Assessing and Comparing Vulnerability Detection Tools for Web Services: Benchmarking Approach and Examples

Nuno Antunes, *Member, IEEE* and Marco Vieira, *Member, IEEE Computer Society*

Abstract—Selecting a vulnerability detection tool is a key problem that is frequently faced by developers of security-critical web services. Research and practice shows that state-of-the-art tools present low effectiveness both in terms of vulnerability coverage and false positive rates. The main problem is that such tools are typically limited in the detection approaches implemented, and are designed for being applied in very concrete scenarios. Thus, using the wrong tool may lead to the deployment of services with undetected vulnerabilities. This paper proposes a benchmarking approach to assess and compare the effectiveness of vulnerability detection tools in web services environments. This approach was used to define two concrete benchmarks for SQL Injection vulnerability detection tools. The first is based on a predefined set of web services, and the second allows the benchmark user to specify the workload that best portrays the specific characteristics of his environment. The two benchmarks are used to assess and compare several widely used tools, including four penetration testers, three static code analyzers, and one anomaly detector. Results show that the benchmarks accurately portray the effectiveness of vulnerability detection tools (in a relative manner) and suggest that the proposed benchmarking approach can be applied in the field.

Index Terms—Benchmarking, vulnerability detection, penetration testing, static analysis, and runtime anomaly detection

1 Introduction

Web services (WS) are nowadays widely used to support many enterprise systems, linking suppliers and clients in sectors such as banking, transportation, and manufacturing, just to name a few [1]. Web services are a key element in service oriented architectures (SOA) and consist of standard-based self-describing components that can be used by other software across the web in a platform-independent manner. This makes web services the *lingua franca* for systems integration.

The security of web applications is, in general, quite poor [2]. Web services are no exception and are frequently deployed with code vulnerabilities (as shown in [3], [4]). This is confirmed by the field study presented in [5], which describes an experimental evaluation of the security vulnerabilities in 300 publicly available web services. Four well-known vulnerability scanners have been used to identify security flaws in the services implementations and a large number of vulnerabilities has been observed (25 of the tested services presented some type of security vulnerability that could be exploited), confirming that many services (more than 8 percent) are deployed without proper security testing. A key observation was that injection vulnerabilities are particularly frequent [2]. These consist of improperly coded applications that allow the attacker to inject and execute commands in the vulnerable service, enabling, for instance, access to critical data. Vulnerabilities allowing SQL Injection and XPath Injection are especially relevant, as web services frequently use a data persistence solution supported by a relational database [6] or a XML solution [7].

Web services are so widely exposed that any security vulnerability will most probably be uncovered and exploited by hackers. This way, to prevent vulnerabilities, developers should apply coding best practices, perform security reviews of the code, use static code analyzers, execute penetration tests, etc. [8]. However, most times, developers focus on the implementation of functionalities to satisfy the user's requirements and the time-to-market constraints, thus disregarding security aspects. In this context, vulnerability detection tools provide an easy and low cost way to test web services for vulnerabilities.

Vulnerability detection tools are widely used by web services' developers to support automated security checking and comprise some of the best examples of critical tools for secure software development. Different techniques for vulnerabilities detection have been proposed in the past [8], including penetration testing and static code analysis, which are the two most used ones. Due to time constraints or resource limitations, developers frequently have to select a specific tool from the large set of tools available (usually without really knowing how good each tool is) and strongly rely on that tool to detect potential security problems in the code being developed.

Previous work shows that the effectiveness of many of these tools is quite low [5], [9], [10]. In fact, the low coverage and the high number of false positives frequently observed highlight the limitations of many vulnerability detection tools. Furthermore, it is clear that the performance of a given tool strongly depends on the specificities of the

• *The authors are with the University of Coimbra, 3030-329 Coimbra, Portugal. E-mail: {nmsa, mvieira}@dei.uc.pt.*

*Manuscript received 6 Aug. 2013; revised 22 Jan. 2014; accepted 15 Feb. 2014. Date of publication 10 Mar. 2014; date of current version 10 Apr. 2015.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSC.2014.2310221*

application scenario (i.e., the class of target web services (e.g., SOAP, REST), the types of vulnerabilities to detect, etc.), and that the same tool may have different performance levels in different scenarios. Although some studies focused on the evaluation bug detection tools [11], [12], [13], none has provided a systematic way to evaluate and compare vulnerability detection tools. This way, developers urge the definition of a practical approach that helps them assessing and comparing alternative tools concerning their ability to detect vulnerabilities.

This paper proposes an approach for benchmarking vulnerability detection tools for web services. This approach specifies all the components and steps needed to define benchmarks to assess and compare alternative tools, with particular focus on two metrics: precision (ratio of correctly detected vulnerabilities to the number of all detected vulnerabilities) and recall (ratio of correctly identified vulnerabilities to the number of all known vulnerabilities). These are proven and well known metrics that are widely used in several domains, although originally proposed for information retrieval systems [14]. Additionally, it defines the other required components, which include a workload (work that the vulnerability detectors under testing have to do, in the form of a set of web services that should be searched for vulnerabilities) and a well-defined benchmarking procedure (set of steps that have to be followed for conducting a benchmarking campaign, ranging from the preparation of the experiments to the ranking of the tools). A key aspect is that the proposed approach is generic and can be used to specify different benchmarks for different application domains and types of vulnerabilities.

The benchmarking approach has been used to define two concrete benchmarks. The first targets tools capable of detecting SQL Injection vulnerabilities in SOAP web services, including detection approaches based on penetration testing, static code analysis, and runtime anomaly detection. This benchmark uses a well defined and large set of web services adapted from standard performance benchmarks, and includes both vulnerable and non-vulnerable versions of the services. The main limitation is that, although based on a well-defined set of rules, it is not protected against "*gaming*" (i.e., adaptations/tuning that allow producing optimistic or biased results). In fact, as the workload is well known, providers can easily tune their tools to maximum effectiveness in the context of the benchmark, while failing in different scenarios.

To demonstrate an alternative approach, we propose a second benchmark for penetration testing tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This benchmark circumvents the "*gaming*" problem by allowing the benchmark user to specify the workload (i.e., the workload is not predefined and is unknown to the tools' providers) that best represents his specific development conditions, thus providing more realistic (and specific to the development environment) results. To support the user in the task of defining the workload, the benchmark includes a procedure and a tool to identify vulnerabilities in the target web services, thus avoiding the need for conducting such analysis manually.

When compared with related benchmarking works, our approach inovattes in the following. Contrarily to benchmarks for bug detection tools, the services used as workload should work correctly from a functional point of view, although containing vulnerabilities that may be exploited by a security attack, which raises difficult challenges when defining workloads. Also, the focus on the web services environment brings the need for the benchmarks to be useful for providers and consumers. This way, the metrics should be easy to understand for both parties and must allow comparing different types of vulnerability detection tools. Due to the diversity of potential users, the procedure should be the most automated possible. Finally, considering that the efficiency of vulnerability detection tools depend on the context where they are applied, it is very important to support benchmarks that allow users to define their own workloads, making the results of their campaigns much more useful.

To demonstrate the benchmarking approach and the two concrete benchmarks, several widely used commercial and open-source vulnerability detection tools have been benchmarked, including four penetration testers, three static code analyzers, and one anomaly detector. The results allowed us to successfully rank the tools according to several criteria, while fulfilling key properties such as repeatability, portability, representativeness, non-intrusiveness, and simplicity of use. This suggests that the proposed approach can be applied in the field.

In summary, the *contributions of this paper are*:

- A generic benchmarking approach for vulnerability detection tools for web services. The approach is based on the clear definition of the benchmarking domain (i.e., of the characteristics of the target tools) and defines the components and metrics needed to specify concrete benchmarks. This approach is based on a preliminary proposal presented in [15], which has been detailed and extended to support the definition of benchmarks based on user-specific workloads (the original version considered only the use of predefined workloads).
- A concrete benchmark for penetration testing, static code analysis, and runtime anomaly detection tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This benchmark is based on a set of predefined web services and has been used to assess and rank a set of tools, including four penetration testers, three static code analyzers, and one anomaly detector. This benchmark is based on the preliminary proposal in [15].
- A new benchmark (not present before) for penetration testing tools capable of detecting SQL Injection vulnerabilities in SOAP web services. This benchmark allows the user to define the workload (thus preventing "*gaming*") and includes a tool for identifying the existing vulnerabilities in such workload. The benchmark was used to compare four penetration testers.

The outline of this paper is as follows. Section 2 presents background and related work. Section 3 discusses the benchmarking approach. Section 4 introduces the tools used to demonstrate the concrete benchmarks. Section 5 presents the benchmark based on the predefined workload and Section 6 discusses the benchmark based on the user-defined workload. Both include the experimental evaluation and the benchmark properties discussion. Section 7 concludes the paper.

2 Background and Related Work

Published studies show that, in general, web applications present dangerous security flaws. For example, the NTA's Annual Security Report 2008 [16] states that 25 percent of the companies tested presented one or more high-risk vulnerabilities. This number is lower than the 32 percent reported in 2007 [17]. Nevertheless, the overall vulnerabilities found increased in some critical sectors (finance, government, legal, retail and utilities). The NTA's Annual Web Application Security Report 2011 [18], focused on web applications, states that 8 percent of the applications tested contained at least one high-risk vulnerability and that 26 percent of them contained medium risk vulnerabilities. These results cannot be generalized to web services, but show a high number of software applications being deployed without proper security cautions, including web applications.

In a web services environment, the vulnerability distribution might be slightly different from typical web sites, but web services are also frequently deployed containing security vulnerabilities [3], [4]. The study presented in [5] shows that numerous public web services have some type of vulnerability that can be exploited, confirming that many are deployed without proper security testing. To mitigate this, developers should use appropriate tools to detect vulnerabilities. The problem is that even state-of-the-art detectors frequently present low effectiveness both in terms of vulnerability detection coverage (ratio between the number of vulnerabilities detected and the total number of existing vulnerabilities) and false positives (ratio between the number of true vulnerabilities detected and the total number of vulnerabilities reported) [5], [9], [10]. This way, benchmarking approaches that allow developers to select the most effective tools for each particular scenario are of utmost importance.

2.1 Vulnerability Detection

Penetration testing and static code analysis are two well-known techniques frequently used by developers to identify security vulnerabilities in web services [8]. Although penetration testing is based on the effective execution of the code, vulnerabilities detection consists in the analysis of the responses, which limits the visibility on the internal service behavior. On the other hand, static code analysis is based on the analysis of the source code (or the bytecode in more advanced analyzers), which allows identifying specific code patterns prone to security vulnerabilities. However, it lacks a dynamic view of the service behavior in the presence of a realistic workload.

Penetration testing tools provide an automatic way to test an application for vulnerabilities, based on specifically tampered input values. Previous research shows that the effectiveness of these tools in web services is very poor. For example, the work presented in [5] shows several limitations, namely: large differences in the vulnerabilities detected by each tool, low coverage (less than 20 percent for two scanners), and high number of false positives (35 and 40 percent in two cases). These limitations are also confirmed by the studies presented in [9], [10].

Static code analyzers provide an automatic manner for highlighting possible coding errors without actually executing the software [19]. In [20] authors evaluated three tools and compared their effectiveness with the effectiveness of code reviews. The tools achieved higher efficiency than the reviews in detecting software bugs (the study did not consider security issues in particular) in five Java-based applications, but all the tools presented false positive rates higher than 30 percent. This is also confirmed in [9].

Runtime anomaly detection consists in the search for deviations from an historical profile of valid commands and is an alternative approach for vulnerability detection. For example, in [21] an approach is proposed that combines penetration testing with anomaly detection for uncovering SQL Injection vulnerabilities. Another example is Analysis and Monitoring for NEutralizing SQL-Injection Attacks (AMNESIA) [22], a tool that combines static analysis and runtime monitoring to detect and avoid SQL injection attacks. The problem is that these approaches are typically based on a learning phase whose completeness is difficult to guarantee, thus the model representing the valid/expected behavior may be incomplete (guaranteeing complete learning is extremely difficult), leading to false positives and undetected vulnerabilities [21], [22].

2.2 Benchmarking

Computer benchmarks are standard tools that allow evaluating and comparing different systems or components according to specific characteristics (e.g., performance, dependability, etc.) [23]. The work on performance benchmarking has started long ago [23]. Ranging from simple benchmarks that target very specific hardware systems or components to very complex benchmarks focusing complex systems (e.g., database management systems, operating systems), performance benchmarks have contributed to improve successive generations of systems. Research on dependability benchmarking boosted in the beginning of this century [24]. Several works have been done by different groups and following different approaches (e.g., experimental, modeling, fault injection) [24]. Finally, work on security benchmarking is a new topic with many open questions [25], [26].

Several studies show the importance of evaluating testing techniques using controlled experiments. In fact, in [27] the authors present the difficulties behind creating controlled environments, and introduce an infrastructure for supporting controlled experimentation with software testing and regression testing. An attempt to create a benchmark containing multithreaded bugs is presented [11]. The benchmark uses a data set of applications that was built using students assigned to write buggy multithreaded Java programs and document those bugs. Undocumented bugs, i.e., introduced unintentionally, were also considered in the data set. Although this study has obvious problems in terms of representativeness, the problems found in the detection tools showed the utility of this kind of benchmarks. Regarding the evaluation of bug detecting approaches, in [12] the authors present "BugBench," a data set consisting of real-life applications with varying sizes and containing bugs. The bugs were manually collected and most of them are related to memory. Conversely, iBUGS [13] is an approach to semiautomatically extract data sets of real bugs and corresponding tests from the history of a project. The authors demonstrated the approach extracting 369 bugs from the history of AspectJ Project and then used these bugs to evaluate one bug localization tool. However, these studies always focused on classical bugs, which differ from the problematic raised by security vulnerabilities.

Although several works have tried to assess the effectiveness of vulnerability detection tools (e.g., [5], [9], [10], [20], [28]) none has proposed a generic and standard approach that allows the comparison of results. In fact, existing works compare different tools under very specific conditions, which cannot be generalized or easily replicated, thus results are of limited use. A first attempt to define a benchmarking approach for vulnerability detection tools was presented at [15], which included a concrete benchmark for tools able to detect SQL Injection vulnerabilities. The current work extends that approach and proposes a new benchmark that addresses the problem of "*gaming*" of results. Another relevant work is presented in [10]. It proposes a method to evaluate web vulnerability scanners using software fault injection techniques. Software faults are injected in the application code and the tool under evaluation is executed, showing its strengths and weaknesses concerning coverage of vulnerability detection and false positives. However, this study was focused on a specific family of applications, namely database centric web-based applications written in PHP, and the benchmarking approach cannot be generalized and applied to other domains.

2.3 Benchmarking Properties

Computer benchmarking is primarily an experimental approach. As an experiment, its acceptability is largely based on two salient facets of the experimental method: 1) the ability to reproduce the observations and the measurements, either on a deterministic or on a statistical basis, and 2) the capability of generalizing the results through some form of inductive reasoning. The first aspect gives confidence in the results and the second makes the benchmark results meaningful and useful beyond the specific setup used in the benchmarking process. In practice, benchmarking results are normally reproducible in a statistical basis. On the other hand, the necessary generalization of the results is inherently related to the representativeness of the benchmark experiments. The notion of representativeness is manifold and touches almost all the aspects of benchmarking, as it really means that the conditions used to obtain the measures are representative of what can be found in the real world.

To achieve acceptance by the computer industry or by the user community a benchmark should fulfill a set of key properties [23]: representativeness, portability, repeatability, non-intrusiveness, and simplicity of use. These properties must be taken into account from the beginning of the definition of the components and must be validated after the benchmark has been completely defined.

To be credible, a benchmark for vulnerability detection tools must report similar results when run more than once over the same tool. However, *repeatability* has to be understood in statistical terms, as it might be impossible to reproduce exactly the same conditions concerning the tool and the web services state during the benchmark run. In practice, small deviations in the measurements in successive runs are normal and just reflect the non-deterministic nature of web applications.

Another important property is *portability*, as a benchmark must allow the comparison of different tools in a given domain. In practice, the workload is the component that has more influence on portability, as it must be able to exercise the vulnerability detection capabilities of a large set of tools in the domain.

In order to report relevant results, a benchmark must represent real world scenarios in a realistic way. In our work, *representativeness* is mainly influenced by the workload, which must be based on realistic code and must include a realistic set of vulnerabilities. This can more easily be taken into account in the case of benchmarks based on a predefined workload, as it is possible to address representativeness issues during the benchmark specification. However, this may be a issue in the case of user-defined workloads, as the benchmark user may not be aware of the representativeness issues of the services considered and, consequently, of the results obtained.

A benchmark must require minimum changes (or no changes at all) in the target tools. If the implementation or execution of the benchmark requires changes in the tools (either in the structure or in the behavior) then the benchmark is *intrusive* and the results might not be valid.

Finally, to be accepted, a benchmark must be as *easy to implement and run* as possible. Ideally, the benchmark should be provided in a form ready to be used or, if that is not possible, as a document specifying in detail how the benchmark should be implemented and executed. In addition, the benchmark execution should take the smallest time possible (preferably not more than a few hours per tool). This is obviously easier to achieve in benchmarks based on a predefined workload, as in the case of user-defined workloads the benchmark user has the added work of defining and characterizing the workload.

3 Approach for Benchmarking Vulnerability Detection Tools

Our proposal to benchmark vulnerability detection tools is inspired on measurement-based techniques. The basic idea is to exercise the tools under benchmarking using web services code with and without vulnerabilities and, based on the detected vulnerabilities, calculate a small set of measures that portray the tools' detection capabilities.

Due to the high diversity of web services, types of vulnerabilities, and vulnerability detection approaches, the definition of a benchmark for all vulnerability detection tools is an unattainable goal. This way, as recommended in [23], a benchmark must be specifically targeted to a particular domain. In fact, the division of the spectrum into well-defined areas is necessary to make it possible to make choices during the definition of the benchmark components. In the context of this work, the *definition of the benchmarking domain* includes selecting the class of web services, the type of vulnerabilities, and the vulnerability detection approaches for the target tools under benchmarking, which mainly influence the definition of the workload (see Section 4.2). This way, the main components of a benchmark are:

- *Metrics*. Characterize the effectiveness of the tools under benchmarking in detecting the vulnerabilities that exist in the workload services. The metrics must be easy to understand and must allow comparison of tools.
- *Workload*. Represents the work that a tool must perform during the benchmark execution. In practice, it consists of a set of services (with and without vulnerabilities) that will be used to exercise the vulnerability detection tools during the benchmarking process. Depending on the goal of the benchmark, the workload can be predefined (i.e., defined in the benchmark specification itself) or provided by the benchmark user.
- *Procedure*. Describes the procedure and rules that must be followed when executing the benchmark.

The procedure and rules have to be specified during the definition of the benchmark. In fact, those procedures and rules are the core of the benchmark specification. Although this is, obviously, dependent on the specific benchmark, in the following points we identify some guidelines on specific aspects needed in most of the cases:

- *Standardized procedures* for "translating" the workload defined in the specification into the actual workload that will be applied to the tools under benchmarking. These procedures guarantee that the different users understand and use the benchmark in a consistent way.
- *Uniform conditions* to build the experimental benchmark setup, perform initialization tasks that might be defined in the specification, and run the benchmark according to the specification.
- *Rules* related to the collection of the experimental results. These rules may include, for example, available possibilities for system instrumentation, degree of interference allowed, common references and precision for timing measures, etc.
- *Rules* for the production of the final measures from the experimental results, such as formulas, ways to deal with uncertainties, errors and confidence intervals, etc.
- *Disclosures* required for interpreting the benchmark measures. Similarly to what happens in other domains [24], a report may be required for the results to be considered compliant with the specification. The goal is to allow the reproduction of the experiments in other sites using the same vulnerability detection tools.
- *Rules* to avoid "*gaming*" to produce optimistic or biased results. For example, rules regarding the use and/or definition of the workload.

3.1 Metrics

The benchmark metrics should be computed from the information collected during the benchmark run and must follow the well-established measuring philosophy typically used in performance benchmarking [23]. In fact, benchmarks should provide relative measures (i.e., measures related to the conditions disclosed in the benchmark report) that can be used for comparison or for improvement and tuning. For example, it is well known that performance benchmark results do not represent an absolute measure of performance and cannot be used for planning capacity or to predict the actual performance of the system in field. In a similar way, the measures in a benchmark for vulnerability detectors must be understood as results that can be useful to characterize the tools in a relative fashion (e.g., to compare alternative tools).

A key difficulty related to the metrics is that different detection tools report vulnerabilities in different ways. For example, for penetration testing tools vulnerabilities are reported for each vulnerable input. On the other hand, for static analysis tools vulnerabilities are reported for each vulnerable line in the code. Due to this dichotomy, it is very difficult (or even imposable) to compare the effectiveness of tools that implement different vulnerability detection approaches, based on the number of vulnerabilities reported for the same piece of code. Our proposal is to characterize vulnerability detection tools using the F-Measure proposed by van Rijsbergen [14], which is largely independent of the way vulnerabilities are counted. In fact, it represents the harmonic mean of two very popular measures (precision and recall), which, in the context of vulnerability detection, can be defined as:

- *Precision*. The ratio of correctly detected vulnerabilities to the number of all detected vulnerabilities:

$$precision = \frac{TP}{TP + FP} \tag{1}$$

- *Recall*. The ratio of correctly detected vulnerabilities to the number of known vulnerabilities:

$$recall = \frac{TP}{TV} \tag{2}$$

where:

- True positives (TP) is the number of true vulnerabilities detected;
- Fault positives (FP) is the number of vulnerabilities detected that, in fact, do not exist;
- True vulnerabilities (TV) is the total number of vulnerabilities that exist in the code.

Assuming an equal weight for precision and recall, the formula for the *F-Measure* is:

$$F\text{-}Measure = \frac{2 \cdot precision \cdot recall}{precision + recall} \tag{3}$$

A highly effective tool will generate a high F-Measure (which obviously ranges between 0 and 1). For example, consider a set of 100 vulnerabilities in a piece of code.

A tool that achieves a precision of 0.7 is able to detected vulnerabilities with a probability of 70 percent. A recall of 0.8 expresses that 80 percent of all the known vulnerabilities are detected and that 20 percent are missed. In this case the F-Measure would be approximately 0.7466.

The three metrics can be used to establish a ranking of tools depending on the purposes of the benchmark user. Note that these are proven metrics that are typically used to portray the effectiveness of many computer systems [14], particularly in information retrieval. Thus, they are easy to be understood by most users.

3.2 Workload

The workload defines the work that has to be done by the vulnerability detection tools during the benchmark execution. In other words, the workload should include the code that will be used to exercise the vulnerability detection capabilities of the tools under benchmarking. It is mainly influenced by three factors:

- The *class of web services* (e.g., SOAP, REST), which allows defining the characteristics of the services that will be used to exercise the tools under benchmarking.
- The *types of vulnerabilities* (e.g., SQL Injection, XPath Injection) to be detected by the tools. This defines the vulnerabilities that must exist in the workload.
- The *detection approaches* (e.g., penetration testing, static analysis), which specify the approaches used by the tools under benchmarking to detect vulnerabilities.

Three different types of workloads can be considered for benchmarking purposes:

- *Real workloads*. These are made of applications used in real environments having real vulnerabilities. Real workloads are expected to be quite representative. However, many applications and vulnerabilities may be needed to achieve good representativeness and those applications frequently require some adaptation.
- *Realistic workloads*. Artificial workloads that are based on the adaptation of real applications in the domain. Although artificial, realistic workloads still reflect real situations and are more portable than real workloads.
- *Synthetic workloads*. A synthetic workload can be a set of randomly selected code elements in which vulnerabilities are artificially injected. Synthetic workloads are easy to use but their representativeness is questionable.

In summary, the workload includes the web services code that will be used to exercise the vulnerability detection capabilities of the tools under benchmarking. For example, the workload for a benchmark targeting static code analysis tools capable of detecting SQL Injection vulnerabilities in web services must include the source code of web services with realistic (and well identified) SQL Injection vulnerabilities. On the other hand, for penetration testers access to the source code is not needed.

Two options are available regarding the definition of the workload (these apply to the three types of workloads):

- *Predefined workload*. The benchmark includes a predefined set of web services with vulnerabilities.
- *User-defined workload*. Leaves to the user the responsibility of selecting the target set of services.

As mentioned before, while the first approach guarantees some level of standardization and uniformity of results across different executions of the benchmark, the second allows circumventing the "*gaming*" problem and best represents the user specific development conditions, thus providing more realistic results. A key aspect is that the workload should include both vulnerable and non-vulnerable services in order to better characterize the tools under assessment (e.g., vulnerable services are useful to gather coverage metrics while non-vulnerable services help on assessing false positive rates).

In both cases, information about the vulnerabilities that exist in the target web services is needed in order to be able to calculate the metrics. This can be obtained by extensively searching the web services for vulnerabilities, using different techniques, including penetration testing, code inspection, static analysis, etc. For the case of predefined workloads this information must be provided together with the benchmark specification. On the other hand, for user-defined workloads, the benchmark may provide only guidelines on how to perform the workload characterization or include tools to facilitate the work.

As different vulnerability detection approaches report vulnerabilities in different ways, different characterizations about the existing vulnerabilities may be required (e.g., the number of vulnerable inputs is needed for penetration testing tools, while the number of vulnerable lines of code (LoC) is required for static analysis tools), depending on the vulnerability detection approaches of tools targeted by the benchmark (as defined in the benchmark domain).

3.3 Procedure

Although the detailed procedure depends on the specificities of the benchmark, we proposed three main phases:

1. *Preparation*. Prepare the execution, including:
   a. *Workload selection and characterization*. Required only when the benchmark leaves to the user the responsibility of selecting the target services. In such case, the user has to define the web services and characterize the existing vulnerabilities.
   b. *Tools identification*. Identify the vulnerability detection tools to be benchmarked.
2. *Execution*. Use the tools under benchmarking to detect vulnerabilities in the workload services.
3. *Comparison*. Characterize the tools benchmarked. This includes two steps:
   a. *Metrics calculation*. Analyze the vulnerabilities reported by the tools (i.e., confirm true positives and identify false positives) and calculate the metrics.

TABLE 1
Tools Under Benchmarking

Code	Provider	Tool	Technique
VS1	HP	WebInspect	Penetration testing
VS2	IBM	Rational AppScan	
VS3	Acunetix	Web Vuln. Scanner	
VS4	Univ. Coimbra	IPT-WS	
SA1	Univ. Maryland	FindBugs	Static code analysis
SA2	SourceForge	Yasca	
SA3	JetBrains	IntelliJ IDEA	
RAD	Univ. Coimbra	CIVS-WS	Anomaly detection

b. *Ranking and selection*. Rank the tools using F-Measure, precision, and recall. Select the most effective tool (or tools) using the preferred ranking.

In the case of benchmarks based on a predefined workload Step 1.a is not required, as the target web services are characterized in the benchmark specification (including the number of existing vulnerabilities). On the other hand, for benchmarks based on a user-defined workload Step 1.a is extremely relevant, as it greatly influences the benchmark results (e.g., if the workload does not contain representative vulnerabilities then the measures will not be representative of the tools effectiveness).

The benchmark execution is a straightforward process and consists of using each tool to detect vulnerabilities in the workload code. Depending on the tool under benchmarking this may require some configuration of the tool parameters. After executing the benchmark it is necessary to compare the vulnerabilities detected by the tool with the ones that effectively exist in the workload code. Vulnerabilities correctly detected are counted as true positives and vulnerabilities detected but that do not exist in the code are counted as false positives. This is the information needed to calculate the precision and recall of the tool, and consequently the F-measure.

4 Case Study

To demonstrate the proposed approach we designed two different benchmarks:

- *VDBenchWS-pd*. Benchmark based on a predefined workload, targeting vulnerability detectors based on penetration testing, static analysis, and runtime anomaly detection, able to detect SQL Injection in SOAP web services. This benchmark is presented in Section 5.
- *PTBenchWS-ud*. Benchmark based on a user-defined workload, targeting penetration testing tools, able to detect SQL Injection vulnerabilities in SOAP web services. This benchmark is presented in Section 6.

These benchmarks have been used to assess and compare a set of vulnerability detection tools, which are summarized in Table 1 (the *code* is used to refer the tools during experimental evaluation and results description). As shown, four *penetration testing tools* have been benchmarked, including three well-known commercial tools. The last penetration tester considered implements the approach proposed in [29]. An important aspect is that, when allowed by the testing tool, information about the domain of each input parameter was provided. If the tool requires an exemplar invocation per operation, the exemplar respected the input domains. All the tools in this situation used the same exemplar to guarantee a fairness.

Three vastly used *static code analyzers able* to detect vulnerabilities in Java applications' source or bytecode have also been considered in this study, namely: FindBugs, Yasca, and IntelliJ IDEA. During the experiments the static analyzers were configured to fully analyze the services code. For the analyzers that use binary code, the deployment ready version was used.

The last tool, named Command Injection Vulnerability Scanner for Web Services (CIVS-WS) [21], combines *runtime anomaly detection* with penetration testing for uncovering SQL Injection vulnerabilities in web services.

5 Example 1: Benchmark Based on a Predefined Workload (VDBenchWS-pd)

In this section we present a benchmark (VDBenchWS-pd) targeting the following domain:

- *Class of web services*. SOAP web services implemented in Java, which are nowadays widely used for data exchange and systems integration [30].
- *Type of vulnerabilities*. SQL Injection. Vulnerabilities allowing SQL Injection are particularly relevant in web services [2], as these frequently use a data persistence solution based in a relational database.
- *Detection approaches*. penetration testing, static code analysis, and runtime anomaly detection [31], [32], [33].

5.1 Workload Definition

As mentioned before, the workload is the component most influenced by the benchmarking domain and strongly determines the benchmark results. In order to define a representative workload we have decided to adapt code from three standard benchmarks developed by the Transactions processing Performance Council, namely: TPC-App, TPC-C, and TPC-W (see details http://www.tpc.org). TPC-App is a performance benchmark for web services infrastructures and specifies a set of web services accepted as representative of real environments. TPC-C is a performance benchmark for transactional systems and specifies a set of transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at warehouses. Finally, TPC-W is a benchmark for web-based transactional systems. The business is a retail store over the Internet where several clients access the website to browse, search, and process orders.[1]

As an adaptation of real applications the proposed workload follows the *realistic workloads* approach, and thus needs to include realistic SQL Injection vulnerabilities. Although feasible, artificial vulnerabilities injection [10] would

1. Although TPC-C and TPC-W do not define the transactions in the form of services, they can easily be implemented as such.

TABLE 2
Vulnerabilities Found in the Workload

BM	Service Name	Vuln. Inputs	Vuln. Queries	LoC	Avg. C.
TPC-App	ProductDetail	0	0	121	5
	NewProducts	15	1	103	4.5
	NewCustomer	1	4	205	5.6
	ChangePaymentMethod	2	1	99	5
TPC-C	Delivery	2	7	227	21
	NewOrder	3	5	331	33
	OrderStatus	4	5	209	13
	Payment	6	11	327	25
	StockLevel	2	2	80	4
TPC-W	AdminUpdate	2	1	81	5
	CreateNewCustomer	11	4	163	3
	CreateShoppingCart	0	0	207	2.67
	DoAuthorSearch	1	1	44	3
	DoSubjectSearch	1	1	45	3
	DoTitleSearch	1	1	45	3
	GetBestSellers	1	1	62	3
	GetCustomer	1	1	46	4
	GetMostRecentOrder	1	1	129	6
	GetNewProducts	1	1	50	3
	GetPassword	1	1	40	2
	GetUsername	0	0	40	2
Total		56	49	2654	-

introduce complexity and suffer from representativeness issues. When possible, the option should be to consider code with real vulnerabilities, (inadvertently) introduced by the developers during the coding process. This way, for the present work we invited an external developer to implement the TPC-App web services (without disclosing the objective of the implementation) and successfully searched the web for publicly available implementations of TPC-C and TPC-W, which were adapted to the form of web services by the same external developer (this consisted basically on the encapsulation of the transactions as web services). Obviously, this was a risky choice as there was some probability of getting code without vulnerabilities. However, as expected, the final code includes several SQL Injection vulnerabilities (see Table 2), which is representative of the current situation in real web services development (as shown in [5], [18]).

The workload services are currently implemented in Java. Although this is not relevant when benchmarking penetration testing tools, it limits the application of the benchmark to static code analyzers and runtime anomaly detectors that look for vulnerabilities in Java code. This way, to increase the domain of the benchmark, we would need to develop the workload in more languages. Nevertheless, the current implementation is sufficient to demonstrate our benchmarking approach, as Java is a language widely used to implement web services and there are many vulnerability detection tools for Java code.

To characterize the vulnerabilities that exist in the workload code, we invited a team of three external developers, with two or more years of experience in security of database centric applications to conduct a formal inspection of the code looking for vulnerabilities. As the different vulnerability detection approaches considered report vulnerabilities in different ways (penetration testers report the vulnerable inputs, while static code analyzers and runtime anomaly detectors report the vulnerable lines of code), we asked the security experts to identify both the input parameters and the source code lines prone to SQL Injection attacks. Table 2 presents the summary of the vulnerabilities detected by the security experts, the total number of lines of code per service, and the average Cyclomatic Complexity [34] (Avg. C.) of the code (calculated using SourceMonitor [35]. The results show a total of 56 vulnerable inputs and of 49 vulnerable SQL queries in the set of services considered.

In order to exercise the tools under benchmarking in a more exhaustive and realistic manner we decided to generate additional versions of the web services. The first step consisted of creating a new version for each service with all the known vulnerabilities fixed. Then we generated several versions for each service, each one having only one vulnerable SQL query. This way, for each web service we have one version without known vulnerabilities, one version with N vulnerabilities, and N versions with one vulnerable SQL query each. This accounts for a total of 80 versions, with *158 vulnerable inputs and 87 vulnerable queries*, which we believe is enough to exercise detection tools (see Section 5.3 for the properties).

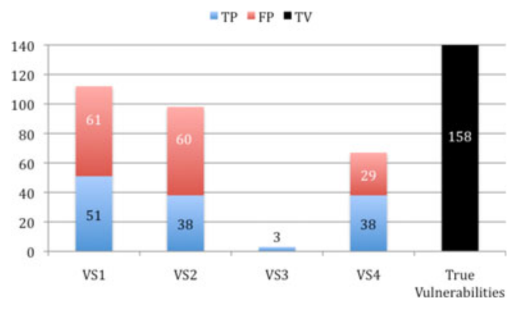
It is important to emphasize that we are aware of the limitations of the workload code. In fact, this code may not be representative of all the SQL Injection vulnerability patterns found in real web services. However, what is important is to define the benchmark components in such a way that allow characterizing the effectiveness of the tools under benchmarking in a relative manner (i.e., that allow establishing comparisons between tools). Based on the extensive experimental evaluation conducted, and in particular on the benchmark properties discussion, we believe that the proposed workload is sufficient to assess and compare the effectiveness of SQL Injection vulnerability detection tools for web services. Nevertheless, the proposed benchmark can easily be extended to include more services. Readers can find details at [36].

5.2 Experimental Evaluation

The tools under benchmarking (selected in *Phase 1*: *Preparation* and presented in Table 1) were run over the workload code (*Phase 2. Execution*). The vulnerabilities detected were then compared with the existing ones and used to calculate the benchmark metrics and rank the tools (*Phase 3. Comparison*). Vulnerabilities correctly detected by the tools were counted as true positives. Vulnerabilities detected by the tools that did not match any of the known ones were manually analyzed before being classified as false positives. Although this is not a step included in the benchmarking approach, it was useful to validate the result of the code reviews conducted by the security experts. The outcome was that no additional vulnerabilities were identified by any of the tools, which gives us some guarantee that the code reviews were conducted in an appropriate manner and identified all the vulnerabilities.

TABLE 3
VDBenchWS-pd Benchmarking Results

Tool	F-Measure	Precision	Recall
VS1	0.378	0.455	0.323
VS2	0.297	0.388	0.241
VS3	0.037	1	0.019
VS4	0.338	0.567	0.241
SA1	0.691	0.923	0.552
SA2	0.780	0.640	1
SA3	0.204	0.325	0.149
CIVS-WS	0.885	1	0.793



Tool	% TP
VS1	32.28%
VS2	24.05%
VS3	1.9%
VS4	24.05%

Tool	% FP
VS1	54.46%
VS2	61.22%
VS3	0%
VS4	43.28%

Fig. 1 VDBenchWS-pd results for the penetration testing.

5.2.1 Overall Benchmarking Results

Table 3 presents the overall benchmarking results. As we can see, the anomaly detection tool (CIVS-WS) is the one that presents the higher F-Measure. Additionally, two of the static code analysis tools (SA1 and SA2) present better results than the penetration testing tools. SA3 and VS3 are the tools with the lowest F-Measure.

The benchmark measures can be easily used to rank the tools under benchmarking (*Step 3.b: Ranking and Selection*) according to three criteria: precision, recall, and F-Measure (a balance between precision and recall). Table 4 presents a possible ranking for the tools. We divide the ranking in two, considering the approach used to report vulnerabilities (vulnerable inputs or vulnerable SQL queries), as defining a single ranking for tools that report vulnerabilities in different ways may not be meaningful (nevertheless, the benchmark measures allow such a ranking). Tools presented in the same cell are ranked in the same position due to the similarity of the results.

As we can see in Table 4, CIVS-WS seems to be the most effective tool considering both F-Measure and precision. However, the most effective tool when we consider recall is SA2, being CIVS the second best. VS3 seems to be the least effective tool in terms of F-Measure and recall. However, it has a very good precision (in fact it reported no false positives, but detected only three of the existing vulnerabilities). Excluding SA3, static analysis appears to be a better option than penetration testing.

5.2.2 Results for Penetration Testing

Fig. 1 shows the vulnerabilities reported by the penetration testing tools (the last bar in the graph presents the total number of vulnerabilities in the workload code). As we can see, the different tools reported a different number of vulnerabilities. An important observation is that all the tools detected less than 35 percent of the known vulnerabilities. VS1 identified the higher number of vulnerabilities (≈32 percent of the total vulnerabilities). However, it was also the scanner with the higher number of false positives.

VS2 and VS4 reported the same number of vulnerabilities (although not exactly the same vulnerabilities), but VS2 reported a much higher number of false positives. The very low number of vulnerabilities detected by VS3 can be partially explained by the fact that this tool does not allow the user to set any information about input domains, nor it accepts any exemplar request. This means that the tool generates a completely random workload that, probably, is not able to test the parts of the code that require specific input values.

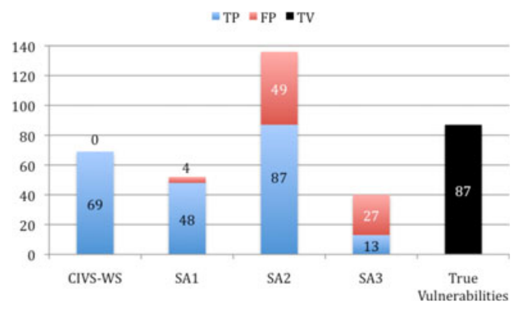
TABLE 4
VDBenchWS-pd Tools Ranking

	Criteria	1st	2nd	3rd	4th
Inputs	F-Measure	VS1	VS4	VS2	VS3
	Precision	VS3	VS4	VS1	VS2
	Recall	VS1	VS2/VS4		VS3
Queries	F-Measure	CIVS	SA2	SA1	SA3
	Precision	CIVS	SA1	SA2	SA3
	Recall	SA2	CIVS	SA1	SA3

5.2.3 Results for CIVS-WS and Static Analysis

Fig. 2 shows the number of vulnerable SQL queries identified by the static analyzers and by the CIVS-WS tool. As shown, in general, CIVS-WS presents better results than the static analyzers (although it has a lower true positives rate than SA2).

Considering only the static analyzers, SA2 detected the higher number of vulnerabilities, with 100 percent of true positives (an excellent result), but identified 49 false positives, which represents ≈36 percent of the vulnerabilities pointed. The high rate of false positives is, in fact, a problem shared by SA3, which reported more than ≈67 percent of false positives. The reason is that these tools detect certain patterns that usually indicate vulnerabilities, but many times they detect vulnerabilities that do not exist, due to intrinsic limitations of the static profile of the code.



Tool	% TP
CIVS	79.31%
SA1	55.17%
SA2	100%
SA3	14.94%

Tool	% FP
CIVS	0%
SA1	7.69%
SA2	36.03%
SA3	67.50%

Fig. 2 VDBenchWS-pd results for CIVS-WS and static analysis.

TABLE 5
Results for Third-Party Web Services

Tool	TP	FP	F-Measure	Precision	Recall
VS1	31	5	63.9%	86.1%	50.8%
VS2	22	1	52.4%	95.7%	36.1%
VS3	6	0	17.9%	100%	9.8%
VS4	28	0	62.9%	100%	45.9%
SA1	23	7	79.3%	76.7%	82.1%
SA2	28	10	84.9%	73.7%	100%
SA3	11	4	51.2%	73.3%	39.3%
CIVS-WS	28	0	100%	100%	100%

5.3 Properties Discussion

The *representativeness* of the workload greatly influences the representativeness of the benchmark. As mentioned before, we are aware of the limitations of the workload code, as it may not be representative of all the SQL Injection vulnerability patterns found in web services. However, the services used were adapted from implementations of benchmarks that are widely accepted as representative of real systems. This way, we take advantage of years of reasoning and maturity around these systems. Additionally, this code is not biased, as it was implemented by external developers with the goal of evaluating other characteristics of the system. Our thesis is that the workload should be good enough to allow the comparison of vulnerability detection tools. In fact, what is important is that the benchmark results accurately portray the tools effectiveness in a relative manner. Additionally, the process to extend the benchmark by adding services or upgrading the existing ones only requires that the new services are ready to run and have their vulnerabilities documented by an inspection team.

Comparing the benchmarking results with the effectiveness of the tools under benchmarking in different scenarios allows us to check if the benchmark accurately portrays the effectiveness of vulnerability detection tools in a relative manner. This way, we used of the tools to detect vulnerabilities in a small set of third-party web services. Eight web services implementing 28 operations have been used in the study. To avoid selecting services that fit the characteristics of the benchmark workload (and thus get biased results) we adopted the services from a previous study (see [21]). Due to space constraints we do not present details on the services (those can be found in [21]).

To perform a correct evaluation we extensively reviewed the source code looking for vulnerabilities. Sixty one vulnerable parameters and 28 vulnerable queries were identified (false positives were eliminated by cross-checking the results from different experts). Table 5 presents a summary of the results reported by the tools. As expected, the measures are not equal to the ones reported by the benchmark. This is normal as this new set of services has different code characteristics and different SQL Injection vulnerabilities. Table 6 presents the tools ranking obtained using this data (tools in the same cell are ranked in the same position). Comparing this ranking with the one proposed using the benchmark measures (see Table 4) we can observe the following: 1) the ranking based on the F-Measure is precisely the same; 2) the ranking based on precision differs for VS2 and VS1; and 3) the ranking based on recall is the same. This suggests that the tools' ranking derived from the benchmarking campaign adequately portrays the relative effectiveness of the tools. However, to prove the property and improve the benchmark representativeness, more vulnerable web services need to be added to the workload.

TABLE 6
Ranking Based on Third-Party Services

	Criteria	1st	2nd	3rd	4th
Inputs	F-Measure	VS1	VS4	VS2	VS3
	Precision	VS3/VS4		VS2	VS1
	Recall	VS1	VS4	VS2	VS3
Queries	F-Measure	CIVS	SA2	SA1	SA3
	Precision	CIVS	SA1	SA2	SA3
	Recall	SA2/CIVS		SA1	SA3

TABLE 7
VDBenchWS-pd Repeatability Results

	VS1			SA2		
	Run 0	Run 1	Run 2	Run 0	Run 1	Run 2
F-Measure	0.378	0.381	0.378	0.78	0.78	0.78
Precision	0.455	0.452	0.455	0.64	0.64	0.64
Recall	0.323	0.329	0.323	1	1	1

The services used to validate our benchmark came from a study performed by the same authors, which may constitute a *threat to the validity* of the benchmark representativeness. However, these services were gathered from the web, in an attempt to create a set of services with vulnerabilities that would allow us to have some insight on the effectiveness of vulnerability detection tools. It is also important to understand that the access to standardized sets of services with vulnerabilities online is very limited. If this sometimes limits the amount of third-party resources to validate our results, it also highlights the importance of this kind of studies and benchmarks that may lay the basis for future comparisons and evaluations.

Regarding *portability*, the benchmark seems to be quite portable. In fact, we were able to successfully benchmark four penetration testers, three static code analyzers, and one anomaly detector. It is important to emphasize that these tools are provided by different vendors and have very different functional characteristics. The benchmark is portable because it is not based on the implementation details of any specific tool.

The proposed benchmark must report similar results when used more than once over the same tool. To check *repeatability* we executed the benchmark for VS1 and SA2 (penetration tester and static code analyzer with the higher F-Measure) two more times. Table 7 presents the results of the three executions. As we can see, the results for the SA2 are always the same. This was expected as static code analyzers analyze the code in a deterministic manner, which removes variance from the results. On the other hand some small variations can be observed for VS1. However, these variations are always under 0.01, which suggests that the benchmark is quite repeatable.

The benchmark does not require any changes to the benchmarked tools, which guarantees the *non-intrusiveness* property. This is possible because the measures portray tools effectiveness from the point-of-view of the service they provide (i.e., vulnerabilities reported) and not based on the internal behavior.

The use of the proposed benchmark is quite *simple to use* (in part, because most steps are automatic). In fact, we have been able to run the benchmark for all the tools in about six man-days, which correspond to an average of 0.75 man-days per benchmarking experiment. Running the benchmark only requires executing the tools and comparing the reported vulnerabilities with the ones that effectively exist. As different tools report vulnerabilities in different formats (e.g., XML file, text file, GUI), to automate the vulnerability comparison step, we need to convert the output of the tools to a common format. Although possible, we decided not to do it in this work (it is just a technical issue with no scientific relevance).

6 Example 2: Benchmark Based on a User-Defined Workload (PTBenchWS-ud)

In this section we present a benchmark targeting the following domain:

- *Class of web services*. SOAP web services [30].
- *Type of vulnerabilities*. SQL Injection [2].
- *Detection approaches*. penetration testing [31].

6.1 Workload Definition and Characterization

The set of web services that will compose the benchmark workload is to be defined by the benchmark user. This should include a number of SOAP web services with and without SQL Injection vulnerabilities. As defined in Section 3.2, this workload can be real, realistic, or synthetic. What is important is to understand that the workload definition determines the benchmark results and properties, thus the user should be aware of the impact of the decisions regarding the web services being considered.

A key aspect is the characterization of the existing vulnerabilities. As the target of the benchmark are penetration testing tools, the number of vulnerable inputs is needed to later calculate the metrics. Such characterization can be based on an extensive *manual analysis* of the selected web services in order to identify the existing vulnerabilities (in a similar way to what we did in Section 5). The problem is that such process can become extremely expensive if the set of services is large and complex. Thus, as an alternative, we propose an *automatic approach for identifying the base set of vulnerabilities*, grounded on the use of a tool that combines attack signatures and interface monitoring to detect SQL Injection vulnerabilities in web services [37]. Although the proposed approach does not guarantee the detection of all existing vulnerabilities, it assures that no false positives are reported. The vulnerabilities detected will serve as reference to estimate the number of true and false positives of the tools under benchmarking, as discussed in the next sections.

A key aspect is that the proposed benchmark can be easily extended to other types of injection vulnerabilities. The only constraint is that the benchmark user has to define a workload containing other types of vulnerabilities

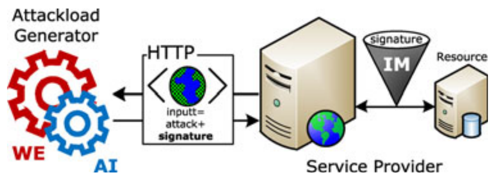


Fig. 3 Simplified representation of the Sign-WS detection tool.

and then manually characterize those vulnerabilities. In fact, although the technique presented next can be easily extended to other Injection vulnerabilities (see [37]), in this work we are targeting only SQL Injection.

6.1.1 Vulnerabilities Identification

For the vulnerability identification, the benchmark includes the Sign-WS tool, which implements the technique proposed in [37]. This technique addresses the limitations of penetration testing by using attack signatures and interface monitoring for the detection of injection vulnerabilities in web services. The goal is to improve the detection process by providing enhanced visibility, yet without needing to access or modify the code of the target service. The key assumption is that most injection attacks manifest, in some way, in the interfaces between the attacked web service and other resources (e.g., database, operating system) and services. For example, a successful SQL Injection attack leads the service to send malicious SQL queries to the database. Thus, it can be observed in the SQL interface between the service and the database server.

Comparing to traditional penetration testing, this approach allows achieving higher effectiveness, as it provides the information needed to increase the number of vulnerabilities detected and completely eliminate the false positives. Still, the application is tested as a black-box as the only requirement is to monitor is the interface between the application and the used resources as allowed in this specific scenario. A workload emulator module analyzes the web service description and generates a set of valid requests, which are afterwards modified by the attack injector module. During this process, the interfaces are monitored to detect the signatures that represent vulnerabilities. Fig. 3 portrays the approach.

6.1.2 Benchmark Metrics Estimation

The signatures and monitoring approach provides information that is not available to the penetration testing tools under benchmarking, thus it is expected to detect more vulnerabilities and present less false positives. In fact, and based on the precise detection of signatures, no false positives are expected (see [37]). Thus, the vulnerabilities identified using interface monitoring can be used as a baseline for evaluating other tools, as explained next.

The detection coverage is the percentage of real vulnerabilities that are detected by a tool. Assuming that the number of vulnerabilities reported by the signatures and monitoring approach is a valid estimation of the total number of existing vulnerabilities, then the percentage of those vulnerabilities that are reported by a given penetration testing tool is also a valid estimation for its vulnerability

TABLE 8
Coverage and False Positives for the Example

Tool	Estimated Coverage Rate	Estimated False Positive Rate
PTA	8 / 10 = 80%	6 / (8 + 6) ~= 43%
PTB	4 / 10 = 40%	1 / (4 + 1) = 20%

detection coverage. Similarly, we can estimate the false positives rate, which represents the percentage of vulnerabilities reported by the tool that in fact do not exist. Considering that the set of vulnerabilities detected using our approach does not include false positives (guaranteed by the adequate signatures), we can estimate the false positives rate of a penetration tester by calculating the difference between the vulnerabilities reported by such tool and the ones identified via interface monitoring.

To better understand our proposal, let's consider a simple scenario. Consider that the signatures system is able to detect 10 SQL Injection vulnerabilities in a given web service and that a penetration testing tool A detects eight of those and six more, and that a penetration testing tool B detects four of those and one more. As shown in Table 8 we can use these values to estimate the coverage and false positives of both tools. Note that, the considered total number of vulnerabilities is only an estimated value, as there is no guarantee of perfect detection coverage from the signatures system. This way, the total number of vulnerabilities will be always equal or superior to the estimated number of vulnerabilities and this fact can diminish the importance of the evaluation in two ways.

First, the coverage rates calculated for the evaluated tools may be overestimated. Although this seems a key problem, it is important to stress that the evaluation of the different tools is done for benchmarking purposes (e.g., to select one) and not for assessing actual effectiveness (as this depends on several factors, including the target application, programing language, type of vulnerability, etc.). Thus, taking a relative perspective of the results (rather than an absolute perspective), the overestimation should be equivalent for all the evaluated tools, affecting them similarly, while maintaining a fair comparison.

Second, the false positive rates for the evaluated penetration testers may also be overestimated. Again, although this seems a major issue, in practice the impact will be minor, as the Sign-WS tool uses extra information on the internal behavior of the web services, provided by the signatures and the interface monitoring, to achieve much higher detection coverage than the penetration testing tools, which are the target of this benchmark. This way, it is highly probable that a vulnerability detected by a tester will also be detected by Sign-WS. In fact, during our experiments only one vulnerability was detected by a penetration testing tool and not detected by Sign-WS, representing less than 1 percent of the total vulnerabilities considered. Thus, the estimation for the false positives should be close to the real values, which again is adequate for a relative view of the results. Finally, it is important to remember that the dependence on the Sign-WS tool can be avoided if the benchmark user has the resources to perform a manual characterization of the workload.

TABLE 9
PTBenchWS-ud Benchmarking Results

Tool	F-Measure	Precision	Recall
VS1	0.437	0.446	0.427
VS2	0.353	0.388	0.325
VS3	0.050	1.000	0.026
VS4	0.413	0.567	0.325

6.2 Experimental Evaluation

To demonstrate the benchmark we considered the set of web services included in the benchmark proposed in Section 5 (this will allow to compare the results of both benchmarking campaigns). However, we assume no knowledge about the existing vulnerabilities. This way, to characterize the workload (*Phase 1. Preparation*) we used the attack signatures and interface monitoring approach. The penetration testing tools under benchmarking (presented in Table 1) were run over the workload code (*Phase 2. Execution*). The vulnerabilities reported were manually confirmed and compared with the ones identified in the preparation phase to calculate the benchmark metrics and rank the tools (*Phase 3. Comparison*).

6.2.1 Characterization of the Workload

The vulnerabilities detected by the Sign-WS tool have been manually confirmed to guarantee the absence of false positives (as claimed in [37]). The tool indeed reported 0 false positives, but the coverage was only of 74.05 percent (*117 true positives* out of 158 true vulnerabilities). As we will show later, although not all the true vulnerabilities are considered in the calculation of the metrics, the ones reported by Sign-WS are enough for a good estimation of the tools effectiveness.

6.2.2 Benchmarking Results

Table 9 presents the benchmark metrics for each tool, considering as base set the 117 vulnerabilities reported by the Sign-WS tool. As we can see, VS1 is the tool with the highest F-Measure, closely followed by VS4. VS2 presents very poor F-Measure results. Regarding precision, VS3 is the best as it reported no false positives, and VS4 presents the best results. Finally, in terms of recall, VS1 has the best results, while VS2 and VS4 perform equally. The recall of VS3 is very low as it detected only three vulnerabilities.

The results presented in Table 9 were used to rank the tools according to the different criteria: F-Measure, Precision, and recall. Table 10 shows the proposed rank.

Fig. 4 shows the vulnerabilities reported by the penetration testing tools (the last bar in the graph presents the number of vulnerabilities detected by the Sign-WS tool). A key observation is that all the tools detected less than 43 percent of the vulnerabilities reported by Sign-WS, which makes the base set of vulnerabilities a good reference. A key aspect is that VS1 reported a true vulnerability that was not reported by Sign-WS, but this was the only case. We will discuss later the impact of this in the metrics calculation when compared to the benchmark based on a predefined workload.

TABLE 10
PTBenchWS-ud Tools Ranking

Criteria	1st	2nd	3rd	4th
F-Measure	VS1	VS4	VS2	VS3
Precision	VS3	VS4	VS1	VS2
Recall	VS1	VS2 / VS4		VS3

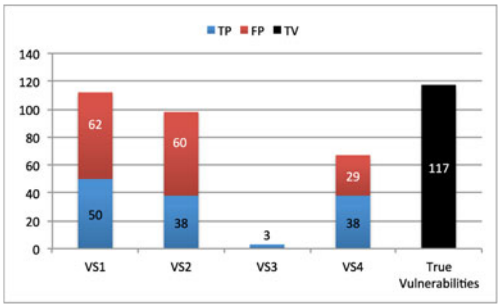
6.2.3 Comparison with the VDBenchWS-pd Benchmark

A key aspect is to compare the results of the present benchmark with the ones of the benchmark based on a predefined workload. Note that, although we are considering the same set of web services, in the benchmark based on a user-defined workload we consider only a subset of the existing vulnerabilities (has reported by the Sign-WS tool). This is obviously also a way for validating the workload characterization and metrics estimation approaches proposed to support the benchmark.

Table 11 summarizes the metrics obtained for both benchmarks (it is a marge of Tables 3 and 9 for the case of the penetration testing tools). As expected, the metrics differ slightly because the base set of true vulnerabilities is different in the two cases. The F-Measure values are consistently lower in VDBenchWS-pd. This is due to the higher values for recall in PTBenchWS-ud, which are related to the lower number of true vulnerabilities considered as reference. Finally, precision is the same in both benchmarks, except for the case of VS1. This is due to the fact that VS1 detected a vulnerability that was not reported by the Sign-WS tool, and thus was not included in the base set of true vulnerabilities. This obviously harms the reported tool precision, but as the coverage of the Sign-WS is very high, the impact is minimum. In fact, it does not affect the relative results and the tools' ranking is precisely the same for both benchmarks (see Tables 6 and 10).

6.3 Properties Discussion

The *representativeness* of the benchmark depends on workload defined by the user. In fact, although leaving to the user the responsibility for defining the workload allows obtaining environment-specific results and prevents "*gamming*," it may also affect the validity of the results if the web services and vulnerabilities in the workload are not representative of real scenarios. Obviously, in the case of the experimental evaluation presented in the previous section, the representativeness issues are as discussed in Section 5.3. The ranking obtained (equal to the benchmark presented in Section 5) suggests that the procedure and the approaches for characterizing the workload and estimating the metrics are quite adequate for characterizing the tools under assessment even when there is no previous knowledge about the existing vulnerabilities.



Tool	% TP
VS1	42.74%
VS2	32.48%
VS3	2.56%
VS4	32.48%

Tool	% FP
VS1	55.36%
VS2	61.22%
VS3	0.00%
VS4	43.28%

Fig. 4 PTBenchWS-ud results for the penetration testing.

TABLE 11
Results for Both Benchmarks

	Tool	F-Measure	Precision	Recall
VDBenchWS-pd	VS1	0.378	0.455	0.323
	VS2	0.297	0.388	0.241
	VS3	0.037	1	0.019
	VS4	0.338	0.567	0.241
PTBenchWS-ud	VS1	0.437	0.446	0.427
	VS2	0.353	0.388	0.325
	VS3	0.050	1.000	0.026
	VS4	0.413	0.567	0.325

Regarding *portability*, the benchmark seems to be quite portable in the specified domain. In fact, we were able to benchmark four penetration testers, from different vendors and having diverse functional characteristics.

In terms of *repeatability* we executed the benchmark for VS1 (penetration tester with the highest F-Measure) two more times. Small variations where observed, but they were always under 0.01, which suggests that the benchmark is quite repeatable. In fact, the repeatability results are similar to the ones discussed in Section 4.3, thus they are not further discussed due to space reasons.

The *non-intrusiveness* property is guaranteed, as the benchmark does not require any changes to the tools.

Although the proposed benchmark is quite *simple to use* (most steps are automatic), the fact that the user has to provide the workload and characterize the existing vulnerabilities, may increase its complexity. Obviously, the approach proposed for the metrics estimation based on the Sign-WS approach makes the work easier. We have been able to run the benchmark for all the tools in about four man-days, which correspond to an average of 1 man-day per experiment. In practice, running the benchmark only requires executing the tools and comparing the reported vulnerabilities with the ones reported by the Sign-WS tool. The problem of different formats in the reports (e.g., XML file, text file, GUI) also applies, and automating the vulnerability comparison step would require converting the output of the tools to a common format.

7 Conclusion

This paper proposed an approach to define benchmarks for vulnerability detection tools in web services. This approach has been used to define two concrete benchmarks targeting tools able to detect SQL Injection vulnerabilities. The first benchmark is based on a predefined workload, while the second leaves to the user the responsibility for defining that workload (thus avoiding "gamming" problems). Several tools have been benchmarked, including commercial and open-source tools.

The results show that the proposed benchmarks can be easily used to assess and compare penetration testers, static code analyzers, and anomaly detectors. In fact, the benchmark metrics provided an easy way to rank the tools under benchmarking, leading to similar rankings in in both cases. The properties of the benchmarks were validated and discussed in detail and suggest that *the proposed benchmarking approach can be applied in the field* to specify benchmarks for vulnerability detection tools targeting different domains.

Future work includes extending the benchmarks to other types of vulnerabilities and applying the benchmarking approach to define benchmarks for other types of web services. The approach can be extended for other domains as long as it is possible to provide a representative set of code with vulnerabilities (workload). The task of gathering and characterizing the workload may be costly, but in some scenarios it certainly may be worth the work as it allows us to understand the effectiveness of the different tools available to detect vulnerabilities. Also, we plan to study of vulnerability injection techniques for the automatic generation of syntactic workloads is an interesting challenge. Finally, we also plan to provide support for the validation of the results and the maintenance of a benchmark. An idea is to define a core branch of the benchmark and variations to more specific scenarios, and devise new rules to avoid *gaming* tactics, and to integrate new applications provided by external contributors and that we believe may be useful and representative, originating new versions of the benchmark.

ACKNOWLEDGMENTS

This work has been partially supported by the project ICIS - Intelligent Computing in the Internet of Services (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro Program and European Union's FEDER and the project CErtification of CRItical Systems (http://www.cecris-project.eu/, CECRIS), Marie Curie IAPP no. 324334, within the context of the EU Seventh Framework Programme (FP7).

REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*. first ed., Springer, 2010.

[2] S. Christey and R.A. Martin, "Vulnerability Type Distributions in CVE," The MITRE Corporation. V1, 1 2007.

[3] L. Lowis and R. Accorsi, "On a Classification Approach for SOA Vulnerabilities," *Proc. 33rd Ann. IEEE Int'l Computer Software and Applications Conf. (COMPSAC '09)*, vol. 2, pp. 439-444, 2009.

[4] M. Jensen, N. Gruschka, R. Herkenhoner, and N. Luttenberger, "SOA and Web Services: New Technologies, New Standards—New Attacks," *Proc. Presented at the Fifth European Conf. Web Services (ECOWS '07)*, pp. 35-44, 2007.

[5] M. Vieira, N. Antunes, and H. Madeira, "Using Web Security Scanners to Detect Vulnerabilities in Web Services," *Proc. IEEE /IFIP Int'l Conf. Dependable Systems Networks (DSN '09)*, pp. 566-571, 2009.

[6] TPC, "TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.3," http://www.tpc.org/tpc_app/, 2014.

[7] W. Meier, "eXist: An Open Source Native XML Database," *Proc. Revised Papers from the NODe 2002 Web and Database-Related Workshops Web, Web-Services, and Database Systems*, pp. 169-183, 2003.

[8] D. Stuttard and M. Pinto, *The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*. John Wiley & Sons, 2007.

[9] N. Antunes and M. Vieira, "Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services," *Proc. 15th IEEE Pacific Rim Int'l Symp. Dependable Computing (PRDC '09)*, pp. 301-306, 2009.

[10] J. Fonseca, M. Vieira, and H. Madeira, "Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks," *Proc. Presented at the 13th Pacific Rim Int'l Symp. Dependable Computing (PRDC '07)*, pp. 365-372, 2007.

[11] Y. Eytani and S. Ur, "Compiling a Benchmark of Documented Multi-Threaded Bugs," *Proc. 18th Int'l Parallel and Distributed Processing Symp.*, p. 266, 2004.

[12] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for Evaluating Bug Detection Tools," *Proc. Workshop the Evaluation of Software Defect Detection Tools*, pp. 1-5, 2005.

[13] V. Dallmeier and T. Zimmermann, "Extraction of Bug Localization Benchmarks from History," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 433-436, 2007.

[14] C.J. Van Rijsbergen, *Information Retrieval*. Buttersworth, 1979.

[15] N. Antunes and M. Vieira, "Benchmarking Vulnerability Detection Tools for Web Services," *Proc. IEEE Int'l Conf. Web Services (ICWS)*, pp. 203-210, 2010.

[16] NTA Monitor, "Annual Web Application Security Report," 2008.

[17] NTA Monitor, "Annual Security Report," May 2007.

[18] NTA Monitor, "Annual Web Application Security Report," 2011.

[19] V.B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proc. 14th Conf. USENIX Security Symp.*, vol. 14, pp. 18-18, 2005.

[20] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing Bug Finding Tools with Reviews and Tests," *Proc. 17th IFIP TC6/WG 6.1 Int'l Conf. Testing of Communicating Systems*, pp. 40-55, 2005.

[21] N. Antunes, N. Laranjeiro, M. Vieira, and H. Madeira, "Effective Detection of SQL/XPath Injection Vulnerabilities in Web Services," *Proc. IEEE Int'l Conf. Services Computing (SCC '09)*, pp. 260-267, 2009.

[22] W.G.J. Halfond and A. Orso, "Preventing SQL Injection Attacks Using AMNESIA," *Proc. 28th Int'l Conf. Software Eng.*, pp. 795-798, 2006.

[23] J. Gray, *The Benchmark Handbook*. Morgan Kaufmann Publishers Inc, 1993.

[24] K. Kanoun and L. Spainhower, *Dependability Benchmarking for Computer Systems*. John Wiley & Sons-IEEE CS Press, 2008.

[25] M. Vieira and H. Madeira, "Towards a Security Benchmark for Database Management Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '05)*, pp. 592-601, 2005.

[26] A.C. d. Araújo Neto and M. Vieira, "Selecting Secure Web Applications Using Trustworthiness Benchmarking," *Int'l J. Dependable and Trustworthy Information Systems*, vol. 2, no. 2, pp. 1-16, 2011.

[27] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Eng.*, vol. 10, no. 4, pp. 405-435, Oct. 2005.

[28] A. Tajpour and M.J. zade Shooshtari, "Evaluation of SQL Injection Detection and Prevention Techniques," *Proc. Second Int'l Conf. Computational Intelligence, Comm. Systems and Networks*, pp. 216-221, 2010.

[29] N. Antunes and M. Vieira, "Detecting SQL Injection Vulnerabilities in Web Services," *Proc. Fourth Latin-Am. Symp. Dependable Computing (LADC '09)*, pp. 17-24, 2009.

[30] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, no. 2, pp. 86-93, Mar./Apr. 2002.

[31] B. Arkin, S. Stender, and G. McGraw, "Software Penetration Testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84-87, Jan./Feb. 2005.

[32] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22-29, Sept./Oct. 2008.

[33] C. Kruegel and G. Vigna, "Anomaly Detection of Web-Based Attacks," *Proc. 10th ACM Conf. Computer and Comm. Security*, pp. 251-261, 2003.

[34] *Handbook of Software Reliability Engineering*. M.R. Lyu, ed., McGraw-Hill, 1996.

[35] Campwood Software, "SourceMonitor Version 2.5," http://www.campwoodsw.com/sourcemonitor.html, 2008.

[36] N. Antunes and M. Vieira, "Benchmarks for Vulnerability Detection tools for Web Services," http://eden.dei.uc.pt/~nmsa/publications/vdbench-ws.zip, 2013.

[37] N. Antunes and M. Vieira, "Enhancing Penetration Testing with Attack Signatures and Interface Monitoring for the Detection of Injection Vulnerabilities in Web Services," *Proc. IEEE Int'l Conf. Services Computing (SCC)*, pp. 104-111, 2011.

Nuno Antunes received the PhD degree in Information Science and Technology from the University of Coimbra, Portugal. He is a post-doctoral researcher at the University of Coimbra, Portugal. His interests include the development of dependable and secure web applications and services, and security benchmarking. He is a member of the IEEE Computer Society and a member of the IEEE.

Marco Vieira received the PhD degree in computer engineering from the University of Coimbra. He is an assistant professor at the University of Coimbra, Portugal. His interests include dependability and security benchmarking, experimental dependability evaluation, fault injection, software development processes, and software quality assurance. He is a member of the IEEE Computer Society.