for Vulnerability Detection

XSS and SQLi have been for many years in the first three places of the OWASP Top 10 of web application security vulnerabilities [13], and they are also two of the most widely exploited ones [17]. A XSS attack consists of the injection of JavaScript in a vulnerable webpage; and a SQLi attack is the injection of code that changes the SQL query sent to the back-end database. These attacks are very dangerous to enterprises and individuals, since they may allow performing undue actions, such as accessing privileged database accounts, manipulating unauthorized database data, impersonating other users to perform actions on their behalf, defacing websites, injecting malware and virus, etc. Many SATs include features for detecting SQLi and XSS vulnerabilities, although their real effectiveness is questionable and not fully understood [8], [18].

SATs inspect the source code of a software program without executing it, to discover potential problems (e.g., security vulnerabilities). These tools are considered by many as the most efficient way to automatically locate vulnerabilities in software [19]–[21]. However, SATs are limited by their nature and often contain bugs themselves [22]. Moreover, some specific programming constructs are difficult to analyze, such as dynamic file inclusion, evaluation of dynamic strings, object-oriented programming (OOP), and automatic typecasts. Therefore, the SAT developers simplify assumptions by producing approximate solutions that frequently lead to false alarms and undetected vulnerabilities [22].

Nowadays, a large number of SATs are available and novel tools are emerging to address new needs. However, different tools have different strengths, depending on the algorithms and technologies used in their development [23], [24]. Due to the diversity of results coming from SATs, especially the tradeoff between soundness and completeness, there is no consensus regarding which one is the best, because false alarms take a lot of time to verify and undetected vulnerabilities may lead to exploits.

In a previous work, we studied this problem arguing that a combination of diverse SATs can improve the vulnerability detection [25]. We evaluated 32 combinations of SATs for different scenarios and vulnerability classes, and showed that the best solution

1) is never composed of all the SATs;
2) changes both across scenarios and classes of vulnerabilities; and
3) in some cases is just a single SAT.

In fact, combining many tools can be counterproductive as this will not lead to the detection of more vulnerabilities, but will increase the number of FPs reported. In the experiments described in [25], we used a benchmark for SATs, for criticality scenarios and a workload of real applications. However, the process of defining and implementing such a benchmark, the problems associated and their solutions were not detailed. This paper fills this gap by proposing and evaluating such benchmark and discussing all the aspects leading to its implementation. It also presents a case study on how to find the most appropriate SAT that satisfies the requirements of each scenario.

C. Benchmarks for SATs

There is no consensus on the metrics to use for evaluating the effectiveness of SATs (coverage, precision, recall, F-Measure, discrimination, etc.). Delaitre *et al.* [8] identified three test case characteristics required to calculate such metrics: statistical significance, ground truth, and relevance. However, in practice, test cases respecting all these characteristics do not exist (or are not publicly available), and creating them is difficult due to the amount of effort that would be required. What we can find are test cases combining two of the characteristics: software with common vulnerability enumeration (CVE) (relevance and ground truth), production software (statistical significance and relevance), and synthetic test cases (statistical significance and ground truth) [8].

Two benchmarks for SATs are the BSA [7] from OWASP and the SAMATE project [6] from NIST. Through the development of tool functional specifications [26], test suites, and tool metrics, the SAMATE project establishes a methodology to understand the capability of SATs against a set of weaknesses. The workload contains test suites for C/C++, Java, and PHP code, and includes a variety of test cases inspired on real applications, applications specifically developed for the benchmark and code written by students. The metrics used to evaluate the tools are the false positive rate, precision, and recall.

The BSA from OWASP is a free and open test suite to evaluate the speed, coverage, and accuracy of automated SATs and services [7]. The workload contains over 20K Java test cases that are fully runnable and exploitable, including 11 classes of vulnerabilities. Each category comprises test cases with and without vulnerabilities. Instead of real applications, the test cases are small pieces of code with less than 100 lines, derived from coding patterns observed in real applications.

Evaluating the effectiveness of SATs using the SAMATE and the BSA benchmarks requires some manual work: running the SATs for detecting vulnerabilities in the synthetic workloads, converting the results of the SATs to a common format, comparing the results with the expected ones, and computing the chosen evaluation metrics. The main limitation of both SAMATE and BSA is the synthetic workload, which is composed mainly by simple small test cases with few programming constructs, that may not be representative of production code, limiting the validity of the results in real conditions [8]. Also, the evaluation procedure does not consider the specific characteristics of the scenario where the tools are to be used. This contrasts with the reality, where applications are large and complex. Thus, with these test cases, it is very difficult to evaluate the real effectiveness of the SATs. In our benchmark, we propose a workload composed of software in production, i.e., real applications that have real vulnerabilities. The workload is built following a process to characterize the applications in term of vulnerable LOCs and nonvulnerable LOCs, and assign them to scenarios based on their software quality.

Kupsch and Miller [27] compared the results of two commercial SATs with an in-depth manual vulnerability assessment. The SATs just found a few of the several vulnerabilities discovered in the manual assessment and missed many vulnerabilities requiring a deep understanding of the source code.

In addition to the workload issues, another limitation of existing works is the use of the same metrics independent of the environment where the vulnerability detection is going to be performed (projects have specific goals and constraints regarding criticality and budget). We aim to improve these aspects by using a representative set of real web applications with real vulnerabilities, and to use different evaluation metrics to rank the tools according to the scenario considered.

D. Software Quality

Several software quality models were proposed and many tools were created to control the development and maintenance of software [28]–[31]. Among others, these tools are used to identify the problems in the source code early in the development process, allowing project managers to take mitigation actions. In fact, several studies show that there is a relation between the quality of the source code and the failures of software products [32]. For example, it is known that the code units that have the highest complexity also tend to contain more defects [33].

Web applications have characteristics in common with traditional software; however, they also have unique characteristics that are related to the distributed nature of the Internet, use and reuse of third-party components developed in multiple languages, web interfaces with users, the speed of access to data, and the security of transactions [34]. Thus, traditional software quality models may not be adequate to fully assess the quality of web applications. Since web applications became an indispensable platform in all sectors of our society, researchers proposed models for assessing the quality characteristics of web-based applications. Nabil *et al.* [35] proposed a software quality model for web-based applications that extends the ISO 9126 software quality model by adding characteristics, such as reusability, scalability, credibility, security, popularity, and profitability, etc. They organized these characteristics in three views: developers, owners, and visitors. Sankar *et al.* [34] proposed common quality attributes for secure web applications organized in four quality categories: design, run-time, system, and user.

Several SCMs, such as the Extended Cyclomatic Complexity Number (CCN2, a variation of the Cyclomatic Complexity (CCN) adapted for OOP [36]) [37], or the number of Logical Lines of Code (LLOC), have been proposed to measure quantitatively the quality of software products [38]. A common method for aggregating SCMs is to build a risk profile based on a set of predefined thresholds for the SCM [39]. This allows developers to focus on software units where SCMs are exceeding the thresholds first and the others later, as units with higher values for several SCMs tend to have more faults [32].

An appropriate use of SCMs requires risk thresholds to determine whether the value of a SCM is acceptable or not. These risk thresholds vary widely in the literature. For example, the limit of 10 for CCN was proposed by McCabe [36], but limits as high as 15 have also been used successfully [40]. In fact, the risk thresholds are defined based on the opinion of software quality experts for particular contexts [41]. For example, in high quality software, it is admissible to have small percentages of source code with high values for some SCMs to express a balance between real needs and idealized design practices [41].

Alves *et al.* [38] proposed a methodology for deriving threshold values for the SCMs based on data analysis from a representative set of applications. This methodology has been successfully used in several works. One example is the method proposed by Baggen *et al.* [42] to rate the maintainability of the source code of applications (from 0.5 to 5.5 stars) based on risk profiles and a set of rating thresholds. The Baggen *et al.* method is applied by the SIG to annually recalibrate its quality model [43], which forms the basis of the evaluation and certification of software maintainability conducted by SIG [44] and TViT [42].

The static analysis community has recognized that the analysis of source code is harder than usually assumed [45]. The participants of the NIST workshop on software measures and metrics to reduce security vulnerabilities (SwMM-RSV)

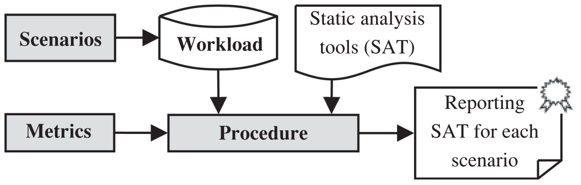

Fig. 1. General architecture of the benchmark.

recommended that code should be amenable to automatic analysis [45]. Therefore, the analyzability (subcharacteristic of maintainability) should be measured and increased to make the code readily analyzable. This contributes to reduce vulnerabilities, as tools tend to perform better in less complex code.

In our benchmark, the rating of applications (to distribute test cases according to the specificities of the development scenario) is done based on the Baggens *et al.* method. In practice, we define four range thresholds to map the applications with the four scenarios. Neither of the other existing benchmarks (SAMATE and BSA) uses software quality models to define representative workloads.

III. Benchmarking Approach

Our benchmarking approach follows a specification-based style, where the specification defines the functions that must be achieved by the target tools, the required inputs (workload), and the outcomes (vulnerabilities and metrics) [46]. Essentially, the idea is to run the target SATs using as input a set of real-world vulnerable software and, after gathering the vulnerabilities identified by the SATs and verifying their correctness, use a small set of metrics that summarize the detection capabilities of the tools to obtain a ranking for each development scenario.

The high variety of applications constructed with heterogeneous components and the diversity of vulnerability classes make it unfeasible to define a benchmark for all SATs in all situations. Therefore, a benchmark should be specifically built or configured for a particular domain to allow making educated choices during the definition of the components [9]. In this paper, defining the benchmark domain directly affects the workload and includes selecting the class of web applications (banking, social networking, etc.) and the classes of vulnerabilities (SQLi, XSS, etc.) to be detected by the target SATs. Also, the strengths and weaknesses of the workload depend on the balance of several criteria, often conflicting. Since no single workload can be strong in all criteria, there will always be a need for considering multiple workloads [46]. Therefore, our proposal is to define a set of workloads according to specific scenarios. Moreover, the workload should be built using a representative set of real software code with vulnerabilities.

The overall SAT benchmark architecture is illustrated in Fig. 1. Our approach is composed of four components that are introduced next and detailed in the following sections:

1) *Scenarios:* Requirements representing real contexts, with constraints with different criticality, where SATs will be used.
2) *Metrics:* Used to characterize and compare the effectiveness of the tools under benchmarking, in each specific scenario.
3) *Workload:* Representative applications, with a set of vulnerabilities, to be used in each scenario. The classes of vulnerabilities (SQLi, XSS, etc.) should be representative of the target application domain.
4) *Procedure:* The process to execute the benchmark using the workload. For each scenario, the benchmark produces a report with the ranking of the SATs under benchmarking, ordered using the relevant metrics.

A. Scenarios

A scenario should be based on the technical needs and business impact of the applications in an organization, by means of requirements in terms of the level of security that should be satisfied and the amount of resources available during development. As an example, for a high-quality scenario (e.g., home banking), one wants to select the SAT with the highest detection rate, even if it raises more false alarms than others, since any vulnerability that is left undetected may have a high impact if successfully exploited. Therefore, all resources that are required to check the warnings produced by the SAT and to fix the vulnerabilities are assumed to be available. On the other hand, for medium-quality scenario (e.g., corporate site), one may want the SAT with the highest detection rate, but that does not raise too many false alarms, since the resources available to deal with those false alarms are not unlimited.

In our approach, four criticality levels representing realistic scenarios are considered. We adapted the names of the scenarios defined by Antunes *et al.* [9] to better represent their requirements, but maintaining their scope:

1) *Highest quality:* Every vulnerability missed may be a big problem due to the criticality of the application. For this scenario, the goal is to select the SAT reporting the highest number of vulnerabilities even if reporting many false alarms.
2) *High quality:* Given that the criticality of applications is not the highest, a few vulnerabilities may be missed if that lowers the number of false alarms. For this scenario, the goal is to select the SAT reporting the highest number of vulnerabilities but not too many false alarms.
3) *Medium quality:* Vulnerabilities may be missed at the cost of reducing the false alarms. For this scenario, the goal is to select a SAT reporting few false alarms at the cost of skipping some vulnerabilities.
4) *Lowest quality:* Every false alarm is an important cause of concern due to tight budget restrictions. The goal for this scenario is to select the SAT reporting the lowest number of false alarms while still reporting vulnerabilities.

The definition and usage of scenarios is very helpful for software developers and decision makers because they can control the acceptable/expected outcomes of the static analysis process for each project that fits in a scenario.

B. Metrics

To compare the results and rank the benchmarked SATs, we propose the use of metrics that are adequate to the vulnerability detection scenario. For each scenario, we propose one main metric to rank the tools and a tiebreaker metric used only when

TABLE I
SUMMARY OF METRICS BY SCENARIO

Scenario	Metric	Tiebreaker
1 - Highest-quality	Recall	Precision
2 - High-quality	Informedness	Recall
3 - Medium-quality	F-Measure	Recall
4 - Low-quality	Markedness	Precision

there is a tie among tools (see Table I), adapted from Antunes *et al.* [9]. In practice, the metrics depend on the vulnerability detection goals, which are related with the amount of available resources to fix the vulnerabilities. For example, in the highest quality scenario the chosen metric is recall, which allows finding the highest number of vulnerabilities at any cost, even ignoring the precision of the results. Only in the case of a tie, the precision is used to rank first the tool that reports less false alarms.

The next paragraphs describe the evaluation metrics and present some arguments why the metrics portray the effectiveness of the SATs in each scenario, considering that the number of Negative (N, nonvulnerabilities) instances are more frequent than the Positive (P, vulnerabilities) instances in the workload. The metrics *Recall* and *F-Measure* are focused on all P instances. The metric *Precision* is focused on some P instances (i.e., the outcomes of the SATs). The metrics *Informedness* and *Markedness* focus on all P and all N instances, by means of true positives (TP, vulnerabilities classified correctly), false positives (FP, nonvulnerabilities classified incorrectly or false alarms), true negatives (TN, nonvulnerabilities classified correctly), and false negatives (FN, vulnerabilities classified incorrectly or missed vulnerabilities) [47].

1) *Recall:* The proportion of true vulnerabilities are correctly identified as such, ranking first the SAT reporting the highest number of TPs required for the highest quality scenario, and for the high-quality and medium-quality scenarios in case of a tie

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \tag{1}$$

2) *Informedness:* How consistently a SAT predicts the outcome of both a TP and a TN, i.e., how informed a SAT is for the specified condition, versus chance. Every TP increases the metric in the proportion 1/P and every FP decreases the metric in the proportion 1/N. Since the prevalence of P instances is less than the prevalence of N instances, the metric prioritizes SATs reporting more vulnerabilities and at the same time not too many FPs, which is the goal of the high-quality scenario

$$\text{Informedness} = \text{Recall} + \text{Inverse Recall} - 1 \tag{2}$$

$$\text{Informedness} = \frac{\text{TP}}{\text{TP} + \text{FN}} + \frac{\text{TN}}{\text{FP} + \text{TN}} - 1. \tag{3}$$

3) *Precision:* Proportion of the classified positive cases that are correctly classified. This metric is used only as tiebreaker. Thus, from a list of tools reporting the same number of vulnerabilities, the best one is the SAT with highest precision (i.e., less FPs reported)

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{4}$$

4) *F-Measure:* The harmonic mean of precision and recall. In this metric, the TPs have twice the weigh of the FPs. Thus, it is suitable for the medium-quality scenario where it is preferable to fix less vulnerabilities than fix more vulnerabilities, and at the same time to consume resources checking FPs

$$F - \text{Measure} = \frac{2 \times \text{TP}}{2 \times \text{TP} + \text{FP} + \text{FN}}. \tag{5}$$

5) *Markedness:* How consistently the SAT has the outcome as a marker, i.e., how marked a condition is for the specified SAT, versus chance. The metric sums the proportions of the positives and the negatives that are correctly identified as such. The Precision (first part of the formula 6) focus on the FPs reported by the SATs and handles only part of the N instances. Therefore, based on the Precision, a SAT reporting no FPs is better than a SAT reporting all vulnerabilities but at least one FP. This fits with the required for the low-quality scenario, where there are no resources for addressing FPs. The inverse precision (second part of formula 6) focus on the FNs (i.e., the vulnerabilities that were left undetected) and handles only part of P instances. Thus, the metric for SATs with the same inverse precision, ranks first the SAT reporting more vulnerabilities. Thus, the metric portrays the required goal for the low-quality scenario

$$\text{Markedness} = \text{Precision} + \text{Inverse Precision} \tag{6}$$

$$\text{Markedness} = \frac{\text{TP}}{\text{TP} + \text{FP}} + \frac{\text{TN}}{\text{FN} + \text{TN}}. \tag{7}$$

C. Workload

The perfect workload is a large set of real applications of diverse sizes, developed according to typical industry practices and with all vulnerabilities identified [45]. However, such workload does not exist and creating it is an unfeasible task that would consume immense resources. To limit this problem, we propose a process based on the results of several SATs combined with manual review for finding vulnerabilities and nonvulnerabilities in real software. The proposed process to build the workload is presented in Fig. 2, and it involves three stages (illustrated by the gray boxes in the figure), which are discussed in the following sections.

Stage 1: Collecting the source code of vulnerable applications The methodology to select a representative set of vulnerable applications to define the workload includes the following steps, represented in Fig. 3:

1) Choosing applications in the benchmarking domain for which source code is available (SATs require the source of the application to detect vulnerabilities).
2) Choosing the classes of vulnerabilities that are relevant in the benchmark domain.

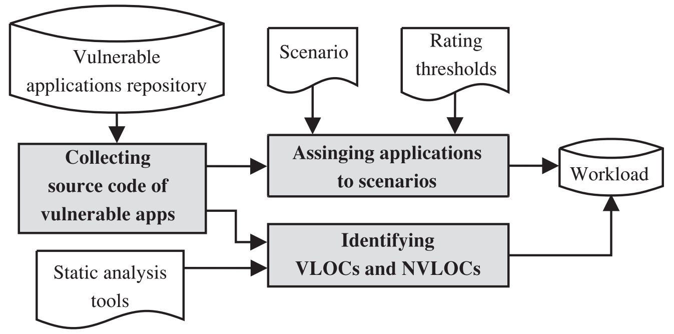
Vulnerable applications repository
Scenario
Rating thresholds
Collecting source code of vulnerable apps
Assinging applications to scenarios
Workload
Identifying VLOCs and NVLOCs
Static analysis tools

Fig. 2. Process to compose the workload.

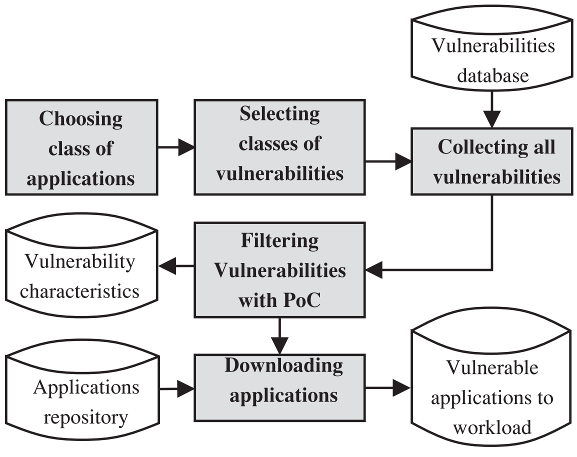
Vulnerabilities database
Choosing class of applications
Selecting classes of vulnerabilities
Collecting all vulnerabilities
Vulnerability characteristics
Filtering Vulnerabilities with PoC
Applications repository
Downloading applications
Vulnerable applications to workload

Fig. 3. Process for collecting vulnerable applications.

3) Collecting all vulnerabilities of the chosen applications registered in their development repository or from vulnerability databases, e.g., WPVD, CVE, and Bugtraq.
4) Selecting only vulnerabilities with a Proof of Concept (PoC), i.e., vulnerabilities for which a proof that they can be exploited exists (i.e., the attack that can be done and the code of the application that is affected by it, are available).
5) Downloading the applications with the vulnerabilities with PoCs from source code repositories.

A major advantage of this methodology over existing benchmarks (like those from NIST and OWASP) is the representativeness of the vulnerabilities since they exist in real applications and are proven to be exploitable.

Stage 2: Assigning applications to scenarios

To compose the workload, we need to assign a representative group of vulnerable applications to each scenario. This process has two steps (see Fig. 4):

1) *Assigning ratings to applications.* This is based on the approach proposed by Baggen *et al.* [42] 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 SCMs (e.g., cyclomatic complexity number (CCN2) [37]). These SCMs are used to measure the software product properties (SPPs) (e.g., Unit Complexity) of the software. Table II lists the SCMs used to measure the SPPs including the level of measure. Table III outlines the SPPs and their relationship with the subcharacteristics of maintainability. The table also includes an example of assigning a rating to an application. The ratings of the subcharacteristics are obtained by averaging the ratings of the selected properties (marked with a "×"). The final rating is obtained by adding 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 approach is implemented in three major steps and requires as input: i) a large set of applications in the benchmarking domain; ii) a set of percentages of code to represent; and iii) a table of rating thresholds for each SCM. For i), we use the applications of our workload, and for ii) and iii), we adopt the same values proposed by Baggen *et al.*, since they were successfully

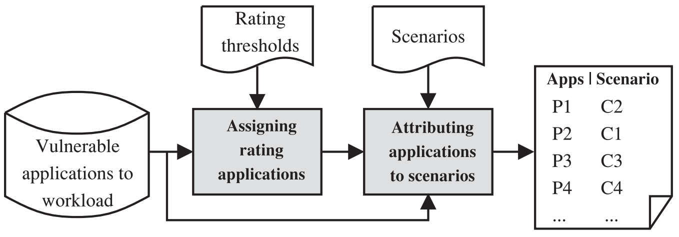
Rating thresholds
Scenarios
Vulnerable applications to workload
Assigning rating applications
Attributing applications to scenarios
Apps | Scenario
P1 C2
P2 C1
P3 C3
P4 C4
... ...

Fig. 4. Process for assigning applications to scenarios.

TABLE II
SCMs FOR EVALUATING THE SPPs

SPP	SCM	Level	Description
Duplication	DLD	App.	Duplicated Line Density [48] [49]
Size	LLOC	Unit	Logical Lines of Code
Size	WMC	Class	Weighted Method Count [50]
Complexity	CCN2	Unit/ Class	Extended Cyclomatic Complexity Number [37]
Coupling	CBO	Class	Coupling Between Objects [51]
Interfacing	NPARM	Unit	Number of parameters in functions and methods
Class Interface	CIS	Class	Number of non-private methods and properties [52]
Testing	NPATH	Unit	Number of execution paths

TABLE III
MAPPING OF SOFTWARE PRODUCT PROPERTIES TO ISO/IEC SUBCHARACTERISTICS OF MAINTAINABILITY AND AN EXAMPLE

	Software Product Properties								
Sub-characteristics	Duplication	Unit complexity	Unit size	Module coupling	Class Complexity	Unit interfacing	Class interface size	Unit Testing	Average
Ratings example	5.0	4.0	3.0	4.0	3.0	3.0	2.0	3.0	
Analyzability	×	×	×						4.0
Changeability	×	×		×	×				4.0
Stability						×	×	×	2.6
Testability		×	×	×				×	3.5
Maintainability rating (average: ⋆ ⋆ ⋆ ⋆)									**3.5**

used in several works. The first step is the extraction of the values of the SCMs. There are many free tools for gathering the SCMs (e.g., PHPdepend [37]). Afterward, using the values of all SCMs of all applications, we derived the ratings for each SCM of each application. Finally, we obtain the ratings of the applications by averaging the ratings of the subcharacteristics of maintainability, as described before.

2) *Assigning applications to scenarios:* This step is based on a simple scheme for mapping the ratings with scenarios. Since the ratings vary from 0.5 to 5.5 in ascending quality order and the scenarios from 1 to 4 in descending 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 (low quality). As shown, we used intervals of one value for mapping the ratings into the scenarios, trying to respect Baggens approach, except for the less stringent (and less relevant) scenario, which accommodates all the ratings below 2.5 (code of lower quality).

Stage 3: Identifying vulnerabilities and nonvulnerabilities

To evaluate a SAT, we need to know which LOCs are vulnerable (i.e., positive instances (P = TP + FN), or VLOCs) and which LOCs are not vulnerable (i.e., negative instances (N = FP + TN), or NVLOCs). However, for large code bases, this is a hard task that requires a thorough review by security experts, and the result may not be completely accurate (as experts can also make mistakes). To address this problem, we propose the procedure discussed next.

A vulnerability may manifest in a restricted set of constructs (e.g., XSS in the PHP echo, mysqli_query, etc.) of the programming language, named sensitive sinks (SSs), which can be viewed as site (location) in the code that can be exploited if some malcrafted input is used as argument [53]. Although the number of VLOCs in an application is limited to the lines that include such constructs, the number of vulnerabilities can potentially be greater than the number of SS, as one SS may have several vulnerabilities. For example, the PHP `echo ''$name $city,''` may have two XSS vulnerabilities (due to the two variables used). In this paper, we count the vulnerabilities at the level of the LOC, meaning that a LOC with one or more vulnerabilities counts as one positive instance. Next, we discuss how to characterize the VLOCs and NVLOCs and present the method to obtain them.

a) Characterizing VLOCs: The initial list of VLOCs of the workload are the vulnerabilities (i.e., the LOCs where the vulnerabilities are located) resulting from the process of collecting source code of vulnerable applications (see Stage 1 of this Section), specifically, exploitable vulnerabilities registered in public vulnerability databases. However, the number of these vulnerabilities is very low, probably lower than the real number, breaking the ground truth of the workload. Our approach to find more VLOCs in the workload is based on running several SATs and on a manual review to confirm the results. Thus, to obtain the VLOCs, we run the SATs to scan for vulnerabilities in the selected applications; then, the outputs are combined and each candidate vulnerability is manually reviewed to determinate if it is a TP (i.e., vulnerability) or a FP (i.e., nonvulnerability). Thus, the initial VLOCs merged with all TPs become the list of positive instances (P) and all FPs becomes part of the list of negative instances (N) or nonvulnerable LOCs (NVLOCs). Note that, if available, other approaches for detecting vulnerable lines of code may be integrated in the benchmarking process.

b) Characterizing NVLOCs: The first list of NVLOCs is limited to the FPs reported by the tools used in the process above. Therefore, if the tools report few FPs, the size of the set will be small, so the values of the metrics that depend on the number of NVLOC (e.g., informedness) would not be representative if only those were considered. A naive way to identify more NVLOCs would be to calculate the difference between all the LOCs and the VLOCs. However, this would result in an extreme unbalance between VLOCs and NVLOCs. In this case, FPs would have a very small (or negligible) effect on the metrics based on the NVLOCs. For example, the results for the informedness metric would become very similar (slightly lower) to the results for the recall metric; thus, loosing usefulness. To overcome this problem, we propose to consider as NVLOCs only the LOCs that constitute a SS with at least one variable, but for which no vulnerability is known or has been detected. SS function calls without any variable are not possible to be vulnerable; thus, they are not considered as NVLOCs. This way, the list of NVLOCs in the workload is created by merging two lists: the first comes from the process of characterizing vulnerable LOCs (previous subsection), which contains the FP identified there, and the second results from a process to identify more NVLOCs based on the SSs in the source code.

c) Obtaining VLOCs and NVLOCs: The method for obtaining VLOCs and NVLOCs includes six steps:

1) Identify the set of SATs to be used for defining the list of VLOCs and NVLOCs. This includes the definition of the configuration settings for the selected tools.
2) Detect vulnerabilities by running the SATs on the workload applications. From this step results a list of candidate VLOCs.
3) Manually verify the vulnerabilities reported by the tools and classify them as VLOCs or NVLOCs.
4) Create the list of VLOCs by merging the initial list of vulnerabilities with PoC that resulted from the process of collecting the source code of vulnerable applications (see Section III-C) and the VLOCs from Step 3).
5) Create the set of NVLOCs by merging all distinct FPs reported by the tools, which will compose the first part of NVLOCs in the workload. The second part is composed by the LOCs where a SS function is called, having at least one variable, but excluding the LOCs that were labeled as VLOCs in the previous step.
6) Characterize the set of VLOCs, including information on the vulnerable file, the LOC, and vulnerable variable, the class of vulnerability, the source inputs, and SSs.

We are aware that the process for extracting VLOCs and NVLOCs may leave some vulnerabilities undetected. Consequently, an issue may occur during the execution of the benchmark if one or more SATs report previously unknown

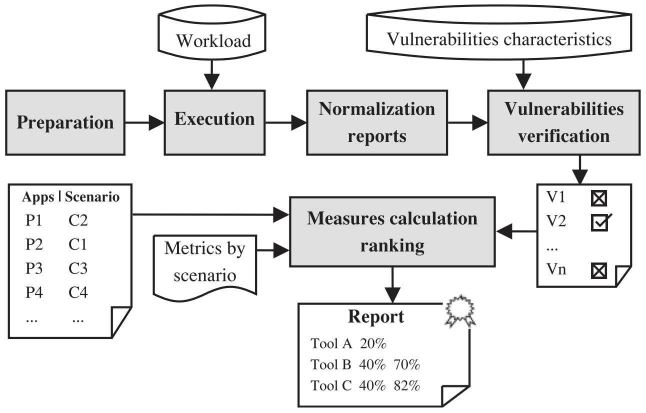

Fig. 5. Benchmarking procedure.

vulnerabilities. This requires a manual review to classify such findings as TPs or FPs. This allows updating the list of VLOCs and NVLOCs, but also changes the values of the metrics/ranking of SATs previously benchmarked, which may also need to be updated. Although a best effort approach, the usage of several SATs in the LOCs characterization process would minimize this problem; therefore, reducing the probability of a SAT to detect unknown vulnerabilities.

D. Procedure

The benchmarking procedure is a well-defined set of steps and rules that must be followed to implement and run the benchmark (see Fig. 5):

1) *Preparation:* Identifying the SATs to be benchmarked. Different tools are executed in different ways, as they have diverse features, configurations, and user interfaces, thus, whenever possible, the tools must be configured according to the characteristics of applications in the benchmark domain.
2) *Execution:* Running the SATs under benchmarking to detect the vulnerabilities in the workload.
3) *Normalization of reports:* As each tool delivers the results in a specific format, they must be normalized and merged into a single report with a standard format, including the following information: the LOCs reported as vulnerable, the files where they were found, the vulnerability class, and the application where they were discovered.
4) *Vulnerability verification:* Analysis of the SATs' results by applying three verifications. First, the vulnerabilities reported by the SATs that belong to the list of VLOCs (i.e., TPs) are automatically verified by a grading program to confirm their correctness. Second, the vulnerabilities reported by the SATs but that belong to the list of NVLOCs (i.e., FPs) are also automatically verified by the grading program. Third, the vulnerabilities reported by the SATs that do not belong to the list of VLOCs or NVLOCs require a manual verification to confirm their vericity, and then update the lists of VLOCs and NVLOCs according to the results of the manual review (i.e., adding such vulnerabilities to the lists: if the vulnerability reported is a TP, the VLOC is updated; otherwise, if it is a FP, the NVLOC list is updated).
5) *Metrics calculation and ranking:* Based on the SAT outputs and their verification (previous step), the benchmark metrics are calculated automatically. Afterward, SATs are ranked according to the metrics recommend for each scenario (see Table I).

IV. Benchmark Instantiation

The benchmarking approach presented before intends to be generic, meaning that it may be applied to any type of application and class of vulnerabilities for the evaluation of any set of SATs. In this section, we present an instantiation of the approach, so we need to specify the workload and targets, as detailed in the following sections.

A. Target Applications and Vulnerability Classes

The target applications are WordPress plugins and the target vulnerabilities are SQLi and XSS, which are two of the most common web application security vulnerabilities [13] and also two of the most widely exploited [17].

WordPress is extensible by means of PHP-based resources such as plugins that allow the addition of new features, templates, functions, etc. They are so common that the WordPress Plugin Directory [54] contains around 50 000 plugins with over 1.5 billion downloads. Besides being so widely used, plugins also pose a serious security problem, as many of them are developed without proper care, as shown by recent research [55]. In fact, they are responsible for lots of vulnerabilities and, since a single plugin may be used in thousands of websites, they are an attractive target for hackers.

B. Workload

Next, we describe the composition of the workload for the benchmark, based on WordPress plugins, and following the process presented in Section III-C.

1) Collecting the Source Code of Vulnerable Applications: We used the online WPScan Vulnerability Database (WPVD) to collect WordPress plugins with SQLi and XSS vulnerabilities, including PoC and more details, like the CVE identifier [56]. The result was a list of 134 WP plugins with 152 SQLi and 67 XSS vulnerabilities registered. A total of 42 of these plugins contain both classes of vulnerabilities, while 79 contain only SQLi and 13 only XSS. To have an idea of their relevance, overall these plugins have been downloaded over 77 million times and they are used in business, e-commerce, monetization, social networking (Google, Facebook, Youtube), photo and video gallery, registration, admin, advertising, email, bookings, reservations, events management, newsletter, e-learning, and document manager. The list of plugins, including rating information, distribution per scenario, vulnerabilities and other details, is available online [57]. It is important to emphasize that, using the workload that we created, researchers can evaluate other SATs with little effort.

TABLE IV
PLUGIN BACKGROUND INFORMATION

Scenarios	OOP	POP	Total	%Tot.	Files	LLOC	%LLOC
Highest-quality	10	2	12	8.9	352	19542	4.2
High-quality	39	17	56	41.8	1687	122835	26.4
Medium-quality	40	11	51	38.1	2208	211297	45.3
Low-quality	14	1	15	11.2	728	112490	24.1
Total	103	31	134	100	4975	466164	100

TABLE V
DISTRIBUTION OF VULNERABILITIES AND NONVULNERABILITIES BY SCENARIOS AND TOOLS/VD

	Scenario	phpSAFE		RIPS		VD	Total	NV	Total	
		TP	FP	TP	FP		FP		P	N
SQLi	1	29	5	0	0	17	5	84	41	89
	2	274	58	43	2	35	60	1068	308	1128
	3	99	50	153	113	22	163	2053	251	2216
	4	36	32	1	0	10	32	1105	46	1137
	Total	438	145	197	115	84	260	4310	646	4570
XSS	1	96	16	113	29	3	43	947	168	990
	2	1149	76	887	188	1	223	5673	1767	5896
	3	951	264	1775	487	4	652	9370	2315	10022
	4	244	33	369	89	5	116	3481	535	3597
	Total	2440	389	3144	793	13	1034	19471	4785	20505
	Total	2878	534	3341	908	97	1294	23781	5431	25075

The plugins were developed in PHP using POP and OOP. Notice that, in PHP, the presence of classes does not imply an object-oriented design. Thus, it is frequent to find procedural code using objects and OOP code using procedural code. Overall, we have 31 POP plugins and 103 OOP plugins (see Table IV). The workload contains 466 164 LLOCs (Logical Lines of Code), where 39.5% are POP, 47.8% OOP, and 12.7% a mix of both. The number of LOCs is 1023 081, where 32% are POP, 57% OOP, and 11% a mix of both.

2) Assigning Applications to Scenarios: For gathering the measures of the SCMs to evaluate the SPPs listed in Table II, we used three tools: PHPdepend v2.5.0 [37] for gathering the LLOC, WCM, CCN2, CBO, CIS, and NPTAH metrics; SonarQube v5.2 [30] for the DLD metric; and PHPMD v2.6.0 for the NPARM metric.

The results of applying the methodology for assigning applications to scenarios (see Section III-C) are presented in Table IV, which shows the number of plugins that compose the workload, distributed over the four scenarios. Scenario 1 (highest quality) has a lower number of plugins compared with 2 (high quality) and 3 (medium quality). This is realistic as code with very high quality is not so common. Moreover, 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 (scenario 4).

The SATs failed analyzing several files of the plugins (see Section V). We observed that the percentage of LLOC where at least one SAT failed the analysis decreases as the quality of the code increases. The percentages are 56% for the low-quality scenario, 50% for the medium-quality scenario, 35% for the highest quality scenario, and 38% for the high-quality scenario. This shows that the code with better quality (i.e., less complex, as recommended by the participants in the SwMM-RSV NISTs workshop [45]) increases the probability of being successfully analyzed by the SATs. As a side effect, this contributes to reducing vulnerabilities in the software, since they are more likely to be detected by the SATs. Because users have different requirement constraints regarding the code quality and this in turn has a direct impact on the ability to detect vulnerabilities, it is very important to have the benchmark configured for scenarios based on the internal software quality.

3) Identifying Vulnerabilities and Nonvulnerabilities: The first part of the list of VLOCs was given by the information collected from the WPScan Vulnerability Database (WPVD). To obtain the second part, we ran two free SATs, RIPS [58], and phpSAFE [55], to scan for vulnerabilities in the workload. The SATs were configured by default for PHP entry points, SS, and sanitization functions (e.g., htmlentities, mysql_real_escape_string). The results were manually reviewed as defined in our design approach (see Section III-C).

The list of NVLOCs considered is the combination of the FPs reported by the tools with the list of LOCs that have at least one SS outputting at least one variable. For this, we developed a PHP script for gathering all SS function calls of the source code files based on their AST. From this list, we removed those already labeled as VLOCs. The script was executed individually for each file. A manual check of random samples has been performed to increase the trust on the accuracy of the NVLOCs identified.

Table V presents the results obtained using our approach. Overall, 7725 (FP: 1,294 + TP: 5,431) LOCs were extracted from the outputs of the tools and manually reviewed (80.8% of TPs and 19.2% of FPs). The table depicts, for each tool, the number of TPs and FPs, followed by the number of vulnerabilities in the WPVD database (column VD) [56]. The column Total FP is the union of the FPs of the tools, and NV shows the NVLOCs obtained in the previous step (VLOC characterization). The two last columns show the number of positive instances and negative instances (combination of the FPs with the NV). These columns are used for calculating the metrics and should be updated during the execution of the *benchmarking procedure* if a SAT under testing reports a new vulnerability.

An important aspect regarding the VLOCs is that the number of vulnerabilities reported in the WPVD (97, see Table V) is far from the reality (5431, see Table V). In fact, we were able to detect a much larger number of true vulnerabilities using the SATs. This emphasizes the capability and relevance of static analysis to detect the vulnerabilities.

V. EXPERIMENTAL EVALUATION

The main goal of this experimental evaluation is to demonstrate the benchmark, validate the benchmarking process and, at the same time, confirm/deny the following hypothesis:

H_1 The best SAT is the same across different scenarios.

H_2 The best SAT is the same across different classes of vulnerabilities.

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. In practice, we evaluated the following tools: RIPS v0.55 [58], Pixy v3.03 [20], phpSAFE [55], WAP v2.0.1 [59], and WeVerca v20150804 [60]. RIPS and Pixy are the two most referenced PHP SATs in the literature, but they are not ready for the OOP analysis. Pixy performs tainted analysis and alias analysis, but has not been updated since 2007, and RIPS has only been developed as open source until 2014. RIPS recently released a commercial version able to fully analyze the OOP code [61]. WAP, phpSAFE, and WeVerca are more recent tools under active development, and they are prepared for OOP code. In terms of configuration, phpSAFE, RIPS, WAP, and Pixy are configured by default for PHP entry points, SSs, and sanitization functions (e.g., *htmlentities*, *mysql_real_escape_string*). WeVerca does not allow configuration and includes, out of the box, a programmed list of entry points, SSs, and sanitization functions.

A. Ranking the SATs

We ran the benchmark for all the SATs searching for XSS and SQLi vulnerabilities in the workload plugins. Overall, WAP was able to analyze all plugins, but seven of them only partially. Pixy analyzed partially 103 plugins (i.e., fails in 1473 files) and WeVerca was not able to analyze 20 source files of 14 plugins. phpSAFE was unable to fully analyze 18 plugins (130 files), taking a very long time on those plugins without returning any results. RIPS outputted the message "*Code is object-oriented. This is not supported yet and can lead to false negatives*" for 76 plugins (2179 files). In practice, the tools could not fully analyze some plugin/files, reporting runtime errors or taking a very long time without any results. This results from limitations of the SATs used, potentially due to the size/complexity of some files.

The results by scenario for SQLi and XSS vulnerabilities are listed in Tables VI and VII, respectively. The columns TP, FP, FN, and TN show the confusion matrix for the corresponding SAT. The data in the table are first ordered by the main metric (*Metric*), and second ordered by the tiebreaker metric (*Tiebreaker*), as recommended for each scenario (in the current case, the tiebreaker metric was not needed, but previous work found cases where a tiebreak metric was useful [9]). The *Plugins* column shows the number of plugins, where the SATs found vulnerabilities. The ratings of the SATs in the tables are relative, not absolute.

Focusing on SQLi (see Table VI) the tool chosen for each scenario was: WAP for the *highest quality* scenario; phpSAFE for the *high quality*; RIPS for the *medium-quality scenario*, despite having detected vulnerabilities in just a few plugins (6); and WAP for the *low-quality scenario*, with few vulnerabilities found (5) and zero FPs. As for XSS vulnerabilities (see Table VII), RIPS was the best SAT for the *highest quality* and

TABLE VI
RANKING OF TOOLS BY SCENARIO: SQLi

Tool	TP	FP	FN	TN	Plugins	Main Metric	Tiebreaker Metric
Highest-quality						Recall	Precision
WAP	49	4	26	83	7	0.653	0.925
phpSAFE	29	5	46	82	5	0.387	0.853
WeVerca	0	0	75	87	0	0.000	-
RIPS	0	0	75	87	0	0.000	-
Pixy	0	0	75	87	0	0.000	-
High-quality						Informdeness	Recall
phpSAFE	274	58	72	1057	30	0.740	0.792
WAP	44	4	302	1111	12	0.124	0.127
RIPS	43	2	303	1113	8	0.123	0.124
WeVerca	18	1	328	1114	6	0.051	0.052
Pixy	16	0	330	1115	7	0.046	0.046
Medium-quality						F-Measure	Recall
RIPS	153	113	114	2101	6	0.574	0.573
phpSAFE	99	50	168	2164	15	0.476	0.371
WAP	72	0	195	2214	11	0.425	0.270
Pixy	54	13	213	2201	4	0.323	0.202
WeVerca	21	34	246	2180	3	0.130	0.079
Low-quality						Markedeness	Precision
WAP	5	0	45	1137	2	0.962	1.000
RIPS	1	0	49	1137	1	0.959	1.000
phpSAFE	36	32	14	1105	7	0.517	0.529
WeVerca	0	0	50	1137	0	-	-
Pixy	0	0	50	1137	0	-	-

TABLE VII
RANKING OF TOOLS BY SCENARIO: XSS

Tool	TP	FP	FN	TN	Plugins	Main Metric	Tiebreaker Metric
Highest-quality						Recall	Precision
RIPS	113	29	55	961	10	0.673	0.925
phpSAFE	102	18	66	972	8	0.607	0.853
Pixy	69	14	99	976	7	0.411	-
WeVerca	44	5	124	985	7	0.262	-
WAP	23	6	145	984	3	0.137	-
High-quality						Informdeness	Recall
phpSAFE	1164	90	678	5735	46	0.617	0.792
RIPS	1013	194	829	5631	46	0.517	0.127
WeVerca	436	50	1406	5775	25	0.228	0.124
Pixy	453	148	1389	5677	28	0.221	0.052
WAP	219	55	1623	5770	18	0.110	0.046
Medium-quality						F-Measure	Recall
RIPS	1812	490	577	9479	43	0.773	0.573
phpSAFE	970	267	1419	9702	41	0.535	0.371
Pixy	717	56	1672	9913	23	0.454	0.270
WeVerca	621	21	1768	9948	19	0.410	0.202
WAP	344	13	2045	9956	18	0.251	0.079
Low-quality						Markedeness	Precision
WAP	62	3	483	3591	6	0.835	1.000
phpSAFE	244	33	301	3561	10	0.803	1.000
WeVerca	73	8	472	3586	7	0.785	0.529
RIPS	377	91	168	3503	10	0.760	-
Pixy	51	7	494	3587	9	0.758	-

the medium-quality scenarios; phpSAFE was the best SAT for the *high-quality scenario*; and WAP was the best SAT for the *low-quality scenario*. Unlike for SQLi, all tools found XSS vulnerabilities in all scenarios.

In general, the results show that the best solution for vulnerability detection depends on the chosen scenario and on the class of vulnerabilities. Therefore, *hypotheses H_1 (the best SAT is the same across different scenarios) and H_2 (the best SAT 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, nor across scenarios even if considering the same class of vulnerabilities. A relevant observation is that, in almost all cases, the SATs analyzed are better at detecting XSS than SQLi.

We also verified whether or not the metrics we used were the best to rank the SATs in each scenario. To confirm this, we simulated the ranking procedure using the other metrics and compared these results with those that we have in Tables VI and VII. The resulting confusion matrices (TP, FP, FN, and TN in Tables VI and VII) show that the metrics we have selected are the best to rank the SATs for each scenario.

B. Results for BSA and SAMATE

In this section, we compare our results with the results using BSA and SAMATE, in order to show the capabilities and limitations of the different metrics for ranking SATs. First, we present BSA and SAMATE metrics, then their results on ranking the five SATs, and finally we compare our rankings with the SAMATE and BSA.

1) SAMATE metrics: The metrics proposed by the SAMATE for evaluating the tools are Precision, F-Score (i.e., F-Measure), Recall, and discrimination rate (DR). The DR is applied to a pair of test cases: the bad and the good. The bad test case has a vulnerability and the good test case is essentially the bad test with the vulnerability fixed. While every TP counts when calculating recall; thus, increasing the metric, for the DR a TP counts if the tool reports a vulnerability in the bad test case and does not report a FP in the good test case [62].

Since DR is applied to pairs of test cases, we would need a vulnerability free version of each plugin (i.e., with all vulnerabilities fixed) to calculate it. However, for many plugins there is no fixed version available, so we could not compute the DR metric. A key aspect is that the DR does not consider the goals of the vulnerability detection in scenarios. For example, in the high-quality scenario every vulnerability found is important. However, ranking the tools using the DR metric will rank first a tool reporting no FPs but less TPs than a tool reporting more TPs but reporting a number of FPs greater than the difference of TPs of the tools (second−first).

2) BSA metrics: The BSA established a scientific way to evaluate and compare tools. It defined a single metric called benchmark accuracy score (BAS) which is equivalent to the Informedness metric normalized to the range −100 to 100 [7]. It is based on the confusion matrix (TP, FP, FN, and TN) and is essentially a Youden Index, which is a standard way of summarizing the accuracy of a set of tests [7]. The metric is calculated

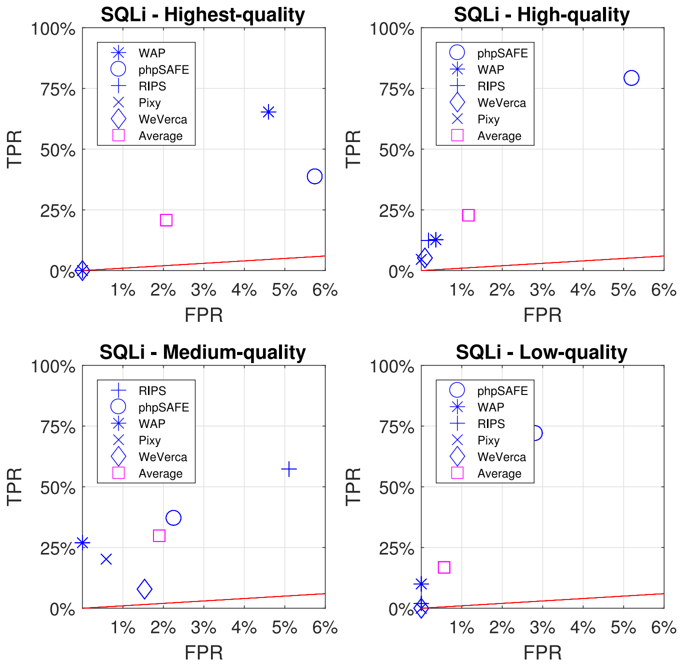

Fig. 6. Benchmark SQLi comparison by scenario.

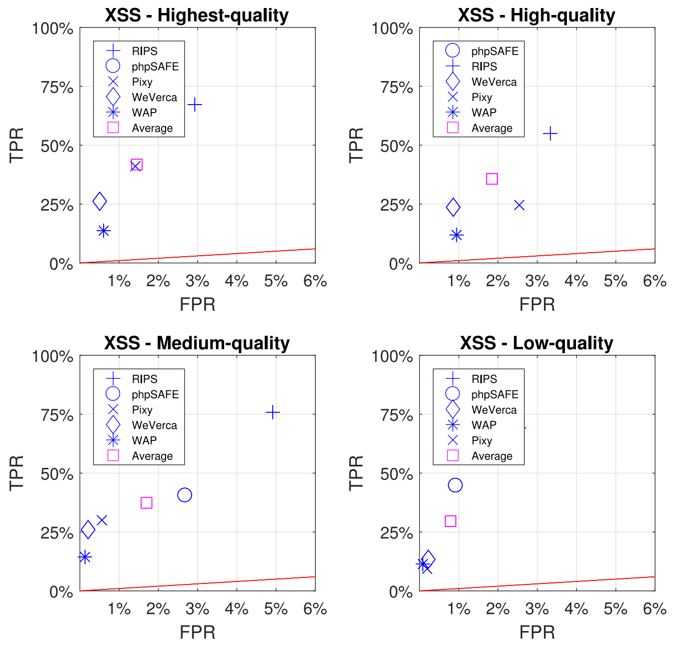

Fig. 7. Benchmark XSS comparison by scenario.

as following:

$$\mathrm{BAS} = (\mathrm{TPR} - \mathrm{FPR}) \times 100. \tag{8}$$

The TPR and FPR metrics are respectively the True Positive Rate or Recall and False Positive Rate as defined in Section III-B.

The BSA established chart plots (scorecard) for visualizing the performance of a tool or tools (see Figs. 6 to 8). The charts plot the TPR versus FPR to provide a visual indication of the results of the tools, showing how well each tool finds TPs and

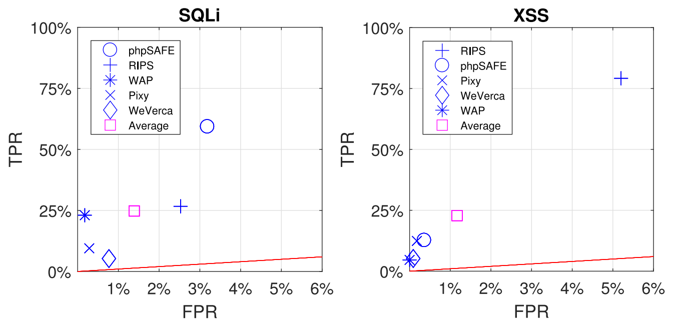

Fig. 8. Benchmark SQLi and XSS comparison without scenarios.

TABLE VIII
Ranking of Tools by Scenario and BAS Metric: SQLi

Highest-quality				High-quality			
Tools	TPR	FPR	BAS	Tools	TPR	FPR	BAS
WAP	65.3	4.6	60.7	phpSAFE	79.2	5.2	74.0
phpSAFE	38.7	5.7	32.9	WAP	12.7	0.4	12.4
WeVerca	0.0	0.0	0.0	RIPS	12.4	0.2	12.2
RIPS	0.0	0.0	0.0	WeVerca	5.2	0.1	5.1
Pixy	0.0	0.0	0.0	Pixy	4.6	0.0	4.6
Medium-quality				Low-quality			
RIPS	57.3	5.1	52.2	phpSAFE	72.0	2.8	69.2
phpSAFE	37.1	2.3	34.8	WAP	10.0	0.0	10.0
WAP	27	0.0	27	RIPS	2.0	0.0	2.0
Pixy	20.2	0.6	19.6	WeVerca	0.0	0.0	0.0
WeVerca	7.9	1.5	6.3	Pixy	0.0	0.0	0.0

avoids FPs. The charts include a slope one diagonal random guess line. It means that a tool on that diagonal reported the same rate of TPs and FPs, and its score is zero. The left-up corner of charts represents an ideal tool and the right-bottom corner the worst. Therefore, going up is good because the tool is reporting TPs and going to the right is bad because the tool is reporting FPs. Tools above the diagonal line have a TPR greater than the FPR and tools below the line have FPR greater than the TPR. On the charts, the BAS metric is the normalized distance from the point (TPR, FPR) down to the diagonal line. The charts can include the results of several tools by vulnerability class or the average of all vulnerability classes of the tools to provide an overall rank of the tools.

3) Results: Except for the DR metric, the SAMATE metrics coincide with some of our metrics. The results for those metrics were presented before in Tables VI and VII. As the remaining metrics of SAMATE (recall, F-Score and precision) are not used for explicitly ranking the tools, we do not include that analysis here (i.e., the F-Score for the scenario highest-quality, the Precision and the Recall for the scenario medium-quality, etc.). However, they can be calculated with the data in the referred tables.

Using the data from Tables VI and VII, we computed the values of the BSA metrics, as shown in Table VIII for SQLi and Table IX for XSS (the tools are sorted using the BAS metric). Note that, since the BAS metric is based on the informedness

TABLE IX
Ranking of Tools by Scenario and BAS Metric: XSS

Highest-quality				High-quality			
Tools	TPR	FPR	BAS	Tools	TPR	FPR	BAS
RIPS	67.3	2.9	64.3	phpSAFE	63.2	1.5	61.6
phpSAFE	60.7	1.8	58.9	RIPS	55	3.3	51.7
Pixy	41.1	1.4	39.7	WeVerca	23.7	0.9	22.8
WeVerca	26.2	0.5	25.7	Pixy	24.6	2.5	22.1
WAP	13.7	0.6	13.1	WAP	11.9	0.9	10.9
Medium-quality				Low-quality			
RIPS	75.8	4.9	70.9	RIPS	69.2	2.5	66.6
phpSAFE	40.6	2.7	37.9	phpSAFE	44.8	0.9	43.9
Pixy	30.0	0.6	29.5	WeVerca	13.4	0.2	13.2
WeVerca	26.0	0.2	25.8	WAP	11.4	0.1	11.3
WAP	14.4	0.1	14.3	Pixy	9.4	0.2	9.2

TABLE X
Ranking of Tools Considering all Plugins and all our Main Metrics

SQLi							
Tools	Recall	Tools	Infor.	Tools	F-M	Tools	Mark.
phpSAFE	0.59	phpSAFE	0.56	phpSAFE	0.66	WAP	0.84
RIPS	0.27	RIPS	0.24	RIPS	0.38	Pixy	0.72
WAP	0.23	WAP	0.23	WAP	0.37	phpSAFE	0.69
Pixy	0.09	Pixy	0.09	Pixy	0.17	RIPS	0.52
WeVerca	0.05	WeVerca	0.05	WeVerca	0.10	WeVerca	0.39
XSS							
RIPS	0.73	RIPS	0.63	RIPS	0.73	WeVerca	0.78
phpSAFE	0.63	phpSAFE	0.48	phpSAFE	0.63	phpSAFE	0.75
Pixy	0.40	Pixy	0.25	Pixy	0.40	RIPS	0.73
WeVerca	0.38	WeVerca	0.23	WeVerca	0.38	WAP	0.72
WAP	0.23	WAP	0.13	WAP	0.23	Pixy	0.70

metric, the ranking of the tools for the high-quality scenario is the same of our benchmark.

For all scenarios and classes of vulnerabilities, the values of the TPR are at least ten times higher than the values of the FPR. This shows the importance of identifying the NVLOCs in production software, to allow characterizing the strengths and limitations of the tools. In fact, the tools are not reporting FPs in many of the places where the SSs are, which are the places where a tool may find a vulnerability.

Figs. 6 and 7 present the chart plots (similar to the ones provided by the OWASP BSA) showing the results of the tools by scenario for SQLi and XSS, respectively. The order of the items in the captions stands for the order of the tools ranked using the BAS metric. The graphs include the average of all tools. As shown, all tools score above the diagonal line, i.e., TPR is greater than the FPR.

Table X details the values of our main metrics for SQLi and XSS vulnerabilities considering the inexistence of scenarios (without assigning scenarios to the plugins). We observed that, depending on the class of vulnerability, the same tool comes first for almost all the metrics: phpSAFE for SQLi and RIPS for XSS.

TABLE XI
RANKING OF TOOLS CONSIDERING ALL PLUGINS AND BAS METRICS

SQLi				XSS			
Tools	TPR	FPR	BAS	Tools	TPR	FPR	BAS
phpSAFE	59.3	3.2	56.2	RIPS	67.1	3.9	63.1
RIPS	26.7	2.5	24.2	phpSAFE	50.2	2.0	48.2
WAP	23.0	0.2	22.9	Pixy	26.1	1.1	25
Pixy	9.5	0.3	9.2	WeVerca	23.7	0.4	23.3
WeVerca	5.3	0.8	4.5	WAP	13.1	0.4	12.7

Table XI presents the results using the BSA metrics not considering the scenarios, and Fig. 8 shows the respective graphs. The results are similar to the results of using our main metrics; thus, phpSAFE is better for SQLi and RIPS for XSS. We verified that both tools are far above the average of all tools (see Fig. 8). However, when using scenarios this distance is much shorter. This means that, without scenarios, both strengths and limitations of the tools may be masked.

4) Comparing Our Results With SAMATE and BSA: As mentioned before, it was not possible to calculate the DR metric from SAMATE. Therefore, we do not provide ranking of the tools using this metric. Next, we compare our results with the remaining SAMATE metrics (precision, F-Measure, and recall).

The SAMATE recall and F-Measure metrics rank first the same SATs for all scenarios and classes of vulnerabilities as our main metrics, with the exception of the low-quality scenario in which recall ranks a different SAT for both classes of vulnerabilities and F-Measure ranks a different SAT for XSS vulnerabilities. Comparing with our benchmark, the SAMATE precision metric, for SQLi, ranks the same SATs for the highest quality and low-quality scenarios and different SATs for the other scenarios. For XSS, it ranks the same SAT for the high-quality scenario and a different SAT for the other scenarios. The SAMATE precision metric reveals that the WAP is more precise for SQLi, and WeVerva for XSS. However, unlike our main metric, markedness, the precision should not be used for ranking the tools as it ignores the P reported by the tools.

Regarding BSA, the results of ranking the tools using the BAS metric are similar to the ranking using our metrics, except in the low-quality scenario and for both SQLi and XSS, where the ranking of the tools is different. In summary, the BAS metric does not provide useful information to the users when they need to choose tools with different security requirements. In fact, tools with the same BAS might have different TPRs and FPRs. However, in projects with more demanding security requirements, the priority may be to find as many vulnerabilities as possible. On the other hand, for projects with tight budgets and where the security is not important, the priority may be to limit the number of results to observe.

Using as workload all the plugins without distributing them across the scenarios leads our metrics and the BAS metric to rank first the same tools (phpSAFE for SQLi and RIPs for XSS), except for the markedness metric. In practice, we obtained the WAP tool for SQLi and the WeVerca tool for XSS. Thus, we can conclude that organizing the workload in scenarios and defining metrics according to their goals is useful since it allows exploring the capabilities of the tools in different contexts. For example, code with poor quality may have unfeasible execution paths, which require more sophisticated analysis to avoid the FPs.

As a final note on the results, we observe that the metrics for evaluating security tools should be improved, considering, for example, the inclusion of different weighs for TPs and FPs. In fact, when we use the recall metric, we are assuming for TPs and FPs a weight of one and zero, respectively.

VI. BENCHMARKING PROPERTIES AND VALIDATION

The results of running the benchmark show that the proposed approach can be used to successfully rank the SATs in different scenarios. In this section, we discuss the key properties of the benchmark instantiation for WordPress plugins and SQLi and XSS vulnerabilities, and discuss the validation of the proposed process.

A. Discussion on the Benchmark Properties

To be accepted, any benchmark should fulfill a set of key properties: representativeness, repeatability, nonintrusiveness, scalability, portability, and simplicity of use [14], [63].

Representativeness: Our workload includes real applications since it is composed of WordPress plugins widely used in different scenarios, with real vulnerabilities. However, the workload across the various scenarios is unbalanced, which may affect the results in some cases. For example, in the highest quality scenario and for SQLi, only two SATs reported vulnerabilities, which may limit our study. Works using other tools are needed for improving the characterization of the vulnerable/nonvulnerable LOCs in the workload. Trust on the representativeness of the metrics is increased by previous works that showed that the different metrics should be considered for different vulnerability detection scenarios [8], [64].

Repeatability: SATs with the same settings always produce the same results as they analyze the static program structure in a deterministic way, making the results of the benchmark deterministic. We also verified this property empirically.

Nonintrusiveness: Our approach is nonintrusive, as it does not require any change to the SATs under benchmarking.

Scalability: The workload can be scaled in the number and in the complexity of the tests, since the load increases proportionally, not exponentially. The benchmark can be applied without any change to the SATs with different functionalities and maturity.

Portability: SATs do not need to run the program being analyzed, so the benchmark can be used for evaluating different SATs able to detect the SQLi and XSS vulnerabilities in PHP code, as demonstrated in the experiments. Addressing other languages and classes of vulnerabilities requires defining a new workload, following the process proposed.

Simplicity of use: Running the benchmark takes three simple steps:

1) configuring and executing the SATs;
2) comparing the results with known vulnerable and nonvulnerable LOC; and

3) calculating the metrics and ranking the tools.

These are quite straightforward, although time consuming in some cases, due to some amount of manual work involved (to verify new vulnerabilities).

B. *Validate the Benchmarking Process*

To validate our benchmarking approach, we need to validate its four main components. The *scenarios* and metrics were previously validated by Antunes *et al.* [9]. The *procedure* is well known and follows existing approaches on performance and dependability benchmarking.

The *workload* is the component that influences most the results, so it should be discussed in greater detail. The proposed process to build the workload allows selecting real applications with known vulnerabilities. The instantiation of the benchmarking approach and the results of the experiments show that it is feasible, but has some limitations/difficulties. The following paragraphs discuss the main issues.

Identifying and collecting vulnerable applications: Since there are many plugins with vulnerabilities, the likelihood of finding plugins with documented vulnerabilities is very high. In fact, results showed that our approach allows the identification of many vulnerable plugins with available source code. However, we also observed that in the WPVD there are many vulnerabilities with incomplete documentation, which is needed to evaluate the SATs, such as the vulnerable file, the LOC, the vulnerable variable, and PoC. In fact, due to this lack of data, the initial number of plugins identified was dramatically reduced from 273 to 134. This problem can be minimized by using more vulnerability databases.

Assigning applications to scenarios: We observed that our workload is unbalanced concerning the number of plugins by scenario. However, this was expected as the number of plugins collected (134) is not very high and the real percentages of plugin with five or one stars is very low. Moreover, the distribution of the plugins by scenario seems to follow a pattern similar to a normal distribution. Adding more plugins to the workload could help mitigating this issue.

Identifying VLOCs and NVLOCs: The process used to identify the lists of VLOCs and NVLOCs requires updating values during the process of benchmarking when the tools report previously unknown vulnerabilities. This occurred for 2 out of 3 tools, and the total number of manual reviews required was 251 (WAP: 168; WeVerca: 83). Therefore, as the number of benchmarked tools increases, the number of required reviews may decrease due the overlap of vulnerability detection between the tools. Moreover, none of the tools reported vulnerabilities in LOCs outside the lists of NVLOCs and VLOCs. This means that the process of identifying the NVLOCs can be trusted.

VII. Conclusion

In this paper, we addressed the problem of choosing adequate SATs for vulnerability detection in web applications. We proposed an approach to design benchmarks for evaluating such SATs considering different levels of criticality. Our approach combines SCM to automatically organize the workload in four scenarios of increasing criticality. Each scenario uses different metrics to rank the tools. To evaluate the approach, we created a benchmark for WordPress plugins and tested it with five free SATs searching for XSS and SQLi vulnerabilities in 134 WordPress plugins with real vulnerabilities, developed in PHP.

The experimental results showed that the best tool changes from one scenario to another and also depends on the class of vulnerabilities being detected. Our novel benchmark approach is a valuable tool to help project managers choosing the best SAT according to their needs and the resources available.

The comparison of the results using our metrics and the metrics from SAMATE and BSA reveals that the use of the same metrics for all scenarios makes more difficult the choice of the most appropriated tool for a project with specific requirements of security. For instance, the DR and BAS metrics may mask the capabilities of the tools when a tool reports FPs. Therefore, the metrics should be chosen according to the vulnerability detection scenario. Moreover, we found that identifying the TPs in the workload helps us to better characterize the tools. However, since the number of negative instances in real applications might be much higher than the number of positive instances, the metrics should be improved to balance the weight of the TPs and FPs in the computation of the metrics.

Future work includes two main directions. First, we would like to improve our workload by adding more plugins in order to provide a balanced set of plugins between scenarios. Second, we would like to investigate the code patterns that SATs are not able to analyze. The goal is to understand the detection capabilities and the demerits of the tools (data/control/path flows, dealing with OOP constructs, complex string replace operations, etc.), to provide means to improve them.

References

[1] [Online]. Available: http://www.acunetix.com/vulnerability-scanner/, Accessed on: Aug. 8, 2015.

[2] "Annual Consumer Studies." 2015. [Online]. Available: http://www.ponemon.org/, Ponemon Institute

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Depend. Security Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[4] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, "Effect of static analysis tools on software security: Preliminary investigation," in *Proc. ACM Workshop Quality. Protection*, 2007, pp. 1–5.

[5] A. Doupe, M. Cova, and G. Vigna, "Why Johnny can't pentest: An analysis of black-box web vulnerability scanners," in *Proc. 7th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2010, pp. 111–131.

[6] [Online]. Available: http://samate.nist.gov/, Accessed on: Jun. 6, 2015.

[7] [Online]. Available: https://www.owasp.org/index.php/Benchmark, Accessed on: Apr. 10, 2016.

[8] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders: Test and measurement of static code analyzers," in *Proc. 1st Int. Workshop Complex faUlts Failures Large Softw. Syst.*, 2015, pp. 14–20.

[9] N. Antunes and M. Vieira, "On the metrics for benchmarking vulnerability detection tools," in *Proc. IEEE/IFIP 45th Annu. Int. Conf. Depend. Syst. Netw.*, Jun. 2015, pp. 505–516.

[10] [Online]. Available: https://w3techs.com/technologies/overview/content_management/all, 2017.

[11] "WP Template.com." [Online]. Available: http://www.wptemplate.com/tutorials/safety-and-security-of-wordpress-blog-infographic.html, Accessed on: May 1, 2016.

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

[13] [Online]. Available: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf, 2017.

[14] C. Ballinger, "TPC-D: Benchmarking for Decision Support," in *The Benchmark Handbook for Database and Transaction Systems*, J. Gray, Ed. 2nd ed. San Mateo, CA, USA: Morgan Kaufmann, 1993.

[15] N. L. de Poel, F. B. Brokken, and G. R. R. de Lavalette, "Automated security review of PHP web applications with static code analysis," Master's thesis, vol. 5, 2010.

[16] M. Vieira, H. Madeira, K. Sachs, and S. Kounev, "Resilience benchmarking," *Resilience Assessment Eval. Comput. Syst.*, pp. 283–301, 2012.

[17] S. Neuhaus and T. Zimmermann, "Security trend analysis with CVE topic models," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 111–120.

[18] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Inf. Softw. Technol.*, vol. 68, pp. 18–33, Dec. 2015.

[19] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proc. 14th Conf. USENIX Security Symp.*, vol. 14, 2005, pp. 18–18.

[20] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Proc. IEEE Symp. Security Privacy*, May 2006, p. 263.

[21] G. Wassermann and Z. Su, "Static detection of cross-site scripting vulnerabilities," in *Proc. IEEE ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 171–180.

[22] W. Landi, "Undecidability of static analysis," *ACM Lett. Programm. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, 1992.

[23] J. Fonseca, M. Vieira, and H. Madeira, "The web attacker perspective—A field study," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 299–308.

[24] J. Fonseca and M. Vieira, "A practical experience on the impact of plugins in web security," in *Proc. IEEE 33rd Int. Symp. Rel. Distrib. Syst.*, 2014, pp. 21–30.

[25] P. Nunes, I. Medeiros, J. Fonseca, N. Neves, M. Correia, and M. Vieira, "On combining diverse static analysis tools for web security: An empirical study," in *Proc. 13th Eur. Dependable Comput. Conf.*, Sep. 2017, pp. 121–128.

[26] P. E. Black, M. Kass, M. Koo, and E. Fong, "Source code security analysis tool functional specification version 1.1," Feb. 2011.

[27] J. A. Kupsch and B. P. Miller, "Manual vs. automated vulnerability assessment: A case study," in *Proc. 1st Int. Workshop Manag. Insider Secur. Threats*, 2009.

[28] "PhpMetrics.org," [Online]. Available: http://www.phpmetrics.org/, Accessed on: Oct. 3, 2016.

[29] "PHPMD - PHP mess detector," [Online]. Available: https://phpmd.org/, Accessed on: Jan. 6, 2017.

[30] "Sonarqube.org," [Online]. Available: http://www.sonarqube.org/, Accessed on: Nov. 3, 2016.

[31] "Software quality enhancement," [Online]. Available: http://www.squale.org/, Accessed on: Nov. 3, 2016.

[32] Ö. F. Arar and K. Ayan, "Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies," *Expert Syst. Appl.*, vol. 61, pp. 106–121, Nov. 2016.

[33] M. Schroeder, "A practical guide to object-oriented metrics," *IT Professional*, vol. 1, no. 6, pp. 30–36, Nov. 1999.

[34] M. Sankar and A. Irudhyaraj, "Software quality attributes for secured web applications," *Int. J. Eng. Sci. Invention*, vol. 3, no. 7, pp. 19–27, 2014.

[35] D. Nabil, A. Mosad, and H. A. Hefny, "Web-based applications quality factors: A survey and a proposed conceptual model," *Egyptian Informat. J.*, vol. 12, pp. 211–217, 2011.

[36] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[37] "pdepend.org," [Online]. Available: https://pdepend.org/, Accessed on: Mar. 11, 2016.

[38] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proc. IEEE Int. Conf. Softw. Maintain.*, 2010, pp. 1–10.

[39] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *Proc. IEEE Joint Conf. 21st Int. Workshop Softw. Meas. 6th Int. Conf. Softw. Process Product Meas.*, 2011, pp. 20–29.

[40] A. H. Watson, T. J. Mccabe, and D. R. Wallace, "Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric," U.S. Dept. Commer./Nat. Inst. Stand. Technol., 1996.

[41] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik, "RTTool: A tool for extracting relative thresholds for source code metrics," in *Proc. IEEE Int. Conf. Softw. Maintain. Evol.*, Sep. 2014, pp. 629–632.

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

[43] J. P. Correia and J. Visser, "Certification of technical quality of software products," in *Proc. Int. Workshop Found. Tech. Open Source Softw. Certification*, 2008, pp. 35–51.

[44] "Software improvement group (SIG)," Jan. 2017. [Online]. Available: https://www.sig.eu

[45] P. E. Black and E. N. Fong, Gaithersburg, MD, USA, Tech. Rep.

[46] J. Kistowski, J. A. Arnold, K. Huppler, P. Cao, and J. L. Henning, "How to build a benchmark," in *Proc. 6th ACM/SPEC Int. Conf. Performance Eng.*, Feb. 2015, pp. 333–336.

[47] D. M. W. Powers, "Evaluation evaluation a Monte Carlo study," *CoRR*, vol. abs/1504.00854, 2015. [Online]. Available: http://arxiv.org/abs/1504.00854

[48] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proc. 6th Int. Conf. Quality Inf. Commun. Technol.*, Sep. 2007, pp. 30–39.

[49] W. Hu, T. Loeffler, and J. Wegener, "Quality model based on ISO/IEC 9126 for internal quality of MATLAB/Simulink/Stateflow models," in *Proc. IEEE Int. Conf. Ind. Technol.*, 2012, pp. 325–330.

[50] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[51] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, Jun. 1974.

[52] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Trans. Softw. Eng.*, vol. 28, no. 1, pp. 4–17, Jan. 2002.

[53] I. Bojanova, P. E. Black, Y. Yesha, and Y. Wu, "The bugs framework (BF): A structured approach to express bugs," in *Proc. IEEE Int. Conf. Softw. Quality, Rel. Security*, pp. 175–182, 2016. [Online]. Available: https://doi.org/10.1109/QRS.2016.29

[54] [Online]. Available: https://wordpress.org/plugins/, Accessed on: Dec. 29, 2016.

[55] P. Nunes, J. Fonseca, and M. Vieira, "phpSAFE: A security analysis tool for OOP web application plugins," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 299–306.

[56] "WPScan Vulnerability Database," 2017. [Online]. Available: https://wpvulndb.com/, Accessed on: Oct. 26, 2015.

[57] [Online]. Available: https://github.com/pjcnunes/ISSRE2017, 2017.

[58] J. Dahse, G. Horst, and T. Holz, "Simulation of built-in PHP features for precise static code analysis," in *Proc. Netw. Distrib. Syst. Security*, 2014, pp. 23–26.

[59] I. Medeiros, N. F. Neves, and M. Correia, "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives," in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 63–74.

[60] D. Hauzar and J. Kofron, "Framework for static analysis of PHP applications," in *Proc. 29th Eur. Conf. Object-Oriented Programm.*, 2015, pp. 689–711.

[61] J. Dahse, N. Krein, and T. Holz, "Code Reuse Attacks in PHP: Automated POP chain generation," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2014, pp. 42–53.

[62] F. G. G. Meade, "CAS Static Analysis Tool Study—Methodology," 2011. [Online]. Available: https://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf

[63] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2008, pp. 41–50.

[64] H. Perl *et al.*, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 426–437.

Paulo Nunes is currently working toward the Ph.D. degree in a Doctoral Program in Information Science and Technology, University of Coimbra, Coimbra, Portugal.

He has been an Adjunct Professor with the Department of Computer Science, Polytechnic Institute of Guarda, since 1998. Since 2008, he has been a Researcher with the Research Unit for Inland Development, IPG, and since 2014, a Research Student with the Centre for Informatics and Systems, University of Coimbra.

Ibéria Medeiros (M'16) received the Ph.D. degree in computer science from the Faculty of Sciences, University of Lisbon, Lisbon, Portugal, in 2016.

She is currently an Assistant Professor with the Department of Informatics, Faculty of Sciences, University of Lisbon. She is a member of the Large-Scale Informatics Systems Laboratory, and the Navigators Research Group. She has been participating in SEGRID and DiSIEM European Projects. Her research interests are concerned with software security, source code static analysis, vulnerability detection, data mining and machine learning, and security.

José C. Fonseca received the Ph.D. degree in informatics engineering from the University of Coimbra, Coimbra, Portugal, in 2011.

Since 2005, he has been with the CISUC as a Researcher. He teaches computer related courses in the Polytechnic Institute of Guarda since 1993. He is the author or coauthor of more than two dozen papers in refereed conferences and journals. His research on vulnerability and attack injection was granted with the DSNs William Carter Award of 2009, sponsored by the IEEE Technical Committee on Fault-Tolerant Computing and IFIP Working Group on Dependable Computing and Fault Tolerance (WG 10.4).

Nuno Ferreira Neves received the Ph.D. degree from University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 1998.

He is currently a Full Professor with the Department of Computer Science, Faculty of Sciences, University of Lisbon, Lisbon, Portugal. He is currently the Head of the Department. He leads the Navigators Research Group and he is on the Executive Board of the LASIGE Research Unit. His main research interests include security and dependability aspects of distributed systems. He is currently a Principal Investigator of the SUPERCLOUD and SEGRID European Projects, and he is involved in projects BiobankClouds and Erasmus+ ParIS. His work has been recognized on several occasions, for example, with the IBM Scientific Prize, and the William C. Carter award. He is on the Editorial Board of the *International Journal of Critical Computer-Based Systems*.

Miguel Correia (SM'87) received the Ph.D. degree from Universidade de Lisboa, Lisboa, Portugal, in 2003.

He is currently an Associate Professor with Instituto Superior Técnico, Universidade de Lisbon, Lisbon, Portugal, and a Senior Researcher at INESC-ID, Distributed Systems Group. He has been involved in several international and national research projects related to cybersecurity, including the SafeCloud, PCAS, TCLOUDS, ReSIST, CRUTIAL, and MAFTIA European projects. He has more than 150 publications.

Marco Vieira (M'06) received the Ph.D. degree from UC, Portugal, in 2005.

He currently is a Full Professor with the University of Coimbra, Coimbra, Portugal. His research interests include dependability and security assessment and benchmarking, fault injection, software processes, and software quality assurance, subjects in which he has authored or coauthored more than 200 papers in refereed conferences and journals. He has participated and coordinated several research projects, both at the national and European level. He has served on program committees of the major conferences of the dependability area and acted as referee for many international conferences and journals in the dependability and security areas.